# Graph Basics[1]

These notes are supposed to be a brush up of mostly definitions about graphs that we will need for the rest of the course. Also added is a section on *priority queues* and their implementation using *heaps*. I am going to use this data structure as a black-box. It is a pretty useful data structure you may have seen in other CS classes.

## Graphs

Formally, a graph is a pair of sets often denoted as $G = (V, E)$. For this course, $V$ is a *finite* set which we often assume to be $\{1, 2, \ldots, n\}$. $E$ is a set whose elements are *pairs* of elements from $V$. Notationally, this is denoted as $E \subseteq V \times V$ where the latter is the Cartesian product of sets. We will often use $m$ as the number of edges.

The above formalism comes to life when we draw a picture of the above definition. On the plane, we draw $n$ points naming them using the elements of $V$. We call these points *vertices*. For every pair $(u, v) \in E$, we draw a line segment from point named $u$ to point named $v$. These line segments are called *edges*. Doing so, we get a *pictorial representation* of the object $G = (V, E)$, something which allows us human beings to comprehend much better than the sets written as sequence of pairs. So much so, that the set $V$ is called the vertex set and the set $E$ is called the edge set of the graph $G$. Here's an example.

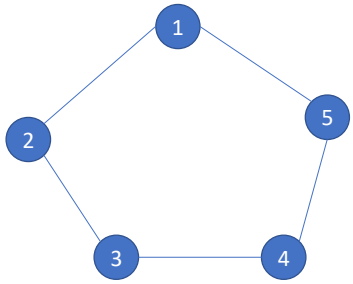


Figure 1: Above is the pictorial representation of the graph $G = (\{1, 2, 3, 4, 5\}, \{(2, 3), (4, 5), (1, 2), (4, 3), (1, 5)\})$. Most of us will easily recognize the picture than parse through the numbers given here to understand it means a cycle on 5 vertices.

**Remark:**

- *As one can see from the formal definition above, the pair $(u, u)$ technically can be in $E$. These pairs are called* loops. *Occasionally, it makes sense to define $E$ as a multi-set, that is, which allows the same edge $(u, v)$ to appear more than once. In the pictorial notation, these would lead to multiple* parallel *line segments between the points. These edges are called* parallel edges. *A graph is called* **simple** *if it has no loops or parallel edges.*

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
*These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!*

- *In the above definition, elements of the edge $E$ have been considered to be* ***unordered*** *pairs. Often, it will make sense to talk of* ***ordered*** *pairs, that is, where $(u, v)$ and $(v, u)$ are different elements. In such cases, in the pictorial notation we draw an edge $(u, v)$ as a line segment from $u$ to $v$ and mark it with an* arrow *pointing towards $v$. The vertex $v$ is called the* head *and the vertex $u$ is called the* tail *of the edge $(u, v)$. Such graphs with arrows are called* **directed graphs.** *Note that simple directed graphs can have so-called* "antiparallel" *edges (the edges $(u, v)$ and $(v, u)$.)*

Graphs have changed the world. It is hard to understate this. What started as recreational puzzles has been the bedrock of things that are used day-to-day a billion times: from getting people and packets from point A to point B fast, to discovering relations between proteins which lead to drug design and disease understanding. To do all these, there are many *computational problems* on graphs that need to be solved and need to be solved fast. Graph algorithms are key and have a very rich theory developed in the last 60 years. In this course we will see some basic graph algorithms which you all should soak in your muscles and blood.

## How does a computer see a graph?

Unlike humans, computers (as of today) don't see any benefit to pictorial representations. To a computer, one needs to describe graphs using the language it understands: matrices, lists, and ultimately bits. There are two standard ways.

**Adjacency Matrix.** Given a graph $G = (V, E)$ with $n$ vertices, the *adjacency matrix* is an $n \times n$ matrix $A_G$ whose columns and rows are indexed by the vertices of $G$. For any two vertices $u$ and $v$ in $V$, we have $A_G[u, v] = 1$ if and only if $(u, v) \in E$. Note if $G$ is undirected, then $A_G$ is symmetric, but for directed graphs $A_G$ need not be. Figure 2 shows an example. As you may be feeling, this way of representing wastes

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | 1 | | | | 1 | 1 | 1 | |
| 2 | 1 | | 1 | | | | | | 1 |
| 3 | | 1 | | 1 | | 1 | | 1 | |
| 4 | | | 1 | | 1 | | | | |
| 5 | | | | 1 | | 1 | | 1 | |
| 6 | 1 | | 1 | | 1 | | 1 | | |
| 7 | 1 | | | 1 | | 1 | | | 1 |
| 8 | 1 | | 1 | | 1 | | | | 1 |
| 9 | | 1 | | | | | 1 | 1 | |

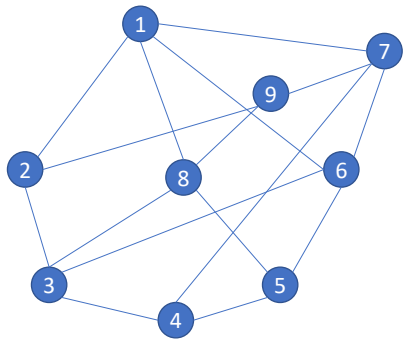


Figure 2: Adjacency Matrix. The empty spaces in the matrix are zeroes.

a lot of space: indeed, why explicitly write a zero when it means not one? Thus, most often one uses the next style of storing graphs.

**Adjacency Lists.** The other way, and the way that we will be mostly using in the course, is that of *adjacency lists*. Given $G = (V, E)$, we have an array $L$ indexed by the vertices which contain pointers to doubly linked lists. For a vertex $u$, the list $L[u]$ contains all the *neighbors* of $u$, that is vertices $v$ such that $(u, v) \in E$, in an arbitrary fashion. Figure 3 shows an example.

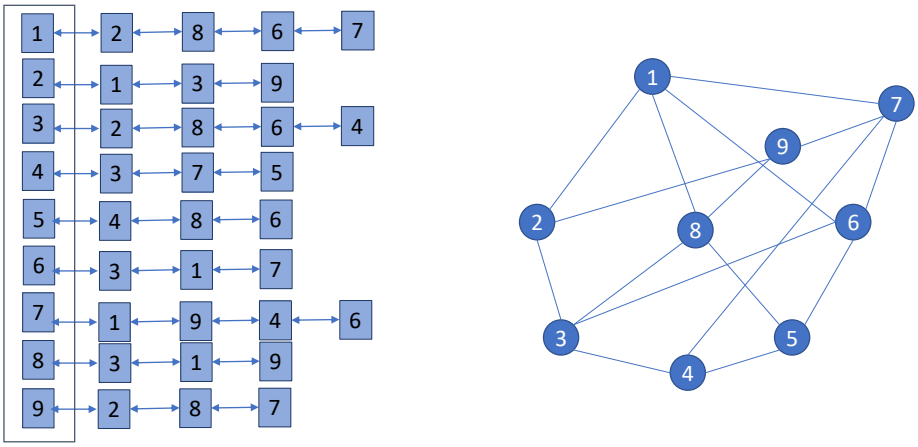


Figure 3: Adjacency Lists.

**Comparison.** The following table shows the comparison of the above two methods.

| Operation | Adj. Matrix | Adj. Lists (naive) | Adj. Lists with Hashing |
|---|---|---|---|
| Is $(u, v)$ an edge? | $O(1)$ | $O(\min(\deg(u), \deg(v))$ | $O(1)$ |
| Iterate over neighbors of $v$ | $O(\|V\|)$ | $O(\deg(v))$ | $O(\deg(v))$ |
| Add an edge $(u, v)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Delete an edge $(u, v)$ | $O(1)$ | $O(\deg(u) + \deg(v))$ | $O(1)$ |
| Space | $O(\|V\|^2)$ | $O(\|E\|)$ | $O(\|E\|)$ |

The third column of the table is the implementation of the adjacent lists using hashing. We probably will not see how this is done in this course, but hashing may be something you may have seen before.

## Some definitions/notations regarding graphs

Given a graph $G = (V, E)$, we often reserve $n$ to denote the number of vertices and $m$ to denote the number of edges. Since $G$ is simple, $m$ can vary from 0 (if the graph is "empty") to $\binom{n}{2} = O(n^2)$. A special case is when $m = O(n)$; such graphs are called *sparse graphs*.

**Degrees.** If $G$ is undirected, the degree of a vertex $v$, denoted as $\deg(v)$ is the number of edges *incident* on $v$. That is, it is the number of edges of the form $(v, u)$. If $G$ is directed, the vertex $v$ has two degrees. The out-degree $\deg^+(v)$ is the number of directed edges $(v, u)$ for which $v$ is the tail. The in-degree $\deg^-(v)$ is the number of directed edges $(u, v)$ for which $v$ is the head. The ***handshake*** lemma states the following (which you should recall from CS 30): For any undirected graph $G = (V, E)$, $\sum_{v \in V} \deg(v) = 2|E|$. For any directed graph $G = (V, E)$, $\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |E|$.

**Walks, Paths, Trails, Cycles, Circuits.** Again, some stuff you may have seen before. A brush-up from CS30. Fix a graph $G = (V, E)$, directed or undirected. Here are some definitions. To illustrate, consider the graph in Figure 2.

- An *alternating sequence* $w = (v_0, e_0, v_1, e_1, \ldots, e_{k-1}, v_k)$ where each $v_i \in V$ and $e_i \in E$ is called a **walk** in the graph $G$ if for all $0 \leq i \leq k-1$, the edge $e_i = (v_i, v_{i+1})$. There is no restriction on repetitions: both vertices and edges could repeat. The *length* of this walk is $k$ which is the number of edges in the walk. The first vertex $v_0$ is called the *source* or *origin* of the walk; the last vertex $v_k$ is the *sink* or *desitnation*. In a simple graph, the walk could just be specified by the vertices as the edges are implied by this; but whenever we describe a walk, we will use this alternating notation.

  For instance, $w = (1, (1,8), 8, (8,5), 5, (5,4), 4, (4,3), 3, (3,8), 8, (8,1), 1, (1,2), 2)$ is a valid walk. Note that the vertices $8, 1$ and the edge $(8,1)$ repeats.

- A walk $w$ is called a **trail** if no edges repeat, but vertices may.

  For instance, $t = (1, (1,8), 8, (8,5), 5, (5,4), 4, (4,3), 3, (3,8), 8, (8,9), 9)$ is a valid trail. The vertex $8$ repeats.

- A trail $C$ is called a **circuit** if the source and sink of the trail are the same. That is, the trail starts and ends at the same location.

  For instance, $C = t \circ ((9,2), 2, (2,1), 1))$ is a valid circuit where $t$ is the example of the trail above, and $\circ$ is the concatenation operator.

- A walk is called a **path** if no vertices repeat.

  For instance, $p = (1, (1,2), 2, (2,3), 3, (3,8), 8)$ is a path. Often a path is just denoted with the vertices like $p = (1, 2, 3, 8)$.

- A circuit $C$ is a **cycle** if no vertices repeat. A singleton vertex $v_0$ is also a cycle of length $0$. Also note that in directed graphs, two antiparallel edges form a valid cycle of length $2$.

  For instance, $C = p \circ ((8,1), 1))$ is a valid cycle.

The following is a simple but very important fact.

**Lemma 1.** Given any walk $w$ from vertex $u$ to $v$, there is also a path from $u$ to $v$ using a subset of the edges of $w$.

*Proof.* We prove by induction on the number of edges in $w$. Let the length of the walk $w$ be $\ell$. That is, $w = (u = v_0, e_0, \ldots, e_{\ell-1}, v_\ell = v)$. If $\ell = 1$, then $w$ is also a path, and the claim is vacuously true (this is the base case).

Now suppose the claim is indeed true for all walks of length $\ell' < \ell$, and we want to prove it for this walk $w$ of length $\ell$. Firstly, we observe that if no vertices repeat in $w$, then $w$ is a path by itself and there is nothing to prove. So, suppose the vertex $v_i$ repeats as $v_j$ for some $j > i$.

Now we do the following *shortcutting* operation on the walk $w$. Construct the following walk

$$w' = (v_0, e_0, v_1, e_1, \ldots, e_{i-1}, v_i, e_j, v_{j+1}, \ldots, e_{\ell-1}, v_\ell)$$

This is a valid walk since $e_j = (v_j, v_{j+1})$ which is the same as $(v_i, v_{j+1})$. Also, the length of $w'$ is *strictly* less than that of $w$. In fact, the length has dropped by $(j - i)$. Finally, $w'$ is also from $u$ to $v$. By Induction, there is a path from $u$ to $v$ which is a subset of $w'$ and thus a subset of $w$. □

Similarly, one can prove (and either recall or try proving as above) the following.

**Lemma 2.** Given a *circuit* $C$, prove there exists a *cycle* $C'$ which is a subset of edges of $C$.

**Lemma 3.** Given a walk $w$ which has at least one vertex repeating, prove there is a *cycle* which is a subset of $w$.

**Lemma 4.** Any walk's edges can be decomposed into a path from the walk's source to destination plus a bunch of cycles.

A graph is called **acyclic** if it contains no cycles. An acyclic undirected graph is called a **forest**. A directed acyclic graph is often called by its acronym, **DAG**[2].

**Connectivity.** This is an important concept. Given a graph $G = (V, E)$ and two vertices $u, v$, we say $v$ is *reachable* from $u$ if and only if there is a path from $u$ to $v$. The notion of graph connectivity is subtly different for undirected and directed graphs.

- An undirected graph $G$ is ***connected*** if every vertex $u$ is reachable from every other vertex $v$. Note that in an undirected graph if $u$ is reachable from $v$ then $v$ is reachable from $u$.

  Given any undirected graph $G$, one can decompose its vertex set $V$ into a partition $V = V_1 \cup V_2 \cup \cdots \cup V_k$ where $V_i \cap V_j = \emptyset$ for $i \neq j$, such that $G[V_i]$ is connected, and for any $u \in V_i$ and $v \in V_j$, $u$ and $v$ are not reachable from each other. These $V_i$'s are called the **connected components** of $G$. Here, $G[V_i]$ is the subgraph of $G$ induced by $V_i$; formally $G[V_i] = (V_i, E_i)$ where $E_i \subseteq E$ with $(x, y) \in E_i$ if and only if $x, y \in V_i$ and $(x, y) \in E$.

- A directed graph $G$ is *weakly connected* if one ignores the arrows on the edges, the underlying undirected graph is connected. This is rather unsatisfactory as it doesn't tell us about what the connectivity is in the *directed* graph. A directed graph is ***strongly connected*** if for any two vertices $u$ and $v$ in $V$, both $u$ is reachable from $v$ *and* $v$ is reachable from $u$. Any directed graph can be partitioned into **strongly connected components**: $V = V_1 \cup \cdots \cup V_k$ such that each $G[V_i]$ is strongly connected and for any $u \in V_i$ and $v \in V_j$ either $u$ is *not* reachable from $v$, or $v$ is not reachable from $u$, or both.

- An undirected graph $G$ is a ***tree*** if it is *acyclic* and *connected*. A *directed* acyclic graph $G$ is a ***rooted (out-)tree***, if (a) the underlying undirected graph (will all arrows removed) is a tree, and (b) there is a particular vertex $r$ (the root) with the property that for any other vertex $v$ in $G$, there is a path from $r$ to $v$ in the directed graph $G$. Note: there cannot be a path from $v$ to $r$ as well since $G$ is acyclic. If in condition (b) instead there was path from $v$ to $r$, for every $v$, then the graph would be a rooted in-tree.

**Some Computation Problems on Graphs.** The following problems are some computation problems that we will tackle first in this course.

REACHABLE?
**Input:** Graph $G = (V, E)$. Two vertices $u, v$.
**Output:** Is $v$ reachable from $u$?

CYCLE?
**Input:** Graph $G = (V, E)$.
**Output:** Does the graph $G$ have a cycle?

[2] https://www.youtube.com/watch?v=zH64dlgyydM

CONNECTED COMPONENTS
**Input:** Undirected Graph $G = (V, E)$.
**Output:** Connected Components of $G$.

STRONGLY CONNECTED COMPONENTS
**Input:** Directed Graph $G = (V, E)$.
**Output:** Strongly Connected Components of $G$.

## Data Structure: Heaps and Priority Queues

We will be using the following data structure, called ***priority queues***, at least once (perhaps twice), and in any case this is worth knowing. A nice exposition can be found in one of the text-books; CLRS is more detailed and verbose, while DPV is terser.

Here is what we want to maintain. There is a set $S$ of objects with (key, value) pairs whose keys come from an $n$ element universe $U$. We may as well assume the keys are numbers from 1 to $n$. We want to allow the following 4 operations:

- INSERT($S, x$): Insert an object $x$ into $S$.
- DELETE($S, x$): Delete an object $x$ from $S$.
- DECREASE-VAL($S, x, v$): Decrease the value of $x \in S$ to $v$ only if $v$ is smaller than $x$'s current value.
- EXTRACT-MIN($S$): Return the $x \in S$ with minimum value and delete it.

Of course one can just use an array $A[1 : n]$ where $A[x]$ stores the value of $x \in S$ and $\perp$ otherwise. The first three operations take $O(1)$ time, however, the last operation takes $\Theta(n)$ time. On the other hand if we store the items as a MIN-HEAP, then *all* the operations take $O(\log n)$ time. Using heaps to implement priority queues is the most common way.

There is another data structure called the FIBONACCI HEAP which can implement the first three operations in $O(1)$ time[3] and the last in $O(\log n)$ time. Seems like the best of both the array-and-the-heap world. The following table encapsulates all this.

| Operation | Array | Heap | Fibonacci Heap |
|---|---|---|---|
| INSERT($S, x$) | $O(1)$ | $O(\log n)$ | $O(1)$ |
| DELETE($S, x$) | $O(1)$ | $O(\log n)$ | $O(1)$ |
| DECREASE-VAL($S, x, v$) | $O(1)$ | $O(\log n)$ | $O(1)$ |
| EXTRACT-MIN($S$) | $O(n)$ | $O(\log n)$ | $O(\log n)$ |

[3] I am lying. The $O(1)$ is only *amortized* over many calls. We, unfortunately, won't cover amortized analysis in this course.