# Graph Theory & Algorithms

Aniket Pandey

23 December 2017

## 1 Introduction

Graph Theory is the study of Graphs, the mathematical objects modelling the pairwise relation of Vertices(also called nodes) and Edges where two nodes are connected by edges. A graph can be directed or undirected, cyclic or acyclic, linear or weighted etc.

Graph Theoretical concepts are widely used to study and model various applications, in different areas. For example, in Computer Science, problems like travelling salesman problem, the shortest spanning tree in a weighted graph and in Mathematics like hamiltonian graphs and Fermat's Little Theorem & Nielson-Schreier Theorem.

## 2 Notations

### 2.1 Big-O Notation

Big-O Notations are used in mathematics to characterize functions according to their growth rate. In Graph Theory, efficiency of an algorithm is measured in terms of the input length $n$ as $n \to \infty$.

Formal definition would be
If $f : N \to N$ and $g : N \to N$ are two functions, then $f = \mathrm{O}(g)$ if and only if $f(n) < c{\cdot}g(n)$ for a constant $c$ as $n \to \infty$.

### 2.2 Other Notations

There are a few more notations which complement Big-O Notation. I will give a brief information about these.

For functions $f$ & $g$ from $N$ to $N$

$$f = \Omega(g) \quad \text{if } g = O(f) \tag{1}$$
$$f = \Theta(g) \quad \text{if } f = O(g) \ \&\ g = O(f) \tag{2}$$
$$f = o(g) \quad \text{if there exists } \varepsilon \text{ such that } f(n) < \varepsilon \cdot g(n) \tag{3}$$
$$f = \omega(g) \quad \text{if } g = o(f) \tag{4}$$

## 3 Terminologies

Here are a few basic terminologies that are used to represent navigation through the Graph.

**Walk** A walk is any route through a graph from vertex to vertex along edges. A walk can end on the same vertex on which it began or on a different vertex. A walk can travel over any edge and any vertex any number of times.

**Path** A path is a walk that does not include any vertex twice, except that its first vertex might be the same as its last.

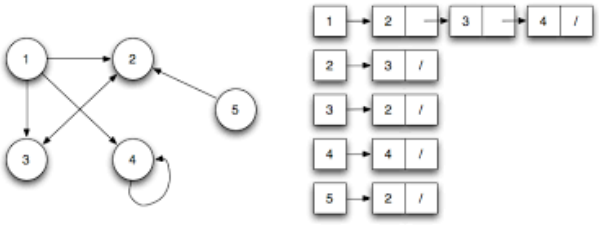
**Trail** A trail is a walk that does not pass over the same edge twice. A trail might visit the same vertex twice, but only if it comes and goes from a different edge each time.

**Cycle** A cycle is a path that begins and ends on the same vertex.

# 4 Data Structures used in Graphs

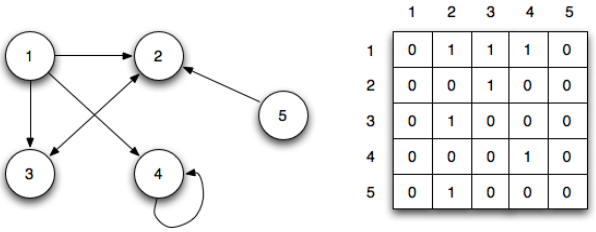
## 4.1 Adjacency List

An Adjacency List is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in the graph.




## 4.2 Adjacency Matrix

An Adjacency Matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.




## 4.3 Stacks and Queues

Stacks and Queues are dynamic sets in which the element removed from the set by the *delete* operation is prespecified. In a Stack, the element deleted from the set is the one inserted most recently, while in a Queue, the element that is to be deleted is the element which has been in the Queue for the longest time.

A **Stack** is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack.

A **Queue** is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle. In the queue only two operations are allowed, enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item.

**STACK**

```
def StackEmpty(S)
    if S.top == 0
        return TRUE
    else return FALSE

def push(S,x)
    S.top = S.top + 1
    S[S.top] = x

def pop(S)
    if StackEmpty(S)
        error "Stack Underflow"
    else S.top = S.top - 1
        return S[S.top+1]
```

**QUEUE**

```
def enqueue(Q,x):
    Q[Q.tail] = x
    if Q.tail == Q.length:
        Q.tail = 1
    else Q.tail = Q.tail + 1

def dequeue(Q):
    x = Q[Q.head]
    if Q.head == Q.length:
        Q.head = 1
    elif Q.head == 0:
        error "Queue Underflow"
    else Q.head = Q.head + 1
    return x
```

# 5 BFS and DFS

## 5.1 Breadth First Search

Breadth First Search (or BFS) is a graph traversal algorithm through which we can determine the shortest path from a source node $V$ to any other node in the graph, which is visitable from $V$. BFS works by starting from the source node, then visits all its neighbours, assignes them the value 1 (distance from $V$). It continues the process by visiting the neighbours of the current node which have not already been visited, assignes them the respective value.

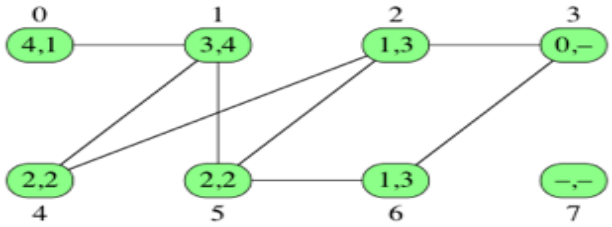
Now the question is, how do we keep track of all visited nodes. Well, for that we use a *boolean array* and a *queue* for storing the status of each node and for storing a node's unvisited neighbours respectively.

Here is a simple pseudocode of BFS.

```
def bfs(graph, S):
  visited = set()
  Q.enqueue(S)

  while Q:
    vertex = Q.dequeue()

    for neighbor in graph[vertex].neighbors:
      if neighbor not in visited:
        visited.add(neighbor)
        Q.enqueue(neighbor)

  return visited
```

**Explanation and Analysis** In the beginning I have declared an empty set *visited*, and enqueued the source node $S$. Then as long as queue $Q$ is not empty (there are unvisited nodes), we'll dequeue it and check all its neighbours, if they have not been visited, enqueue them and mark them as visited. This process will continue till every node is in the visited array. We can further run a test in the boolean array to check if there is a node which cannot be visited from the source node.

For example, in the image shown below, the node 7 cannot be visited from the source node 3.




**Time Complexity** BFS loops through each Vertex and Edge to check the visited condition. So the complexity is,

$$O(V, E) = O(V + E) \tag{5}$$

## 5.2 Depth First Search

Depth First Search (or DFS) is a recursive Graph traversal algorithm using which we can determine the minimum spanning tree, presence of cycles and to check if the graph is bipartite. DFS works on the principle of backtracking, that is, it goes down a path, maintaining visited nodes in a *stack* this time. When it can't find another unvisited node, it backtracks its path to the nearest node which has an unvisited neighbour. It continues this process until it lands back to the source node, implying that the traversal is complete.

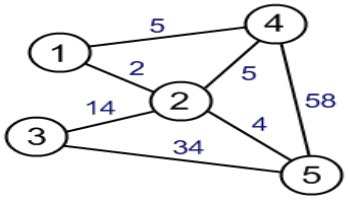
```
def dfs(graph, S, visited = ()):
  visited.push(S)

  for neighbor in graph[S].neighbors:
    if neighbor not in visited:
      dfs(graph, neighbor)

  return visited
```

**Explanation and Analysis** Apart from Graph and source node, the *dfs* function also takes as an input an empty *stack* which is used to keep track of visited chain of nodes, that is, we keep pushing onto the stack the nodes that have been visited, popping them once the recursion is complete. Then the part which checks for unvisited node is same as that in *bfs*, only difference being we call dfs at every iteration.

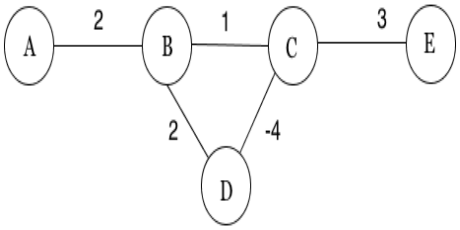
**Time Complexity** Same as that of BFS, the complexity of DFS is $O(V + E)$

# 6 Single Source, Shortest Path

Till now, both algorithms relied on the fact that the weight of each edge is unity. What if the weights were any integer (even negative), in that case, the algorithms discussed above will not work. For instance in the images shown below




Here, the direct path between 4 and 5 (58) is not the shortest one. 4-2-5 will be much shorter.




Here, the cycle $BCD$ has negative length (-1) implying the cycle has infinite negative weight.

To alleviate this issue, I'll discuss two algorithms, namely *Dijkstra's Algorithm* & *Bellman-Ford Algorithm*

## 6.1 Dijkstra's Algorithm

Dijkstras algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source. The graph will have random connected nodes denoted by $u$ and $v$ and the weight of their edge will be $len(u, v)$. Purpose is to find the shortest path of every node from a given source node $s$.

```
def dijkstra(Graph, source):
  dist[source] = 0
  for node in Graph:
    if not node = source:
      dist[node] = infinity
    Q.push(node)

  while Q:
    v = vertex in Q with min dist[v]
    Q.pop(v)
    for u in Graph[v].neighbors:
      alt = dist[v] + len(v, u)
      if alt < dist[u]:
        dist[u] = alt

  return dist[]
```

**Explanation and Analysis** Dijkstra starts off by declaring an array *dist* to store updated shortest distance of a node from the source. Clearly, $len(v, v) = 0$ and for every other node $u$, $dist[u] = \infty$. Now, we select the node which has the least distance from source (say $v$) and check if for a node $u$ $dist[v] + len(u, v) < dist[u]$. If so, then we update the value of $dist[u]$. This process continues till the no more updates can be done (Shortest paths have been found)

**Time Complexity** The code which evaluates vertex $v$ for which $dist[v]$ is minimum can be a binary search of complexity $O(logV)$. For every node $v$, there are $v - 1$ nodes and hence, $v - 1$ possible edges (routes). So the complexity for Dijkstra is,

$$\begin{aligned} O(V, E) &= O(V) + O((E-1)logV) + O(c) && (6) \\ &= O(Elog(V)) && (7) \end{aligned}$$

### 6.2 Bellman-Ford Algorithm

The Bellman-Ford is another algorithm which calculates the shortest paths from a single source. It is slower than Dijkstra, but for appropriate reasons as it also takes into account negative weights. If we put a Graph with negative edge cycles into Dijkstra, it will ignore those cycles and give wrong results.

```
def bellmanFord(Graph, Egde, S):
  for node in Graph:
    dist[node] = infinity
    node.prev = nil
  dist[S] = 0

  for node in Graph:
    for (u,v) in Edge:
      if dist[v] > dist[u] + len(u,v):
        dist[v] = dist[u] + len(u,v)

  for (u, v) in Edge:
    if dist[v] > dist[u] + len(u,v):
      print "A negative weight cycle exists"

  return dist[], Graph.prev[]
```

**Explanation and Analysis** Bellman-Ford works by overestimating the edge lengths of every node, assuming it to be a very large number. $node.prev = nil$ specifies the pointer to previous node which in this case has not yet been initialized. This algorithm relaxes the shortest path of each node by iterating over every possible edge, checks if $dist[v] > dist[u] + len(u, v)$ for adjacent nodes $(u, v)$ and if so, replaces the value of $dist[v]$.

It has been proved that this iteration for $|V|$ times will make sure every shortest path is computed. Now, for the presence of any negative weight cycles, the last iteration

again checks for triangular inequality for every pair $(u, v)$ in the Edge set. If it finds one, reports it as a negative weight cycle.

**Time Complexity** For various relaxations, Bellman-Ford check $|E|$ edges for every node in $|V|$. So the time complexity is,

$$\begin{aligned} O(V, E) &= O(V) + O(V.E) + O(E) + O(c) && (8) \\ &= O(V.E) && (9) \end{aligned}$$