Chapter 22
Basic Stuffs of Graph Algorithms

Graph can be regarded as a structure that generalizes tree, in the sense that, in a graph, there could be more than one paths from one node to another. For example,

In this chapter, we discuss the general concepts of the graph structure, its applications, and several practically useful algorithms, e.g., topological sort, shortest path, minimum spanning tree, and, if time allows, maximum flow.

We begin with a little theory on graphs.
Königsberg problem

In the town of Königsberg, the river Pregel flows around the Kneiphof and Island, which divides into two pieces. There are therefore four land areas, connected with seven bridges.

The Königsberg problem is to determine, when starting at one land area, e.g., $A$, if it is possible to walk across all the bridges exactly once and return to the starting area $A$.

We can certainly try to dig out a solution by walking around, and taking records 😊.

Question: Is there a “better” way?
Tired of walking?

Define a graph, $\mathcal{K}(V, E)$, as follows:

$$V = \{v | v \text{ is a piece of land.}\}$$

$$E = \{(v_1, v_2) | \exists \text{ a bridge between } v_1 \text{ and } v_2.\}$$

Thus, the Königsberg problem has a solution if and only if there exists a circuit (cycle) including every vertex in $V$ such that every edge in $E$ is included exactly once.

In general, if a graph has this property, it is called an Eulerian graph.

**Theorem** (Euler, 1736) A connected graph is Eulerian if and only if each and every vertex in the graph has an even degree.
A piece of cake

By $K_n$, we mean a graph with $n$ vertices such that between any two vertices, there exists exactly one edge.

We notice that there is just one edge in $K_2$, three in $K_3$, six in $K_4$, ten in $K_5$.

If we keep on counting, we will find out there are fifteen edges in $K_6$, and Twenty one in $K_7$.

You must know what we are going to do next. 😊
A natural question

**Question:** How many edges are there in $K_n, n \geq 1$?

Consider $u$, any of the $n$ vertices, there are $n - 1$ edges connecting it to the other $n - 1$ vertices, thus $n(n - 1)$ connections.

Since any of these edges $(u, v)$ is counted twice in the process: one from $u$ and the other from $v$, the total number of edges in $K_n$ is thus half of the connections, i.e., $n(n - 1)/2$. 
A piece of rock

Given six people, Ann, Bran, Charlie, David, Evelyn, and Fred, the following graph shows that while Ann, Bryn and David know each other, all connected in red; Ann, Charlie and Fred don’t know each other as they are all connected blue.

Question: How could we prove that, with six people, either there is a group of three know each other; or they don’t know each other?

Answer: Ask them.
It might not be easy?

**Question:** If we label each edge with either red or blue in a $K_6$, is there always either a red triangle or a blue one?

As there are $2^{15} (= 32,768)$ different ways to label $K_6$, it is not practical to try this out exhaustively. 😞

Consider Ann, one of the six vertices. There are exactly five edges connecting Ann with the other five vertices. An important observation is that at least three edges, $E_3$, will be labeled with the same color, either red or blue

**Question:** Why?

**Answer:** By DeMorgan’s law,

$$\neg[E_3 \lor E_3] = \neg E_3 \land \neg E_3.$$  

Then, at most two edges of each color will be incident to Ann, leading to a maximum of four edges incident to Ann, a contradiction to the fact that there are five such edges. 😞
A piece of cake again

Assume at least three blue edges are connecting Ann to these five vertices, e.g., $(Ann, Charlie)$, $(Ann, Evelyn)$ and $(Ann, Fred)$.

We now consider the edges connecting Charlie, Evelyn, and Fred. Since $(Charlie, Fred)$ is blue, we have a blue triangle $(Ann, Charlie, Fred, Ann)$, showing they don’t know each other. This result holds if any of such an edge is blue.

Otherwise, if none of these edges is blue, they must be all red. Then, we would have a red triangle, i.e., they would know each other.

In general, out of any group of six people, there must be three who are either mutual friends, or total strangers.

This is the special case of a very rich, and challenging, Ramsey theory.

Frank P. Ramsey passed away at 26 in 1930. 😞
The four-color problem

In map making, to distinguish the different regions, it makes sense to use different colors to color adjacent regions.

Obviously at least three colors are needed and it can also been shown that three colors are not sufficient, but five colors are.

It was conjectured that four colors are sufficient, as well. This *four-color problem* remained unsolved for over 100 years.

In 1976, Appel and Haken proved that this conjecture is true by checking out 1,936 maps, *using a computer 😊*
Traveling Salesman Problem

Holding a map, a salesman wants to schedule a trip, in which he can visit all the towns without any repetition, which also minimizes the total distance.

The question is whether this is possible, and if it is, how to schedule this trip.

Define a graph, $S(V, E, \omega)$, as follows, where $\omega$ stands for weight assigned to edges, e.g., distances:

$$
V = \{v|v \text{ is a town.}\}
$$

$$
E = \{(v_1, v_2, w)|d(v_1, v_2) = w\}
$$
Thus solution of the *traveling salesman problem* is a minimum circuit such that it includes every vertex in the graph $S$ exactly once, 25 for the previous one.

In general, if a graph, where *every edge has the same weight*, has this property, it is called an *Hamiltonian graph*.

Even though the weaker Hamiltonian problem looks quite similar to the Königsberg problem (?), there does not exist a simple, useful and elegant solution: Every known algorithm for this problem is $O(2^n)$. 😞

On the other hand, no body has proved that this problem is $\Omega(2^n)$. 😔

This is a major unsolved problem in graph theory, and a typical example of the NP-Complete problems, which we will discuss at the end of this semester, *if we have time*. 😊
Graph terminology

**Definition 1:** A *graph*, $G$, is an ordered pair $(V, E)$. $V$ is a set of vertices and $E$ is a set of edges. $E = \{(u, v) | u, v \in V\}$.

If for all $(u, v) \in E$, $(u, v)$ is an ordered pair, then $G$ is a *directed graph*, or *digraph*; otherwise, it is a *undirected graph*, or *graph*.

Vertices $i$ and $j$ are *adjacent*, if $(i, j) \in E$. The edge $(i, j)$ is also *incident* to vertices $i$ and $j$.

The number of vertices incident to a vertex, $v$, is defined to be the *degree* of that vertex, denoted as $d(v)$.

Recall that we used the notion of *degree* in giving a result for the Eulerian graph on page 3.
A weighted graph is a triplet, \((V, E, \omega)\). Here \(\omega\) is the weight function. For every edge, \(e \in E\), \(\omega(e)\) is the weight assigned to the edge \(e\).

For example, the following map is a weighted graph.

![Graph with weights](image)

**Definition 2:** Given \(G(V, E)\). By saying that \(G\) is complete, we mean that \(\forall u, v \in V, (u, v) \in E\).

Recall that a complete graph with \(n\) nodes has \(\frac{n(n-1)}{2}\) edges (Cf. Page 5).
Definition 3: By saying that $G_1(V_1, E_1)$ is a subgraph of $G_2(V_2, E_2)$, we mean $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. For example, the following graph is a subgraph of the campus graph we saw on Page 1.

![Graph](image)

A subgraph of $G$ that contains all the vertices of $G$ is called a spanning graph of $G$. If such a subgraph is also a tree, it will then be called a spanning tree.

Minimum spanning tree is associated with the least expensive way to connect a bunch of sites, which we will study in the next unit.
**Definition 4:** Given a graph, $G(V, E)$. By saying that $p (= v_1, \ldots, v_n)$ is a *path* in $G$, we mean that, for all $i \in [1, n)$, $(v_i, v_{i+1}) \in E$. A *simple path* is a path in which all the vertices, except the first and the last, are different.

The *length* of $p$ is defined as $n - 1$. If $p$ is empty, its length is defined as 0.

For every edge in a (di)graph, we can associate it with a *length*. The length of a path is then defined to be the sum of the lengths of all the edges in that path.

We are often interested in finding out a *shortest path* from one vertex to another, e.g., the *GPS* app.

We will study this important problem in a later unit.
Definition 5: Given \( G(V, E) \), a digraph. By a cycle, we mean a path in \( G \) with at least one edge and \( w_1 = w_n \). Call it a simple circle, if the path is simple.

A digraph is acyclic, called a dag, if it has no cycles, which we met on Page 39 of the Parallel programming chapter.

Definition 6. Given \( G(V, E) \), a graph. By saying that \( G \) is connected, we mean that for all \( u \) and \( v \in V \), there is a path between \( u \) and \( v \). Otherwise, \( G \) is disconnected.

We are usually interested in connected graph, in studying, e.g., a computer network.

If \( G \) is a digraph and it satisfies the above property, we say it is strongly connected.

If a digraph is not strongly connected, but its underlying graph is connected, we say that the digraph is weakly connected.
Application

Airport system, $A(V, E, \omega)$, can be represented as a weighted digraph.

\( V = \{a | a \text{ is an airport.} \} \)

\( E = \{(a_1, a_2) | \exists \text{ non-stop flight from } a_1 \text{ to } a_2.\} \)

For \( e \in E \), \( \omega(e) \) could be cost, time, etc.

To have an effective transportation system, we may expect a strongly connected digraph, meaning, there is a route from any place to any other place.

To have an efficient system, we may want to find out the “best” travel plan, e.g., the shortest path from one city to another, the plan with an overall cheapest cost, the one will take the least amount of fuel to complete, etc., etc., etc.
Basic properties of a tree

A tree is a graph which is both connected and acyclic. We have looked at the representation of a rooted tree in the BST chapter. In general, we have the following:

**Definition 7:** A tree is a connected graph containing no cycles.

We later need the following basic results.

**Lemma:** Let $T$ be a tree. Then it has at least two leaves.

**Proof:** By construction and contradiction, both $u$ and $v$ must be leaves.
Another property of a tree

**Theorem:** Let $T$ be a finite tree containing $n$ vertices. Then it has $n - 1$ edge.

**Proof:** By construction and induction. When a tree holds just one vertex, it has 0 edges.

Assume this holds for a tree with $n$ vertices. Let $T$ be a tree with $n + 1$ vertices, and let $v$ be a leaf, adjacent to $u$.

Let $T'$ be a tree after removing $v$ and the edge $(u, v)$. Then, $T'$ has $n$ vertices, by the inductive assumption, it has $n - 1$ edges.

Thus, $T$ has $n$ edges, including $(u, v)$. 
Why trees?

It is known that there are $n^{n-2}$ unrooted trees with $n$ labeled nodes; and, $n^{n-1}$ such rooted trees. Here is the case of $n = 3$.

A forest is just a collection of trees, which are not connected to each other.

Let $F(n)$ be the total number of forests with $n$ labeled nodes, which could contain at least one tree, then it has also be found out that

$$\lim_{n \to \infty} \frac{n^{n-2}}{F(n)} = e^{-1/2} \approx 0.6065.$$ 

In other words, about 60% of random forests with a large number of nodes would be trees, i.e., connected. Thus, tree is an important class of graphs.

Check out BST notes for other basic properties of trees, including various traversal operations.
Properties of a graph

Property 1: Let $G(V, E)$ be a graph. Let $|V| = n, |E| = e$, and $d_i$ denotes the degree of $v_i$. Then

$$
\sum_{i=1}^{n} d_i = 2e, \text{ and, (1)}
$$

$$
0 \leq e \leq \frac{n(n - 1)}{2}. \text{ (2)}
$$

Proof: Eq. 1 is true since every edge is incident to exactly two vertices, hence contributing 2 to the sum. We will use this result later on.

There are five edges here, and a degree sum of 10.
Properties of a digraph

Let $G$ be a digraph, $d_i^{\text{in}}$, the \textit{in-degree} of a vertex, $v$, is defined to be the number of edges coming into $v$. We have a similar definition of \textit{out-degree} of $v$.

\textit{Property 2:} Let $G(V,E)$ be a digraph. Let $|V| = n, |E| = e$, and $d_i$ denote the degree of $v_i$. Then

\[
\sum_{i=1}^{n} d_i^{\text{in}} = \sum_{i=1}^{n} d_i^{\text{out}} = |E|, \text{ and,} \quad (3)
\]

\[
0 \leq e \leq n(n - 1). \quad (4)
\]

The proofs are similar to the previous one. Check out Page 23 for an example.

Note that, in a directed graph, the edge $(u, v)$ is different from $(v, u)$. Just think of the one-way streets in Boston (Cf. Course page). 😊
Graph representation

We have to represent a graph so that a computer can work on it.

The adjacency matrix of a graph $G(V, E)$, $|V| = n$, is an $n \times n$ matrix:

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

Such a matrix must be symmetric, i.e., for all $i, j$, $A(i, j) = A(j, i) = d_i = d_j$

Similarly, the adjacency matrix of a digraph $G(V, E)$, $|V| = n$, is an $n \times n$ matrix:

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

It is also easy to see that

$$\sum_{j=1}^{n} A(i, j) = d_i^{\text{out}}, \text{ and,}$$

$$\sum_{i=1}^{n} A(i, j) = d_j^{\text{in}}.$$
Adjacency list

Adjacency matrix is easy to use, but it uses $\Theta(n^2)$ space, which is not space efficient, except for a rather dense graph, such as a complete graph, a rare situation in practice.

An alternative approach is to use a linked list to represent, for every vertex, all the vertices that is adjacent.

If $|v| = n, |E| = e$, such a representation costs $\Theta(n + m)$, where $m = 2e$, when $G$ is a graph, $m = e$ otherwise. For example, $m = e = 8$ in the following structure.

Homework: Exercises 22.1-1, 22.1-2, and 22.1-5.
Search in graphs

Remember PreOrder, InOrder, and PostOrder traversal for a tree? (Cf. Page 5 in the BST notes) There are two primary ways to systematically “visit” all the vertices in a graph: Breadth-First, and Depth-First search; often called BFS and DFS.

With BFS, once we have visited a vertex, we will visit all its neighbors before moving on. We will accomplish this by keeping the adjacent vertices that have not been visited in a queue.

With DFS, once we have visited a vertex, we keep on “digging down”, while visiting all the vertices along the way. When we hit the bottom, if there are still unvisited vertices left, we come back, called “backtrack”, along the vertices that we have already visited until we come to a vertex that is not. We then visit this latter vertex, and dig down from there.

Are we surprised that a stack sits behind DFS?
I want to see...

When applying BFS to the following graph, but not a tree, starting with 1, we visit both of its neighbors 2 and 3.

We continue with the neighbor of 2, e.g., 4, 5, and 6; then visit neighbors of 3, only 7, since 6 is already visited. We proceed with the neighbors of 4, and 5, e.g., 8, 9, 10, and 11. Finally, we will visit neighbors of 6, and 7, e.g., 12, 13, 14, and 15.

So, one BFS sequence could be 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.
How about DFS?

Give the same graph, starting with the top node 1, with DFS, we dig all the way down, 2, 4, until hitting 8.

Since there is nothing underneath, we backtrack to 4, the parent of 8; and, after visiting both 9, and 10, we backtrack to 2, and visit its other child 5, and then 11; then back track to 2, continue with 6, then 12, 13, and 14.

We will then backtrack to 6, 2, and finally 1, and start to dig down on its other side. We start with 3, and, since we already visited 6, we visit 7, and finally wrap up with 15.
Breadth-first search

Given a graph $G(V, E)$, and a source vertex $s$, BFS systematically explores the edges of $G$ to visit every vertex that is reachable from $s$.

When running BFS on a graph, all the vertices at distance $k$ away from $s$ will be discovered before those $k + 1$ away. Thus, when applied to a tree, BFS leads to a level order.

As a by-product, it computes the distance, i.e., the length of a shortest path, from $s$ to each reachable vertex. It also produces a “breadth-first tree” with root $s$ and all the vertices reachable from $s$.

As a result, for every vertex $v$ reachable from $s$, the path in this tree from $s$ to $v$ corresponds to a “shortest path”, with fewest number of edges, from $s$ to $v$ in $G$.

BFS is not useful in finding shortest path in reality, where edges come with different weights.
Black, gray or white?

The breadth-first search algorithm assumes an adjacency list representation (Cf. Page 24) of a graph $G$, and uses color to distinguish nodes during the discovery process.

A node is colored white when it is not discovered; gray when it is discovered by its parent; and black, when all of its children have been discovered.

BFS constructs a tree, initially containing only $s$, whenever a white $v$ is discovered during scanning of an already discovered vertex $u$, both $v$ and $(u, v)$ are added to the tree, and $v$ turns into gray. We say that $u$ is an ancestor of $v$ if $u$ occurs in a path from $s$ to $v$.

We can use a queue to support the implementation of the BFS algorithm... .
Why do you mean?

BFS(G, s)
1. for each u in V-{s} //Get it started...
2. do color[u]<-White //Not seen yet
3. d[u]<-MAXINT // What is d(s, u)?
4. p[u]<-NIL //How to get to u?
5. color[s]<-Gray //Just saw s
6. d[s]<-0 // d(s, s)=0
7. p[s]<-NIL // P(s, s) is empty
8. Initialize(Q) //Get the queue started
9. Enqueue(Q, s) //Throw in s
10. while(!IsEmpty(Q)) //Not done yet
11. do u<-DeQueue(Q) //Take out the next one
12. for each v in Adj[u]
13. do if color[v]=White //Not seen yet
14. then color[v]<-Gray //We now see it
15. d[v]<-d[u]+1 //d(s, v)=d(s u)+1
16. p[v]<-u //Get into v from u
17. EnQueue(Q, v) //Work with v later
18. color[u]<-Black //Done with u
I want to see...

Below shows an execution of the BFS algorithm.

Technically, we only need to tell whether a vertex is white. Once it is changed, it cannot be changed back to “white” again, as its value is fixed.
Algorithm analysis

We notice that a node can only be colored white during the initialization only. Thus, the test in line 13 guarantees that each node can be enqueued only once in Lines 9, and 17; and dequeued only once in line 11. As a result, the queue related operations takes $O(|V|)$ as done in line 10.

Regarding the for loop in line 12, since the adjacency list is scanned only when a node is dequeued, each such list is scanned at most once.

Since the sum of the length of all such lists is $\sum_u d(u) = \Theta(E)$ (Cf. Pages 21 and 24 of this set of notes), the time spent there is $O(|E|)$.

Add on the initialization cost of $\Theta(|V|)$, in Lines 1 through 9, we conclude that the total time of BFS is $O(|V| + |E|)$.

**Homework:** Exercise 22.2-1.
Depth-first search

The strategy of the depth-first search is just the opposite: it looks for nodes in the graph as “deep” as possible.

More specifically, edges are explored out of the most recently discovered vertex \( v \) that still has unexplored edges out of \( v \). When all the edges that exit \( v \) are explored, the algorithm “backtracks” to the parent of \( v \) to continue exploration from there.

When the exploration eventually ends at the starting vertex, the search is over for that component, i.e., the maximum connected subgraph of \( G \). And it may continue with another component, if there are still unexplored vertices.

When applied to a tree, \( DFS \) leads to a Pre-Order traversal... .
The DFS algorithm

DFS(G)
1. for each u in V
2. do color[u]<-White // Get started
3. p[u]<-NIL // What is the parent of u?
4. time<-0 // Start at the beginning
5. for each u in V
6. do if color[u]=White //If it is new
7. then DFS-Visit(u) //Work with it

DFS-Visit(u)
1. color[u]=Gray // We have seen you
2. time<-time+1 // update time
3. d[u]<-time // Got grayed at this updated time
4. for each v in Adj[u] //Check out its neighbors
5. do if color[v]=White // Never saw you before
6. then p(v) <- u //Got to v through u
7. DFS-Visit(v) //Keep on digging...
8. color[u]<-Black //Done with u
9. f[u]<-time<-time+1 //Stamp u with black
I want to see...

Below shows an execution of the DFS algorithm.

**Assignment:** Apply the DFS algorithm on the above graph to verify the process and the result.

You need to completely understand this stuff to do Project 8.
DFS structure

When BFS terminates on a connected graph $G$, we end up with a tree, consisting of all the vertices together with all the edges from a vertex $v$ to its parent $p[v]$.

On the other hand, DFS, when applied to a graph, the resulting predecessor subgraph could form a DFS forest consisting of several trees, each for a component of $G$. The edges collected in these trees are called tree edges.

Instead of giving a distance, the DFS algorithm timestamps each vertex during its execution.

For every $v \in V$, the first timestamp $d[v]$ tells when $v$ is first discovered (grayed); and the second timestamp $f[v]$ tells when the search finishes checking through all the other vertices adjacent to $v$, when it is colored black.

As we always discover first, finish later, for every $u \in V$, $d(u) < f(u)$. 
Algorithm analysis

We notice that the first for loop in DFS(G) takes $\Theta(|V|)$. The procedure DFS-Visit is called only for the white nodes, and, a node can only be colored white during the initialization, and as soon as DFS-Visit is called for such a vertex, $u$, it is colored gray, thus, DFS-Visit is called once for each vertex, $\Theta(|V|)$ in total.

During DFS-Visit($u$), each vertex in $\text{Adj}(u)$ is visited only once in Line 4. Again, since the sum of the length of all such lists is $\Theta(E)$, (Eq. 1 on Page 21) the time spent on the for loop in Line 5 in the DFS(G) procedure is $O(|E|)$.

We thus conclude that the total time of DFS is $O(|V| + |E|)$, linear in the size of the graph.

**Homework:** Exercise 22.3-2.
An interesting property

**Theorem 22.7.** In any DFS search of a graph $G = (V, E)$, for any vertices $u$ and $v$, exactly one of the following holds:


Notice that these are the only three possible cases between $u$ and $v$.

Check Page 35 for examples. For Case 1, $(y, w)$; for Case 2, $(u, v)$; and for Case 3, $(x, y)$. 

38
What is the point?


This property is called \textit{parenthesis property} since if we represent \( d[u], f[u], d[v], \) and \( f[v] \) with (, ), [ and ], respectively, the above result shows that DFS only admits either ()[], [](), ([]), or [[]], but never ([)] or [[]]). 😊

As you will see in \textit{CS 3780}, a \textit{context-free grammar} to generate such a balanced structure is:

\[
\begin{align*}
A & \rightarrow () \\
A & \rightarrow [] \\
A & \rightarrow (A) \\
A & \rightarrow [A] \\
A & \rightarrow AA
\end{align*}
\]

\textbf{Homework:} Exercise 22.3-3.
Topological sort

Given an acyclic digraph, $G(V, E)$, by a topological sort of $G$ on $V$, we mean a linear ordering of all the vertices in $V$ such that, if there is a directed path from $u$ to $v$, then $u$ precedes $v$ in the ordering. Sequences that satisfy this property are topological orders, or topological sequences.

As the sample in the book shows, you have to put on your shirt first before putting on your jacket.

A topological ordering of all the courses as required by a program leads to a sequence that students may take that won’t violate the prerequisite requirement.

Another example is the assembly task, in which some parts must be put together first, before other tasks can be performed.
A couple of notes

1. If the given digraph is not acyclic, then no topological sort exists.(?)

2. There could be more than one such sequences for a given acyclic graph. For example, $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ are both topological sort sequences for the following graph.

There are multiple ways of finishing the Computer Science degree here at Plymouth State.
A simple algorithm

Given a digraph $G(V, E)$, and $v \in V$, by an edge coming into $v$, we mean any edge $(u, v) \in E$. By an edge outgoing from $v$, we mean any edge $(v, w) \in E$.

The following is an algorithm looking for a topological sorting in $G$.

1. Set $n$ to 1
2. While not all vertices labeled yet
3. Find $v$, s.t. there is no edge coming into $v$;
4. Label $v$ with $n$;
5. $n \leftarrow n + 1$
6. Remove $v$ together with all the edges outgoing from $v$;

**Homework:** Exercise 22.4-1.
I want to see...

Below shows one sequence: 1, 2, 5, 4, 3, 7, 6, with the other being 1, 2, 5, 4, 7, 3, 6.
Correctness

**Lemma** If $G$ is a finite acyclic digraph, there exists a vertex in $G$ such that no edge comes into it.

**Proof:** Just assume it is false, i.e., for every vertex $v \in V(G)$, there is an edge $(u, v)$, which comes into $v$.

Let $v_1$ be such a vertex. Then, for some $v_2$, $(v_2, v_1) \in E$. Similarly, for $v_3, \ldots, v_n, \ldots$, we have that $(v_{i+1}, v_i) \in E$.

As $G$ is finite, some vertex that we see in the process must have already occurred. In other words, for some $1 \leq i < j \leq n$, $v_j = v_i$.

Thus, $G$ contains a cycle $(v_i, v_{i+1}, \ldots, v_j(= v_i))$, which contradicts the acyclic assumption. 😊
Main result

**Theorem:** Let $G(V, E)$ be a finite and acyclic digraph. The above algorithm generates a topological sorting for $V$.

**Proof:** Let $v_i$ and $v_j$ be two vertices such that there is a path from $v_i$ to $v_j$. We show that $v_j$ cannot be labeled unless $v_i$ gets labeled first.

We prove the above by inducting on the length of the path from $v_i$ to $v_j$.

If the length is 1, i.e., $(v_i, v_j) \in E$.

Then $v_j$ gets labeled at $t_3$ when no edge comes into $v_j$, i.e., after all such incoming edges have been removed, particularly $(v_i, v_j)$ was removed at $t_2$.

But this removal of $(v_i, v_j)$ and $v_i$ occur after the labeling of $v_i$ at $t_1$. 
The final kick

In general, let $v_i, v_{i+1}, \ldots, v_j$ be a path of $G$. The same argument shows that $v_{i+1}$ will be labeled at $t_2$ after the labeling of $v_i$ at $t_1$. Also, inductive assumption shows that $v_j$ gets labeled, at $t_3$, after $v_{i+1}$ gets labeled.

Thus, $v_j$ is labeled after $v_i$ gets labeled first.

Since labels get increasingly larger, $l(v_i) \leq l(v_j)$, $1 \leq i \leq j \leq |V(G)|$.

We are done. 😊
Implementation

Let $G(V, E)$ be a dag, and let $v \in V$, recall that the in-degree of $v$, $d_{in}(v)$, is defined as the number of edges coming into $v$. Check out Page 22...

Notice $d_{in}(v), v \in V$, can be found out in $\Theta(|E|)$ (How?). Once this information is available, the following algorithm kicks in.

```c
void Toposort(G: graph){
  for(c=1; c<=|V|; c++){
    find v such that its indegree is 0;
    if (v==0) error; // Not acyclic
    else {
      top_num[v]=c;
      for each w adjacent to v do
        //Won’t cut it off
        indegree[w]=indegree[w]-1;
    }
  }
}
```

It is clear that the whole algorithm takes $O(|E|)$. 
I want to see...

Below shows one sequence: 1, 2, 5, 4, 3, 7, 6, with the other being 1, 2, 5, 4, 7, 3, 6.