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 predecessors, and successors, for each node (vertex), connected with edges. For example,

In this chapter, we discuss the general concepts of the graph structure, and several practically useful algorithms.

We begin with a bit graph theory.
Königsberg Problem

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

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

We can certainly keep on walking to explore the possibility… .

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

Leonhard Euler (1707-1783) had a different idea, when he defined a graph, $\mathcal{K}(V,E)$, as follows:

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

Now, 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 iff each and every vertex in the graph has a 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.

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

**A shorter answer:** $\frac{n(n-1)}{2}$.

**A longer one:** Consider any of the $n$ vertices, there are $n - 1$ edges connecting such a vertex to the other $n - 1$ vertices. Thus, we have $n(n - 1)$ edges in total.

But any of these edges $(u, v)$ is counted twice in the process: one from $u$ and the other from $v$. So, we just take half of them.
A piece of rock?

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

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

Consider $v_1$, one of the six vertices. There are exactly five edges connecting $v_1$ with the other five. An important observation is that at least three edges will be labeled with the same color, either red or green(?).

**Answer:** Just assume the opposite. Then, at most two edges of the same color will be incident to $v_1$, a maximum of four. 😞
A piece of cake again

Now, assume at least three red edges are connecting $v_1$ to, e.g., $v_2, v_3$ and $v_4$.

We now consider the edges connecting $v_2, v_3$ and $v_4$. If any of the three edges is red, say $(v_2, v_3)$, we have a red triangle $(v_1, v_2, v_3)$.

Otherwise, all of the three edges must be green, we now have a green triangle $(v_2, v_3, v_4)$.

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

Check out the course page for a similar “Party problem”: How many people should you invite so that it is guaranteed either at least three are mutual friends, or they don’t know each other.
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 the *traveling salesman problem* has a solution iff there exists a circuit such that it includes every vertex in $S$ exactly once, when we also want to achieve the *minimum cycle*.

In general, such a graph, where every edge has the same weight, is called an *Hamiltonian graph*.

Even though the salesman problem looks quite similar to the Königsberg problem, there does not exist a simple, useful and elegant solution. Also, every known algorithm for this problem is $O(2^n)$.

This is a major unsolved problem in graph theory, and a typical example of the NP-Complete problems. 😊
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 $d(v)$. We used this notion when discussing the Königsberg problem.
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.

![Weighted graph example](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 4)
Intuitively, a subgraph is a part of a graph.

**Definition 3:** By saying that $G_1$ is a subgraph of $G_2$, 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 before.

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**, 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 weight, e.g., the distance in between the two locations as represented by the two endpoints of such an edge.

The weight of a path is then defined to be the sum of the weights of all the edges in that path.

We are often interested in finding out the shortest path from one vertex to another. 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 $v_1 = v_n$. Call it a *simple circle*, if the path is simple.

A digraph is *acyclic* if it has no cycles.

**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*.

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*.

A computer network, when represented as a graph, has to be connected to be effective.
Something about a tree

We have looked at the representation of *rooted* tree in the Binary search tree 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.

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

**Proof:** By construction and induction.
Application

Airport system can be represented as a weighted digraph.

\[ A = (V, E, \omega) \]
\[ V = \{ a | a \text{ is an airport. } \} \]
\[ E = \{ (a_1, a_2) | \exists \text{ non-stop flight from } a_1 \text{ to } a_2. \} \]
For \( e \in E \), \( \omega(e) \) could be cost, time, . . . .

To have an effective transportation system, we may expect a strongly connected digraph. This is the basis of a useful transportation system.

To have an efficient system, we may want to find out the “best” travel plan, e.g., the shortest path from one town to another. This is the base of such apps as GPS and Waze.
Properties

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

\[ \sum_{i=1}^{n} d_i = 2e, \text{ and}, \quad (1) \]
\[ 0 \leq e \leq \frac{n(n-1)}{2}. \quad (2) \]

Proof: Eq. 1 is true since every edge is incident to exactly two vertices, hence contributing 2 to the sum.

Obviously, when a graph is empty, $e = 0$, this the lower bound. We have discussed the upper bound earlier. (Cf. Page 4)
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 $v$.

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

$$\sum_{i=1}^{n} d_i^{\text{in}} = \sum_{i=1}^{n} d_i^{\text{out}}, \text{ and,} \quad (3)$$

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

The proofs are similar to the previous one.

Note in the directed graph case, the edge $(u, v)$ is different from $(v, u)$. 
Graph representation

Usually, a graph can be represented as either an adjacency matrix, or an adjacency list.

The adjacency matrix of an $n$–vertex graph, $G$, is an $n \times n$ matrix, such that,

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

It is easy to see, such a matrix must be symmetric.

Similarly, the adjacency matrix of an $n$–vertex digraph, $G$, is an $n \times n$ matrix, such that,

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

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 complete graph, a rare situation in practice.

An alternative approach is to use a linked list to represent, for every vertex, all the vertices it is adjacent. If $|v| = n, |E| = e$, such a representation uses $\Theta(n + m)$, where $m = 2e$, when $G$ is a graph, $m = e$ when $G$ is directed.

Homework: Exercises 22.1-1(*), 22.1-2(*), 22.1-3, and 22.1-5(*).
Search in graphs

There are two primary graph traversal schemes: *Depth-First* and *Breadth-First Traversals*.

In *BFS*, every vertex adjacent to the current vertex is visited first before we move away from it. We will accomplish this by keeping the adjacent vertices that have not been visited in a *queue*.

In *DFS*, we mark a vertex as soon as it is visited and then try to move to an unmarked adjacent vertex. If there are no unmarked vertices left, we backtrack along the vertices we have already visited until we come to a vertex that is adjacent to an unvisited one, visit this latter vertex, and continue the process.

Since DFS is often done recursively, there is a *stack* behind *DFS*. 
Breadth-first search

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

As a byproduct, it computes the distance from $s$ to each reachable vertex. It also produces a “breach-first tree” with root $s$ that contains all the reachable vertices. 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 $G$.

When running BFS on a graph, all the vertices at distance $k$ away from $s$ will be discovered before those that are $k+1$ away from the starting vertex.
Black, gray or white?

The breadth-first search algorithm assumes an adjacency list representation of a graph $G$, and uses color to distinguish nodes during the discovery process. A node is colored *white* when it is not discovered by the algorithm; *gray* when it is discovered, but not labeled; *black*, when it is finally labeled by the algorithm.

BFS constructs a tree, initially containing only $s$, whenever a white vertex $v$ is discovered during scanning of an already discovered vertex $u$, both $v$ and $(u, v)$ is 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 as follows.
An example

Below shows an execution of the BFS algorithm.
How to do it?

BFS(G, s)
1. for each u in V-{s}
2. do color[u]<-White //No idea
3. d[u]<-MAXINT
4. p[u]<-NIL
5. color[s]<-Gray //Seen it
6. d[s]<-0 //Empty path
7. p[s]<-NIL //s is the first
8. Initialize(Q)
9. Enqueue(Q, s)//Start with s. Step a is done.
10. while(!IsEmpty(Q)) //Not done yet
11. do u<-DeQueue(Q) //Take out u
12. for each v in Adj[u]
13. do if color[v]=White //Not seen v yet
14. then color[v]<-Gray
15. d[v]<-d[u]+1 //One edge away from u
16. p[v]<-u //Get to v via u
17. EnQueue(Q, v) //Put in v
18. color[u]<-Black //Done with u in Step b.
Algorithm analysis

We notice that a node can only be colored "white" during the initialization, thus, the test in line 13 guarantees that each node can be enqueued only once, thus dequeued only once. As a result, the queue related operations takes $O(|V|)$.

Regarding 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 $\sum_v d(v) = \Theta(E)$ (Page 17), the time spent there is $O(|E|)$.

Add on the initialization cost of $\Theta(|V|)$, we conclude that the total time of BFS is $O(|V| + |E|)$, linear in terms of the size of the graph.

**Homework:** Exercises 22.2-1(*), 22.2-2 and 22.2-3(*).
Correctness of BFS

Let $\delta(u, v)$ stand for the distance between $u$ and $v$ in a graph $G$, i.e., the length of a shortest path between those two vertices, a key property of BFS is the following:

**Theorem 22.5.** Let $G = (V, E)$ be a digraph or a graph, and suppose BFS is run on $G$ starting with $s \in V$. Then, during its execution, BFS discovers every $v \in V$ that is reachable from $s$, and upon its termination,

$$d[v] = \delta(s, v).$$

Moreover, for every vertex $v(\neq s)$ that is reachable from $s$, one of the shortest paths from $s$ to $v$ starts from $s$ to $p[v]$, followed by the edge $(p[v], v)$. 

27
BFS tree

The BFS procedure actually builds a tree. We can define the resulted *predecessor subgraph* by applying a search algorithm on $G(V, E)$ as $G_p(V_p, E_p)$, where

$$V_p = \{v \in V | p[v] \neq NIL\} \cup \{s\} \subseteq V,$$

and

$$E_p = \{(p[v], v) | v \in V_p - \{s\}\}.$$ 

Such a subgraph is a **BFS tree** if $V_p$ consists of all the vertices reachable from $s$ and for all $v \in V_p$, there is a unique path in $G_p$ from $s$ to $v$ that is also a shortest path from $s$ to $v$ in $G$.

**Lemma 22.6.** When applied to a graph $G = (V, E)$, BFS constructs the predecessor subgraph $G_p$ that is a BFS tree for $G$. 

28
Depth-first search

The strategy of the depth-first search is to look 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 there.

When the backtrack and the following exploration ends at the starting vertex, the search is over for this component, and will continue if there is something left.

Sounds complicated? 😞 Let’s check it out with a tree, again 😊.
DFS structure

As we saw earlier, 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 search, when applied to a graph, the resulting predecessor subgraph forms a DFS forest, consisting of several trees. The edges collected in these trees are called tree edges.

The DFS algorithm also 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 one, $f[v]$, tells when the search finishes checking through all the other vertices adjacent to $v$, when it is colored black.

We have that, for every $u \in V$, $d(u) < f(u)$. 
An example

Below shows an execution of the DFS algorithm.
The DFS algorithm

DFS(G)
1. for each u in V
2. do color[u]<-White //Not seen yet
3. p[u]<-NIL //No parent found yet
4. time<-0 //Ready to start
5. for each u in V
6. do if color[u]=White //Not seen yet
7. then DFS-Visit(u)

DFS-Visit(u)
1. color[u]=Gray //Seen it...
2. time<-time+1 //Start now
3. d[u]<-time // Step a
4. for each v in Adj[u]
5. do if color[v]=White //Not seen yet
6. then p(v)=u //Step b
7. DFS-Visit(v) //Keep on digging
8. color[u]<-Black //Done with u
9. f[u]<-time<-time+1 //Leaving... Step j
Algorithm analysis

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

During \( \text{DFS-Visit}(u) \), \( \text{Adj}(u) \) is done only once. Again, since the sum of the length of all such lists is \( \Theta(E) \), the time spent during lines 4-7 in the \( \text{DFS-Visit} \) procedure, collective for all the vertices (Lines 5-7 of the DFS) is \( O(|E|) \).

Add on the initialization cost of \( \Theta(|V|) \), the total time of DFS is \( O(|V| + |E|) \), linear in the size of the graph.

**Homework:** Exercises 22.3-1(*) and 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:


- If $v$ is a descendant of $u$, then $d[u] < d[v] < f[v] < f[u]$.

- If $u$ is a descendant of $v$, then $d[v] < d[u] < f[u] < f[v]$.
What is the point?


This property is called parenthesis property since if we represent $d[u], f[u], d[v]$, and $f[v]$ with (, ), [ and ], respectively, the above results show that DFS only admits either ()[], [](), (][), or [()], but never (][).

Other application of such a balanced language include the composition of arithmetic expressions and program statements.

You will learn in CS3780 Intro to Computational theory how to use Context-free grammar, and Push-down automata, to automatically generate, and accept, this important class of languages.

**Homework:** Exercises 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 (sequence) of all the vertices in $V$ such that if there is a directed path from $u$ to $v$, then $u$ precedes $v$ in the sequence. Sequences that satisfy this property are topological orders, or topological sequences.

A topological ordering of all the courses leads to a sequence which respects the prerequisite requirement. (Check out the CS course requirement on the course page)

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

Do we have to put on socks before shoes?
A couple of notes

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

   Just assume that a cycle contains $u$ and $v$. Since a path goes from $u$ to $v$, in such a sort, $u$ must proceed $v$. by the same token, in the same sort, $v$ proceeds $u$. 😉

2. There could be multiple 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 for the following graph.
A simple algorithm

Given a digraph $G$ 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. no edge coming into $v$;
4. Label $v$ with $n$;
5. $n<-n+1$
6. Retract $v$ and all the edges outgoing from $v$;

The only way to understand an algorithm is...

Homework: Exercise 22.4-1(*).
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 that all the vertices having edges coming into themselves.

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, for $i \geq 2$, $(v_{i+1}, v_i) \in E$.

As $G$ is finite, so is $|V| = n$, we have to run out of choices at some point.

Thus, in other words, for some $1 \leq i < j$, $v_j = v_i$, when $G$ contains a cycle. This contradicts the acyclic assumption.
Theorem If $G$ is a finite and acyclic digraph, then the algorithm gives the topological sorting.

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.

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 only if every edge coming into $v_j$, particularly $(v_i, v_j)$, has been retracted. By the algorithm, this won’t happen unless $v_i$ is retracted, after its labeling.

In general, let $(v_i, v_{i+1}, \cdots, v_j)$ be a path of $G$. Similar argument shows that $v_{i+1}$ will be labeled after the labeling of $v_i$. Also, inductive assumption shows that $v_j$ won’t be labeled unless $v_{i+1}$ gets labeled first.

Thus, $v_j$ won’t be labeled unless $v_i$ gets labeled first, i.e., \textit{the label of $v_i$ is less than that of $v_j$}. 
Implementation

Let $G = (V, E)$, and let $v \in V$. By $d_{in}(v)$, the in degree of $v$, is defined as the number of edges coming into $v$.

```c
void Toposort(G: graph){
    for(c=1; c<=|V|; c++){
        //The following might take O(|V|).
        find v such that its indegree is 0;
        if (v cannot be found) error;
        else {
            top_num[v]=c;
            for each w adjacent to v do
                indegree[w]=indegree[w]-1;
        }
    }
}
```

This algorithm takes $O(|V|^2)$. It can be improved to a $O(|V| + |E|)$ with a better algorithm, as we only need to check for those nodes whose indegree is just decremented to see if it is 0.