Towards the mathematical notion of “graph”

**Definition** A **graph** $G$ is a pair $G = (V, E)$, where $V$ is the set of vertices, while $E$ is the set of (undirected) edges. An (undirected) edge connecting vertices $u, v$ is denoted by $\{u, v\}$. We require $u \neq v$ here.

An example is

$$G = (\{1, 2, 3, 4, 5\}, \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 4\}\})$$

which is a graph with $|V| = 5$ vertices and $|E| = 4$ edges. A possible **drawing** is as follows:

```
1 -- 2
  |   |
  |   |
  |   |
3 -- 4 -- 5
```

We see that $G$ has two “connected components”.

Directed graphs

A “graph” is synonymous with “undirected graph”. Now for a “directed graph” the edges are directed:

**Definition** A **directed graph** (or digraph) $G$ is a pair $G = (V, E)$, where $V$ is again the set of vertices, while $E$ is the set of directed edges or arcs. A directed edge from vertex $u$ to vertex $v$ is denoted by $(u, v)$. Again we require $u \neq v$ here.

An example is

$$G = (\{1, 2, 3, 4, 5\}, \{(1, 2), (3, 1), (2, 4), (4, 3)\})$$

which is a graph with $|V| = 5$ vertices and $|E| = 4$ edges. A possible **drawing** is as follows:

```
1 → 2
  ↓   ↓
  3 ← 4 ← 5
```

General remarks

- We consider **graphs**, an important data structure.
- The two main algorithms are **breadth-first search** (BFS) and **depth-first search** (DFS).
- We consider one application of BFS, the computation of **shortest paths**.
- And we consider one application of DFS, **topological sorting** of graphs.

Reading from CLRS for week 4

- Chapter 22
- Plus appendices
  - B.4 “Graphs”
  - B.5.1 “Free trees”
Remarks on these fundamental notions

A loop is an edge or arc connecting a vertex with itself:

- The notion of graph doesn’t allow loops; only the extension of “graphs with loops” allows them, but we do not consider them here.
- Also the notion of directed graph doesn’t allow loops (different from the book we keep the symmetry between “graphs” and “directed graphs”); the extension of “directed graphs with loops” allows them, but again we do not consider them.

For a graph $G$ with $n = |V|$ vertices we have

$$|E| \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

while for a directed graph $G$ we have

$$|E| \leq n(n-1).$$

Adjacency list representation

For (undirected) graphs this representation uses two list elements for each edge (since both directions are present); for directed graphs this is only the case when antiparallel edges are present.

The total space required for this representation is $O(V + E)$ (that is, $O(|V| + |E|))$. This representation is suited to sparse graphs, where $E$ is much less than $V^2$.

Adjacency matrix representation

For (undirected) graphs the matrix is symmetric.

The total space required for this representation is $O(V^2)$. This representation is suited to dense graphs, where $E$ is on the order of $V^2$.

Representations of graphs

The above notions of graphs and digraphs yield mathematical objects. These are the eternal platonic ideas, which have many different representations in computers.

There are two main ways to represent a graph $G = (V, E)$:

- **Adjacency lists**: The graph is represented by a $V$-indexed array of linked lists, with each list containing the neighbours of a single vertex.
- **Adjacency matrix**: The graph is represented by a $|V| \times |V|$ bit matrix where the $A_{i,j} = 1$ if and only if vertex $i$ is adjacent to vertex $j$. 

Adjacency list representation

```
  1 2 4 6
  2 3 5 7
  3 2 4
  4 2 3 5 6
  5 3 4 7
  6 2 3 5
  7 2 4 5
```

Adjacency matrix representation

```
A =
0 1 0 0 0 0 0
1 0 1 1 0 0 0
0 1 0 1 1 0 0
0 1 1 0 1 1 1
0 0 1 1 0 0 1
0 0 0 1 0 0 0
0 0 0 1 1 0 0
```
Comparing the two representations

Apart from the different space requirements needed for sparse and dense graphs, there are other criteria for deciding on which representation to use.

The choice of representation will depend mostly on what questions are being asked.

For example, consider the time required to decide the following questions using the two different representations:

- Is vertex \( v \) adjacent to vertex \( w \) in an undirected graph?
- What is the out-degree of a vertex \( v \) in a directed graph?
- What is the in-degree of a vertex \( v \) in a directed graph?

Spanning trees

**Definition** Consider a connected graph \( G \) with at least one vertex. A spanning tree of \( G \) is a tree \( T \) with \( V(T) = V(G) \) and \( E(T) \subseteq E(G) \).

So spanning trees just leave out edges which are not needed to connect the whole graph. For example consider the graphs

\[
G_1 = \begin{array}{c}
1 & 2 \\
| & |
\end{array}, \quad G_2 = \begin{array}{c}
1 & 2 \\
\downarrow & \\
3 & 4
\end{array}.
\]

\( G_1 \) has 4 different spanning trees, while \( G_2 \) has \( 4 + 4 = 8 \).

The notion of a tree

You might have seen already “rooted trees” (and you will in the next week) — trees are basically just like rooted trees, but without a designated root.

**Definition** A tree is a graph \( T \) with at least one vertex, which is connected and does not have a cycle.

Intuitively, a graph \( G \) has a cycle if there are two different vertices \( u, v \) in \( G \) such that there are (at least) two essentially different ways of getting from \( u \) to \( v \). And thus going from \( u \) to \( v \) the one way, and from \( v \) to \( u \) the other way, we obtain the “round-trip” or “cycle”.

We have the following fundamental characterisation of trees:

**Lemma 1** A graph \( G \) is a tree if and only if \( G \) is connected, \( |V| \geq 1 \) and \( |E| = |V| - 1 \) holds.

So trees realise minimal ways of connecting a set of vertices.

Computing spanning trees

We will see two algorithms, BFS and DFS, for computing spanning trees:

- In both cases actually rooted spanning trees are computed, that is, additionally a vertex is marked as “root”.
- When drawing such rooted spanning trees, one starts with the root (otherwise one could start with an arbitrary vertex!), going from the root to the leaves.
- For such rooted spanning trees, one typically speaks of nodes instead of vertices.
- Both algorithms compute additional data, besides the rooted spanning trees.
- The DFS version will be extended to compute a spanning forest: It can be applied to non-connected graphs, and computes a (rooted) spanning tree for each connected component.
Representing rooted trees

A rooted tree, that is, a tree together with a root $T$, will be represented by BFS and DFS as follows:

- Now there is a “direction” in the tree, either going from the root towards the leaves, or from the leaves towards the root.
- We obtain the usual notion of the children of a node (without a root, in a “pure tree”, there is no such thing).
- The leaves are the nodes without children.
- And we speak of the(!) parent of a node (note that every node can have at most one parent).
- The root is the only vertex without a parent.
- Now specifying the root for each non-root vertex is sufficient to represent the tree.
- This is done in BFS and DFS by an array $\pi$ (like “parent”).

The algorithm

**Input:** A graph $G$ with vertex set $V(G)$ and edges represented by adjacency lists $Adj$.

BFS($G, s$)
1. for each $u \in V(G)$
2. \hspace{1em} $d[u] = \infty$
3. $\pi[s] = \text{NIL}$
4. $d[s] = 0$
5. $Q = (s)$
6. while $Q \neq ()$
7. \hspace{1em} $u = \text{DEQUEUE}[Q]$
8. \hspace{1em} for each $v \in \text{Adj}[u]$
9. \hspace{1em} \hspace{1em} if $d[v] = \infty$
10. \hspace{1em} \hspace{1em} $d[v] = d[u] + 1$
11. \hspace{1em} \hspace{1em} $\pi[v] = u$
12. \hspace{1em} $\text{ENQUEUE}(Q, v)$

Breadth-first search

Searching through a graph is one of the most fundamental of all algorithmic tasks.

**Breadth-first search (BFS)** is a simple but very important technique for searching a connected graph.

Such a search starts from a given source vertex $s$ and constructs a rooted spanning tree for the graph, called the breadth-first tree (the root is $s$).

It uses a (first-in first-out) queue as its main data structure.

BFS computes the parent $\pi[u]$ of each vertex $u$ in the breadth-first tree (with the parent of the source being NIL), as well as its distance $d[u]$ from the source $s$ (initialised to $\infty$), which is the length of the path from $s$ to $u$ in the breadth-first tree, which is a shortest path between these vertices.
Analysis of BFS

Correctness Analysis: At termination of BFS($G,s$), for every vertex $v$ reachable from $s$:
- $v$ has been encountered;
- $d[v]$ holds the length of the shortest path from $s$ to $v$;  
- $\pi[v]$ represents an edge on a shortest path from $v$ to $s$.

Time Analysis:
- The initialisation takes time $\Theta(V)$.
- Each vertex is \texttt{ENQUEUE}ed once and \texttt{DEQUEUE}d once; these queueing operations each take constant time, so the queue manipulation takes time $\Theta(V)$ (altogether).
- The Adjacency list of each vertex is scanned only when the vertex is \texttt{DEQUEUE}d, so scanning adjacency lists takes time $\Theta(E)$ (altogether).
- The overall time of BFS is thus $\Theta(V + E)$.

Running BFS on directed graphs

We can run BFS also on a digraph $G$:
- For digraphs there are no “spanning trees” anymore.
- But only the vertices reachable from the chosen start vertex following the directions of the edges are in the rooted tree.
- Still the paths in the rooted tree are shortest possible, given that they have to obey the given directions of the edges.

The edges have (implicitly) a length of one unit.

- If arbitrary non-negative lengths are allowed, then we have to generalise BFS to \textit{Dijkstra's algorithm}.
- This generalisation must keep the essential property, that the distances encountered are the final ones and are non-decreasing.
- But now not all edges have unit-length, and thus instead of a simple queue we need to employ a \textit{priority queue}.

Why do we get shortest paths?

Is it really true that we get always shortest paths (that is, using the minimum number of edges)? Let’s assume that to some vertex $v$ there exists a shorter path in $G$ from $s$ to $v$ than found by BFS. Let this length be $d' < d[v]$.

- $v \neq s$, since the distance from $s$ to $s$ is zero (using the path without an edge), and this is correctly computed by BFS.
- Consider the predecessor $u$ on that shorter path.
- If also $d[u]$ would be wrong (that is, too big), than we could use $u$ instead of $v$. Thus w.l.o.g. $d[u]$ is correct.
- Now when exploring the neighbours of $u$, in case $v$ is still unexplored, it would get the correct distance $d' = d[u] + 1$.
- So it must have been explored already earlier.
- However the distances $d[w]$ set by BFS are non-decreasing!

Restart needed for digraphs to cover all vertices

Consider the simple digraph

```
1 ←← 2
\downarrow
3
```

In order to cover all vertices, one needs to run BFS at least two times (and if the first time you start it with $s = 1$, then you need to run it three times).

- Note that the obtained “things” (given by $\pi$) overlap.
- We can still call them “rooted trees”, but now they are situated in a digraph, and thus the edges have already given a direction (it is not upon the algorithm to choose one).
- Still these rooted trees are shortest-path trees, from the root to all other nodes. Note that a path by definition now follows the direction of the arc.
Depth-first search

Depth-first search (DFS) is another simple but very important technique for searching a graph.

Such a search constructs a spanning forest for the graph, called the depth-first forest, composed of several depth-first trees, which are rooted spanning trees of the connected components.

DFS recursively visits the next unvisited vertex, thus extending the current path as far as possible; when the search gets stuck in a “corner” it backtracks up along the path until a new avenue presents itself.

DFS computes the parent \( \pi[u] \) of each vertex \( u \) in the depth-first tree (with the parent of initial vertices being NIL), as well as its discovery time \( d[u] \) (when the vertex is first encountered, initialised to \( \infty \)) and its finishing time \( f[u] \) (when the search has finished visiting its adjacent vertices).

### The algorithm

**DFS(G)**

1. for each \( u \in V(G) \)
2. \( d[u] = \infty \)
3. \( time = 0 \)
4. for each \( u \in V(G) \)
   5. if \( d[u] = \infty \)
     6. \( \pi[u] = NIL \)
     7. DFS-Visit\( (u) \)
   8. \( f[u] = time \)

**DFS-Visit\( (u) \)**

1. \( time = time + 1 \)
2. \( d[u] = time \)
3. for each \( v \in Adj[u] \)
   4. if \( d[v] = \infty \)
     5. \( \pi[v] = u \)
     6. DFS-Visit\( (v) \)
     7. \( time = time + 1 \)

**Analysis:** DFS-Visit\( (u) \) is invoked exactly once for each vertex, during which we scan its adjacency list once. Hence DFS, like BFS, runs in time \( \Theta(V + E) \).

### Running DFS on directed graphs

Again (as with BFS), we can run DFS on a digraph \( G \).

- Again, no longer do we obtain “spanning trees” of the connected components of the start vertex.
- Again, in the rooted tree constructed we find exactly all vertices reachable from the root when following the directions of the edges.

However, DFS trees do not contain shortest paths — to that end their way of exploring a graph is too “adventuresomely” (while BFS is very “cautious”).

Nevertheless, the information gained through the computation of discovery and finish times is very valuable for many tasks. One example follows.
Directed acyclic graphs

An important applications of digraphs $G$ is with scheduling:

- The vertices are the jobs (actions) to be scheduled.
- A directed edge from vertex $u$ to vertex $v$ means a dependency, that is, action $u$ must be performed before action $v$.

Now consider the situation where we have three jobs $a$, $b$, $c$ and the following dependency digraph:

$$G = a \rightarrow b \rightarrow c$$

Clearly this can not be scheduled!

In general we require $G$ to be acyclic, that is, $G$ must not contain a directed cycle.

A directed acyclic graph is also called a dag.

Topological sorting via DFS

Lemma 2 After calling DFS on a dag, for every edge $(u, v)$ we have $f[u] > f[v]$.

Proof Consider when the DFS hits the edge $(u, v)$.

- If $d[v] = \infty$, then DFS will be called recursively on $v$, $f[v]$ will be defined, and the recursive call will terminate; only later will $f[u]$ be defined, and thus we get $f[u] > f[v]$.
- If $d[v] \neq \infty$, then $f[v]$ must already be defined, for otherwise we would be on a path of the depth-first tree which contains $v$, and so this edge $(u, v)$ would complete a cycle in the graph. $f[u]$ has yet to be defined, and so when it does, we shall get $f[u] > f[v]$.

Corollary To topologically sort a dag $G$, we run DFS on $G$ and print the vertices in reverse order of finishing times. (We can put each vertex on the front of a list as they are finished.)

Topological sorting illustrated

Consider the result of running the DFS algorithm on the following dag.

Listing the vertices in reverse order of finishing time gives the following topological sorting of the vertices:

$$u : p\ n\ o\ s\ m\ r\ y\ v\ w\ z\ x\ u\ q\ t$$

$$f[u] : 28\ 26\ 25\ 24\ 20\ 19\ 18\ 17\ 16\ 15\ 12\ 8\ 5\ 4$$

The two possible topological sortings of $G$ are $a, c, b$ and $c, a, b$.

Topological sorting

Given a dag $G$ modelling a scheduling task, a basic task is to find a linear ordering of the vertices (“actions”) such that all dependencies are respected.

- This is modelled by the notion of “topological sorting”.
- A topological sort of a dag is an ordering of its vertices such that for every edge $(u, v)$, $u$ appears before $v$ in the ordering.

For example consider

$$G = a \rightarrow b \rightarrow c$$

The two possible topological sortings of $G$ are $a, c, b$ and $c, a, b$. 