We turn toward the algorithmic problem of connectivity: given vertices $s$ and $t$ in a graph $G$, is there a path from $s$ to $t$? Can we find it algorithmically?

**Breadth-first search** The idea: start at $s$, explore all neighbors one ‘layer’ at a time; at the end can check whether $t$ is in one of the layers. In this way, BFS produces a rooted tree with root $s$.

Breadth-first search (BFS) takes as input a graph $G$ and a vertex $s$ in $G$. It produces a rooted tree with root $s$ and stratifies the graph into layers $L_0, L_1, \ldots$ as follows.

\[
L_0 = \{s\}
\]
\[
L_1 = \{\text{neighbors of } s\}
\]
\[
L_2 = \{v \notin L_0 \cup L_1 \text{ that are adjacent to a vertex in } L_1\}
\]
\[
\vdots
\]
\[
L_{i+1} = \{v \notin L_0 \cup \cdots \cup L_i \text{ that are adjacent to a vertex in } L_i\}.
\]

The distance between two vertices $x$ and $y$ in a graph is the length of the shortest path from $x$ to $y$ (if there is no path we say the distance is $\infty$). We can give a different definition of the layers of BFS by setting

\[
L_i = \{\text{vertices of distance } i \text{ from } s\}.
\]

BFS is an instance of the following underspecified algorithm for exploring a graph from a vertex $s$:

Starting with $R = \{s\}$, iteratively grow $R$ by adding new vertices adjacent to a vertex in $R$, until you run out of such vertices. One way to implement this is to grow $R$ in ‘layers’ as BFS does.

**Depth-first search.** Another such algorithm is suggested by a natural approach to exploring a maze: explore until you reach a dead-end and then backtrack.

---

**Algorithm 1:** the explore subroutine

1. explore($v$):
2. \hspace{1em} visited($v$) = true;
3. \hspace{2em} // (previsit work)
4. \hspace{2em} foreach neighbor $u$ of $v$ do
5. \hspace{3em} if not visited($u$) then
6. \hspace{4em} explore($u$);
7. \hspace{2em} // (postvisit work)

---

**Algorithm 2:** Depth-first search

1. DFS($G, s, t$):
2. \hspace{1em} **Input:** a graph $G = (V, E)$, a start vertex $s$ and a target vertex $t$
3. \hspace{1em} **Output:** whether $s$ is connected to $t$
4. \hspace{1em} // initialize
5. \hspace{2em} foreach $v \in V$ do
6. \hspace{3em} set visited($v$) = false.
7. \hspace{1em} explore($s$);
8. \hspace{1em} if visited($t$) then
9. \hspace{2em} return true
If you visit $w$ in virtue of its being a neighbor of $v$, then you can also add the edge $(v, w)$ to a tree. At the end, you will have a *spanning tree* of the connected component rooted at the start vertex $s$. This is called the DFS tree. (The algorithm is underspecified; you should use an ordering on the vertices to break ties.)

**Problem 1.**
(a) Perform BFS on the graph in (1) starting at any vertex. Identify the layers $L_i$.
(b) Perform DFS on the graph in (1) starting at any vertex.

![Graph](image)

**Problem 2.** Prove the following.
(a) If $x$ and $y$ are adjacent vertices and $x \in L_i$ and $y \in L_j$, then $|i - j| \leq 1$.
(b) The set of vertices visited by BFS is exactly the connected component containing $s$.
(c) Explain how edges can be kept track of in the execution of BFS in such a way that BFS produces a *subtree* of $G$ that *spans* the connected component containing $s$. (Hint: give the vertices an ordering.)

Item (a) says that edges not in the BFS tree pass between vertices in the same layer or between vertices in adjacent layers.

**Definition.** Recall that a *rooted tree* is a tree (connected, acyclic graph) with a specified vertex $r$, called the *root*.

Recall that a graph $T$ is a tree iff between any two vertices of $T$ there is a *unique* path. Thus in a rooted tree every vertex has a unique path to the root.

**Definition.** If $x$ and $y$ are vertices in a rooted tree $(T, r)$, then we say...
(a) $x$ is a *descendant* of $y$ (or $y$ is an *ancestor* of $x$) if $y$ occurs on the unique path from $x$ to the root $r$.
(b) $x$ is a *child* of $y$ (or $y$ is a *parent* of $x$) if $x$ is a descendant of $y$ and a neighbor of $y$.

**Problem 3.** Draw a picture explaining the definition.

**Problem 4.**
(a) Explain how the DFS and the BFS algorithms can be “enriched” to keep track of the edges of a DFS tree and a BFS tree, both rooted trees with root $s$.
(b) With the graph in (1), which starting vertex gives the BFS tree with the minimum height?

**Problem 5.** Look at the graph below. If the blue (solid) portion is the DFS tree starting at $s$, why can’t the graph have an edge indicated by the dashed line? (Certainly the DFS tree cannot have that edge, but the original graph can’t either!)
**Problem 6.** Run \( \text{DFS} \) on the graph in (†), starting at \( S \) and then using the alphabetical ordering. Draw the \( \text{DFS} \) tree. Verify that every non-tree edge is incident to a vertex and its descendant.

![Graph](attachment:image.png)

**Problem 7.** Prove the following assertion: In a call to \( \text{explore}(v) \), all vertices marked ‘visited’ between calling and the end of the execution are descendants of \( v \) in the \( \text{DFS} \) tree \( T \).

Explain why the following Corollary is true.

**Corollary.** Suppose that \( T \) is the \( \text{DFS} \) tree rooted at \( r \) in a graph \( G \). If \( x \) and \( y \) are vertices of \( T \) and \( x \) and \( y \) are adjacent in \( G \), then one of \( x \) and \( y \) is a descendant of the other in \( T \).

**Definition.** \( H \) is a subgraph of \( G \) if \( V(H) \subseteq V(G) \) and \( E(H) \subseteq E(G) \). A spanning tree of \( G \) is a subgraph \( H \) such that (1) \( H \) is a tree and (2) \( V(H) = V(G) \).

**Theorem.** A graph is connected if and only if it has a spanning tree.

**Problem 8.** Prove the theorem. *(Hint: What might this have to do with algorithms?)*

**On implementation and running time** In practice, it is often better to represent graphs by adjacency lists rather than adjacency matrices. In the adjacency list representation, to each vertex \( v \) is associated a list \( L[v] \) of the neighbors of \( v \).

**Problem 9.** While the adjacency-matrix representation takes \( O(n^2) \) space for an \( n \)-vertex graph, the adjacency-list representation requires \( O(m + n) \) space, which for sparse graphs (i.e., graphs for which \( m \) is small relative to \( n \)) is less. Explain.

**Problem 10.** Explain why BFS runs in \( O(m + n) \) time on a graph with \( n \) vertices and \( m \) edges, assuming the graph is given by its adjacency list representation.

**Problem 11.** By consulting one of the CS majors in your group, complete and explain the following analogy.

\[
\text{BFS : queue :: DFS : } \_
\]

**Problem 12.** Give a (vague but convincing) explanation for why \( \text{DFS} \) runs in linear time, using adjacency lists.

**Testing bipartiteness with BFS** *(Assume that \( G \) is connected; if not, we can work component-by-component.)*

**Definition.** A graph \( G \) is bipartite iff there is a partition \( V = R \sqcup B \) of its vertices into two (disjoint) pieces \( R \) and \( B \) so that every edge is incident to one vertex in \( R \) and one in \( B \).

This presents an algorithmic problem: determine whether a graph \( G \) is bipartite and, if it is, find a bipartition.

**Theorem.** \( G \) is bipartite if and only if \( G \) has no odd cycle.

**Problem 13.** We’re about to prove the harder direction \( \Rightarrow \); why is the other direction true?

*(Hint: Suppose that you have an odd cycle and that the vertices are partitioned into \( R \) and \( B \). How many \( R-B \) edges can appear in the cycle?)*
Here is the idea: start with $s$ and declare $s \in B$. Put all neighbors of $s$ into $R$. Color their neighbors Blue. Continue. Hope for the best. This will either produce a bipartition or should yield a blue edge or a red edge.

Do BFS starting at $s$ and obtain layers $L_0, L_1, \ldots$

$$B = L_0 \cup L_2 \cup L_4 \cup \cdots$$
$$R = L_1 \cup L_3 \cup L_5 \cup \cdots$$

(*)

At the end review the $m$ edges to check whether there’s an $R$–$R$ or $B$–$B$ edge.

**Proposition.** Let $G$ be a connected graph and $L_0, L_1, \ldots$ the layers produced by BFS. Then exactly one of the following holds.

(i) There is no edge $(x, y)$ of $G$ with $x, y \in L_i$. This implies that $G$ is bipartite with bipartition (*).

(ii) There is an edge $(x, y)$ of $G$ with $x, y \in L_i$. This implies that $G$ has an odd cycle and is therefore not bipartite.

**Problem 14.**

(a) Why can’t both conditions hold?

(b) Suppose that $x, y \in L_i$ are adjacent, as in condition (ii). The vertices $x$ and $y$ certainly have a common ancestor in the BFS tree, namely the root $r$. Explain why there may not be a cycle including $x$, $y$, and $r$, though. So we choose a common ancestor $z$ of $x$ and $y$ in $L_j$ for $j$ maximal.

(c) There are unique paths from $x$ to $z$ and from $y$ to $z$ in $T$. How long are they?

(d) You should have produced an odd cycle. Verify that you have.

(e) Be sure that you have completed the proof of the Proposition.

**Corollary.** If $G$ has no odd cycle, then $G$ is bipartite.

**Problem 15.** Explain why the Corollary follows from the Proposition.

**Problem 16.** By running BFS starting at vertex $A$ in each case, determine whether each of these graphs is bipartite and if not which odd cycle the algorithm exhibits.