# Data structures and algorithms

## Tutorial 8

Amr Keleg

Faculty of Engineering, Ain Shams University

April 17, 2019

Contact: amr_mohamed@live.com

# Outline

# Outline

```
bool dfs(int node, int des, const vector<vector<int> > & adj_matrix,
  vector<bool> & visited){

  if (node ==des)
    return true;

  visited[node] = true;

  for (int next_node=0; next_node<adj_matrix[node].size(); next_node++){

    if (adj_matrix[node][next_node] == 0 || visited[next_node])
      continue;

    if (dfs(next_node, des, adj_matrix, visited))
      return true;
  }

  return false;
}
```

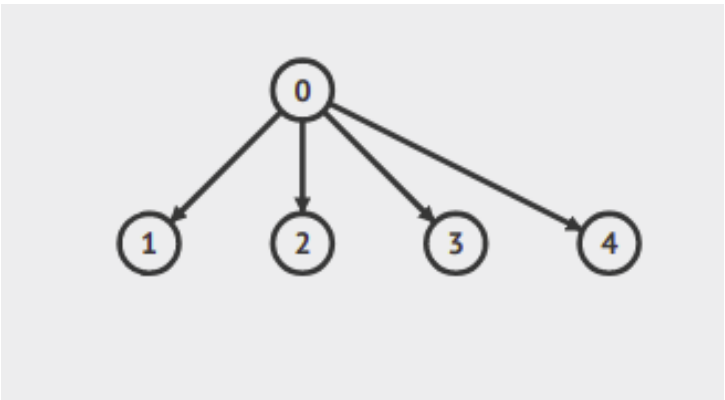
# Outline

```
bool dfs(int node, int des, const vector<vector<int> > & adj_matrix,
  vector<bool> & visited){

  stack<int> nodes_stack;
  nodes_stack.push(node);

  while(!nodes_stack.empty()){

    int cur_node = nodes_stack.top();
    nodes_stack.pop();
    visited[cur_node] = 1;

    if (cur_node==des)
      return true;

    for (int next_node=0; next_node<adj_matrix[cur_node].size(); next_node++){

      if (adj_matrix[cur_node][next_node] == 0 || visited[next_node])
        continue;

      nodes_stack.push(next_node);
    }
  }
  return false;
}
```

What is the complexity of the DFS on adjacency matrix?
The graph has N nodes and E edges.
TODO: A TREE WITH ROOT AND SINGLE LAYER Diagram

What is the complexity of the DFS on adjacency matrix?
The graph has N nodes and E edges.

$$O(N^2) \tag{1}$$




For a simple graph like this one, DFS using adjacency matrix will make 25(5*5) iterations/checks.

# Outline

# Outline

- Adjacency matrix isn't suitable for sparse graphs.
- Adjacency matrix can only represent a single edge between any two nodes.
- Instead of using an adjacency matrix, We will use an adjacency list. For each node, store only information about its adjacent nodes.

- For each node, store only information about its adjacent nodes.

How to read a description of the graph and load it into an adjacency matrix?

```
// An undirected unweighted graph

3 3
0 1
0 2
1 2
```

# Outline

```
// For an Undirected Unweighted graph
int n, m;
cin>>n>>m;
// Initialize the rows as empty vectors.
vector<vector<int> > adj_list(n, vector<int> ());
int a, b;
for (int i=0; i<m ;i++){
   cin>>a>>b;
   adj_list[a].push_back(b);
   adj_list[b].push_back(a);
}
```

```
// For an Directed Unweighted graph
int n, m;
cin>>n>>m;
vector<vector<int> > adj_list(n, vector<int> ());
int a, b;
for (int i=0; i<m ;i++){
   cin>>a>>b;
   adj_list[a].push_back(b);
}
```

```
// For an Undirected Weighted graph
int n, m;
cin>>n>>m;
vector<vector<pair<int,int>>> adj_list(n, vector<pair<int,int>> ());
int a, b, c;
for (int i=0; i<m ;i++){
  cin>>a>>b>>c;
  adj_list[a].push_back({b, c});
  adj_list[b].push_back({a, c});
}
```

```
// For an Directed Weighted graph
int n, m;
cin>>n>>m;
vector<vector<pair<int,int>>> adj_list(n, vector<pair<int,int>>());
int a, b, c;
for (int i=0; i<m ;i++){
  cin>>a>>b>>c;
  adj_list[a].push_back({b, c});
}
```

# Outline

The complexity of adding a node at the end of STL's list is O(1):
http://www.cplusplus.com/reference/list/list/push_back/

```
// For an Undirected Unweighted graph
int n, m;
cin>>n>>m;
// Initialize the rows as empty vectors.
vector<list<int> > adj_list(n, list<int>());
int a, b;
for (int i=0; i<m ; i++){
   cin>>a>>b;
   // Push node b to the end of
   // the linked list for node a
   adj_list[a].push_back(b);
   adj_list[b].push_back(a);
}
```

```
// For an Directed Unweighted graph
int n, m;
cin>>n>>m;
// Initialize the rows as empty vectors.
vector<list<int> > adj_list(n, list<int>());
int a, b;
for (int i=0; i<m ;i++){
   cin>>a>>b;
   // Push node b to the end of
   // the linked list for node a
   adj_list[a].push_back(b);
}
```

```
// For an Undirected Weighted graph
int n, m;
cin>>n>>m;
// Initialize the rows as empty vectors.
vector<list<pair<int,int>>> adj_list(n, list<pair<int,int> >());
int a, b, c;
for (int i=0; i<m ;i++){
  cin>>a>>b>>c;
  // Push node b to the end of
  // the linked list for node a
  adj_list[a].push_back({b, c});
  adj_list[b].push_back({a, c});
}
```

```
// For an Directed Weighted graph
int n, m;
cin>>n>>m;
// Initialize the rows as empty vectors.
vector<list<pair<int,int>>> adj_list(n, list<pair<int,int> >());
int a, b, c;
for (int i=0; i<m ;i++){
  cin>>a>>b>>c;
  // Push node b to the end of
  // the linked list for node a
  adj_list[a].push_back({b, c});
}
```

# Outline

```
// Using an array of arrays for adjacency list.

bool dfs(int node, int des, const vector<vector<int> > & adj_list,
  vector<bool> & visited){

  if (node ==des) return true;

  visited[node] = true;

  for (int next_node_index=0; next_node_index<adj_list[node].size(); next_node_index++){

    int next_node = adj_list[node][next_node_index];

    if (visited[next_node])
      continue;

    if (dfs(next_node, des, adj_list, visited))
      return true;
  }

  return false;
}
```

```
// Using an array of lists for adjacency list.

bool dfs(int node, int des, const vector<list<int>> & adj_list,
  vector<bool> & visited){

  if (node ==des) return true;

  visited[node] = true;

  for (list<int>::iterator it = adj_list[node].begin(); it!=adj_list[node].end(); it++){

    int next_node = *it;

    if (visited[next_node])
      continue;

    if (dfs(next_node, des, adj_list, visited))
      return true;
  }

  return false;
}
```

# Outline

BFS is a traversing algorithm where you should:

- start traversing from a selected node (source or starting node)
- traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node).
- You must then move towards the next-level neighbour nodes.

# Outline

```
bool bfs(int node, int des, const vector<vector<int>> & adj_mat,
  vector<bool> & visited){

  queue<int> nodes_q;
  nodes_q.push(node);

  while(!nodes_q.empty()){
    node = nodes_q.front();
    nodes_q.pop();
    visites[node] = true;
    if (node==des)
      return true;

    for (int next_node = 0; next_node<adj_mat[node].size(); next_node++){

      if (visited[next_node] || adj_mat[node][next_node] == 0)
        continue;

      nodes_q.push(next_node);
    }
  }

  return false;
}
```

# Outline

```
// Using an array of lists for adjacency list.

bool bfs(int node, int des, const vector<list<int>> & adj_list,
  vector<bool> & visited){

  queue<int> nodes_q;
  nodes_q.push(node);
  visited[node] = true;
  while(!nodes_q.empty()){
    node = nodes_q.front();
    nodes_q.pop();
    visited[node] = true;
    if (node==des)
      return true;

    for (list<int>::iterator it = adj_list[node].begin(); it!=adj_list[node].end(); it+-

      int next_node = *it;

      if (visited[next_node])
        continue;

      nodes_q.push(next_node);
    }
  }

  return false;
}
```

# Outline

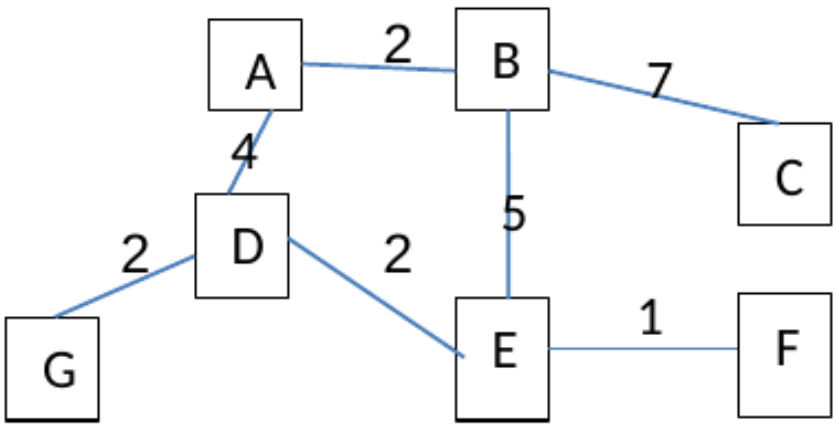
What does the first path found by BFS represent?

What does the first path found by BFS represent?
In case of unweighted graphs, the first path found by BFS is actually the shortest one.

# Outline

# Outline

Q3. Apply both depth first and breadth first algorithms starting from vertex D



DFS: D - A - B - C - E - F - G
BFS: D - A - E - G - B - F - C

# Outline

Q2. Create an algorithm to determine whether a undirected graph is connected or not (i.e. any node/vertex can be reached from any other node/vertex)

- Start from any node.
- Apply DFS or BFS.
- Count the number of visited nodes.
- If all the nodes where visited then the graph is connected. Otherwise, the graph isn't connected.

```
void dfs(int src, vector<bool> & vis, ....);

int main(){
  int n,m;
  .....
  vector<bool> visited(n, false);
  dfs(0, visited, ...);
  int vis_count = 0;
  for(int i=0;i<n;i++)
    vis_count += visited[i];

  if (vis_count==n)
    cout<<"Connected";
  else
    cout<<"Not connected";
}
```

# Outline

Q1. A degree of a vertex is the number of other vertices connected to it. Show that the sum of all of the graph vertex degrees is always even.

- Assume that initially the graph has no edges (Summation of degree $= 0$ "even").
- For each new edge, the degree of two nodes will be increased by one.
- Thus the summation of the degree will increase by two (will still be even).

# Outline

# Outline

- Priority Queue is an extension of queue with following properties.
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- `http://www.cplusplus.com/reference/queue/priority_queue/`

# Outline

```
int shortest_distance(int src, int des,
  vector<vector<pair<int,int>>> adj_list){
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>q;

  int n = adj_list.length();
  vector<int> d(n, INF);
  vector<int> p(n, -1);

  d[src] = 0;
  nodes_q.push({0, {src, src}});
  while(!nodes_q.empty()){
    pair<int,int> p = nodes_q.top();
    nodes_q.pop();
    int cur_node = p.second.first;
    int cur_prev_node = p.second.second;
    int cur_dis = p.first;

    if (d[cur_node] != INF)
      continue;

    d[cur_node] = cur_dis;
    p[cur_node] = cur_prev_node;

    for (int i=0; i< adj_list[cur_node].size(); cur_node++){
      int next_node = adj_list[cur_node][i].first;
      int weight = adj_list[cur_node][i].second;
      if (d[next_node] !=INF)
        continue;
      nodes_q.push({cur_dis + weight, {next_node, cur_node}});
    }
  }

  return d[des];
}
```

# Outline

How can we print the path using the p vector?

```
void print_path(int src, int des, vector<int> p);
```

```
void print_path(int src, int des, vector<int> p){
  stack<int> path_nodes;
  int node = des;
  do{
    path_nodes.push(node);
    node = p[node];
  }while(p[node] != node);

  while(!path_nodes.empty()){
    cout<<path_nodes.top()<<endl;
    path_nodes.pop();
  }
}
```

Classical problems on graphs:

- BFS for shortest path: `https://www.hackerrank.com/challenges/bfsshortreach/problem`

Feedback form: `https://forms.gle/KMGs3wTFtLBsbo7N8`