COMPSCI 330: Design and Analysis of Algorithms 02/09/2016 and 02/11/2016

Graph Representations and Traversal

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song*

1 Overview

This lecture covers basic definitions about graph, graph representations, Depth First Search (DFS), Breadth First Search (BFS) and introduction to shortest paths.[1]

2 Definitions

A *graph* $G = (V, E)$, where V is the set of vertices, and E is the set of edges. An edge $e \in E$ is an unordered pair (u, v) in undirected graphs, where $u, v \in V$. In directed graphs, an edge e is an ordered pair. A *path* from a vertex u to a vertex v is a sequence of vertices $(w_0, w_1, ...w_k)$, where $u = w_0$, $v = w_k$ and $(w_{i-1}, w_i) \in E$ for all $1 \leq i \leq k$. The path is a *cycle* if $u = v$. The *length* of a path in an unweighted graph is the number of edges on the path. The *length* of a path in a weighted graph is the sum of the weights of all edges on the path. The *distance* between two vertices u and v is defined as the length of *the shortest path* connecting them.

3 Graph Representations

Adjacency Matrix: For a graph $G = (V, E)$, we number the vertices as $1, 2, ..., |V|$. Construct a $|V| \times |V|$ matrix A, where $A[i, j] = 1$ if $(i, j) \in E$, otherwise $A[i, j] = 0$. The space requirement of the adjacency matrix is $\Theta(|V|^2)$.
Adjacency List: For a graph $G = (V, E)$, we construct an array of $|V|$ lists and one for each vertex. A vertex v is in the list of a vertex u iff $(u, v) \in E$. The space requirement of the adjacency list is $\Theta(|V| + |E|)$.

4 Applications

There are many applications of graphs such as:

- *Map Coloring*: Map coloring is the problem of coloring the vertices of a graph such that no two neighboring vertices have the same color. The name stems from the old problem of coloring countries on a geographical map such that the countries' borders can be distinguished by using different colors for neighboring countries. In that problem, each vertex represents a country, and there is an edge between two countries whenever they share a border. Another example of the application of the map coloring algorithm is the exam scheduling problem. In this problem, exams for classes need to be scheduled in such a way that if a student is taking multiple classes, his exams will not conflict with one another. A good way to reduce this problem to graph coloring is to use a vertex to represent a

[1] Most of the material in this note is from previous notes by Ang Li, Roger Zou and Wenshun Liu for this class in Fall 2014.

class, a edge between two vertices when there is at least one student taking both classes, and color to represent time slots.

- *Facebook Graph*: A node represents a person and an edge represents the friendship between two people.
- *PageRank*: A node represents a webpage and an edge represents a hyperlink.

5 Depth First Search (DFS) and DFS tree

5.1 Depth First Search (DFS)

Pseudo-code of DFS:

Algorithm 1 Depth first search (DFS)

```
1: function DFS(G)
2:     for each vertex v ∈ G.V do
3:         if v is unmarked then
4:             DFS-Trav(G, v)
```

Algorithm 2 Depth first search (DFS) from a source vertex

```
1: function DFS-TRAV(G, v)
2:     Push v into stack S
3:     Mark v
4:     for each w such that (v, w) ∈ G.E do
5:         if w is unmarked then
6:             DFS-Trav(G, w)
7:     Pop v from S
```

The running time of DFS is $O(|V|+|E|)$ using adjacency list, where $G=(V,E)$.

5.2 Pre Order and Post Order

The *pre order* of a vertex is the time when the vertex is pushed into the stack during DFS. The *post order* of a vertex is the time when the vertex is popped out of the stack.

5.3 DFS tree

A DFS tree can be constructed during DFS. An edge (u,v) is in the tree if a vertex is found for the first time by exploring edge (u,v). There are four types of edges clarified by DFS tree: tree edge, forward edge, cross edge, back edge as shown in Figure 1. A forward edge is an edge from a vertex to one of its descendants that are not tree edge. A cross edge is an edge from a vertex u to a vertex v such that the subtrees rooted at u and v have no overlap. A back edge is an edge from a vetex to one of its ancestors. Table 1 is a chart of edge types with their corresponding $pre/post$ orders.

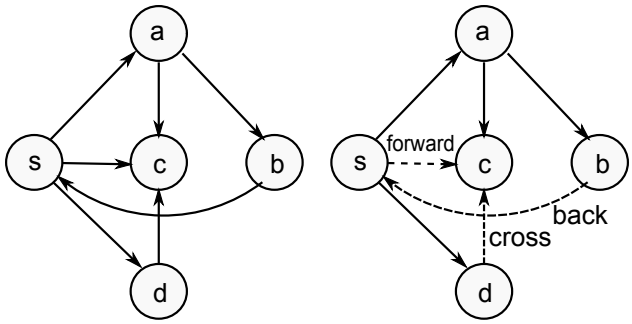

Figure 1: A graph is shown on the left. We run DFS on the graph using vertex s as the source and the edges can be categorized into four types as shown on the right: the solid edges are tree edges.

Table 1: Edge Type and *Pre/Post* Order

Edge Type (u,v)	***pre/post* order**
Tree/forward	$pre(u) < pre(v) < post(v) < post(u)$
Back	$pre(v) < pre(u) < post(u) < post(v)$
Cross	$pre(v) < post(v) < pre(u) < post(u)$

6 Topological Sorting for Directed Acyclic Graphs (DAGs)

Theorem 1. *A directed graph G is acyclic iff DFS on this graph has no back edges.*

Proof: One direction: If there is a back edge, there is a cycle. The other direction: If there is a cylce, let vertex v is the first vertex in the cylce that is visited by DFS, then all other vertices in the cycle are descendents of v in the DFS tree including a vertex u such that edge (u,v) exists in the graph, which results in a back edge.

We can run DFS on a DAG and sort the vertices in decreasing order of their post order values such that all edges point to one direction. This kind of sorting is called *topological sorting*.

7 (Strong) Connectivity

Definition 1. *In a directed graph $G = (V,E)$, a vertex u is connected to a vertex v if there exists a path from u to v.*

Definition 2. *In a directed graph $G = (V,E)$, a vertex u is **strongly** connected to a vertex v if there exists a path from u to v, and s path from v to u.*

Strong connectivity of vertices in a directed graph $G = (V,E)$ is an *equivalence relation* because its reflexivity, transitivity, and symmetry properties.

7.1 Strong Connected Components (SCC)

Definition 3. *In a directed graph $G = (V,E)$, a Strongly Connected Component (SCC) is a maximal subset $C \subseteq V$ s.t. any two vertices $u,v \in C$ are strongly connected.*

Any directed graph can be partitioned into SCCs. We can abstract each SCC as a new vertex and get a DAG of SCCs. A *source* SCC is a SCC that has no edges going into it. A *sink* SCC is a SCC that has no edges going out.

7.2 An Efficient Algorithm for SCCs

To come up with an efficient algorithm to find all the SCCs of a directed graph, observe that DFS from any vertex v in a sink SCC C_{sink} only visits vertices in C_{sink}. Thus, if we have some way of finding such a vertex v, we can do DFS from v, delete all vertices DFS visits, and repeat. However, the problem is finding such a vertex v. The solution arises when one observes that if we run DFS on a directed graph G, the vertex with highest post value belongs to a source SCC. By reversing the direction of all edges in G to construct a new graph G^R, a source SCC in G^R corresponds to a sink SCC in G.

Algorithm to compute SCCs in directed graph $G = (V, E)$

1. Reverse all edges in G to construct graph G^R.
2. Run DFS on G^R, keep a *post* value table, and find the vertex v with the largest *post* value.
3. Run DFS from v on G.
4. Let C_i be the set of all vertices visited by DFS starting from v. Set $V \leftarrow V \setminus C_i$ and remove related edges.
5. Check the *post* value table for the vertices remaining in V and identify the vertex v with the largest *post* value. Go to step 3. Repeat until $V = \emptyset$.

Running Time: Constructing G^R from G (step 1) can be performed in $O(m)$. Finding a vertex in the *sink* SCC can be done in $O(m)$. These two steps can be done before the main loop (steps 3-5). Each iteration i of steps 3-5 takes $O(m_i)$, where m_i is the size of C_i. Because once a vertex and associated edges are deleted they can never be visited again, this means that $O(\sum_i m_i) = O(m)$. Thus the worst case running time is $O(m)$.

8 Breadth First Search (BFS)

8.1 Description

Pseudocode of BFS:

Algorithm 3 Breadth First Search (BFS)

```
1: function BFS(G, s)
2:     mark(s)
3:     enqueue s in Q
4:     while Q ≠ ∅ do
5:         dequeue v from Q
6:         for all neighbors u of v do
7:             if u is unmarked then
8:                 mark(u)
9:                 enqueue u in Q
```

A BFS tree can be constructed during BFS. An edge (u,v) is in the tree if a vertex is found for the first time by exploring edge (u,v). The BFS tree marks the distance from the source vertex to any reachable vertex. We expand the above pseudocode such that distances can be recorded:

Algorithm 4 BFS tree with distances recorded

```
1:  function BFST(G, s)
2:      mark(s)
3:      dist(s)= 0
4:      enqueue s in Q
5:      while Q ≠ ∅ do
6:          dequeue v from Q
7:          for all neighbors u of v do
8:              if u is unmarked then
9:                  mark(u)
10:                 put edge (v,u) in the BFS tree
11:                 dist(u)= dist(v)+1
12:                 enqueue u in Q
```

Lemma 2. *A BFS tree does not contain any forward edges.*

Proof. We prove the lemma by contradiction. If such a *forward edge* (u,v) existed, v would have been put into the queue at an earlier time. The edge (u,v) would have become a *tree edge*. □

8.2 Running Time

The running time of BFS is $O(n+m)$, where n is the number of vertices and m is the number of edges in the graph.

9 Shortest Paths

BFS computes shortest paths in unweighted graphs. However, for weighted graphs, BFS is not an efficient algorithm to find shortest paths. Dijkstra's algorithm is able to find the shortest paths from a source vertex to all vertices for any graph with non-negative edges.

Algorithm 5 Dijkstra's Algorithm

```
1:  function DIJKSTRA(G, s)
2:      dist[s] = 0;
3:      dist[v] = +∞ for all v ≠ s
4:      add all vertices to a heap H
5:      while H ≠ ∅ do
6:          v = EXTRACT_MIN(H)
7:          for all edges (v,u) do
8:              if dist[u] > dist[v] + l(u,v) then
9:                  dist[u] = dist[v] + l(u,v)
```

9.1 Running Time Analysis

The running time of Dijkstra's Algorithm is $O((m+n)\log n)$ using a heap, where n is the number of vertices and m is the number of edges in the graph. With a Fibonacci heap, the running time can be reduced to $O(m+n\log n)$.