Data structures and algorithms

Tutorial 8 - Part 1

Amr Keleg

Faculty of Engineering, Ain Shams University

April 17, 2020

Contact: amr_mohamed@live.com

Outline

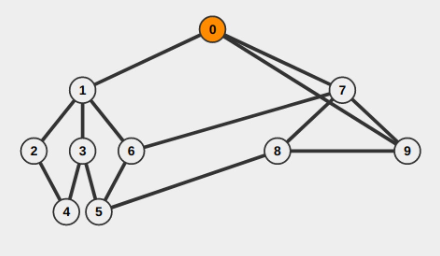
1 Adjacency List
- How to represent a graph using basic data-structures?
- First option - An array(vector) of linked list(lists)
- Second option - An array(vector) of arrays(vectors)
- Searching algorithms in graph - DFS - Depth First Search

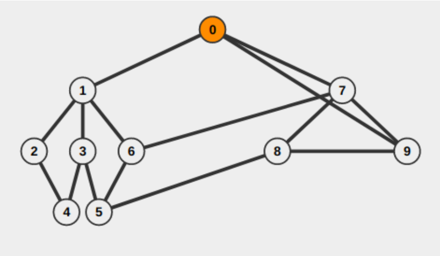
2 Breadth First Search (BFS)

3 Sheet 4 Questions

Outline

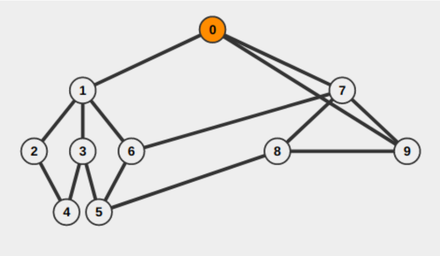
- Adjacency matrix isn't suitable for sparse graphs.
- Adjacency matrix can only represent a single edge between any two nodes.
- Instead of using an adjacency matrix, We will use an adjacency list. For each node, store **only** information about its adjacent nodes.


0
1
7
2
3
6
8
9
4
5



Adjacency Matrix

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



Adjacency Matrix

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

Adjacency List

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

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

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);
}
```

Outline

What values to we need to store for each edge?

What values to we need to store for each edge?

- Source: Implicitly known from the index of the linked list

What values to we need to store for each edge?

- Source: Implicitly known from the index of the linked list
- Destination

What values to we need to store for each edge?

- Source: Implicitly known from the index of the linked list
- Destination
- Weight

What values to we need to store for each edge?

- Source: Implicitly known from the index of the linked list
- Destination
- Weight

First option: OOP approach (used in the slides)

First option: OOP approach (used in the slides)

```
class Connection{
  public:
    int destination;
    int weight;
};
```

First option: OOP approach (used in the slides)

```
class Connection{
  public:
    int destination;
    int weight;
};

int main(){
  int n; // No. of Nodes
  cin>>n;
```

First option: OOP approach (used in the slides)

```
class Connection{
  public:
    int destination;
    int weight;
};

int main(){
  int n; // No. of Nodes
  cin>>n;

  vector<list<Connection> > adj_list(n);
}
```

Second option: Using STL class (Pair)[1]

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
  cout<<p2.first<<" "<<p2.second;
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
  cout<<p2.first<<" "<<p2.second;

  pair<int, int> p3;
  p3 = {10, 20};
  cout<<p3.first<<" "<<p3.second;
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
  cout<<p2.first<<" "<<p2.second;

  pair<int, int> p3;
  p3 = {10, 20};
  cout<<p3.first<<" "<<p3.second;

  // Building the graph using pair
  int n; // No. of Nodes
  cin>>n;
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
  cout<<p2.first<<" "<<p2.second;

  pair<int, int> p3;
  p3 = {10, 20};
  cout<<p3.first<<" "<<p3.second;

  // Building the graph using pair
  int n; // No. of Nodes
  cin>>n;

  //Change this: vector<list<Connection> > adj_list(n);
```

[1] http://www.cplusplus.com/reference/utility/pair/

Second option: Using STL class (Pair)[1]

```
#include<utility>

int main(){
  pair<int, int> p1(10, 1);
  cout<<p1.first<<" "<<p1.second;

  pair<string, int> p2("Apple", 30);
  cout<<p2.first<<" "<<p2.second;

  pair<int, int> p3;
  p3 = {10, 20};
  cout<<p3.first<<" "<<p3.second;

  // Building the graph using pair
  int n; // No. of Nodes
  cin>>n;

  //Change this: vector<list<Connection> > adj_list(n);
  vector<list<pair<int,int> > > adj_list(n);
}
```

[1] http://www.cplusplus.com/reference/utility/pair/

Some **minor** advantages for using pair (STL classes):

Some **minor** advantages for using pair (STL classes):

- C++ knows how to sort them containers of those classes.

Some **minor** advantages for using pair (STL classes):

- C++ knows how to sort them containers of those classes.
- This is also beneficial in case of using data-structures like map, set, priority_queue that rely on comparing the items.

Some **minor** advantages for using pair (STL classes):

- C++ knows how to sort them containers of those classes.
- This is also beneficial in case of using data-structures like map, set, priority_queue that rely on comparing the items.

```
#include <vector> // vector
#include <utility> // pair
#include <algorithm> // sort
#include <iostream> // cout
using namespace std;

int main(){
  vector<pair<int, int> > v;
  v.push_back({2, 2});
  v.push_back({1, 1});
  sort(v.begin(), v.end());

  for(int i=0; i< v.size(); i++)
    cout<<v[i].first <<" "<<v[i].second<<endl;

  /* The output is:
  1 1
  2 2
  */

  return 0;
}
```

```
#include <vector> // vector
#include <algorithm> // sort
#include <iostream> // cout
using namespace std;

class CustomPair{
public:
  int first;
  int second;
  CustomPair(int f, int s):first(f), second(s){}
};

int main(){
  vector<CustomPair > v;
  v.push_back(CustomPair(2, 2));
  v.push_back(CustomPair(1, 1));
  sort(v.begin(), v.end());

  for(int i=0; i< v.size(); i++)
    cout<<v[i].first<<" "<<v[i].second<<endl;

  return 0;
}
```

OUTPUT?

```
#include <vector> // vector
#include <algorithm> // sort
#include <iostream> // cout
using namespace std;

class CustomPair{
public:
  int first;
  int second;
  CustomPair(int f, int s):first(f), second(s){}
};

int main(){
  vector<CustomPair > v;
  v.push_back(CustomPair(2, 2));
  v.push_back(CustomPair(1, 1));
  sort(v.begin(), v.end());

  for(int i=0; i< v.size(); i++)
    cout<<v[i].first<<" "<<v[i].second<<endl;

  return 0;
}
```

OUTPUT?

C++ does not know how to compare items of CustomPair.

```
amr:[~]: g++ custom_pair.cpp                                                                                    [133/136]
In file included from /usr/include/c++/7/bits/stl_algobase.h:71:0,
                 from /usr/include/c++/7/vector:60,
                 from custom_pair.cpp:1:
/usr/include/c++/7/bits/predefined_ops.h: In instantiation of 'constexpr bool __gnu_cxx::__ops::_Iter_less_iter::operator()(_Iterator$
, _Iterator2) const [with _Iterator1 = __gnu_cxx::__normal_iterator<CustomPair*, std::vector<CustomPair> >; _Iterator2 = __gnu_cxx::_$
normal_iterator<CustomPair*, std::vector<CustomPair> >]':
/usr/include/c++/7/bits/stl_algo.h:81:17:   required from 'void std::__move_median_to_first(_Iterator, _Iterator, _Iterator, _Iterato$
, _Compare) [with _Iterator = __gnu_cxx::__normal_iterator<CustomPair*, std::vector<CustomPair> >; _Compare = __gnu_cxx::__ops::_Iter$
less_iter]'
/usr/include/c++/7/bits/stl_algo.h:1921:34:   required from '_RandomAccessIterator std::__unguarded_partition_pivot(_RandomAccessIter$
tor, _RandomAccessIterator, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<CustomPair*, std::vector<CustomPair>
>; _Compare = __gnu_cxx::__ops::_Iter_less_iter]'
/usr/include/c++/7/bits/stl_algo.h:1953:38:   required from 'void std::__introsort_loop(_RandomAccessIterator, _RandomAccessIterator,
_Size, _Compare) [with _RandomAccessIterator = __gnu_cxx::__normal_iterator<CustomPair*, std::vector<CustomPair> >; _Size = long int;
_Compare = __gnu_cxx::__ops::_Iter_less_iter]'
/usr/include/c++/7/bits/stl_algo.h:1968:25:   required from 'void std::__sort(_RandomAccessIterator, _RandomAccessIterator, _Compare)
[with _RandomAccessIterator = __gnu_cxx::__normal_iterator<CustomPair*, std::vector<CustomPair> >; _Compare = __gnu_cxx::__ops::_Iter$
less_iter]'
/usr/include/c++/7/bits/stl_algo.h:4836:18:   required from 'void std::sort(_RAIter, _RAIter) [with _RAIter = __gnu_cxx::__normal_ite$
ator<CustomPair*, std::vector<CustomPair> >]'
custom_pair.cpp:17:25:   required from here
/usr/include/c++/7/bits/predefined_ops.h:43:23: error: no match for 'operator<' (operand types are 'CustomPair' and 'CustomPair')
       { return *__it1 < *__it2; }
                ~~~~~~~^~~~~~~~
In file included from /usr/include/c++/7/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/7/vector:60,
                 from custom_pair.cpp:1:
/usr/include/c++/7/bits/stl_iterator.h:891:5: note: candidate: template<class _IteratorL, class _IteratorR, class _Container> bool __$
nu_cxx::operator<(const __gnu_cxx::__normal_iterator<_IteratorL, _Container>&, const __gnu_cxx::__normal_iterator<_IteratorR, _Contai$
er>&)
     operator<(const __normal_iterator<_IteratorL, _Container>& __lhs,
     ^~~~~~~~
/usr/include/c++/7/bits/stl_iterator.h:891:5: note:   template argument deduction/substitution failed:
In file included from /usr/include/c++/7/bits/stl_algobase.h:71:0,
```

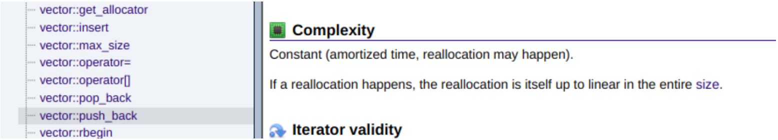
```
#include <vector> // vector
#include <algorithm> // sort
#include <iostream> // cout
using namespace std;

class CustomPair{
public:
  int first;
  int second;
  CustomPair(int f, int s):first(f), second(s){}
  bool operator <(const CustomPair & other_pair){
    return (this-> first < other_pair.first)
         (this-> first == other_pair.first &&
        this->second <=other_pair.second);
  }
};

int main(){
  vector<CustomPair > v;
  v.push_back(CustomPair(2, 2));
  v.push_back(CustomPair(1, 1));
  sort(v.begin(), v.end());

  for(int i=0; i< v.size(); i++)
    cout<<v[i].first<<" "<<v[i].second<<endl;

  return 0;
}
```

```
// 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

```
// 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);
}
```



vector::get_allocator
vector::insert
vector::max_size
vector::operator=
vector::operator[]
vector::pop_back
vector::push_back
vector::rbegin

Complexity

Constant (amortized time, reallocation may happen).

If a reallocation happens, the reallocation is itself up to linear in the entire size.

Iterator validity

http://www.cplusplus.com/reference/vector/vector/push_back/

```
// 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});
}
```

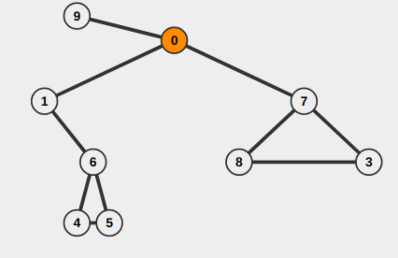
Outline

```
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;
}
```

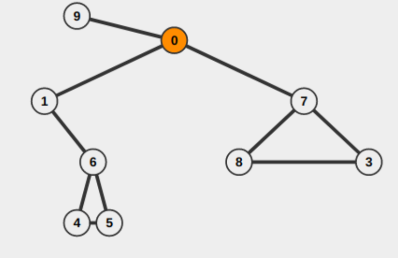
```
// Using an array of arrays for adjacency list.

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

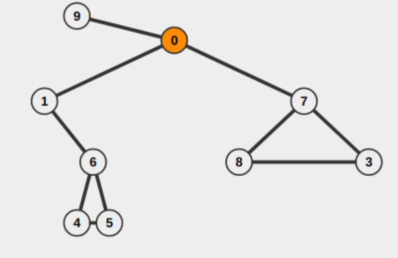


BFS is a traversing algorithm where you should:



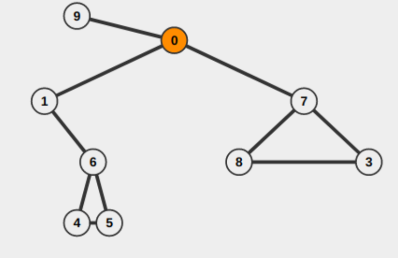
BFS is a traversing algorithm where you should:

- start traversing from a selected node (source or starting node).



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).



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).
- after exploring all the directly connected neighbour nodes, start exploring 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;

    // Loop over neighbouring nodes
    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;
}
```

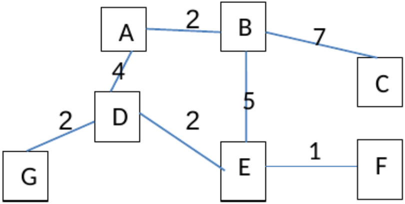
Outline

```
// 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;

    // Loop over neighbouring nodes
    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;
}
```

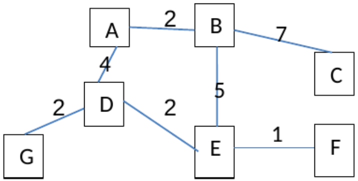
Outline

1. Adjacency List
   - How to represent a graph using basic data-structures?
   - First option - An array(vector) of linked list(lists)
     - Unweighted graphs
     - Weighted graphs
   - Second option - An array(vector) of arrays(vectors)
   - Searching algorithms in graph - DFS - Depth First Search
2. Breadth First Search (BFS)
   - Adjacency Matrix
   - Adjacency List
   - Sheet 4 - Question 3
   - One important application of BFS
3. Sheet 4 Questions
   - Question 2
   - Question 1

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



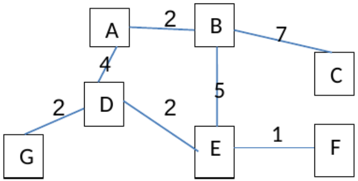
Note: The paths will depend on the way nodes are pushed to the queue/ stack and whether you mark the node on push/ on pop. State your assumptions clearly when you answer such questions unless the rules are stated in the question's statement.



Visited Array:

A	B	C	D	E	F	G

Queue:



Visited Array:

A	B	C	D	E	F	G

Queue:

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

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

1. Adjacency List
   - How to represent a graph using basic data-structures?
   - First option - An array(vector) of linked list(lists)
     - Unweighted graphs
     - Weighted graphs
   - Second option - An array(vector) of arrays(vectors)
   - Searching algorithms in graph - DFS - Depth First Search
2. Breadth First Search (BFS)
   - Adjacency Matrix
   - Adjacency List
   - Sheet 4 - Question 3
   - One important application of BFS
3. Sheet 4 Questions
   - Question 2
   - Question 1

Q2. Create an algorithm to determine whether an 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";
}
```

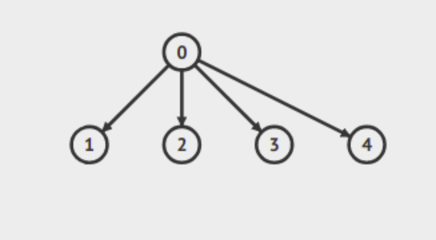
Comments:

- Connectivity in directed graphs is different.

[2] https://en.wikipedia.org/wiki/Strongly_connected_component

Comments:

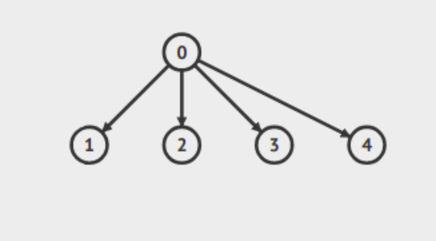
- Connectivity in directed graphs is different.
- Is this graph connected?



[2] https://en.wikipedia.org/wiki/Strongly_connected_component

Comments:

- Connectivity in directed graphs is different.
- Is this graph connected?



- We define another term (Strongly connected component)[2] and another harder algorithm is needed to check if the graph is strongly connected or not.

[2] https://en.wikipedia.org/wiki/Strongly_connected_component

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).

To be continued!