ACA Summer School

Data Structures and Algorithms
Lecture 3

Vijay Keswani

IIT Kanpur

June 22, 2016

Outline

Graph Definitions

Definition

A graph G is denoted by a set of vertices V and a set of edges E that connect the vertices.

Graph Definitions

Definition

A graph G is denoted by a set of vertices V and a set of edges E that connect the vertices.

- If vertex u is connected to v by an edge, then v is called a neighbour of u, and vice-versa.

Graph Definitions

Definition

A graph G is denoted by a set of vertices V and a set of edges E that connect the vertices.

- If vertex u is connected to v by an edge, then v is called a neighbour of u, and vice-versa.
- **Degree** of a vertex if the number of neigbours of the vertex.

Graph Definitions

Definition

A graph G is denoted by a set of vertices V and a set of edges E that connect the vertices.

- If vertex u is connected to v by an edge, then v is called a neighbour of u, and vice-versa.
- **Degree** of a vertex if the number of neigbours of the vertex.
- Maximum number of edges $|E| < |V|^2$.

Graph Terminologies

- **Path** : A sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.

Graph Terminologies

- **Path** : A sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.
- **Cycle** : A cycle is a path such that the start vertex and end vertex on the path are same.

Graph Terminologies

- **Path** : A sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.
- **Cycle** : A cycle is a path such that the start vertex and end vertex on the path are same.
- Two vertices u and v are connected if G contains a path from u to v. Otherwise, they are disconnected.

Graph Terminologies

- **Path** : A sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.
- **Cycle** : A cycle is a path such that the start vertex and end vertex on the path are same.
- Two vertices u and v are connected if G contains a path from u to v. Otherwise, they are disconnected.

Connected graph

A graph is called **connected** if every pair of vertices in the graph are connected.

Types of Graphs

- A **tree** is a connected graph without any cycles.

Types of Graphs

- A **tree** is a connected graph without any cycles.
- A **directed graph** is a graph with orientations in the edges.

Types of Graphs

- A **tree** is a connected graph without any cycles.
- A **directed graph** is a graph with orientations in the edges.
- An **undirected graph** is a graph without any orientations in the edges.

Types of Graphs

- A **tree** is a connected graph without any cycles.
- A **directed graph** is a graph with orientations in the edges.
- An **undirected graph** is a graph without any orientations in the edges.
- A **regular graph** is a graph in which every vertex has the same degree.

Uses of Graphs

Many problems of practical interest can be represented by graphs. For example

- If the vertices represent cities, and edges represent the roads between cities, then a graph can be used to model the road transport network of an area.

Uses of Graphs

Many problems of practical interest can be represented by graphs. For example

- If the vertices represent cities, and edges represent the roads between cities, then a graph can be used to model the road transport network of an area.
- If the vertices represent brain regions, and edges represent the series of neurons connecting the brain regions, then a graph can be used to model the brain neural network.

Uses of Graphs

Minimum distance between two cities: Suppose you have a road structure which connects various cities by road. You need to find the minimum distance road-path between two places.

This problem can be framed as a weighted graph. The places represents the vertices and edges are the connected roads between two places.
The distance between two cities is the weight given to the particular edge.

Outline

Graph Terminologies

Adjacency Matrices and Lists

Graph Traversal

Shortest Path Algorithms

Adjacency Matrix

Given a undirected graph G with n vertices and m edges, the adjacency matrix of G is an $n \times n$ matrix such that

$$\begin{aligned} A[i,j] &= 1, \text{ if } i \text{ and } j \text{ are connected,} \\ &= 0, \text{ otherwise} \end{aligned}$$

Adjacency Matrix

Given a undirected graph G with n vertices and m edges, the adjacency matrix of G is an $n \times n$ matrix such that

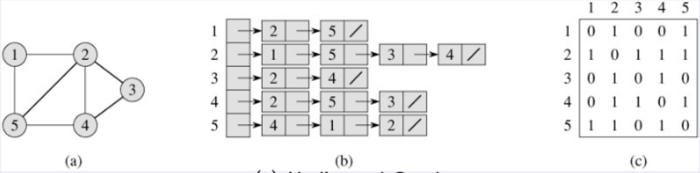
$$A[i,j] = 1, \text{ if } i \text{ and } j \text{ are connected,}$$

$$= 0, \text{ otherwise}$$

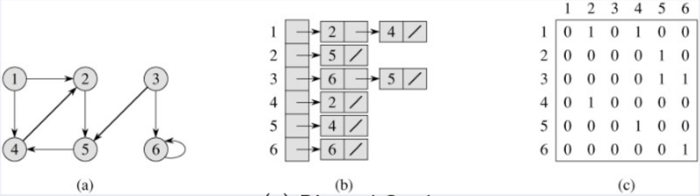
Adjacency Lists

Given a undirected graph G with n vertices and m edges, the adjacency list of a vertex u contains all the vertices that are neighbours of u in G. The array of these adjacency lists can be used to represent G.

Adjacency Matrices and Lists



(t) Undirected Graph



(u) Directed Graph

Adjacency Matrices and Lists

- For weighted graphs, the weights of the edges can be put in the lists or in the entries of the matrix.

Adjacency Matrices and Lists

- For weighted graphs, the weights of the edges can be put in the lists or in the entries of the matrix.
- For directed graphs, the sum of the lengths of all the adjacency lists is $|E|$.

Adjacency Matrices and Lists

- For weighted graphs, the weights of the edges can be put in the lists or in the entries of the matrix.
- For directed graphs, the sum of the lengths of all the adjacency lists is $|E|$.
- For undirected graphs, sum of the lengths of all the adjacency lists is $2|E|$.

Adjacency Matrices and Lists

- For weighted graphs, the weights of the edges can be put in the lists or in the entries of the matrix.
- For directed graphs, the sum of the lengths of all the adjacency lists is $|E|$.
- For undirected graphs, sum of the lengths of all the adjacency lists is $2|E|$.
- Adjacency Lists are more useful for sparse graphs.

Adjacency Matrices and Lists

- For weighted graphs, the weights of the edges can be put in the lists or in the entries of the matrix.
- For directed graphs, the sum of the lengths of all the adjacency lists is $|E|$.
- For undirected graphs, sum of the lengths of all the adjacency lists is $2|E|$.
- Adjacency Lists are more useful for sparse graphs.
- Adjacency matrix is more useful for dense graphs

Adjacency Matrices and Lists

- Adjacency List

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$
  - Time taken to determine if u is neighbour of $v = O(degree(v))$

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$
  - Time taken to determine if u is neighbour of $v = O(degree(v))$
- Adjacency Matrix

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$
  - Time taken to determine if u is neighbour of $v = O(degree(v))$
- Adjacency Matrix
  - Space : $\Theta(|V|^2)$

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$
  - Time taken to determine if u is neighbour of $v = O(degree(v))$
- Adjacency Matrix
  - Space : $\Theta(|V|^2)$
  - Time taken to list all neigbhours of vertex $v = \Theta(|V|)$

Adjacency Matrices and Lists

- Adjacency List
  - Space : $\Theta(|V| + |E|)$
  - Time taken to list all neigbhours of vertex $v = \Theta(degree(v))$
  - Time taken to determine if u is neighbour of $v = O(degree(v))$
- Adjacency Matrix
  - Space : $\Theta(|V|^2)$
  - Time taken to list all neigbhours of vertex $v = \Theta(|V|)$
  - Time taken to determine if u is neighbour of $v = \Theta(1)$

Outline

Graph Terminologies

Adjacency Matrices and Lists

Graph Traversal

Shortest Path Algorithms

Breadth First Search

Given a graph $G = (V, E)$ and a source vertex s.

BFS

Breadth-First Search is a graph traversal technique to systematically explore all the vertices that are reachable from s in G.

Informally, the BFS does the following

Breadth First Search

Given a graph $G = (V, E)$ and a source vertex s.

BFS

Breadth-First Search is a graph traversal technique to systematically explore all the vertices that are reachable from s in G.

Informally, the BFS does the following

- It begins at the source node and explores all the neighboring nodes.

Breadth First Search

Given a graph $G = (V, E)$ and a source vertex s.

BFS

Breadth-First Search is a graph traversal technique to systematically explore all the vertices that are reachable from s in G.

Informally, the BFS does the following

- It begins at the source node and explores all the neighboring nodes.
- Then for each of those nearest nodes, it explores their unexplored neighbor nodes.

Breadth First Search

Given a graph $G = (V, E)$ and a source vertex s.

BFS

Breadth-First Search is a graph traversal technique to systematically explore all the vertices that are reachable from s in G.

Informally, the BFS does the following

- It begins at the source node and explores all the neighboring nodes.
- Then for each of those nearest nodes, it explores their unexplored neighbor nodes.
- It continues step 2, until it explores the entire graph.

Breadth First Search

Key concepts

- It uses a FIFO queue to store the discovered vertices.

Breadth First Search

Key concepts

- It uses a FIFO queue to store the discovered vertices.
- To keep track of progress, it colors each vertex - white, gray or black.

Breadth First Search

Key concepts

- It uses a FIFO queue to store the discovered vertices.
- To keep track of progress, it colors each vertex - white, gray or black.
- All vertices start white.

Breadth First Search

Key concepts

- It uses a FIFO queue to store the discovered vertices.
- To keep track of progress, it colors each vertex - white, gray or black.
- All vertices start white.
- A vertex discovered first time during the search becomes gray.

Breadth First Search

Key concepts

- It uses a FIFO queue to store the discovered vertices.
- To keep track of progress, it colors each vertex - white, gray or black.
- All vertices start white.
- A vertex discovered first time during the search becomes gray.
- A vertex whose all the adjacent vertices have been discovered is colored black.

Breadth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.

Breadth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.
- P[u] : Stores the predecessor of u

Breadth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.
- P[u] : Stores the predecessor of u
- d[u] : The distance from the source s to vertex u computed by the algorithm.

Breadth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.
- P[u] : Stores the predecessor of u
- d[u] : The distance from the source s to vertex u computed by the algorithm.
- A queue Q to store the next elements to be visited.

Breadth First Search

```
for each u in V - {s}                    \\ for each vertex except s.
   do color[u] = WHITE
       d[u] = infinity
       P[u] = NIL
color[s] = GRAY                           \\ Source vertex discovered
d[s] = 0                                  \\ initialize
P[s] = NIL                                \\ initialize
Q = {}                                    \\ Clear queue Q
ENQUEUE(Q, s)
while Q is non-empty
do u = DEQUEUE(Q)                            \\ That is, u = head[Q]
     for each v adjacent to u
          do if color[v] = WHITE
             then  color[v] = GRAY
                   d[v] = d[u] + 1
                   P[v] = u
                   ENQUEUE(Q, v)
     color[u] = BLACK
```

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.
- Enqueue and Dequeue takes $O(1)$ time.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.
- Enqueue and Dequeue takes $O(1)$ time.
- Total time taken by queue operations is $O(n)$.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.
- Enqueue and Dequeue takes $O(1)$ time.
- Total time taken by queue operations is $O(n)$.
- The adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.
- Enqueue and Dequeue takes $O(1)$ time.
- Total time taken by queue operations is $O(n)$.
- The adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.
- Sum of lengths of all adjacency lists is $O(m)$, so total time in this step is $O(m)$.

Breadth First Search

Say the graph has n vertices and m edges.

- Initialization takes $O(n)$.
- Every vertex is enqueued only once.
- Enqueue and Dequeue takes $O(1)$ time.
- Total time taken by queue operations is $O(n)$.
- The adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.
- Sum of lengths of all adjacency lists is $O(m)$, so total time in this step is $O(m)$.
- Running time of BFS $= O(m + n)$.

Depth First Search

DFS

Depth First Search is another graph traversal technique which starts at the source node and explores as far as possible along each branch before backtracking.

Depth First Search

DFS

Depth First Search is another graph traversal technique which starts at the source node and explores as far as possible along each branch before backtracking.

Depth First Search

DFS

Depth First Search is another graph traversal technique which starts at the source node and explores as far as possible along each branch before backtracking.

The idea behind DFS is the following :

- DFS progresses by expanding the first node that appears.

Depth First Search

DFS

Depth First Search is another graph traversal technique which starts at the source node and explores as far as possible along each branch before backtracking.

The idea behind DFS is the following :

- DFS progresses by expanding the first node that appears.
- It goes deeper and deeper from one node to its child until it hits the node that has no children.

Depth First Search

DFS

Depth First Search is another graph traversal technique which starts at the source node and explores as far as possible along each branch before backtracking.

The idea behind DFS is the following :

- DFS progresses by expanding the first node that appears.
- It goes deeper and deeper from one node to its child until it hits the node that has no children.
- The search backtracks, returning to the most recent node it hasn't finished exploring.

Depth First Search

- As in BFS, vertices are colored during the search to indicate their state.

Depth First Search

- As in BFS, vertices are colored during the search to indicate their state.
- Each vertex is initially white.

Depth First Search

- As in BFS, vertices are colored during the search to indicate their state.
- Each vertex is initially white.
- It is colored gray when discovered in the search.

Depth First Search

- As in BFS, vertices are colored during the search to indicate their state.
- Each vertex is initially white.
- It is colored gray when discovered in the search.
- Blackened when its adjacency list has been examined completely.

Depth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.

Depth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.
- P[u] : Stores the predecessor of u

Depth First Search

The following Data Structures are used :

- color[u] : Stores the color of each vertex $u \in V$.
- P[u] : Stores the predecessor of u
- d[u] : Stores the timestamp when u is first discovered.

Depth First Search

```
DFS (V, E)
      for each vertex u in V[G]
         do color[u] = WHITE
                 P[u] = NIL
      time = 0
      for each vertex u in V[G]
         do if color[u] = WHITE
                 then DFS-Visit(u)

DFS-Visit(u)
      color[u] = GRAY
      time = time + 1
      d[u] = time
      for each vertex v adjacent to u
         do if color[v] = WHITE
                 then P[v] = u
                         DFS-Visit(v)
      color[u] = BLACK
      time = time + 1
```

Depth First Search

- The loops in line 1-3 and 5-7 in DFS function are executed $\Theta(n)$ times.

Depth First Search

- The loops in line 1-3 and 5-7 in DFS function are executed $\Theta(n)$ times.
- During the execution of DFS-VISIT(v), the loop in 4-7 is executed $|Adj(v)|$ times.

$$\sum_{u \in V} |Adj(v)| = \Theta(m)$$

Depth First Search

- The loops in line 1-3 and 5-7 in DFS function are executed $\Theta(n)$ times.
- During the execution of DFS-VISIT(v), the loop in 4-7 is executed $|Adj(v)|$ times.

$$\sum_{u \in V} |Adj(v)| = \Theta(m)$$

- Time taken by the function DFS-VISIT is $\Theta(m)$.

Depth First Search

- The loops in line 1-3 and 5-7 in DFS function are executed $\Theta(n)$ times.
- During the execution of DFS-VISIT(v), the loop in 4-7 is executed $|Adj(v)|$ times.

$$\sum_{u \in V} |Adj(v)| = \Theta(m)$$

- Time taken by the function DFS-VISIT is $\Theta(m)$.
- Total time taken in DFS is $\Theta(m + n)$.

Problems on DFS and BFS

- Finding a cycle in the graph

Problems on DFS and BFS

- Finding a cycle in the graph
- Checking if the graph is connected

Problems on DFS and BFS

- Finding a cycle in the graph
- Checking if the graph is connected
- Finding the diameter of a tree

Problems on DFS and BFS

- Finding a cycle in the graph
- Checking if the graph is connected
- Finding the diameter of a tree
- Finding minimum distance for all pairs in an unweighted graph

Outline

Finding the Shortest Path

- To find the shortest path from vertex s to v in an unweighted graph, we can use BFS.

Finding the Shortest Path

- To find the shortest path from vertex s to v in an unweighted graph, we can use BFS.
- Dijkstra's algorithm is a well-known algorithm used to find the shortest path between two vertices in a weighted graph.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array *dist*[] which will store the distance of all vertices from s.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- Visit all the unvisited neighbours of the current node, and update their tentative distance.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- Visit all the unvisited neighbours of the current node, and update their tentative distance.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- Visit all the unvisited neighbours of the current node, and update their tentative distance.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
- If the target node has been visited, then report the distance to that node.

Dijkstra's Algorithm

The basic idea behind Dijkstra's Algorithm is the following :

- Maintain a tentative distance array $dist[]$ which will store the distance of all vertices from s.
- Initialize $dist[s]$ to be zero and set it to infinity for all other vertices.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- Visit all the unvisited neighbours of the current node, and update their tentative distance.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
- If the target node has been visited, then report the distance to that node.
- Otherwise, choose any node in the univisted set and mark it as the current noode and repeat Step 4.

Dijkstra's Algorithm

```
function Dijkstra(G, s):

    Create empty set Q

    for each vertex v :
        dist[v] = INFINITY
        add v to Q

    dist[s] = 0
    while Q is not empty:
        u = vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] = alt

    return dist[]
```

Dijkstra's Algorithm

- Using Fibonacci heaps or self-balancing binary search trees, the above algorithm can run in $O(m + n \log n)$ time.

Reading Assignment

- All-Pair Shortest Path Algorithms

Reading Assignment

- All-Pair Shortest Path Algorithms
- Minimum Spanning Tree

Reading Assignment

- All-Pair Shortest Path Algorithms
- Minimum Spanning Tree
- Directed Graph Traversal

Reading Assignment

- All-Pair Shortest Path Algorithms
- Minimum Spanning Tree
- Directed Graph Traversal
- Topological Sorting