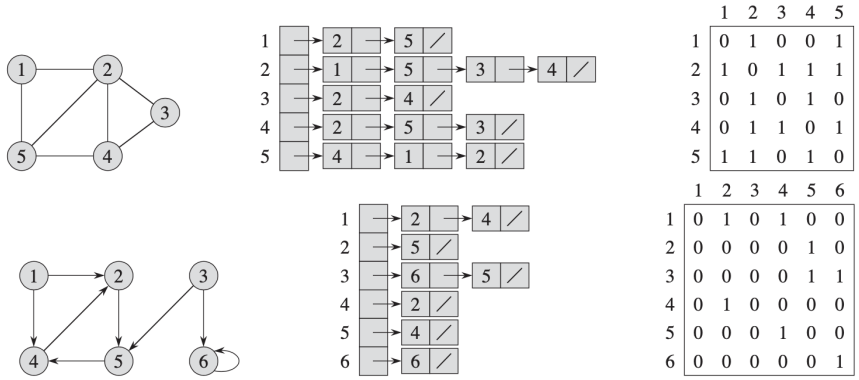
Intro Algo Chapter 22

Mengxi Wu (mw4355@nyu.edu)

September 27, 2020

1 Representation of Graph

Two ways to represent a graph, $G = (V, E)$. It could be adjacency-list or adjacency-matrix. Usually, we use adjacency-list to represent sparse graphs ($|E| << |V^2|$). We may prefer adjacency matrix when the graph is dense ($|E|$ is closer to $|V|^2$) or we need to be able to find whether an edge exist between two vertices quickly.



Adjacency-list Representation
G consists of an array Adj of $|V|$ lists. For each $u \in V$, the adjacency list Adj[u] contains all the vertices adjacent to u in G (there is an edge $(u,v) \in E$). If G is a directed graph, the sum of lengths of all adjacency lists is $|E|$. If G is an undirected graph, the sum of lengths of all adjacency lists is $2|E|$. The memory required for adjacency list is $\Theta(V + E)$.

Adjacency-matrix Representation
G consists of a $|V| \times |V|$ matrix. The entry $a_{ij} = 1$ if $(i, j) \in E$; otherwise, $a_{ij} = 0$. The adjacency matrix requires $\Theta(V^2)$ memory.

2 Breath-first search

Breath-first search constructs a breadth-first tree. Whenever the search discovers a undiscovered vertex v in the course of scanning the adjacency list of an already discovered vertex u, the edge (u,v) are added to the tree. u is the predecessor or parent of v. Each vertex is discovered at most once, so it has at most one parent. If u is on the simple path in the tree from root s to vertex v, then u is an ancestor of v.

Line 1-9 initialize the attributes and add the source vertex to the queue. The queue consists of the set of grey vertices (which are discovered but their adjacency lists has not been fully examined). Line 10-18 iterates as long as there remain gray vertices.

Line 13 ensures that each vertex is added to the queue at most once. Line 11, and 14-17 takes $O(1)$ each time, so total time for these lines is $O(V)$. The procedure scans the adjacency list of each vertex only when the vertex is dequeued, so it scans the adjacency list at most once. The sum of the lengths of all

adjacency lists is $\Theta(E)$. The total time on scanning the lists is $O(E)$. Line 12-13 take $O(E)$ in total. The initialization takes $O(V)$ so $O(V+E)$ in total for BFS.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

3 Depth-first search

Unlike breath-first search, depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edged leaving it. Once all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. The process continues until we have discovered all the vertices that reachable from the original source vertex. If any undiscovered vertices remain, the depth-first search select one of them as a new source, and then do search on the new source. In fact, BFS can also be done in this way in order to traverse the whole graph not only the vertices can be reached from tne source.

Breath-first search has predecessor subgraph forms a tree. The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first tree. We define the forest as $G(V, E_\pi)$ where $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$. When DFS returns, every vertex has its discovery time $u.d$ and finishing time $u.f$.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

Lines 5-7 in DFS check each vertex in V in turn, and when a white vertex is found, visiting it using

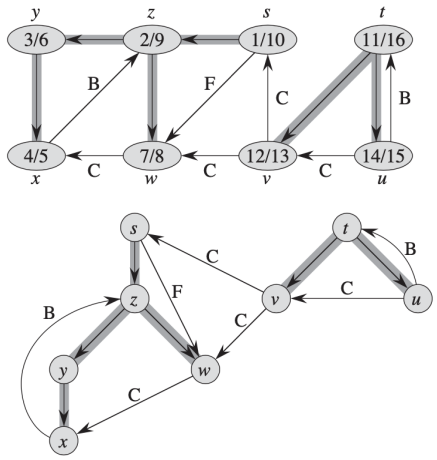
DFS-VISIT. Every time DFS-VISIT is called, vertex u becomes the root of a new tree in depth-first forest. The loops on lines 1-3 and lines 5-7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. The length of the adjacent list is $\Theta(E)$. Lines 4-7 in DFS-VISIT execute $\Theta(E)$ in total. Thus, the running time for DFS is $O(V + E)$

Properties of DFS
1. Vertex v is a descendant of vertex u in depth-first forest if and only if v is discovered during the time in which u is gray.
2. Discovery and finishing times have parenthesis structure. The interval $[u.d, u.f] \subset [v.d, v.f]$, then u is a descendant of v in a depth-first tree.

Nesting of descendants' interval
Vertex v is a proper descendant of vertex u in depth-first forest for graph G if and only if $u.d < v.d < v.f < u.f$.

White-path theorem
In a depth-first forest of a graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Classification of edges



1. **Tree edges**: When we do DFS, only some edges (the ones for which visited[v] is false and v was first discovered by exploring edge (u, v)) will be traversed. These edges will form a depth-first search tree of G starting at the given source s, and the edges in this tree are called tree edges.

The other edges of G can be divided into three categories:
2. **Back edges**: In G, an edge (u, v) that connects u with its ancestor v in a DFS tree and form a cycle is called back edge.
3. **Forward edges**: In G, an edge (u, v) (not tree edge) that connects u with its descendant v is called forward edges.
4. **Cross edge**: In G, an edge (u, v) that connects u to a previously visited vertex v which is neither an ancestor nor a descendant of u.

For an edge (u, v), the color of v indicates the type of the edge. White indicates a tree edge. Gray indicates a back edge. Black indicates a forward edge or cross edge (a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$). In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge. Let (u, v) be an arbitrary edge of G and assume $u.d < v.d$. If u discovers v, the edge is a tree edge. If v finds u while u is still gray, the the edge is a back edge. Forward edge becomes back edge and cross edge becomes tree edge.

4 Topological sort

A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. If the graph contains a cycle, then no linear ordering is possible. We use depth-first search to perform topological sort of a directed **acyclic** graph.

TOPOLOGICAL-SORT(G)
1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

The running time of the algorithm is $O(V + E)$ since DFS takes $O(V + E)$ and inserts each vertex in the front of list takes $O(1)$.

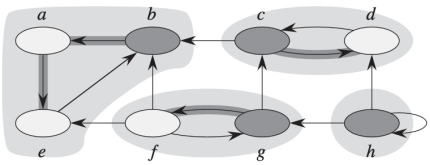
Lemma 22.11
A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.
Given that a depth-first search produces a back edge (u, v), the vertex v is an ancestor of vertex u. Thus, G has a path from v to u, and the back edge (u, v) completes a cycle.
Given G contains a cycle c. Let v be the first vertex discovered in c and (u, v) is the preceding edge in c. At time $v.d$, the vertices in c form a path of white vertices from v to u. By the white-path theorem, vertex u is a descendant of v. Thus, (u, v) is a back edge.

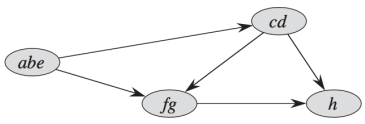
Theorem 22.12
Topological-Sort produces a topological sort of the directed acyclic graph provided as its input.
Consider any edge $(u.v)$ explored by DFS. When this edge is explored v cannot be grey ((u, v) cannot be a back edge so v cannot be u's ancestor). v must be either white or black. If v is white, v is a descendant of u. u will be finish after v is finished so $v.f < u.f$. If v is black, then v is alreay finished but u is still exploring so $v.f < u.f$. For any edge (u, v) in the dag, $v.f < u.f$ so the algorithm is correct.

5 Strongly connected components

This is a classic application of depth-first search: decomposing a directed graph into its strongly connected components. Many algorithms that works with directed graphs begin with such decomposition. Usually, after decomposing we run algorithm on each of the components then we combine together.

Strongly connected component
A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subset V$ such that for every pair of vertices u and v in C, we have them can reach each other.





Reverse/Transpose graph
$G^R = (V, E^R)$, where $E^R = \{(u, v) : (v, u) \in E\}$. That is, E^R consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^R is $O(V + E)$ (visit each list plus each vertex). G and G^R have exactly the same strongly connected components. u and v are reachable from each other if and only if they are reachable from each other in G^R.

Strongly connected components in G will form a component graph called G^{SCC} which is a acyclic graph. To extract strongly connect components in G, we need to first find the vertex in the last component. The last component refers to the component only has incoming edges from other components. When we

do DFS on the last component, the depth-first tree will be formed only by the vertices in this component so we can extract this component.

We cannot directly find a vertex in the last component but the vertex with largest finishing time is guaranteed to be in the first component. The first component refers to the component only has outgoing edges. Only all other components are visited the first component can be finished. Thus, we reverse the original graph G. In reverse graph G^R, the first component in G becomes the last component in G^R. Therefore, we can first find the vertex v with largest finishing time by doing DFS on the original graph G; then, we use v as the source vertex to do DFS on reverse graph G^R in order to extract the strongly connected components.

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
separate strongly connected component

In line 1, store the result in a list. Every time when a vertex is finished, inserted the vertex in the front of the list. In this way, we can get a list of vertices presented in decreasing finishing time. In line 2, after extract a component, pick the vertex that is not visited and has the largest finishing time to do DFS again.

Suppose G has strongly components $C_1, C_2, ...C_k$. The vertex set V^{SCC} is $\{v_1, v_2, ...v_k\}$, and it contains a vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$.

Lemma 22.13
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \to u'$. Then G cannot also contain a path $v' \to v$.
If $v' \to v$, then $u \to u' \to v'$ and $v' \to v \to u$. Thus u and v' can reach each other, contradicting the assumption that C and C' are distinct strongly connected components.

Lemma 22.14
Let C and C' be distinct strongly connected components in directed graph G. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v' \in C$. Then $f(C) > f(C')$. ($d(C) = min_{u \in C}\{u.d\}$ and $f(C) = max_{u \in C}\{u.f\}$)

Corollary 22.15
Let C and C' be distinct strongly connected components in directed graph G. Suppose that there is an edge $(u, v) \in E^R$, where $u \in C$ and $v \in C'$. Then, $f(C) < f(C')$.

To make 22.14 and 22.15 more clear, consider three strongly connected components C_1, C_2, C_3 in G where $f(C_1) > f(C_2) > f(C_3)$. In original graph G, the outgoing edges of C_2 will be towards C_3. However, in reverse graph G^R, the outgoing edges of C_2 will be towards C_1.

Corollary 22.15 explains why the strongly connected component C with maximum finishing time should be extracted first, since it has no outgoing edge towards any other strongly connected component. The vertex x in C can only reach the other vertices in C and will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 selects as a root a vertex from some other strongly connected component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by Corollary 22.15, the only edges in G^R from C' to any other component must be to C, which we have already visited. Each depth-first tree will be exactly one strongly connected component.