Lecture 32

Graphs IV

- Graph Traversal Revisited
  - Directed DFS and BFS
- Topological Sort
  - IKEA Furniture
- All-Pairs Shortest Paths
  - Floyd-Warshall Algorithm
Graph Traversal
Revisited
Exercise: Recursive Depth-First Search

Write a recursive implementation of depth-first search.

- Use pseudocode or Java syntax with the `structure` package.
- Don’t refer to your notes.

Notes:
- Visit the nodes as you find them.

Work by yourself for 3 minutes.
Java code from the textbook appears on the left, and pseudocode appears on the right.

- Both mark vertices as unvisited and then visited, although the pseudocode uses the terms undiscovered and discovered, and also marks them as processed (which is optional; it helps with some applications of DFS).
- Each recursive call examines a single vertex: n in the Java code; start in the pseudocode.
- Iterate over neighbors of the vertex: g.neighbors(n) in Java; neighbors(start) in pseudocode.
  - If a neighbor is unvisited, then recursively run the algorithm on that neighbor. These neighbors must also be marked as visited. Notice that the pseudocode does this before recursing, and the Java does it after.

The Java code adds the vertices to a list l in the order they are visited. This will be helpful later in these slides.
Directed DFS and BFS
Directed Graph Traversals

When working with directed graphs, it often makes sense to consider *directed graph traversals*. These traversals only follow edges in one orientation.

- **Outward depth-first search** follows outgoing edges.
- **Inward depth-first search** follows inward edges.

Breadth-first searches can also be directed. They reach the same vertices as directed DFS but in a different order.

Starting from vertex \( v \), outward searches visit every vertex \( w \) that is reachable by a directed path from \( v \) to \( w \).

Similarly, inward searches visit every vertex \( u \) in which there is a directed path from \( u \) to \( v \).

To implement these algorithms, we simply need to pay attention to the orientation of edges to the neighbors.

---

We can implement `outDFS` by only considering edges in their outward orientation. This can be done in a couple of different ways. (The `structure` package does the first way by default.)

```plaintext
function outDFS(G, start)
    for all u in outneighbors(start)
        if state[u] == "undiscovered" …
```

```plaintext
function outDFS(G, start)
    for all u in neighbors(start)
        if (start \(\rightarrow\) u) in edges(E) and state[u] == "undiscovered" …
```
Topological Sort
IKEA Furniture
IKEA Furniture

Have you ever had the following question when assembling IKEA furniture?

- Why didn’t they put this step earlier in the instructions?

Let’s think about this question using graphs.

- What will the vertices and edges be?
- What does a valid order of instructions correspond to in the graph?
We know that ①②③④⑤⑥ is a valid order. Which other orders are valid or invalid?

- How can you describe these orders?
- Computer Science: graph is a DAG (directed acyclic graph); valid orders are topological sorts.
- Mathematics: graph illustrates a partial order; valid orders are its linear extensions.
Topological Sort

Given a directed acyclic graph $G$, a topological sort is an order of its vertices in which the following is true:

If there is a directed path from $u$ to $v$ in $G$, then $v$ is earlier than $u$ in the order.

Notice that converse does not need to be true. That is, if $v$ is earlier than $u$ in a topological sort, then there isn’t necessarily a directed path from $u$ to $v$ in the graph $G$. In the example on the right there is no directed path from ① to ② or from ② to ①, but one of them must appear earlier than the other.

How can directed DFS help find a topological sort?

• If there is a directed path from $u$ to $v$, then we must place $v$ earlier in any topological sort.
• We can find all directed paths starting from $u$ by running outward DFS (or BFS) starting from $u$. Therefore, reached vertices should be ordered before $u$. Any unreached vertices can be ordered after $u$. Why?
The `topoSort` method runs (outward) DFS on all vertices of graph \( g \) via the \( g\text{.elements()} \) iterator. The vertices in \( g \) will be visited and added in order to a list \( l \) which stores the topological sort. Notice that vertices are skipped over by the `if` statement when they have already been visited and ordered in \( l \).

To understand how `topoSort` creates a topological sort, we need to look closely at the DFS method.

1. When run on a directed graph, the \( \text{neighbors} \) iterator only follows outgoing edges.
2. DFS adds the vertex (i.e, \( n \)) to the end of the list \( l \) after recursing on its (outward) neighbors.

These two points imply the following:

- If there is a directed path from \( n \) to another vertex \( v \), then \( v \) is placed earlier in \( l \).
All-Pairs Minimum-Weight Paths
All-Pairs Minimum Distance Paths (without Negative-Weight Cycles)

In this problem, we want to find the minimum-weight or distance paths between all pairs of vertices. Furthermore, negative-weight edges are allowed, although negative-weight cycles are not allowed.

- If the graph is directed, then interpret negative-weight cycle to be directed negative-weight cycle.
- If the graph is undirected, then a single negative weight edge counts as a negative-weight cycle.

Our single-source path algorithms from this course can be used to solve this problem.

- If the graph has no negative-weight edges, then Dijkstra's algorithm can be repeated n times.

That approach is slower than the $O(n^3)$-time algorithm known as Floyd-Warshall that we’ll look at.

---

### All-Pairs Minimum-Weight Paths without Negative-Weight Cycles

| Input: | A weighted graph $G = (V, E, w)$. There are no negative-weight cycles, i.e. each cycle $C$ has $\sum_{e \in C} w(C) \geq 0$. |
| Output: | Minimum-weight paths from $u$ to $v$, for all $u, v \in V$. Or the minimum-weight of a path from $u$ to $v$, for all $u, v \in V$. |

The graph can be directed or undirected.

Negative-weight edges are only allowed if $G$ is directed, and in that case negative-weight cycles are not allowed.

Example input and output.

<table>
<thead>
<tr>
<th>from</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>-3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

The shortest distance (i.e. minimum weight) from $b$ to $c$ is -1. The table only gives the minimum-weights and not the paths.
Floyd-Warshall for All-Pairs

The Floyd-Warshall algorithm sets initial shortest distances between pairs of vertices in $A$ (see the first two loops). The base cases use no waypoints (i.e., the paths have no intermediate nodes).

It then decreases these distances by adding to the set of allowable waypoints (see the outer $w$ loop).

- For each pair of vertices $u$ and $v$, it considers the old distance and a new distance obtained by using $w$ as a waypoint (see the inner loops).

The algorithm runs in $O(n^3)$-time, where $n = |V|$ is the number of vertices in the graph.

In CSCI 256 you will likely do the following:

- Prove that the algorithm works.
- Understand it as dynamic programming.
- Use the final distances to construct the paths.
- See how it can detect negative weight cycles.

\[
\text{function AllPairs}(G) \\
\text{// Fill in initial distances.} \\
\text{for } u \text{ in } V \\
\text{for } v \text{ in } V \\
\text{if } u == v \text{ then} \\
\quad A[u,v] = 0 \quad // \text{same vertex} \\
\text{else if } uv \in E \text{ then} \\
\quad A[u,v] = w(u,v) \quad // \text{uv edge} \\
\text{else if } uv \notin E \text{ then} \\
\quad A[u,v] = \infty \quad // \text{no uv edge} \\
\text{// Decrease distances by adding each} \\
\text{// vertex $w$ as an allowable waypoint.} \\
\text{for } w \text{ in } V \quad // w \text{ is the new waypoint} \\
\text{for } u \text{ in } V \\
\text{for } v \text{ in } V \\
\quad old = A[u,v] \quad // \text{not via } w \\
\quad new = A[u,w] + A[w,v] \quad // \text{via } w \\
\quad A[u,v] = \min(old, new) \\
\text{// return the shortest distances} \\
\text{return } A
\]