Graph Theory & Algorithms

Aniket Pandey

July 28, 2018

1 Introduction

Graph Theory is the study of Graphs, the mathematical structures modelling the pairwise relation of Vertices (also called nodes) and Edges, where two nodes are connected by an edge. 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 Minimal spanning tree in a weighted graph and in Mathematics like Hamiltonian graphs, 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 portray navigation through a Graph.

Walk A walk is any route through a graph from vertex to vertex along edges. It can end on the same vertex on which it began or on a different vertex and there is no limit on how many times a vertex is convered.

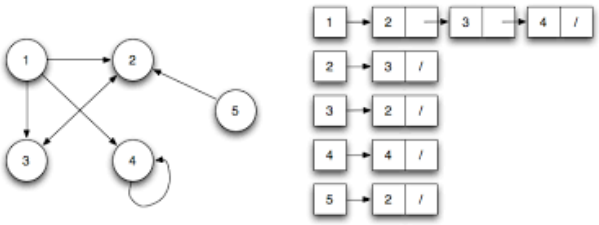
Path A path is a walk that does not include any vertex twice, except that its first vertex might be the same as its last.

Cycle A cycle is a path that begins and ends on the same vertex, like a closed loop.

4 Data Structures used in Graphs

4.1 Adjacency List

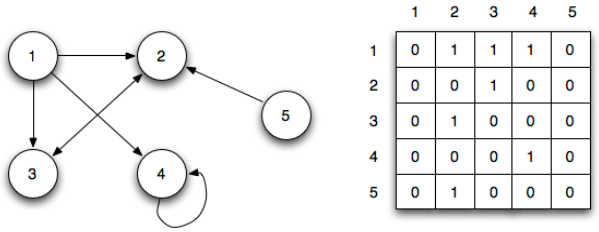
An Adjacency List is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a vertex in the graph.



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.

$$a(i,j) = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$



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.

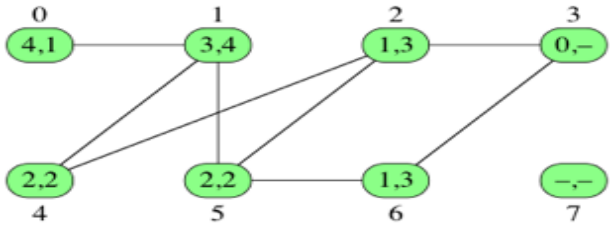
Now the question is, how do we keep track of all visited nodes. Well, for that we use a *boolean array* and a *queue* for storing the status of each node and for storing a node's unvisited neighbours respectively.

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 minimal 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.

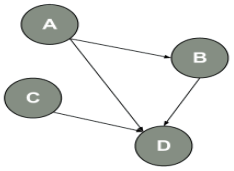
```
def dfs(graph, S, visited = ()):
  visited.push(S)

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

  return visited
```

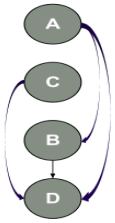
Explanation and Analysis Apart from Graph and source node, the *dfs* function also takes as an input an empty *stack* which is used to keep track of visited chain of nodes, that is, we keep pushing onto the stack the nodes that have been visited, popping them once the recursion is complete. Then the 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 $O(V + E)$

5.3 Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.





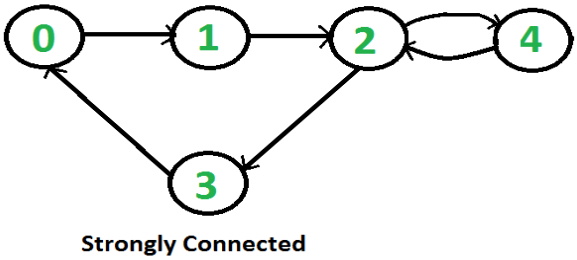

```
def topological(graph(V)):
    list <int> L;
    bool mark[V];
    foreach mark[V]:
        mark[v] = false;

    for v in mark[]:
        if not mark[v]:
            dfs(graph, v, visited)

        L.insert(v);
    return L;

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 Topological sort uses the *finishing time* of a DFS call to determine the order of sorted vertices. The node with the last finishing time is first in the list, as there is no direct path to it from any other vertex. As soon as all edges of a vertex v are marked visited, we add it to the concerned list.

Time Complexity of Topological Sorting is same as DFS, i.e $O(V + E)$ as it is obvious.

6 Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. Finding out if a Graph is Strongly Connected or not is a classic application of Depth First Search.



Strongly Connected

6.1 Kosaraju's Algorithm for verifying SCC

Kosaraju's Algorithm does BFS twice, once with the original Graph and other with the inverted Graph. Idea is that, if it is possible to reach all other vertices from a source vertex and reach the source vertex from all other vertices (reason for inverting the graph), the the Graph is *maximal strongly connected* itself. That is, any vertex is reachable from any other vertex.

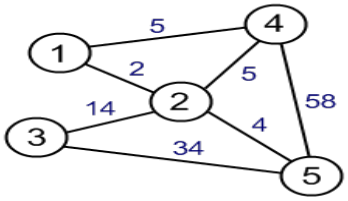
```
def isSCC(Graph):
    bool Visited(V, false)
    traverseDFS(0, visited)

    if visited.len() != V:
        return false

    revGraph = Graph.reverse()
    Visited(V, false)
    revGraph.traverseDFS(0, visited)

    if visited.len() != V:
        return false
    return true
```

Explanation and Analysis Proof of how this works is quite simple. Let u and v be any two vertices. Two confirm the *SCC* property, we need to show that both u & v are reachable from each other. Following the above algorithm, if the vertices are reachable and can be reached from/to a given source node *src*, then we can use the following analogy to prove the correctness of the algorithm.

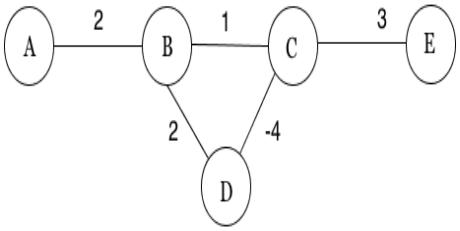
$$U \longrightarrow Src \longrightarrow V \qquad V \longrightarrow Src \longrightarrow U \tag{6}$$

7 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 integers (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*

7.1 Dijkstra's Algorithm

Dijkstra's 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

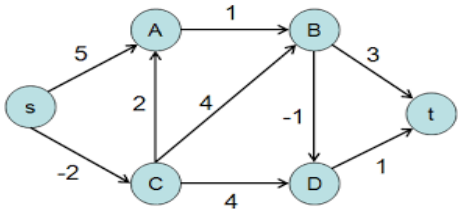
source (say v) and check if for a node u $dist[v] + len(u, v) < dist[u]$. If so, then we update the value of $dist[u]$. This process continues till the no more updates can be done. (Shortest paths have been found)

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,

$$O(V, E) = O(V) + O((E-1)logV) + O(c) \tag{7}$$
$$= O(Elog(V)) \tag{8}$$

7.2 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)
        v.prev = u

  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 $dist[v]$.

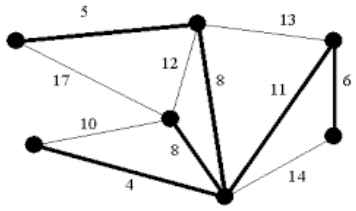
It has been proved that this iteration for $|V|$ times will make sure every shortest path is computed. Now, for the presence of any negative weight cycles, the last iteration again checks for triangular inequality for every pair (u, v) in the Edge set. If it finds one, reports it as a negative weight cycle. Graph.$prev$[] can be used to formulate the shortest path found out by this algorithm, by finding the predecessors of each node starting from the destination node.

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

$$\begin{aligned} O(V, E) &= O(V) + O(V.E) + O(E) + O(c) && (9) \\ &= O(V.E) && (10) \end{aligned}$$

8 Minimum Spanning Tree

A **Spanning Tree** of a Graph is set of edges, essentially forming a tree such that all the vertices of the graph are connected and there is no formation of a cycle in the process. For a weighted graph, we can define a *Minimum Spanning Tree* to be a spanning tree with the additional property that the sum of all weighted edges in minimum.



Mathmematically, we can define the MST problem as followes. Consider a Graph $G = (V, E)$, where V and E are usual notations for Set of all vertices and edges respectively. Since the Graph is weighed, for an edge $(u, v) \in E$, we have a weight function $w(u, v)$ specifying the distance between the edges. We wish to find an acyclic subset $T \subset E$ that connects all vertices and

$$w(T) = \min \sum_{(u,v) \in T} w(u, v) \qquad (11)$$

gives the total length of the MST.

8.1 Building a minimum spanning tree

Algorithms to construct a Minimum Spanning Tree are greedy by nature. This can be proved by simple induction and the generic approach that is taken to build it. The two algorithms to grow an MST are **Kruksal's** and **Prim's** algorithm.

Let A be a subset of some minimum spanning tree. We define a **safe edge** to be an edge (u, v) such that $A \bigcup (u, v)$ does not violate the acyclic property of a mimimum spanning tree. We keep doing this till all the vertices have been taken into account. Folowing is the pseudo-code for a *Generic MST* building, taken from CLRS.

```
GENERIC-MST(G(V,E),w)
A = {}

# Loop till A connects all vertices
while A.len != (V - 1):
    edge = min((u,v))
    A = A.append(edge)
return A
```

Both *Kruksal* and *Prim*'s algorithm build upon this generic approach.

8.2 Kruskal's Minimum Spanning Tree

Krusksal's algorithm works by building a *forest*, i.e disjoint trees. For every iteration, it picks up the edge with the least weight with the vertices necessarily being from separate trees. Care is taken not to connect two vertices from same tree, else a *cycle* would be created, eventually breaking the spanning tree condition.

```
def kruksal(G(E,V)):
    A = {}
    Sorted_Edge(u,v) = sort(G.E(u,v))
    for v in G.V:
        MAKE-SET(v)

    for (u,v) in Sorted_Edge(u,v):
        # Check if u and v are in the same tree or not
        if FIND-SET(u) != FIND-SET(v):
            A = A.append({u,v})
            UNION(u,v)
    return A
```

Explanation and Analysis Kruskal sorts the edges in the decreasing order by their weight. It then simply follows the rule of:

1. Check if addition of edge (u, v) does not form a cycle, via union-find(u, v).
2. If it does not, add the edge to MST-SET() (The Set of Minimum spanning tree).
3. Follow this procedure until all vertices have been connected through $V - 1$ edges.

Time Complexity Sorting the edges takes $O(ElogE)$ time, and the Disjoint tree check condition takes $O(logV)$ time, hence:

$$T(V,E) = O(1) + O(V) + O(ElogE) + O(VlogV) \tag{12}$$
$$= O(ElogE) + O(VlogV) \tag{13}$$

Since there can only be one edge connecting any two vertices, therefore $|E| >= |V| - 1$

$$T(V,E) = O(ElogE) + O(VlogV) \tag{14}$$
$$= O(ElogE) \tag{15}$$

8.3 Prim's Minimum Spanning Tree

Prim's algorithm to obtain an MST from a Graph is also a Greedy algorithm, just like *Kruksal*. In operation it is similar to Dijsktra in terms of initialising all edge distances "*key*" from a selected source vertex and then updating the keys as we go about choosing minimum weight edges.

The algorithm maintains two sets, the first set contains the vertices already included in the MST, called *MST-Set*, the other set contains the vertices not yet included. The idea is to transfer the vertices one by one to the former following the above algorithm.

```
def prim(G, w, src):
    mst_set = {}
    for u in G.V:
        u.key = INFY

    src.key = 0
    mst_set.append(src)

    Q = G.V
    while Q is not None:
        u = Extract-Min(Q)
        for v in G.Adj[u]:
            if (v in Q) and w(u,v) < v.key:
                v.key = w(u,v)
                mst_set.append(v)
    return mst_set
```

Explanation and Analysis Prim initializes the *key* for every vertex v as ∞ except for the source vertex, for which the *key* is 0. So our set is initialised as $\{0, \infty, \infty, ...\}$.

Now, in the main iteration, it checks if a node is in non MST-Set Q, if not then it simply extracts the node with minimum key and updates it according to its neighbors. This process is continued until all the V vertices are in *mst_set*, which is accordingly returned.

9 Conclusion

So these were the 8 major Graph Algorithms I studied for the project. After careful analysis, I realized the application of these algorithms is much more prevalent than the problems which they seem to elucidate. I was able to solve most of the problems concerning graphs by simply using the trivial BFS or DFS. However, on certain occasions, I had to go for the weighted giants.

For the problems with application of either BFS or DFS, I had to observe how far I'll have to travel down the graph. For quick results, which are only concerned with examining a section of graph, I preferred DFS as its recursive nature allowed me to cover the relevant portions quickly. It also reduced the net complexity by not scanning every neighbour of every node.

Most of the problems involved application of these algorithms like topological sorting, checking for bipartite graph, finding shortest route, finding fastest route, detecting cycles and so on. I rarely encountered negative weights so never had to go for Bellman-Ford. Although I believe it will be useful in many advanced problems.