1 Introduction

Graph traversal algorithms help in exploring nodes and edges of a graph systematically. Two fundamental algorithms for this purpose are **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. These are commonly used in pathfinding, connectivity checks, and solving various computational problems. **Flood Fill**, a variation of these, is widely used in image processing and gaming applications.

Without specified mentioned, assume `arr` is declared as a signed integer array. The following code is added to the beginning of all C++ programs.

```cpp
#include <bits/stdc++.h>
using namespace std;
```

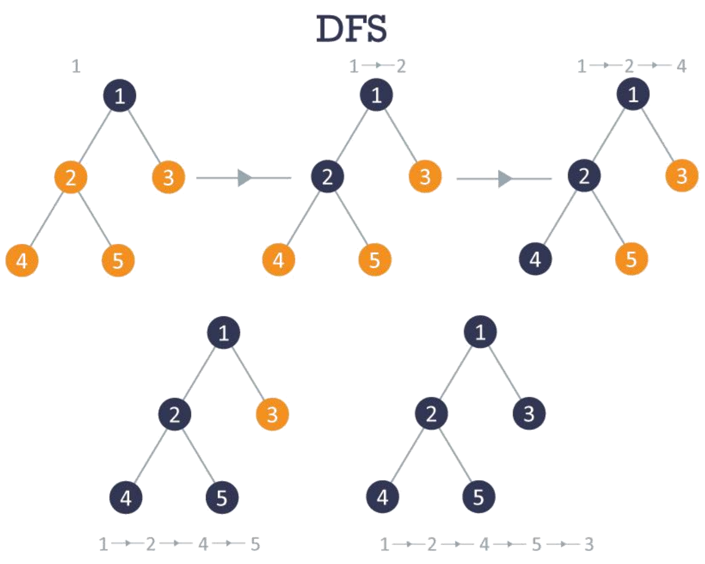
2 Depth-First Search (DFS)

Definition

DFS explores a graph by **going as deep as possible** before backtracking. It uses a **stack** or via **recursion** to keep track of visited nodes.

***Do not mix it with the DFS technique on branch and bound.**

- **Approach**: Start from a node, visit an adjacent unvisited node, and repeat until no more unvisited neighbors remain. Then, backtrack and continue.
- **Time complexity:** (V = vertices, E = edges)
  - **adjacency list: O(|V| + |E|)**
  - **adjacency matrix: O(|V|^2)**
- **Use Cases**: Solving mazes, detecting cycles in graphs, topological sorting, and pathfinding.



Cycle detection

- During our DFS process, if there exists a visited neighbor of current node except its “parent”, the graph contains a cycle.
- For directed graph, you can use Disjoint Set Union-Find to find cycles.
- If you have learnt about topological sort, you can use Kahn’s algorithm to find the existence of a cycle in the graph.

C++ Code Implementation

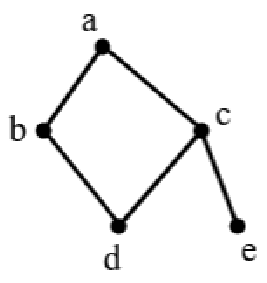
```
// By recursion
vector<vector<int>> adj;  // adjacency_list
vector<bool> vis;         // record whether the vertices has been traversed

void dfs(int u) {
  vis[u] = true;
  for (int v : adj[u])
    if (!vis[v]) dfs(v)
}
```

```
// By stack
vector<vector<int>> adj;  // adjacency_list
vector<bool> vis;         // record whether the vertices has been traversed

void dfs(int x) {
  stack<int> S;
  S.push(x);
  vis[x] = true;

  while (!S.empty()) {
    int u = S.top();
    S.pop();

    for (int v : adj[u]) {
      if (!vis[v]) {
        vis[v] = true;  // to ensure no replicated elements are in the stack
        S.push(v);
      }}}
```

Exercises

[CSP-J1 2021 Q14]

Taking a as the starting point, perform a depth-first traversal of the undirected graph on the right. The number of points b, c, d, and e that may be the last traversed is ( ).



A. 1 B. 2 C. 3 D. 4

[CSP-J1 2020 Section B Q3]

```
int n;
int d[50][2];
int ans;
void dfs(int n, int sum) {
  if (n == 1) {
    ans = max(sum, ans);
    return;
  }
  for (int i = 1; i < n; ++i) {
    int a = d[i - 1][0], b = d[i - 1][1];
    int x = d[i][0], y = d[i][1];
    d[i - 1][0] = a + x;
    d[i - 1][1] = b + y;
    for (int j = i; j < n - 1; ++j)
      d[j][0] = d[j + 1][0], d[j][1] = d[j + 1][1];
    int s = a + x + abs(b - y);
    dfs(n - 1, sum + s);
    for (int j = n - 1; j > i; --j)
      d[j][0] = d[j - 1][0], d[j][1] = d[j - 1][1];
    d[i - 1][0] = a, d[i - 1][1] = b;
    d[i][0] = x, d[i][1] = y;
  }
}
int main() {
  cin >> n;
  for (int i = 0; i < n; ++i)
  cin >> d[i][0];
  for (int i = 0; i < n;++i)
     cin >> d[i][1];
  ans = 0;
  dfs(n, 0);
  cout << ans << endl;
  return 0;
}
```

Assume the input *n* is a positive integer no more than 50, and *d*[*i*][0], *d*[*i*][1] are positive integers no more than 10000.

1. (1 mark)

   If the input *n* is 0, the program may fall into infinite loop or cause run-time error. (True / False)

2. (1 mark)

   If the input *n* is 20 and the remaining input are all 0, the output is 0. (True / False)

3. (1 mark)

   The output number will definitely not be less than any one of the inputs *d*[*i*][0] and *d*[*i*][1]. (True / False)

4. (2 marks)

   If the input *n* is 20, and 20 ‘9’s and 20 ‘0’s is input, the output is

   A. 1890 B. 1881 C. 1908 D. 1917

5. (2 marks)

   If the input *n* is 30, and 30 ‘0’s and 30 ‘5’s is input, the output is

   A. 2000 B. 2010 C. 2030 D. 2020

6. (4 marks)

   If the input *n* is 15, and ‘1’ to ‘15’, and ’15’ to ‘1’ is input, the output is

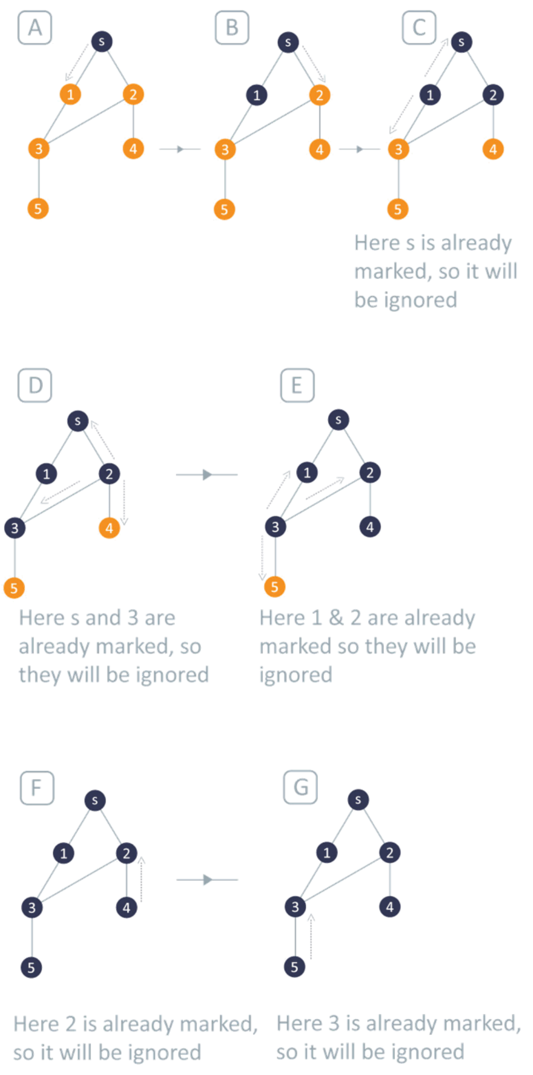
   A. 2440 B. 2220 C. 2240 D. 2420

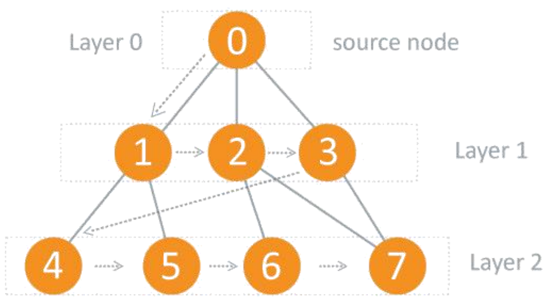
3 Breadth-First Search (BFS)

Definition

BFS explores a graph layer by layer using a queue. It visits all neighbors of a node before moving to the next layer. The algorithm process can be viewed as the process of fire propagation on a graph: initially only the starting point is on fire, and at every moment, the node with fire spreads the fire to all its adjacent nodes.

- **Approach**: Start from a node, visit all its unvisited neighbors, then move to their neighbors, and so on.
- **Time complexity:** (V = vertices, E = edges)
    - **adjacency list: O(|V| + |E|)**
    - **adjacency matrix: O($|V|^2$)**
- **Use Cases**: Shortest path in an unweighted graph, network broadcasting, and web crawling.


A
B
C
Here s is already marked, so it will be ignored
D
E
Here s and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored
F
G
Here 2 is already marked, so it will be ignored
Here 3 is already marked, so it will be ignored



Shortest path for unweighted graph

The so-called breadth-first approach is to try to visit the nodes in the same layer each time. If all nodes in the same layer have been visited, then visit the next layer. The result of this is that **the path found by the BFS algorithm is the shortest legal path from the starting point**. In other words, the path contains the smallest number of edges.
At the end of BFS, every node has been visited by the shortest path from the starting point to that point.

- We can maintain an array `dist[]` to find the shortest path from a fixed vertex. This method can be both applicable to unweighted graph and weighted graph.
- For the shortest path for a weighted graph, you should learn more advanced algorithm such as Dijkstra's algorithm or Floyd-Warshall algorithm.

Finding vertices contributes to the shortest path between two vertices

- Maintain two array d_A and d_B, storing the shortest distance from vertex A and B respectively.
- For a vertex v, if $d_A[v] + d_B[v] = d_A[B]$, then v contributes to the shortest path between vertex A and B.

The idea above can be extended to find edges contributes to the shortest path between two vertices.

Extension

BFS is the foundation of shortest path algorithm. The idea can be further extended:

- BFS + Double-ended queue = 0-1 BFS
- BFS + Priority queue = Dijkstra's Algorithm
- …

C++ Code Implementation

```
// By queue
vector<int> adj[5010];
vector<bool> vis;
queue<int> Q;

void bfs(int u)
{
    vis[u] = true;
    Q.push(u);
    while (!Q.empty())
    {
        u = Q.front();
        cnt++;
        Q.pop();
        for (int v : adj[u])
        {
            if (!vis[v]) {
                Q.push(v);
                vis[v] = true;
            }
        }}}
```

Exercises

[CSP-J1 2022 Q10]

The following statement about data structure is inappropriate:

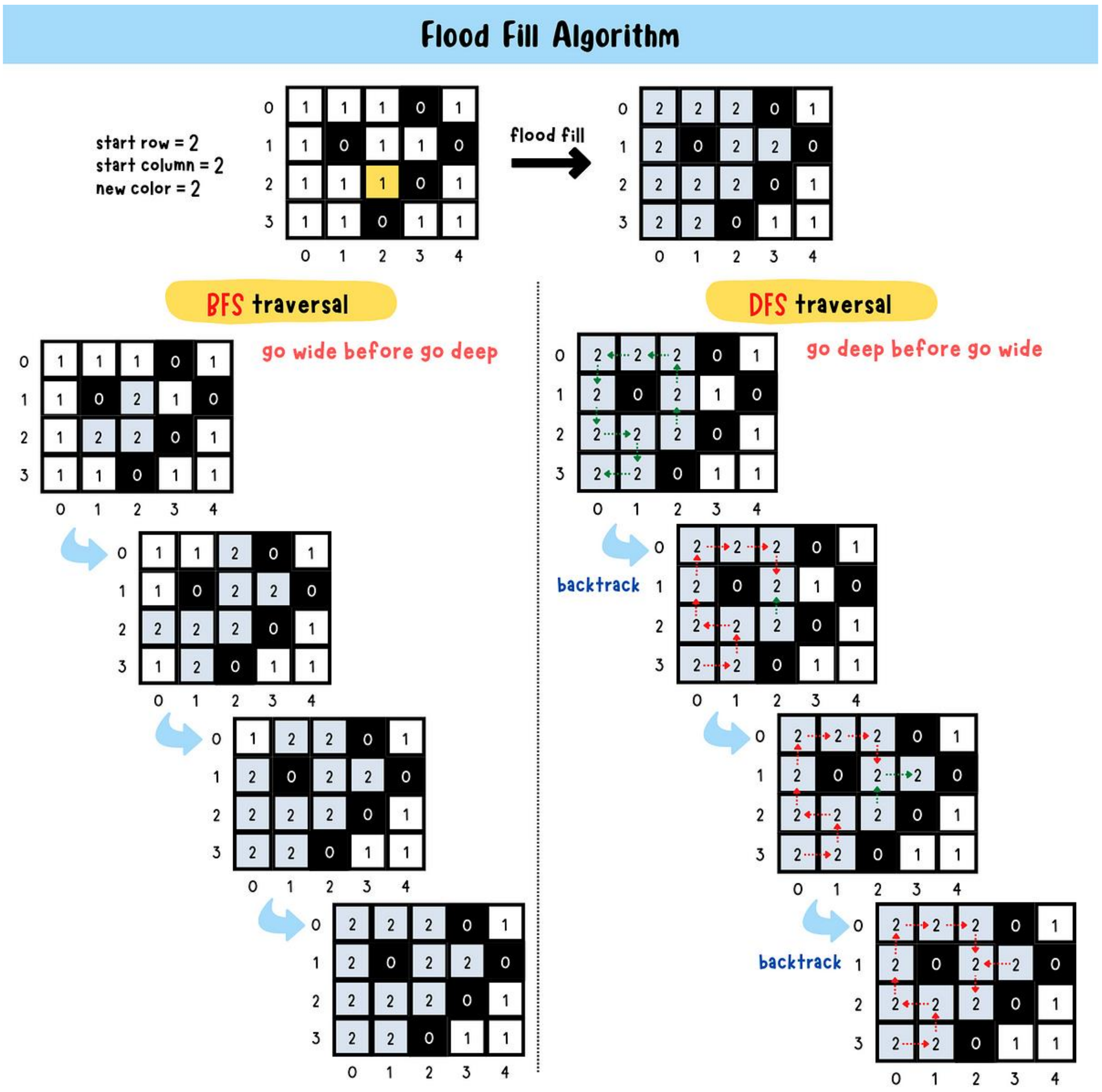
A. The data structure commonly used in the depth-first traversal algorithm of a graph is a stack.
B. The access principle of the stack is last in, first out, and the access principle of the queue is first in, first out.
C. Queues are often used in breadth-first search algorithms.
D. Stacks and queues are fundamentally different, and queues cannot be implemented using stacks.

4 Flood Fill

Definition

Flood Fill is a technique used to fill connected regions in a grid-based structure (like an image or a game map). It is similar to DFS or BFS but applied in a **2D grid** instead of a general graph. A grid can be modelled as an undirected graph, both **DFS and BFS** can be used to implement Flood Fill, however, **DFS** is used more often.

- **Approach**: When we detect an unvisited empty cell, we run DFS/BFS on it and count the number of cells in this region. After visiting a region, we perform DFS/BFS on the remaining unvisited empty cells until all empty cells are visited.
- **Time Complexity**: **O(N × M)** (for an N × M grid)
- **Use Cases**: Paint bucket tool in graphics software, solving connected component problems, and game map filling.



Exercises

[CSP-J1 2022 Q40-44]

Given an 8x8 image where pixel colors are marked by characters, the color fill operation is described as follows: Given the position of a starting pixel and a target color, replace the starting pixel and all reachable pixels (reachable defined as: pixels that can be reached through one or more moves in the four directions of up, down, left, and right, where the endpoint and all pixels along the path have the same color as the starting pixel) with the given color.

Complete the program below.

```
#include <bits/stdc++.h>
using namespace std;

const int ROWS = 8;
const int COLS = 8;

struct Point {
  int r, c;
  Point(int r, int c) : r(r), c(c) {}
};

bool is_valid(char image[ROWS][COLS], Point pt,
              int prev_color, int new_color) {
  int r = pt.r;
  int c = pt.c;
  return (0 <= r && r < ROWS && 0 <= c && c < COLS &&
          ① && image[r][c] != new_color);
}

void flood_fill (char image[ROWS][COLS], Point cur, int new_color) {
  queue<Point> queue;
  queue.push(cur);

  int prev_color = image[cur.r][cur.c];
  ②;

  while (!queue.empty()) {
    Point pt = queue.front();
    queue.pop();

    Point points[4] = {③, Point(pt.r - 1), pt.cm
                       Point(pt.r, pt.c + 1), Point(pt.r, pt.c - 1)};
    for (auto p : points) {
      if (is_valid(image, p, prev_color, new_color)) {
        ④;
        ⑤;
      }
    }
  }
}

int main() {
  char image[ROWS][COLS] = {{'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g'},
                            {'g', 'g', 'g', 'g', 'g', 'g', 'r', 'r'},
                            {'g', 'r', 'r', 'g', 'g', 'r', 'g', 'g'},
                            {'g', 'b', 'b', 'b', 'b', 'r', 'g', 'r'},
                            {'g', 'g', 'g', 'b', 'b', 'r', 'g', 'r'},
                            {'g', 'g', 'g', 'b', 'b', 'b', 'b', 'r'},
                            {'g', 'g', 'g', 'g', 'g', 'b', 'g', 'g'},
                            {'g', 'g', 'g', 'g', 'g', 'b', 'b', 'g'}};
  Point cur(4, 4);
  char new_color = 'y';

  flood_fill(image, cur, new_color);

  for (int r = 0; r < ROWS; r++) {
    for (int c = 0; c < COLS; c++) {
      cout << image[r][c] << ' ';
    }
    cout << endl;
  }
  // output:
  // g g g g g g g g
  // g g g g g g r r
  // g r r g g r g g
  // g y y y y r g r
  // g g g y y r g r
  // g g g y y y y r
  // g g g g g y g g
  // g g g g g y y g

  return 0;
}
```

① should be filled in by

A. `image[r][c] == prev_color`

B. `image[r][c] != prev_color`

C. `image[r][c] == new_color`

D. `image[r][c] != new_color`

② should be filled in by

A. `image[cur.r+1][cur.c] = new_color`

B. `image[cur.r][cur.c] = new_color`

C. `image[cur.r][cur.c+1] = new_color`

D. `image[cur.r][cur.c] = prev_color`

③ should be filled in by

A. `Point(pt.r, pt.c)`

B. `Point(pt.r, pt.c+1)`

C. `Point(pt.r+1, pt.c)`

D. `Point(pt.r+1, pt.c+1)`

④ should be filled in by

A. `prev_color = image[p.r][p.c]`

B. `new_color = image[p.r][p.c]`

C. `image[p.r][p.c] = prev_color`

D. `image[p.r][p.c] = new_color`

⑤ should be filled in by

A. `queue.push(p)`

B. `queue.push(pt)`

C. `queue.push(cur)`

D. `queue.push(Point(ROWS,COLS))`