Graphs
- Represent more complex relations (in contrast to, e.g., linear ordering) among data, for example.
  - Airline route map
  - Flowcharts
  - Computer networks

Problems on graphs
- Easy (running time linear in the number of vertices)
- Medium (polynomial)
- Hard (NP-complete)

Note: one of the fascinating aspects of graph problems is that very slight changes in the way a problem is formulated can often radically affect the problem’s difficulty level.

Definitions
- Graph (or undirected graph)
  - A pair \( G = (V, E) \) where \( V \) is a set whose elements are called vertices, and \( E \) is a set of unordered pairs of distinct elements of \( V \).
- Directed graph (or digraph)
  - A pair \( G = (V, E) \) where \( V \) is a set whose elements are called vertices, and \( E \) is a set of ordered pairs of distinct elements of \( V \).
- Weighted graph
  - A triple \( G = (V, E, W) \) where \( V \) is a graph (directed or undirected) and \( W \) is a function from \( E \) into the real numbers. For an edge \( e \), \( W(e) \) is called the weight of \( e \).
- Complete graph
  - There is an edge between each pair of vertices.

Definitions
- A path from \( v \) to \( w \) in a graph \( G = (V, E) \) is a sequence of edges \( v_0, v_1, v_2, ..., v_k = w \), such that \( v_0 = v \) and \( v_k = w \). The length of the path is \( k \). A simple path is a path such that \( v_0, v_1, ..., v_k \) are all distinct.
- Connected graph
  - A graph is connected if and only if, for each pair of vertices \( v \) and \( w \), there is a path from \( v \) to \( w \). If the graph is a digraph, then it is called strongly connected.

Representation
- Adjacency matrix
  - \( V = \{1, 2, ..., n\} \)
  - \( A[i, j] = 1 \), if \((i, j) \in E\)
  - \( 0 \), otherwise
  - Space usage: \( \Theta(V^2) \)

Representation
- Adjacency list
  - Space usage: \( \Theta(V + E) \)
Representation

- Adjacency matrix (weighted graphs)

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 10 & 14 & 6 & 8 \\
2 & 10 & 0 & 14 & 16 & 10 \\
3 & 14 & 16 & 0 & 10 & 14 \\
4 & 6 & 16 & 10 & 0 & 10 \\
5 & 8 & 10 & 14 & 16 & 0 \\
6 & 10 & 14 & 16 & 10 & 0 \\
7 & 8 & 10 & 14 & 16 & 10 \\
\end{array}
\]

Comparison Adjacency Lists and Adjacency Matrices

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster to test if ((x, y)) exists?</td>
<td>Matrices</td>
</tr>
<tr>
<td>Faster to find vertex degree?</td>
<td>lists</td>
</tr>
<tr>
<td>Less memory on small graphs?</td>
<td>lists ((V+E)) vs. (V^2)</td>
</tr>
<tr>
<td>Less memory on big graphs?</td>
<td>matrices</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>matrices (O(1))</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>lists (V+E) vs. (V^2)</td>
</tr>
<tr>
<td>Better for most problems?</td>
<td>lists</td>
</tr>
</tbody>
</table>

Traversing Graphs

- Systematic search of every edge and vertex of graph (directed or undirected)
  - Efficiency: each edge is visited no more than twice
  - Correctness: no vertex is missed

DFS

- Running time
  - For each vertex \(u\), DFS-visit is called
  - During DFS-visit\((u)\), loop on line 4-6 is executed \(|\text{Adj}[u]|\) times.
  - \(\sum_{u \in V} |\text{Adj}[u]| = O(E)\)
  - Therefore, the total running time of DFS is \(O(V+E)\)

DFS: Parenthesis Theorem

- Represent discovery of \(u\) with left parenthesis \("(u)\"
- Represent finishing \(u\) by right parenthesis \("u)""
- The history of discoveries and finishings makes a well-formed expression, i.e., the parentheses are properly nested.

Or more formally, for any two vertices \(u\) and \(v\), exactly one of the following three conditions holds:

- Interval \([\text{discovery}[u], \text{discovery}[v]]\) and \([\text{finish}[u], \text{finish}[v]]\) are entirely disjoint
- Interval \([\text{discovery}[u], \text{discovery}[v]]\) is contained entirely within \([\text{discovery}[v], \text{finish}[v]]\), and \(u\) is a descendant of \(v\) in the depth-first search tree
- Interval \([\text{discovery}[v], \text{finish}[v]]\) is contained entirely within \([\text{discovery}[u], \text{finish}[u]]\), and \(v\) is a descendant of \(u\) in the depth-first search tree.
White-path theorem: In any DFS of a graph \( G \), a vertex \( w \) is a descendant of a vertex \( v \) in a depth-first search tree if and only if, at the time vertex \( v \) is discovered (just before coloring it gray), there is a path in \( G \) from \( v \) to \( w \) consisting entirely of white vertices.

**Proof:**
- (Only if) If \( w \) is a descendant of \( v \), by the parenthesis theorem, the path of tree edges from \( v \) to \( w \) is a white path.
- (If) By induction on \( k \), the length of a white path from \( v \) to \( w \).

**Topological sort**

**Directed Acyclic Graph (DAG)**

Topological order of a dag \( G = (V, E) \) is
- a linear ordering of all vertices
- If \( G \) contains an edge \((u,v)\), then \( u \) appears before \( v \) in the ordering

**TopologicalSort(G)**

1. Call DFS\((G)\) to compute finishing times \( f[u] \) for each vertex \( u \).
2. As soon as each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices.
Theorem: A directed graph $G$ is acyclic if and only if a DFS of $G$ yields no back edges.

Proof:
- If DFS($G$) yields a back edge $(u,v)$, then there is a cycle in $G$. (Why?)
- Suppose $G$ has a cycle $c$. Edge $(u,v)$ is part of $c$, and $v$ is the first vertex on $c$ discovered by DFS($G$). All other vertices on $c$ form a white path from $v$ to $u$. By White path theorem, $u$ is a descendant of $v$, and $(u,v)$ becomes a back edge. (Because DFS-visited($v$) won’t return to $v$ until all reachable vertices are reached. When it reaches $u$, $(u,v)$ is a back edge.)

Running time:
- DFS part: $O(V + E)$
- Insert each of the $|V|$ vertices onto the front of the linked list: $O(V)$
- Total time: $O(V + E)$

Correctness:
- If $G$ is a dag, then for any edge $(u,v) \in E \rightarrow f[u] < f[v]$ when $(u,v)$ is explored, $u$ is GREY.
- 1. $v$ is RED. Then $(u,v)$ is a back edge. This means $G$ is not a dag.
- 2. $v$ is WHITE. Then $v$ is a descendant of $u$. Therefore $v$ is finished before $u$, namely $f[v] < f[u]$.
- 3. $v$ is BLACK. Then $v$ is already finished, i.e., $f[v] < f[u]$.

Example:

Strongly connected component

Definition: A SCC of a digraph $G = (V,E)$ is a maximal set of vertices $U \subseteq V$ such that every pair of vertices are reachable from each other in $G$ (why not just in $U$)?

Q: Is it possible that $u$ and $v$ are in a SCC and there are edges $(u,v)$ and $(v,u)$, but $x$ is not in the same SCC?

SCC($G$):
1. Call DFS($G$) to compute finishing times $f[u]$ for each vertex $u$; push $u$ onto $finish$ stack when it is finished.
2. Compute $G^T$.
3. Call DFS($G^T$), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ as computed in step 1.
4. Output vertices of each tree (in the depth-first forest of step 3) as a separate SCC.

Exercise: Condensation graph (or called component graph) is a dag.

Each tree yielded by DFS($G^T$) is a SCC.
**Strongly connected component**

Running time of SCC($G$):
- DFS($G$): $O(V + E)$
- Compute $G^T$: $O(V + E)$ (Exercise)
- Call DFS($G^T$): $O(V + E)$

Total time: $O(V + E)$

Space usage: $O(V + E)$ using adjacent list representation.

Correctness:
1. Each DF tree can contain one or more SCCs, never contains partial SCC. (DFS push a topological ordering of condensation graph of $G$ onto the finish stack)
2. In DFS($G$), each DF tree can contain only one SCC, because there is no edge to go to the next SCC. (all edges in condensation graph of $G$ are reversed)

**DFS on undirected graphs**

Similar to DFS on directed graphs
- No cross edges. Why?

Exercise: Design an algorithm to determine whether or not a given undirected graph contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

**Biconnected component**

- Biconnected component of an undirected graph: largest subgraph that remains connected after removal of any one vertex
- Articulation point: its removal disconnects the graph
- Theorem 7.13 In a depth-first search tree, a vertex $v$, other than the root, is an articulation point if and only if $v$ is not a leaf and some subtree of $v$ has no back edge incident with a proper ancestor of $v.$

**Breadth-first search**

- Starts from a source $s$.
- Explores every vertices reachable from $s$, in order of increasing distance from $s$:
  - Visits all vertices at distance $k$ from $s$ before discovering any vertices at distance $k + 1$.
- Three colors:
  - White: vertices undiscovered
  - Gray: vertices discovered but have white adjacent vertices
  - Black: vertices discovered and all neighbors also discovered.

**Biconnectivity**

- Biconnected graph: $G$ is biconnected if and only if $G$ has no articulation points.
- DFS in a biconnected graph:
  - When a back edge is found, continue the DFS from that vertex.
  - When a cross edge is found, do a DFS from that vertex.

**Breadth-First Search**

Computes link distance of all vertices from root $s$.
Use FIFO queue (enqueue at back, dequeue at front)

BFS($G$, $s$):
- for each vertex $u$ in $G$:
  - color[$u$] = WHITE
  - $u$ = null
- color[$s$] = GRAY
- color[$s$] = GRAY
- $Q$ = empty
- while $Q$ is non-empty:
  - do $u$ = dequeue($Q$)
  - for each $v$ in $Adj(u)$:
    - if color[$v$] = WHITE
      - color[$v$] = GRAY
      - color[$v$] = GRAY
      - enqueue($Q$, $v$)
    - if color[$v$] = BLACK
BFS example

Breadth-First Search

Running time:
- Initialize as WHITE: O(V)
- Each vertex is enqueued and dequeued once: O(1)
- V vertices: O(V)
- Adjacent list of each vertex is scanned only once: O(E)

Total time: O(V+E)