Depth-first search, topological sorting
Outline

1 Recap

2 Applications of BFS
   - Testing bipartiteness

3 Depth-first search (DFS)

4 Applications of DFS
   - Cycle detection
   - Topological sorting
1 Recap

2 Applications of BFS
   - Testing bipartiteness

3 Depth-first search (DFS)

4 Applications of DFS
   - Cycle detection
   - Topological sorting
Review of the last lecture

- Graphs (directed, undirected, weighted, unweighted)
  - Notation: $G = (V, E), |V| = n, |E| = m$
- Representing graphs
  1. Adjacency matrix
  2. Adjacency list
- Trees, bipartite graphs, the degree theorem
- Linear graph algorithms
- Breadth-first search (BFS)

Claim 1.

Let $T$ be a BFS tree, let $x$ and $y$ be nodes in $T$ belonging to layers $L_i$ and $L_j$ respectively, and let $(x, y)$ be an edge in $G$. Then $i$ and $j$ differ by at most 1.
Breadth-first search (BFS($G, s$)): explore $G$ starting from $s$ outward in all possible directions, adding reachable nodes one layer at a time.

- First add all nodes that are joined by an edge to $s$: these nodes form the first layer. \textit{If $G$ is unweighted, these are the nodes at distance 1 from $s$.}

- Then add all nodes that are joined by an edge to a node in the first layer: these nodes form the second layer. \textit{If $G$ is unweighted, these are the nodes at distance 2 from $s$.}

- And so on and so forth.
Today

1 Recap

2 Applications of BFS
   - Testing bipartiteness

3 Depth-first search (DFS)

4 Applications of DFS
   - Cycle detection
   - Topological sorting
Testing bipartiteness

- **Input:** a graph $G = (V, E)$
- **Output:** yes if $G$ is *bipartite*, no otherwise

Equivalent problem (*why?*)

- **Input:** a graph $G = (V, E)$
- **Output:** yes if and only if we can color all the vertices in $G$ using at most 2 colors—say red and white—so that no edge has two endpoints with the same color.
**Fact:** If a graph contains an odd-length cycle, then it is not 2-colorable.

So a **necessary** condition for a graph to be 2-colorable is that it does not contain odd-length cycles.

*Is this condition also **sufficient**, that is, if a graph does not contain odd-length cycles, then is it 2-colorable?*

*In other words, are odd cycles the only obstacle to bipartiteness?*
Algorithm for 2-colorability

BFS provides a natural way to 2-color a graph $G = (V, E)$:

- Start BFS from any vertex; color it red.
- Color white all nodes in the first layer $L_1$ of the BFS tree. If there is an edge between two nodes in $L_1$, output no and stop.
- Otherwise, continue from layer $L_1$, coloring red the vertices in even layers and white in odd layers.
- If BFS terminates and all nodes in $V$ have been explored (hence 2-colored), output yes.
Upon termination of the algorithm

- either we successfully 2-colored all vertices and output *yes*, that is, declared the graph bipartite;

- or we stopped at some level because there was an edge between two vertices of that level and output *no*; in this case, we declared the graph non-bipartite.

This algorithm is **efficient**. *Is it a correct algorithm for 2-colorability?*
To prove correctness, we must show the following statement.

If our algorithm outputs

1. **yes**, then the 2-coloring it returns is a valid 2-coloring of $G$;
2. **no**, then indeed $G$ cannot be 2-colored by any algorithm (e.g., because it contains an odd-length cycle).

The next claim proves that this is indeed the case by examining the possible outputs of our algorithm. Note that the output depends solely on whether *there is an edge in $G$ between two nodes in the same BFS layer.*
Claim 2.

Let $G$ be a connected graph, and let $L_1, L_2, \ldots$ be the layers produced by BFS starting at node $s$. Then exactly one of the following is true.

1. There is no edge in $G$ joining two nodes in the same BFS layer. Then $G$ is bipartite and has no odd length cycles.

2. There is an edge in $G$ joining two nodes in the same BFS layer. Then $G$ contains an odd length cycle, hence is not bipartite.

Corollary 1.

A graph is bipartite if and only if it contains no odd length cycle.
1. **Assume** that no edge in $G$ joins two nodes of the same layer of the BFS tree.

By Claim 1, all edges in $G$ not belonging to the BFS tree are
- either edges between nodes in the same layer;
- or edges between nodes in adjacent layers.

Our assumption implies that all edges of $G$ not appearing in the BFS tree are between nodes in adjacent layers.

Since our coloring procedure gives such nodes different colors, the whole graph can be 2-colored, hence it is bipartite.
2. **Assume** that there is an edge \((u, v) \in E\) between two nodes \(u\) and \(v\) on the same layer.

Obviously \(G\) is not 2-colorable by our algorithm: both endpoints of edge \((u, v)\) are assigned the same color.

Our algorithm returns **no**, hence declares \(G\) non-bipartite.

*Can we show existence of an odd-length cycle and prove that \(G\) indeed is not 2-colorable by any algorithm?*
Proof of correctness, part 2

- Let $u, v$ appear at layer $L_j$ and edge $(u, v) \in E$.

- Let $z$ be the common ancestor at max depth of $u$ and $v$ in the BFS tree ($z$ might be $s$). Suppose $z$ appears at layer $L_i$ with $i < j$.

- Consider the following path in $G$: from $z$ to $u$ follow edges of the BFS tree, then to $v$ via edge $(u, v)$ and back to $z$ following edges of the BFS tree. This is a cycle starting and ending at $z$, consisting of $(j - i) + 1 + (j - i) = 2(j - i) + 1$ edges, hence of odd length.
Today

1 Recap

2 Applications of BFS
   - Testing bipartiteness

3 Depth-first search (DFS)

4 Applications of DFS
   - Cycle detection
   - Topological sorting
Depth-first search (DFS): starting from a vertex $s$, explore the graph as deeply as possible, then backtrack

1. Try the first edge out of $s$, towards some node $v$.
2. Continue from $v$ until you reach a dead end, that is a node whose neighbors have all been explored.
3. Backtrack to the first node with an unexplored neighbor and repeat 2.

Remark: DFS answers $s$-$t$ connectivity
Similarities

- Linear-time algorithms that essentially can be used to perform the same tasks

Differences

- DFS is more impulsive: when it discovers an unexplored node, it moves on to exploring it right away; BFS defers exploring until all nodes in the layer have been discovered.

- DFS is naturally recursive and implemented using a stack.
  - A stack is a LIFO (Last-In First-Out) data structure implemented as a linked list: insert (push)/extract (pop) the top element requires $O(1)$ time.
An undirected graph $G_1$
Dashed edges belong to the graph but not to the DFS tree. Ties are broken by considering nodes by increasing index.
A directed graph $G$
Dashed edges belong to $G$ but not to the trees in the DFS forest. $(start, finish)$ intervals appear to the right of every node.
Pseudocode for DFS exploration of the entire graph

\textbf{DFS}(G = (V, E))

\begin{verbatim}
    for u ∈ V do
        explored[u] = 0
    end for
    for u ∈ V do
        if explored[u] == 0 then Search(u)
    end if
    end for
\end{verbatim}

\textbf{Search}(u)

\begin{verbatim}
    previsit(u)
    explored[u] = 1
    for (u, v) ∈ E do
        if explored[v] == 0 then Search(v)
    end if
    end for
    postvisit(u)
\end{verbatim}

R\textbf{unning time} for DFS if previsit, postvisit take \(O(1)\) time?
Graph edges that do not belong to the DFS tree(s) may be

1. **forward**: from a vertex to a *descendant* (other than a *child*)

2. **back**: from a vertex to an *ancestor*
   
   Examples: edges (3, 1), (7, 5) in \( G \)

3. **cross**: from right to left (no ancestral relation), that is
   - from tree to tree (example: edge (5, 4) in \( G' \))
   - between nodes in the same tree but on different branches
Cross and forward edges do not exist in undirected graphs.

In undirected graphs, DFS only yields back and tree edges.
Subroutines \texttt{previsit}(u), \texttt{postvisit}(u) may be used to maintain a notion of \textit{time}:

- In \texttt{DFS}(G), initialize a counter \textit{time} to 0.
- Increment the counter by 1 every time \texttt{previsit}(u), \texttt{postvisit}(u) are accessed.
- Store the times \textit{start}(u) and \textit{finish}(u) corresponding to the first and last time \( u \) was visited during \texttt{DFS}(G).

\begin{align*}
\texttt{previsit}(u) & : \\
   \textit{time} &= \textit{time} + 1 \\
   \textit{start}(u) &= \textit{time} \\
\end{align*}

\begin{align*}
\texttt{postvisit}(u) & : \\
   \textit{time} &= \textit{time} + 1 \\
   \textit{finish}(u) &= \textit{time} \\
\end{align*}
If we use an explicit stack, then

- $\text{start}(u)$ is the time when $u$ is pushed in the stack
- $\text{finish}(u)$ is the time when $u$ is popped from the stack (that is, all of its neighbors have been explored).

1. How do intervals $[\text{start}(u), \text{finish}(u)]$, $[\text{start}(v), \text{finish}(v)]$ relate?

2. What do the contents of the stack correspond to in the DFS tree (and the graph), if $s$ was the first vertex pushed in the stack and $v$ the last?
1. Intervals $[\text{start}(u), \text{finish}(u)]$ and $[\text{start}(v), \text{finish}(v)]$
   - either contain each other ($u$ is an ancestor of $v$ or vice versa)
   - or they are disjoint.

2. If $s$ was the first vertex pushed in the stack and $v$ is the last, the vertices currently in the stack form an $s$-$v$ path.

Claim 3 (Back edges).

Let $(u, v) \in E$. Edge $(u, v)$ is a back edge in a DFS tree if and only if

$$\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v).$$
Proof of Claim 3 (identifying back edges)

Proof.

If \((u, v)\) is a back edge, the claim follows.

Otherwise, \(v\) was pushed in the stack before \(u\) and is still in the stack when \(u\) is pushed into it. Then there is a \(v - u\) path in the DFS tree, so \(v\) is an ancestor of \(u\) and \((u, v)\) is a back edge.
What conditions must the start and finish numbers satisfy if

1. \((u, v) \in E\) is a forward edge in the DFS tree?
2. \((u, v) \in E\) is a cross edge in the DFS tree?
What conditions must the start and finish numbers satisfy if

1. \((u, v) \in E\) is a forward edge in the DFS tree?
2. \((u, v) \in E\) is a cross edge in the DFS tree?

1. Edge \((u, v) \in E\) is a forward edge if

\[
\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u).
\]

2. Edge \((u, v) \in E\) is a cross edge if

\[
\text{start}(v) < \text{finish}(v) < \text{start}(u) < \text{finish}(u).
\]
1 Recap

2 Applications of BFS
   - Testing bipartiteness

3 Depth-first search (DFS)

4 Applications of DFS
   - Cycle detection
   - Topological sorting
Claim 4.

\[ G = (V, E) \text{ has a cycle if and only if } \text{DFS}(G) \text{ yields a back edge.} \]

Proof.

If \((u, v)\) is a back edge, together with the path on the DFS tree from \(v\) to \(u\), it forms a cycle.

Conversely, suppose \(G\) has a cycle. Let \(v\) be the first vertex from the cycle discovered by \(\text{DFS}(G)\). Let \((u, v)\) be the preceding edge in the cycle. Since there is a path from \(v\) to every vertex in the cycle, all vertices in the cycle are now discovered and fully explored before \(v\) is popped from the stack. Hence the interval of \(u\) is contained in the interval of \(v\).

By Claim 1, \((u, v)\) is a back edge.
An undirected acyclic graph has an extremely simple structure: it is a tree, hence a sparse graph ($O(n)$ edges).

A directed acyclic graph (DAG) may be dense ($\Omega(n^2)$ edges): e.g., $V = \{1, \ldots, n\}$, $E = \{(i, j) \text{ if } i < j \}$. 

![Diagram of a DAG](image)
Topological sorting: motivation

**Input:**
- a set of tasks \( \{1, 2, \ldots, n\} \) that need to be performed
- a set of dependencies, each of the form \((i, j)\), indicating that task \(i\) must be performed before task \(j\).

**Output:** a valid order in which the tasks may be performed, so that all dependencies are respected.

**Example:** tasks are courses and certain courses must be taken before others.

*How can we model this problem using a graph? What kind of graph must arise and why?*
Definition 2.

A topological ordering of $G$ is an ordering of its nodes as $1, 2, \ldots, n$ such that for every edge $(i, j)$, we have $i < j$.

- All edges point **forward** in the topological ordering.
- It provides an order in which all tasks can be safely performed: when we try to perform task $j$, all tasks required to precede it have already been done.
Example of DAG and its topological sorting

A DAG (top left), its topological sort (top right) and a drawing emphasizing the topological sort (bottom).
Claim 5.

*If* $G$ *has a topological ordering, then* $G$ *is a DAG.*

**Proof:** By contradiction (*exercise*).

A visual proof is provided by the linearized graph of the previous slide: vertices appear in increasing order, edges go from left to right, hence no cycles.

*Is the converse true: does every DAG have a topological ordering? And how can we find it?*
In a DAG, can every vertex have

- an outgoing edge?
- an incoming edge?

**Definition 3 (source and sink).**

A source is a node with no incoming edges.
A sink is a node with no outgoing edges.

**Fact 4.**

*Every DAG has at least one source and at least one sink.*
How can we use Fact 4 to find a topological order?

The node that we label *first* in the topological sorting must have no incoming edges. Fact 4 guarantees that such a node exists.

Fact 5.

*Let $G'$ be the graph after a source node and its adjacent edges have been removed. Then $G'$ is a DAG.*

**Proof:** removing edges from $G$ cannot yield a cycle!

This gives rise to a recursive algorithm for finding the topological order of a DAG. Its correctness can be shown by induction (use Facts 4, 5 to show induction step).
Algorithm for topological sorting

TopologicalOrder($G$)
1. Find a source vertex $s$ and order it first.
2. Delete $s$ and its adjacent edges from $G$; let $G'$ be the new graph.
3. TopologicalOrder($G'$)
4. Append the order found after $s$.

Running time: $O(n^2)$. Can be improved to $O(n + m)$. 
Let $G = (V, E)$ be a DAG.

- Run $\text{DFS}(G)$; compute $\text{finish}$ times.
- Process the tasks in decreasing order of $\text{finish}$ times.

Running time: $O(m + n)$
The task $v$ with the largest \textit{finish} has no incoming edges (if it had an incoming edge from some other task $u$, then $u$ would have the largest \textit{finish}). Hence $v$ does not depend on any other task and it is safe to perform it first.

The same reasoning shows that the task $w$ with the second largest \textit{finish} has no incoming edges from any other task except (maybe) task $v$. Hence it is safe to perform $w$ second.

And so on and so forth.
Formal proof of correctness

By Claim 4 there are no back edges in the DFS forest of a DAG. Thus every edge \((u, v) \in E\) is either

1. **forward/tree**: \(\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u)\)

2. or **cross** edge: \(\text{finish}(v) < \text{start}(u) < \text{finish}(u)\)
Hence for every \((u, v) \in E\), \(\text{finish}(v) < \text{finish}(u)\).

Consider a task \(v\). All tasks \(u\) upon which \(v\) depends, that is, all tasks \(u\) such that there is an edge \((u, v) \in E\), satisfy \(\text{finish}(v) < \text{finish}(u)\).

Since we are processing tasks in decreasing order of finish times, all tasks \(u\) upon which \(v\) depends have already been processed before we start processing \(v\).