SULEYMAN SULEYMANZADE

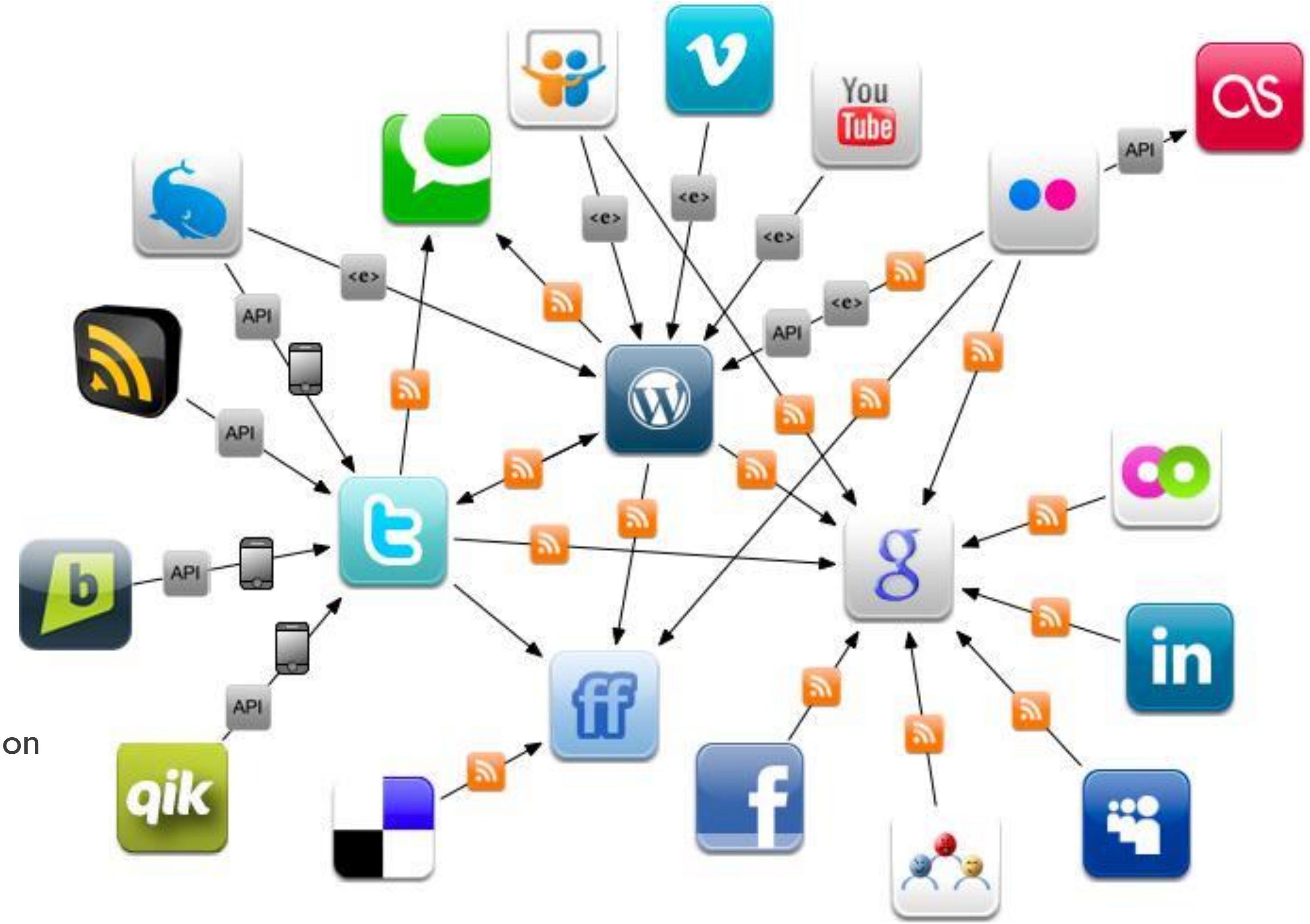
GRAPHS: DFS, BFS

DATA STRUCTURES AND ALGORITHMS

GRAPHS DATA STRUCTURE AND ALGORITHMS

Graphs: DFS, BFS

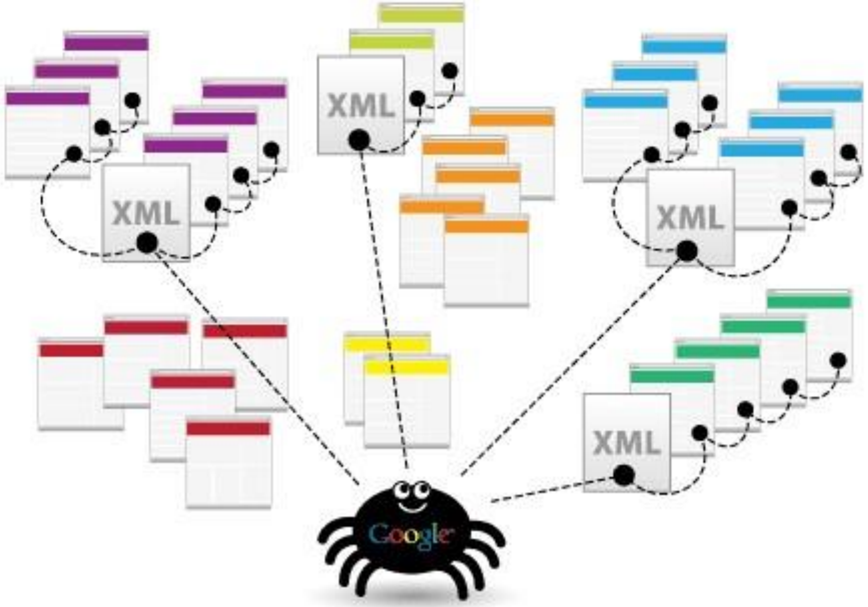
- Traversing
  - Types
  - Overview DFS BFS
  - Application
- DFS implementation
  - Adjacency Matrix version
  - Adjacency List version
- BFS implementation
  - Adjacency Matrix implementation
  - Adjacency List Implementation



DFS, BFS USAGE APPLICATIONS AND USAGE

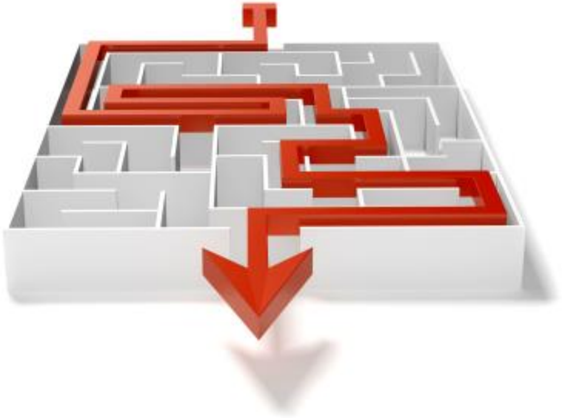
Applications

- Web crawlers
  - Google web crawler
- Path finding algorithms
- Search system
- Network Flows
- Connection and transitive closures



Facebook friend list relations

Bing



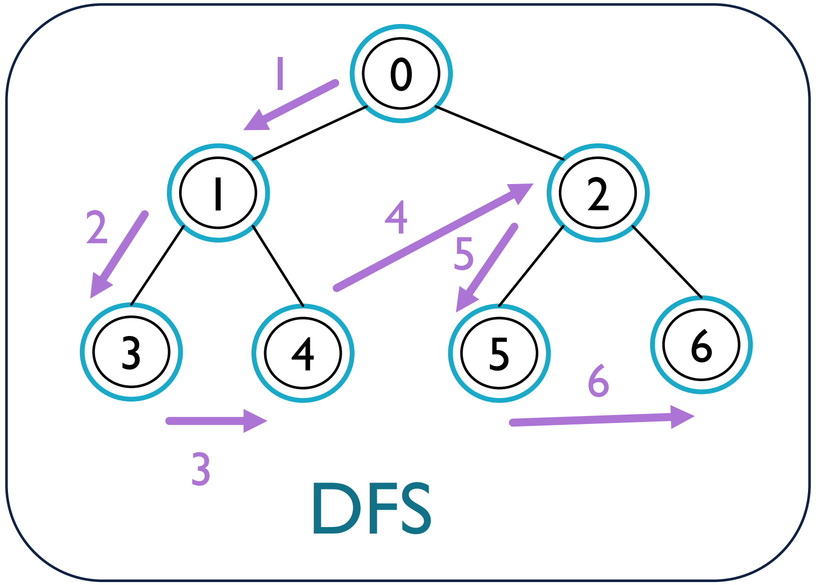
DFS AND BFS

DFS

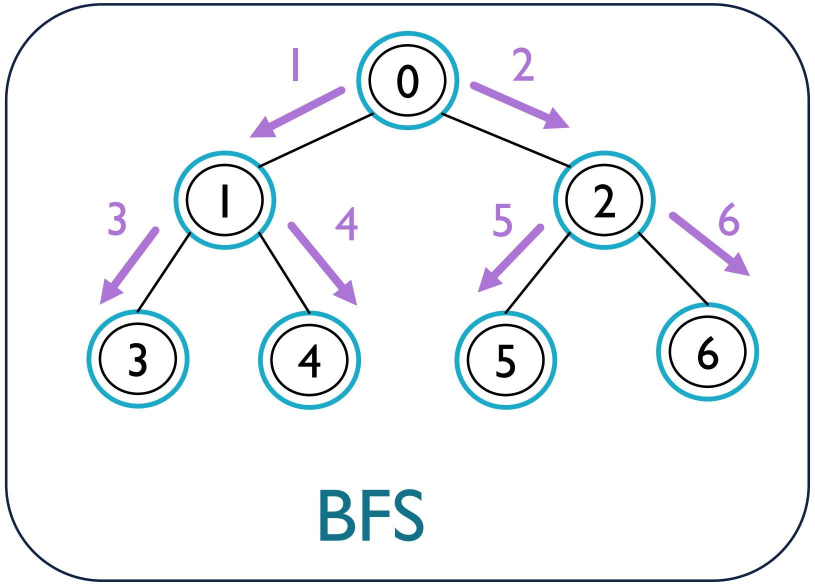
- Depth First Search
- Uses Stack structure

BFS

- Breath First Search
- Uses Queue structure



Starting from a distinguished source vertex, DFS will traverse the graph 'depth-first'. Every time DFS hits a branching point (a vertex with more than one neighbors), DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper



Starting from a distinguished source vertex, BFS will traverse the graph 'breadth-first'. That is, BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

DFS BFS BASED ALGORITHMS

The shortest path (in unweighted graph) finding

Connected Component

Cycles detection

Network flows

Max Flow Min-cut problems

For-Fulkerson, Edmond Karp

Dinic algorithms

DFS BFS Based Algorithms

Topological sorting

All paths finding

LCA problem

Strongest connected component

Kosaraju algorithm

Dijkstra algorithm

Bridges finding

Max Biparted graphs

Articulation points finding

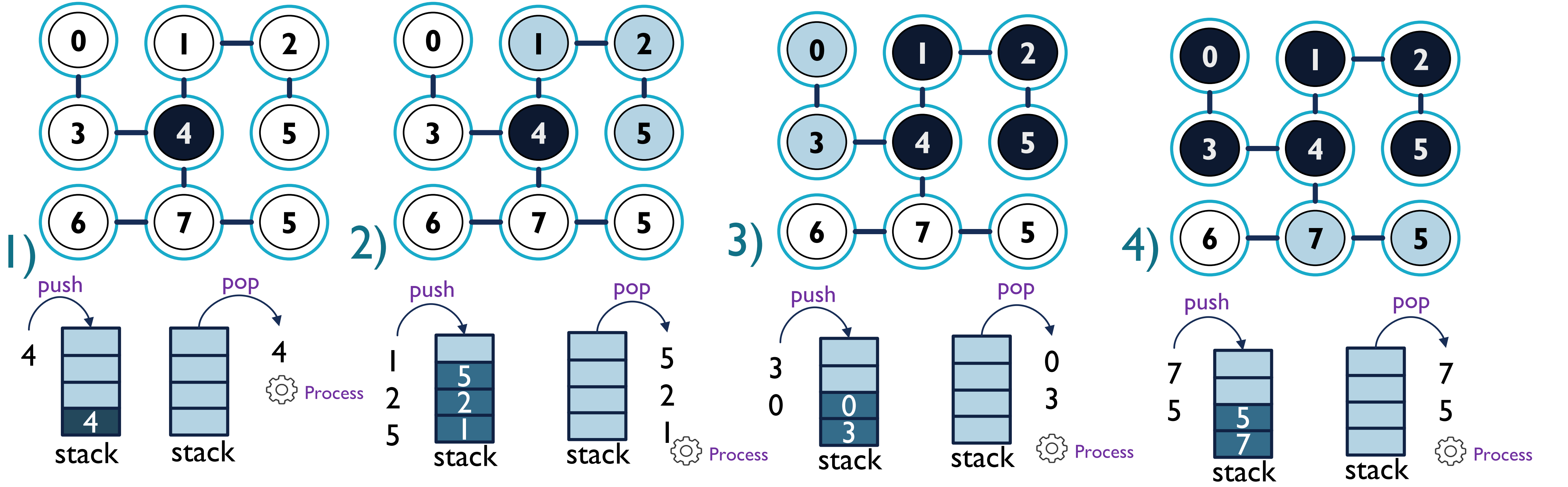
Biparted graphs finding

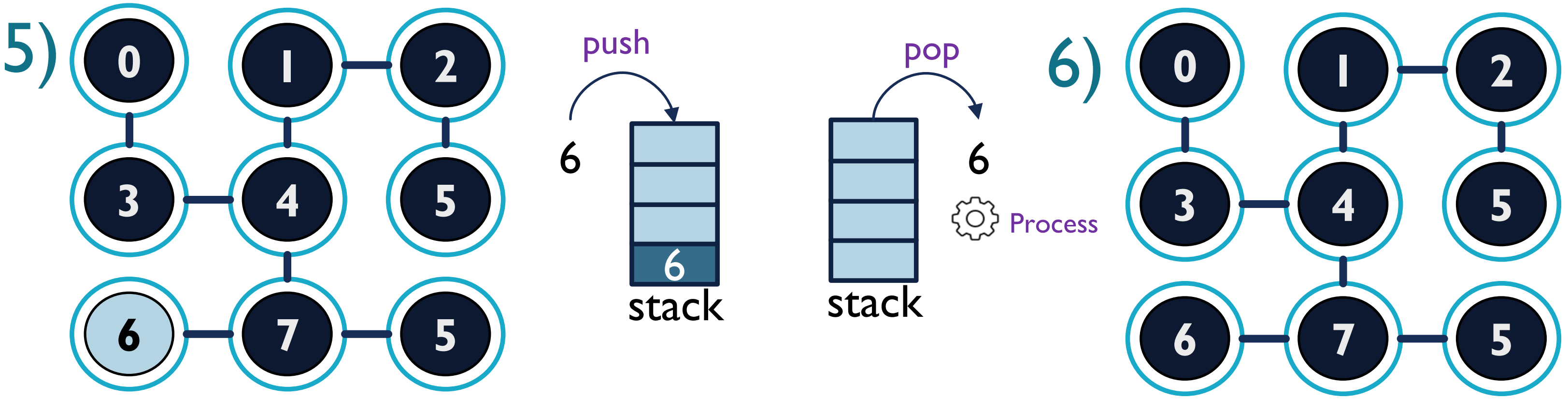
Flood-fill algorithm

DFS

ALGORITHM, IMPLEMENTATIONS, USAGE

DFS: ALGORITHM





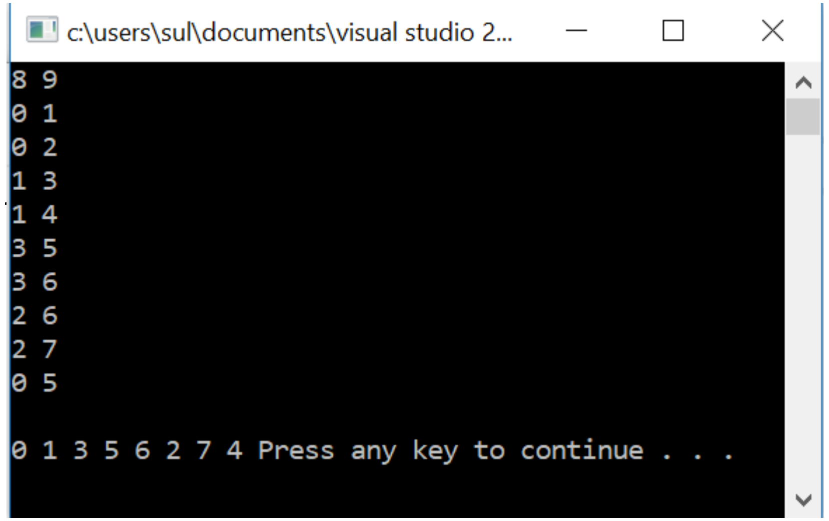
DFS

An order of data processing scheduled by stack data structure (or stack in memory). Traversing starts from one point, then all adjust edges pushed to the stack. Item processing is done while pulling from the stack

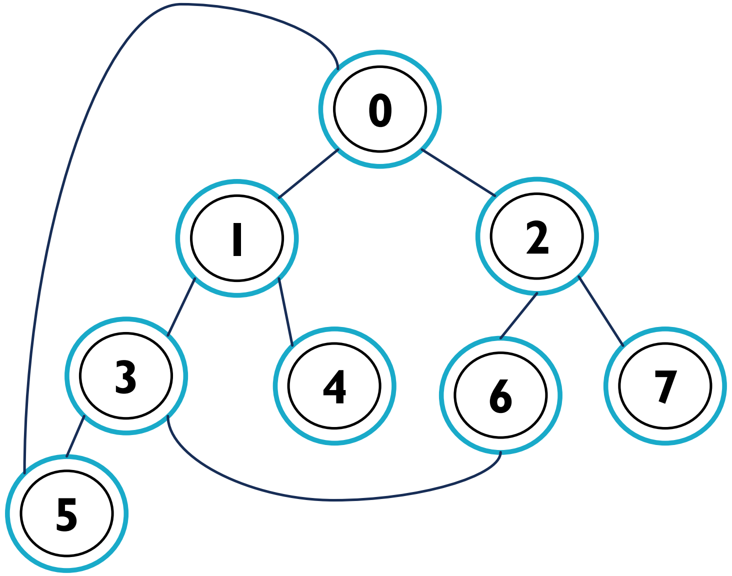
DFS: ADJACENCY MATRIX IMPLEMENTATION

```
#include<iostream>
using namespace std;

int adj[128][128];
bool visited[128];
int n,m;

void dfs(int u){

  visited[u] = true;
  cout<<u<<" ";

  for (int v = 0; v < n; v++){

    if( !visited[v] && adj[u][v])
          dfs(v);
  }
}
```

```
int main(){

  memset(adj, 0, sizeof(adj));
  memset(visited, false, sizeof(visited))
  cin>>n>>m;
  int from, to;

  for (int i = 0; i < m; i++){
    cin>>from>>to;
    adj[from][to] = 1;
    adj[to][from] = 1;
  }

  cout<<endl;
  dfs(0);

  system("pause");
  return 0;

}
```



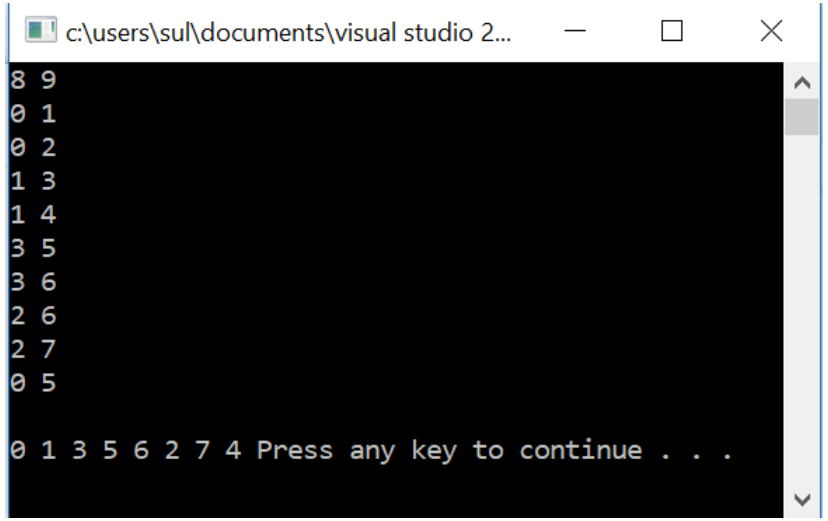
```
8 9
0 1
0 2
1 3
1 4
3 5
3 6
2 6
2 7
0 5

0 1 3 5 6 2 7 4 Press any key to continue . . .
```



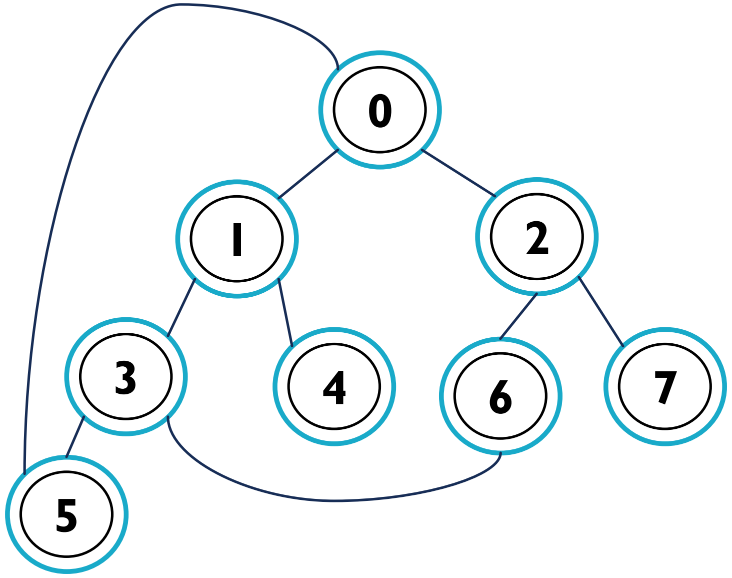
DFS: ADJACENCY LIST IMPLEMENTATION

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

typedef vector<int> vi;
vector<vi> adj;
vector<bool> visited;

void dfs(int u){

  visited[u] = true;
  cout<<u<<" ";
  for (int i = 0; i < adj[u].size(); i++){
    int v = adj[u][i];
    if(!visited[v])
        dfs(v);
  }
}
```

```
int main(){

  int n,m;
  cin>>n>>m;
  visited.assign(n, false());
  adj.assign(n, vi());
  int from, to;
  for (int i = 0; i < m; i++){
    cin>>from>>to;
    adj[from].push_back(to);
    adj[to].push_back(from);
  }
  cout<<endl;
  dfs(0);
  system("pause");
  return 0;
}
```



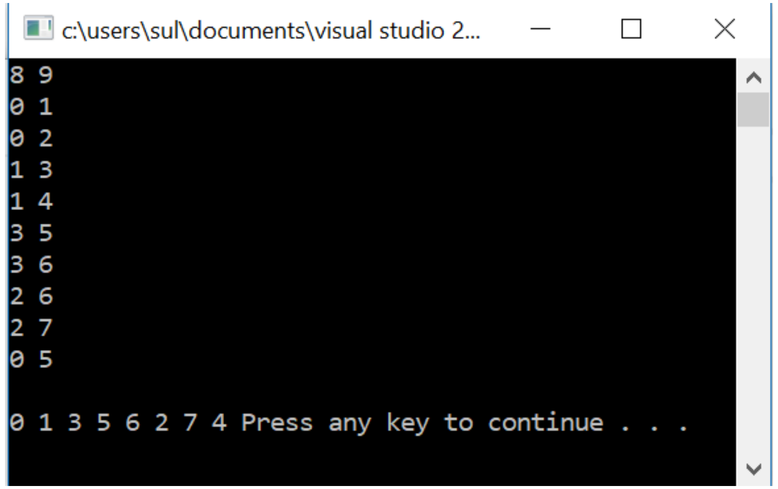
```
8 9
0 1
0 2
1 3
1 4
3 5
3 6
2 6
2 7
0 5

0 1 3 5 6 2 7 4 Press any key to continue . . .
```



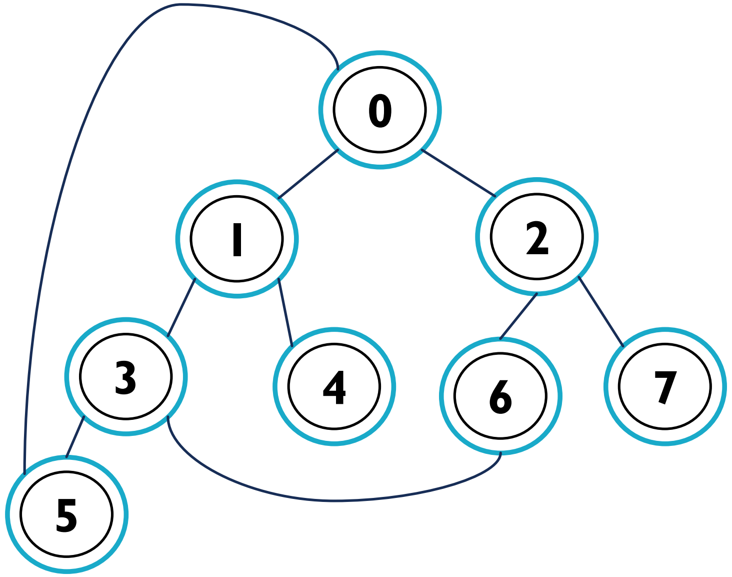
DFS: ADJACENCY LIST WITH EXPLICIT STACK

```
#include<iostream>
#include<stack>
#include<vector>
using namespace std;
typedef vector<int> vi;
vector<vi> adj;
vector<bool> visited;

void dfs(int s){
     visited[s] = true;
     stack<int> st;
     st.push(s);
     while (!st.empty()){
          int u = st.top(); st.pop();
          cout<<u<<" ";
          for (int i = 0; i < adj[u].size(); i++){
               int v = adj[u][i];
               if(!visited[v]){
                    visited[v] = true;
                    st.push(v);
               }
          }
     }
}
```

```
int main(){
  int n,m;
  cin>>n>>m;
  visited.assign(n, false());
  adj.assign(n, vi());
  int from, to;
  for (int i = 0; i < m; i++){
    cin>>from>>to;
    adj[from].push_back(to);
    adj[to].push_back(from);
  }
  cout<<endl;
  dfs(0);
  system("pause");
  return 0;
}
```



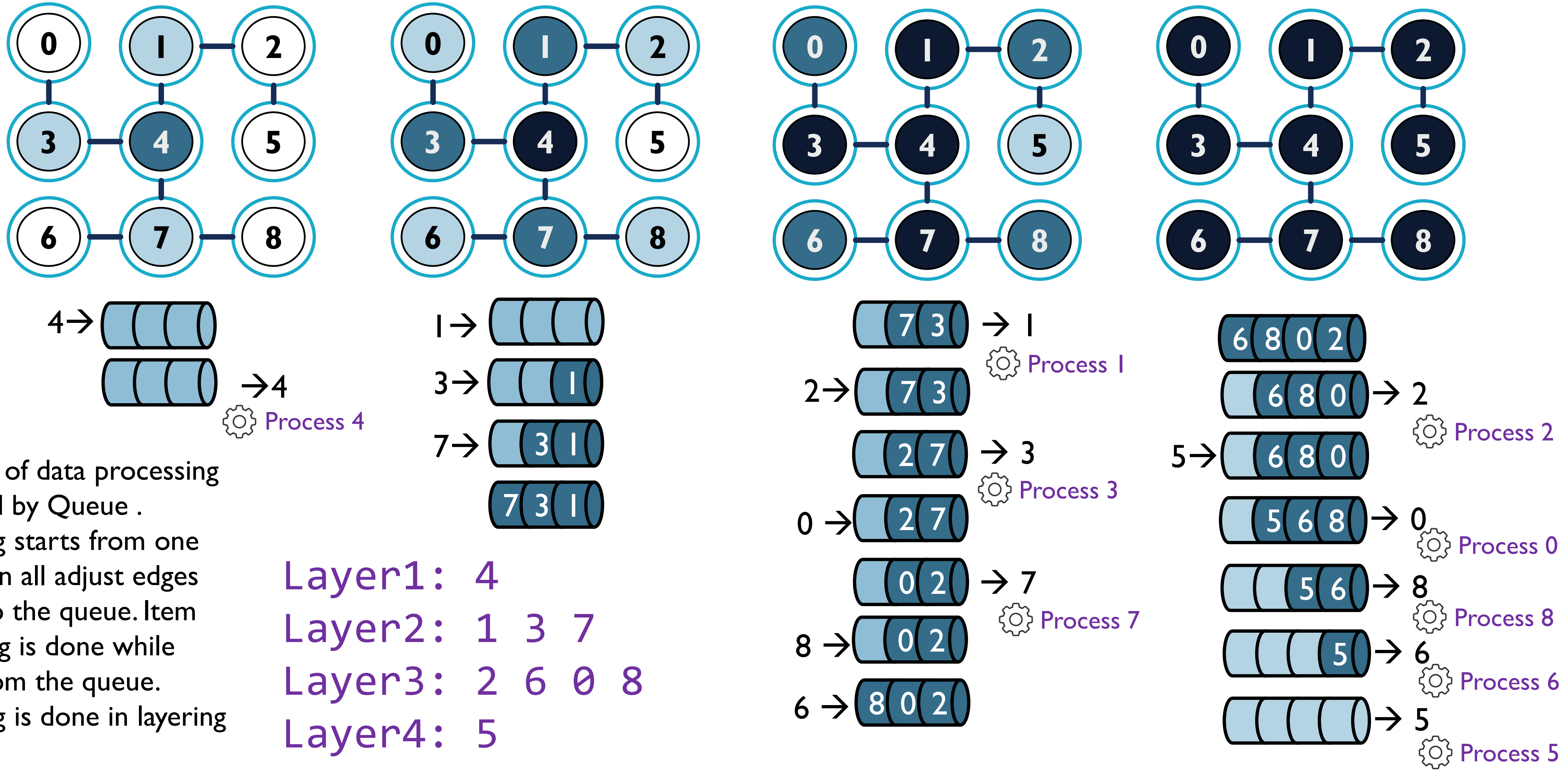
```
8 9
0 1
0 2
1 3
1 4
3 5
3 6
2 6
2 7
0 5

0 1 3 5 6 2 7 4 Press any key to continue . . .
```



BFS

ALGORITHM, IMPLEMENTATIONS, USAGE

BFS: ALGORITHM



BFS

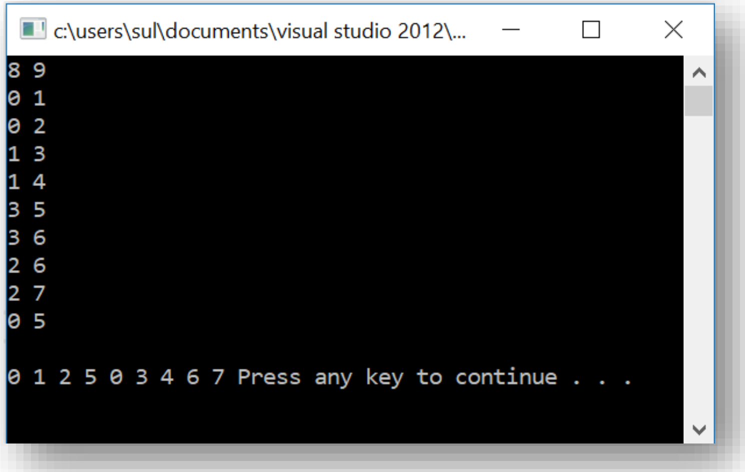
An order of data processing scheduled by Queue . Traversing starts from one point, then all adjust edges pushed to the queue. Item processing is done while pulling from the queue. Traversing is done in layering form

Layer1: 4
Layer2: 1 3 7
Layer3: 2 6 0 8
Layer4: 5

BFS :ADJUST MATRIX IMPLEMENTATION

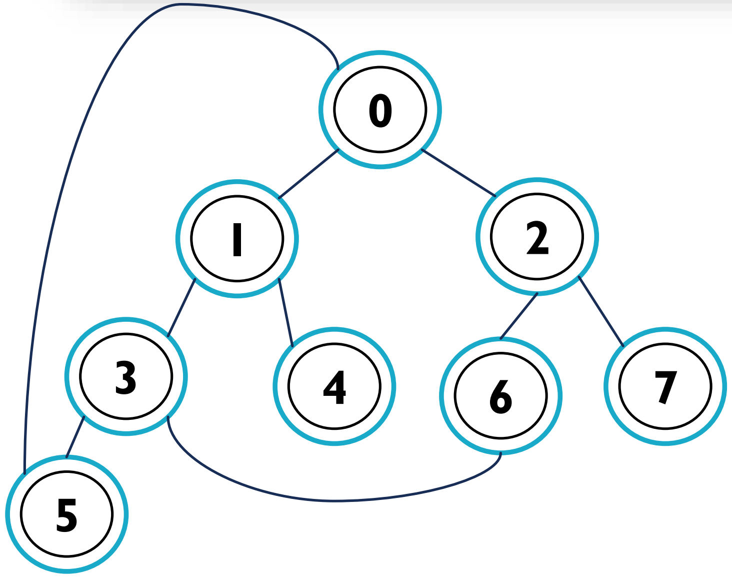
```
#include<iostream>
#include<queue>
using namespace std;
int adj[128][128];
int n,m;
bool visited[128];

void bfs(int i){
   visited[i] = true;
   queue<int> q;
   q.push(i);
   while (!q.empty()){
      int u = q.front(); q.pop();
      cout<<u<<" ";
      for (int v = 0; v < n; v++){
          if( !visited[v] && adj[u][v]){
                visited[v] = true;
               q.push(v);
          }
      }
   }
}
```

```
int main(){

  memset(adj, 0, sizeof(adj));
  memset(visited, false, sizeof(visited));
  cin>>n>>m;
  int from, to;
  for (int i = 0; i < m; i++){
    cin>>from>>to;
    adj[from][to] = 1;
    adj[to][from] = 1;
  }
  cout<<endl;
  bfs(0);
  system("pause");
  return 0;
}
```



c:\users\sul\documents\visual studio 2012\...

```
8 9
0 1
0 2
1 3
1 4
3 5
3 6
2 6
2 7
0 5

0 1 2 5 0 3 4 6 7 Press any key to continue . . .
```



BFS ADJACENCY LIST IMPLEMENTATION

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

typedef vector<int> vi;
vector<vi> adj;
vector<bool> visited;
vector<int> par;
vector<int> dist;

void print_path(int to){
  if(!visited[to])
     cout<<"there is no path to: "<<to<<endl;
  else{
     vector<int> path;
    for (int i = to; i != -1; i = par[i]){
      path.push_back(i);
    }
    reverse(path.begin(), path.end());
    cout<<"path: ";
    for (int i = 0; i < path.size(); i++)
          cout<<path[i]<<" ";

  }
}
```

```
void bfs(int s){
  dist[s] = 0;
  par[s] = -1;
  visited[s] = true;
  queue<int> q;
  q.push(s);
  while (!q.empty()){
    int u = q.front(); q.pop();
    for (int i = 0; i < adj[u].size(); i++){
      int v = adj[u][i];
      if(!visited[v]){
         dist[v] = dist[u] + 1;
         par[v] = u;
         visited[v] = true;
         q.push(v);
      }
    }
  }
}
```

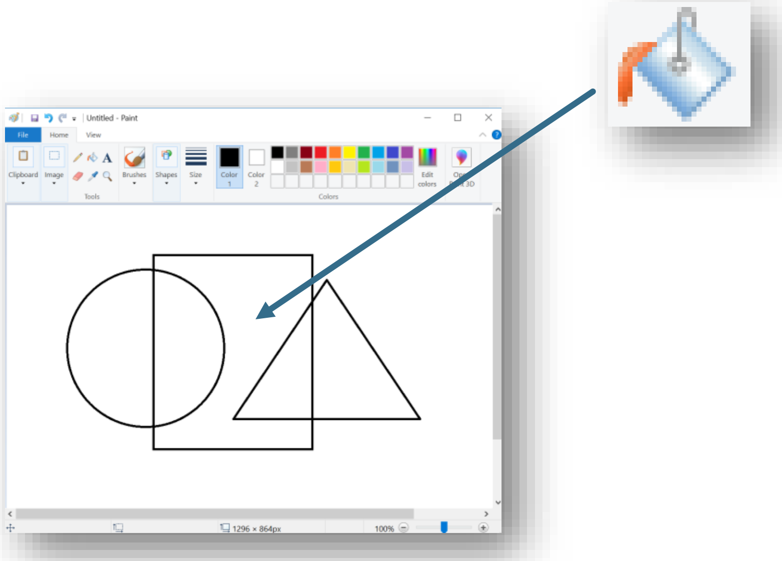
```
int main(){
  int n,m;
  cin>>n>>m;
  visited.assign(n, false());
  par.assign(n, int());
  dist.assign(n, int());
  adj.assign(n, vi());
  int from, to;
  for (int i = 0; i < m; i++){
    cin>>from>>to;
    adj[from].push_back(to);
    adj[to].push_back(from);
  }
  cout<<endl;
  bfs(0);
  print_path(7);
  system("pause");
  return 0;
}
```

FLOOD FILL ALGORITHM

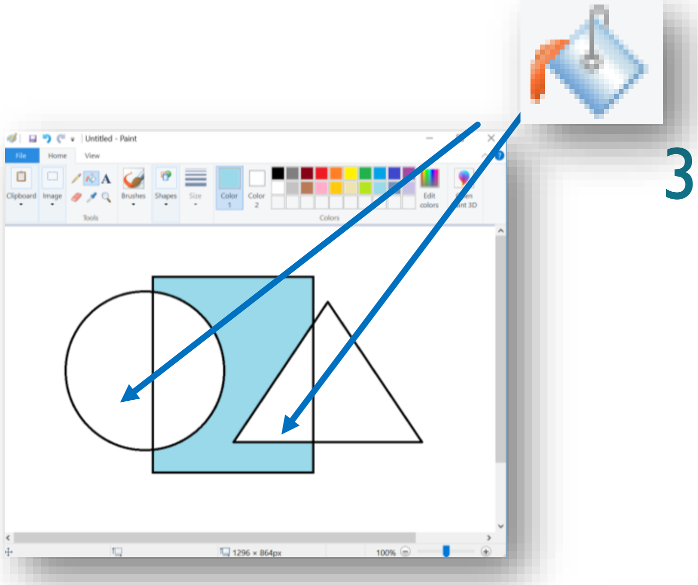
BASED ON DFS

FLOOD FILL ALGORITHM IN APPLICATION

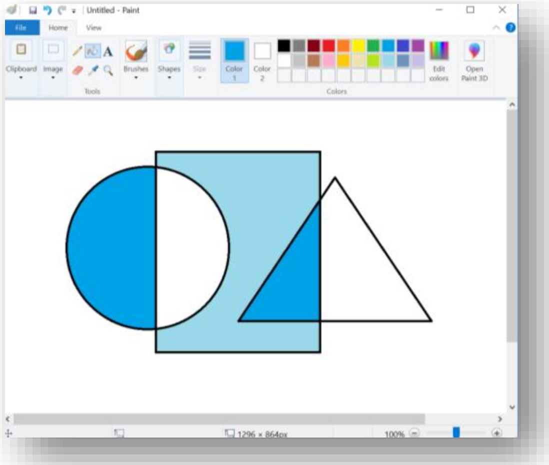
1)



2)



3)

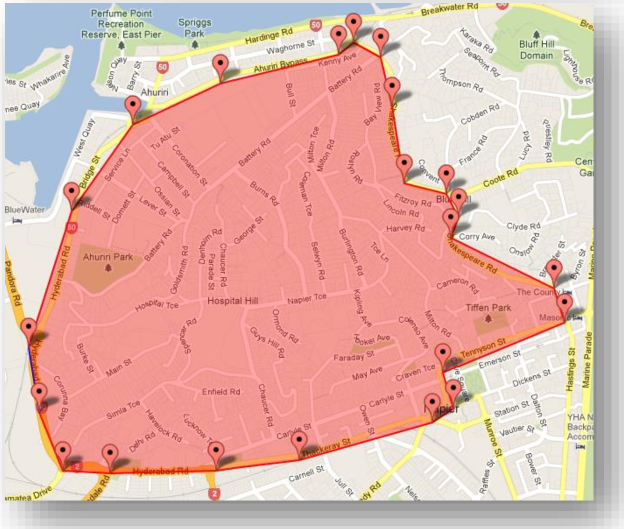


Fill tool

The “Fill” tool in drawing applications such as paint Adobe illustrator use Flood Fill algorithm

Google Maps

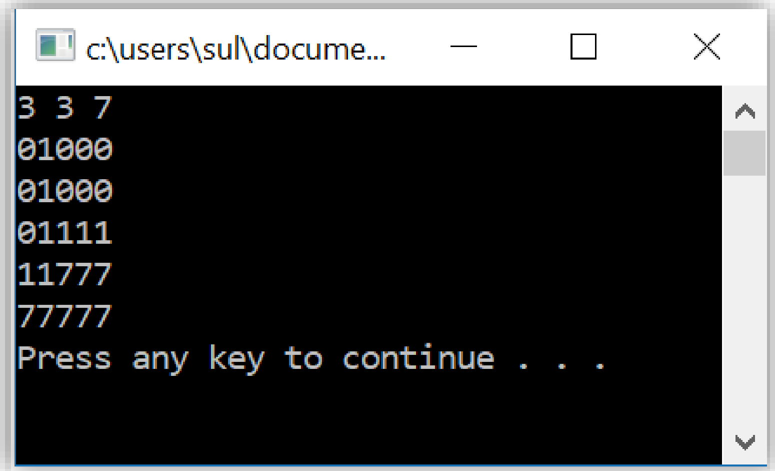
In Google Maps the “Flood Fill” algorithm uses to calculate the area of selected map



FLOOD FILL ALGORITHM'S IMPLEMENTATION

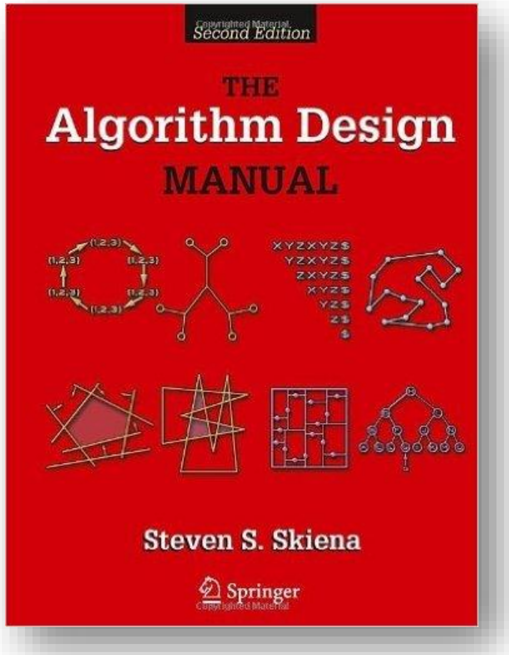
```cpp
#include<iostream>
using namespace std;

int arr[5][5] = {
    {0,1,0,0,0},
    {0,1,0,0,0},
    {0,1,1,1,1},
    {1,1,0,0,0},
    {0,0,0,0,0},
};

void flood_fill(int r, int c, int change){
    if(arr[r][c] == change || arr[r][c] == 1)
        return;
    if(r < 0 || c < 0 || r > 4 || c > 4)
        return;
    arr[r][c] = change;
    flood_fill(r+1, c, change);
    flood_fill(r-1, c, change);
    flood_fill(r, c+1, change);
    flood_fill(r, c-1, change);
}
```

```cpp
int main(){
    int r,c, change;
    cin>>r>>c>>change;
    flood_fill(r,c,change);
    for (int i = 0; i < 5; i++){
        for (int j = 0; j < 5; j++){
            cout<<arr[i][j];
        }
        cout<<endl;
    }
    system("pause");
    return 0;
}
```

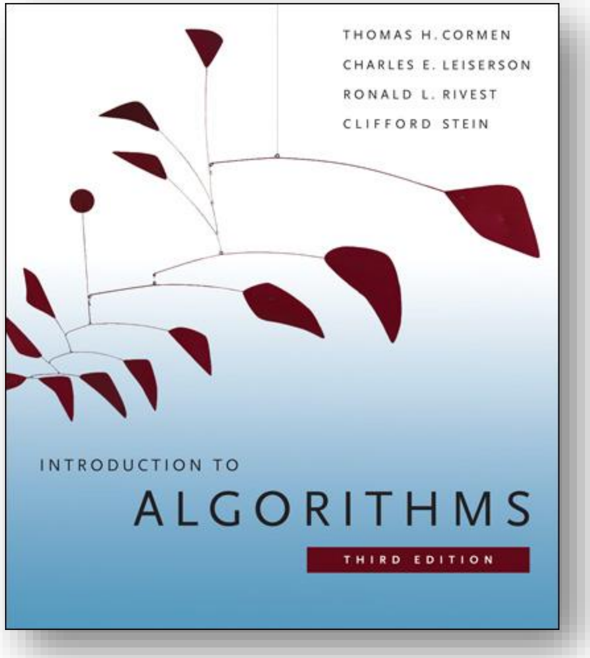


```
3 3 7
01000
01000
01111
11777
77777
Press any key to continue . . .
```

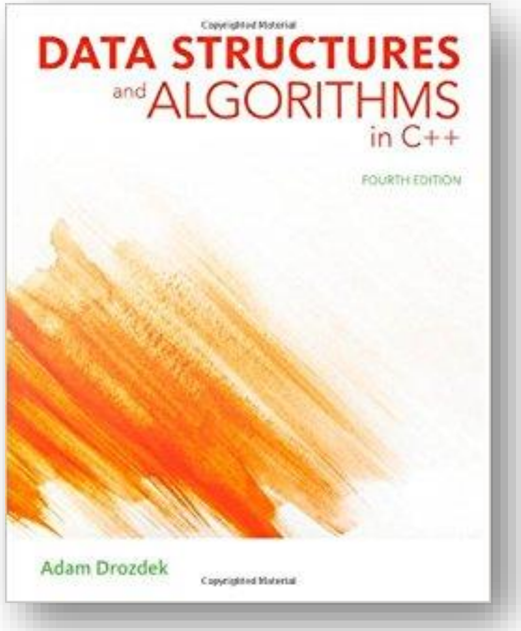
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 5: Graph Traversal
Page 145



Thomas H. Cormen
Introduction to Algorithms
Chapter VI Graph Algorithms
Page 587.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 8: Graphs
Page 391