Graph Introduction

Alison Liu

Graph is used heavily in the field of math and theoretical computer science to model relations between objects.

1 Terminologies

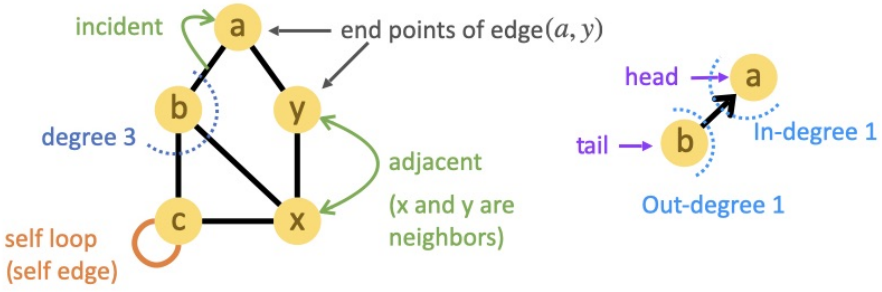
First, we give the formal definitions of terminologies that will be used in this lecture. A graph $G = (V, E)$ consists of a set V of *vertices* and a set E of *edges*, where an edge $(u, v) \in E$ connects two vertices $u, v \in V$. Sometimes, people use $n = |V|$ and $m = |E|$ to represent the number of vertices and edges, respectively. If there exists an edge between two vertices u and v, we say that

- u and v are *adjacent*,
- u and v are neighbors to each other,
- u and v are *end points* of the edge, and
- (u, v) is incident to u (and v).

The *degree* of a vertex u is the number of edges where u is one of the edges.

The edges in a graph can be associated with real numbers, which is called *weight*. A graph in which each edge has a weight of 1 is called a *unweighted* graph, otherwise, it is called a *weighted* graph.

A graph can be *directed* or *undirected*. In a directed graph, the edge (u, v) is an ordered set. That is, $(u, v) \neq (v, u)$. In a directed graph, given an edge (u, v), u is called the *tail* of the edge, and v is called the *head* of the edge. The *in-degree* of a vertex v is the number of edges where v is the tail, and the *out-degree* of v is the number of edges where v is the head.



1.1 Graph representation

There are two ways to represent a graph, *adjacency list* and *adjacency matrix*. In the adjacency list, there is a size-$|V|$ array, where each entry refers to one vertex. An entry of vertex v links to a linked list of its neighbors. In the adjacency matrix, there is a $|V| \times |V|$ matrix, where the entry $a_{u,v}$ is 1 if and only if (u, v) is an edge.

Query complexity and space complexity. Checking if two vertices u and v are adjacent takes $O(|V|)$ time in an adjacency list and $O(1)$ time in an adjacency matrix. Listing all neighbors of a vertex v, on the other hand, takes $O(\text{degree of } v)$ time in an adjacency list and $O(|V|)$ time in an adjacency matrix. Space-wise, the adjacency list needs $O(|V|+|E|)$ units, and the adjacency matrix needs $O(|V|^2)$ units. More complexities can be found in the following table.

	Adjacency list	Adjacency matrix
Space	O(|V|+|E|)	O(|V|²)
Add an edge	O(1)	O(1)
Delete an edge	O(max degree) = O(|V|)	O(1)
List all neighbors of a vertex	O(# of neighbors) = O(|V|)	O(|V|)
Check adjacency of vertices u and v	O(max degree) = O(|V|)	O(1)
List all edges	O(|E|)	O(|V|²)

2 Depth-first-search

One of the algorithms that traverse the graph is *depth-first-search (DFS)*. Initially, all vertices are *not-discovered*. Throughout the process, a vertex is *discovered* when they are accessed by the algorithm for the first time and *finished* when all its neighbors are discovered. Intuitively, the DFS algorithm first starts with discovering a not-discovered vertex, and recursively discovering its undiscovered neighbor. When there is no undiscovered neighbor left, the DFS algorithm backtracks to the last vertex. See Algorithms 1 and 2.

DFS and recurrence. There is a time step *time* in the DFS algorithm. For every vertex v, the variable $d[v]$ is the time when this vertex is visited by the DFS algorithm for the first time, and the variable $f[v]$ is the time when all its neighbors are visited. Recall that DFS recursively visits an undiscovered neighbor of the current vertex. The variable $\pi[v]$ keeps the information that from which vertex v is visited by the algorithm. When all neighbors of v are visited, DFS backtracks to the vertex $\pi[v]$. Moreover, since DFS is a recurrence

Algorithm 1 Depth-first-search

```
for each u ∈ V do
    Mark u as not-discovered
    π[v] ← NIL
time ← 0
for each vertex u ∈ V do
    if u is not-discovered then DFS-VISIT(u)
```

Algorithm 2 `DFS-VISIT`(u)

```
Mark u as discovered
time ← time + 1
d[u] ← time
for each neighbor v of u do
    if v is not-discovered then
        π[v] ← u
        DFS-VISIT(v)
Mark u as finished
time ← time + 1
f[u] ← time
```

algorithm, there is a system stack. All discovered but not finished vertices are kept in the memory. More specifically, v enters the memory at the time step $d[v]$ and leaves the memory at the time step $f[v]$.

Time complexity. In DFS, each edge is visited at most twice (from each direction). A vertex is discovered once and finished once. Therefore, the time needed for DFS is $O(|V| + |E|)$, which is *linear* for graph algorithms.

2.1 Interesting properties of DFS

Parenthesis theorem. Recall that $d[v]$ is the time when v enters the memory, and $f[v]$ is the time when v leaves the memory. Due to the recurrence nature of DFS, for any two vertices u and v, the time intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ form a parenthesis shape. That is, either $[d[u], f[u]] \subset [d[v], f[v]]$, $[d[v], f[v]] \subset [d[u], f[u]]$, or $[d[u], f[u]] \cap [d[v], f[v]] = \phi$.

Theorem 1. (Parenthesis Theorem)
In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, v is a proper descendant of u in the depth-first forest if and only if $d[u] < d[v] < f[v] < f[u]$.

Edge classification. Recall the $\pi[v]$ variables in the DFS algorithm that keep the information from which vertex $\pi[v]$ v is discovered. The *predecessor subgraph* of a DFS is $G_\pi = (V, E_\pi)$, where $E_\pi = \{(\pi[v], v) \mid v \in V \text{ and } \pi[v] \neq \phi\}$.

Note that $G\pi$ is a forest. According to the G_π, each edge in E can be classified as one of the following four types:

- Tree edges: the edges in E_π
- Back edges: the edges that points from a descendant to an ancestor in the depth-first forest
- Forward edges: non-tree-edges pointing from an ancestor to a descendant in the depth-first forest
- Cross edges: all other edges

It is possible to identify the edge classes during the DFS. An edge (u, v) is a tree edge if at the moment it is checked (and u naturally is visited but not finished), the vertex v is not-discovered. If v is another visited but not finished vertex, (u, v) is an edge pointing from a descendent to an ancestor. That is, (u, v) is a back edge. If at the moment (u, v) is checked, v is a finished vertex, there are two cases: 1) v is in the same depth-first tree with u, or 2) u and v will not be in the same depth-tree. The case distinction can be done by checking if $f[v] > d[u]$. If $f[v] > d[u]$, there is an overlap between $[d[u], f[u]]$ and $[d[v], f[v]]$. By Theorem 1, v is a descendent of u. Therefore, (u, v) is a forward edge. Otherwise (that is, if $f[v] < d[u]$), (u, v) is a cross edge. Consider that we initially mark all vertices in white, mark a vertex in grey once it is visited, and mark a vertex in black when it is finished. The summary of the edge classes can be found in the following table. The color of a vertex indicates their states when the edge incident to it is checked.

Type of (u, v)	Relation between d[u], f[u], d[v], f[v]	Intervals [d[u], f[u]] and [d[v], f[v]]	Color of v at the time (u, v) is explored
Tree edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v
Back edge	d[v] < d[u] < f[u] < f[v]	[d[u], f[u]] [d[v], f[v]]	u → v
Forward edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v
Cross edge	d[v] < f[v] < d[u] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v

White-path theorem. In the white/grey/black color coding for the vertices, a tree edge in the depth-first forest points from a grey vertex to a white vertex. Naturally, if at the moment when u is discovered by DFS, there is a path from u to vertex v consisting entirely of white vertices, v will be a descendent of u in the depth-first forest. Surprisingly, the opposite is also correct. That is, if v is a descendant of u in the depth-first forest, there must be a white path from u to v at the time when u is discovered.

Theorem 2. (White-Path theorem) *In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and only if at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices.*

Proof. We first show that if vertex v is a descendant of vertex u, then at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices. Since v is a descendant of u, for any vertex w on the path from u to v in the DFS tree, $d[u] < d[w] < f[w] < f[u]$ (we assume that w is not u). That is, at the moment when u is discovered, w is not-discovered (white). It proves the claim.

For the opposite direction, we prove it by contradiction. Assume there is a white path starting from u. Suppose, on the contrary, that v is the first vertex on this white path which does not become a descendant of u in the depth-first tree. Let w be v's predecessor. Since w and u may be the same vertex, $f[w] \leq f[u]$. Since v is white at the time $d[u]$, $d[u] < d[v]$. Moreover, w cannot be marked as finished if v is not-finished. Therefore, $f[w] > f[v]$. Since every node is finished after being discovered, $d[u] < d[v] < f[v] < f[w] < f[u]$. By the parenthesis theorem, since $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$, v is a descendant of u. □

2.2 Cycle detection – A simple application using properties of DFS

The properties of DFS can be used to design graph algorithms. The following is an example that detects if there exists a directed cycle in a given directed graph.

Theorem 3. *Given a directed graph G, there is a* directed cycle *if and only if a DFS of G yields at least one back edge.*

Proof. We first show that if there is a back edge, then there is a directed cycle. If there is a back edge (u, v), v is an ancestor of u in a depth-first tree. That is, there is a path from v to u in the tree. The path, together with the edge (u, v), forms a cycle.

Next, we prove the opposite direction. Suppose there is a cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$ in G, where v is the first vertex discovered by DFS. That is, at time $d[v]$, all the other vertices in c are not-discovered. Therefore, there is a white-path from v to u. By the white-path theorem, u is a descendant of v in the depth-first tree. Hence, the edge (u, v) points from a descendant to an ancestor and is a back edge. □

According to this theorem, we can decide that there is a directed cycle as long as we find a back edge during the DFS. Therefore, we can answer this question in linear time.

3 Topological sort

One application of graphs in scheduling is to present tasks as vertices. There is an edge from u to v if the task corresponding to u needs to be finished before

the task corresponding to v. A reasonable set of tasks should have no directed cycle in the corresponding graph:

Definition 1. *A* directed acyclic graph (DAG) *is a directed graph that does not have any directed cycles.*

Given a DAG, a scheduler wants to find a "good" order of the tasks such that one can follow this order and finish the tasks without any trouble. The order is called a topological order.

Definition 2. *A* topological sort *of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that for any edge $(u, v) \in E$, u appears before v in the ordering.*

To find the topological order of a given DAG, one can apply the DFS algorithm and return the vertices in decreasing order of their finish time. Note that instead of sorting the vertices by their finish time/ after the DFS, we can put a vertex to the front of the list of finished vertices once it is finished. Therefore, the topological sort only takes $O(|V| + |E|)$ time. See Algorithm 3.

Algorithm 3 Topological-Sort(G)

Linked list $L_{\text{sort}} \leftarrow$ `NIL`
Call `DFS`(G)
As each vertex v is finished, put v to the front of L_{sort}
Return L_{sort}

In the following, we show that the algorithm correctly returns a topological order.

Theorem 4. *After the procedure* `Topological-Sort`*, for any two distinct vertices u and v, if there is an edge (u, v), in the resulting list, u appears before v.*

Proof. Equivalently, we show that for any edge $(u, v) \in E$, $f[v] < f[u]$. Since G is a DAG, there is no back edge in E. Therefore, for any edge (u, v), either v is a descendent of u or (u, v) is a cross edge. If v is a descendent of u, by the parenthesis theorem, $f[v] < f[u]$. Otherwise, if (u, v) is a cross edge, at the moment when it is checked, v is finished. Hence, $f[v] < f[u]$. □

4 Strongly connect components

Definition 3. *A* strongly connected component *of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, and are reachable from each other.*

To find the strongly connected components in a given directed graph, we need the concept of a transpose graph:

Definition 4. *A* transpose *of a directed graph* $G = (V, E)$ *is a directed graph* $G^T = (V, E^T)$*, where* $E^T = \{(u, v) \mid (v, u) \in E\}$.

Note that G^T can be constructed in $O(|V| + |E|)$ time. If there is a path from u to v in G, there is a path from v to u in G^T. For two vertices u and v in a strongly connected component, there is a path from u to v in G and a path from u to v in G^T.

Algorithm 4 `Strongly-Connected-Components`(G)

Call `DFS` and compute finish time $f[u]$ for each u
Compute G^T
while some vertex in G^T is not-discovered **do**
  $u \leftarrow$ not-discovered vertex with the latest $f[u]$
  Call `DFS-VISIT`(u) on G^T
The depth-first trees in `DFS`(G^T) are the strongly connected components.

Theorem 5. *The* `STRONGLY-CONNECTED-COMPONENTS`(G) *algorithm correctly computes the strongly connected components of a directed graph* G.

Proof. We prove this theorem by showing that the first trees produced by `DFS`(G^T) are strongly connected components. In the base case, where $k = 0$, the claim is true.

Assume that the first k trees are strongly connected components. We show that 1) for any vertex u in the $(k + 1)$-th connected component, it is in the $(k + 1)$-th tree, and 2) for any vertex w outside this tree, there is no path from any vertex v in the $(k + 1)$-st tree to w in G^T. First, by the inductive hypothesis, there are strongly connected components $C_1, C_2, \cdots, C_k$. Assume that the $(k + 1)$-th tree is reached by `DFS`(G^T) calling `DFS-VISIT`(v). Let C be the strongly connected component containing v. The vertex v has the largest finish time among all not-discovered vertices (vertices in $G^T \setminus \{C_1, C_2, \cdots, C_k\}$. Therefore, any vertices in C are not-discovered at time $d[v]$. By the White-path theorem, all these vertices are v's descendants. That is, The $(k + 1)$-th tree contains all vertices in C.

From any vertex u in $C_1, C_2, \cdots, C_k$, there is no path from u to any vertex in C (in G^T). That is, in G, for any $1 \leq i \leq k$, there is no path from C to C_i. Since v has the largest finishing time among all not-discovered vertices, in G^T, there is no edge from any vertex in C (or C_i) to any other not-discovered strongly connected components $C' \neq C_1, C_2, \cdots, C_k$. Hence, in G, there is no path from C' to C. □