Graphs

July 17, 2018
Graph – Definitions

- **Graph** – a mathematical construction that describes objects and relations between them.
- A graph consists of a set of *vertices* and a set of *edges* that connect the vertices.
- \( G = (V, E) \) where \( V \) is the set of vertices (nodes) and \( E \) is the set of edges (arcs).
- In a *directed graph*, each edge is an ordered pair \((u, v)\) where \( u, v \in V \).
- In an *undirected graph*, each edge is a set \( \{u, v\} \).
- For *weighted graphs* (directed or undirected), each edge is associated with a weight \( W \).
- Vertex \( v \) is *adjacent* to vertex \( u \) if and only if \((u, v) \in E\) for a directed graph, or \( \{u, v\} \in E\) for an undirected graph.
A Directed Graph Example

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$$
$$E = \{(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10),
(V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2),
(V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1)\}$$
Definitions

- **Path**: a sequence of vertices $v_1, \ldots, v_n$ connected by edges such that \( \{v_i, v_{i+1}\} \in E \) for each $i = 1, \ldots, n$
- **Number of vertices**: $n$
- **Number of edges**: $m$
- **Path length**: the number of edges on the path
- **Weighted path length**: in a weighted graph, the sum of the costs of the edges on the path
- **Cycle**: a path that begins and ends at the same vertex and contains at least one edge
Graph Representation

- Use a 2-dimensional array called *adjacency matrix*, $a[u][v] = \text{edge cost}$
- Nonexistent edges initialized to $\infty$
- For *sparse* graphs, use an *adjacency list* that contains a list of adjacent indices and weights
## Adjacency Matrix or Adjacency List

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test if ((x, y)) is in graph?</td>
<td>adjacency matrix</td>
</tr>
<tr>
<td>Find the degree of a vertex?</td>
<td>adjacency list</td>
</tr>
<tr>
<td>Less memory on sparse graphs?</td>
<td>adjacency list (\Theta(m + n)) vs. (\Theta(n^2))</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>adjacency matrices (O(1)) vs. (O(d))</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>adjacency list (\Theta(m + n)) vs. (\Theta(n^2))</td>
</tr>
<tr>
<td>Better for most problems?</td>
<td>adjacency list</td>
</tr>
</tbody>
</table>
The most fundamental graph problem is to visit every edge and vertex in a graph in a systematic way.

Key idea: Mark each vertex when we first visit it, and keep track of what we have not completely explored.

Each vertex is in one of three states:

1. Undiscovered
2. Discovered: The vertex has been found, but we have not yet checked out all its incident edges
3. Processed: We have visited all its incident edges

A data structure is maintained to hold the vertices that we have discovered but not yet completely processed:

- A queue for BFS
- A stack for DFS
Initially, only the start vertex $s$ is considered to be discovered
- Put $s$ in the data structure

Remove a vertex $u$ from the data structure of discovered vertices

Inspect every edge incident upon $u$

If an edge leads to an undiscovered vertex $v$, mark $v$ as discovered and add it to the data structure

If an edge lead to a processed vertex, ignore this edge

If an edge leads to a discovered but not processed vertex, ignore this edge
Sometimes we want to visit all the adjacent nodes to some node before we visit other nodes. This is called Breadth-first search (BFS).

When searching an undirected graph by breadth-first, we assign a direction to each edge.

Vertex $u$ is the *parent* of vertex $v$.

The start vertex is the root of the search tree.

All other vertices have exactly one parent.
Implementation of BFS

- Associate with a Vertex object
  - A status data member, with possible values of undiscovered, discovered, and processed
  - A parent data member
- Initialize a queue to hold only the start vertex
- Mark the start vertex as discovered, with no parent
- Loop
  - Dequeue a vertex $u$, mark it as processed
  - Loop through the adjacency list of $u$ for each of the edges $(u, v)$
    - If $v$ is undiscovered, mark $v$ as discovered and enqueue, and make $u$ the parent of $v$
- $O(n + m)$ running time
BFS Example, starting at $V_2$

- visit $V_2$ (processed): Three outbound edges, to $V_0$, $V_3$ and $V_5$
- visit edge $V_2-V_0$ (tree edge, since $V_0$ unseen. Consider $V_0$ seen now, but not yet processed)
- visit edge $V_2-V_3$ (tree edge, $V_3$ is now discovered)
- visit edge $V_2-V_5$ (tree edge, $V_5$ is now discovered)
BFS Example

- visit $V_0$ (processed): Two outbound edges, to $V_1$ and $V_3$
- visit edge $V_0 - V_1$ (tree edge, $V_1$ now discovered)
- visit edge $V_0 - V_3$ (non-tree edge, since $V_3$ discovered before)
- visit $V_3$ (processed): 3 outbound edges, to $V_4$, $V_5$, and $V_6$
- visit edge $V_3 - V_4$ (tree edge)
- visit edge $V_3 - V_5$ (non-tree edge, since $V_4$ discovered before)
- visit edge $V_3 - V_6$ (tree edge)
- visit $V_5$ (...)
- ...

Mohammad Hadian
Advanced Data Structures and Algorithms
July 17, 2018 12 / 42
BFS Starting from $V_2$

Initialize: Mark first vertex as reachable with 0 edges

The graph after all the vertices whose path length from the starting vertex is 1 have been found
BFS Starting from $V_2$

The graph after all the vertices whose path length from the starting vertex is 2 have been found (final)
BFS Example

- We end up with a BFS with 2 levels, $V_2$ on the top level, $V_0$, $V_3$ and $V_5$ on the second level, $V_1$, $V_4$ and $V_6$ on the third level.
- We have a tree defined by the BFS using a subset of edges from the graph.
- The nodes at the second level are all one hop away from the source node $V_2$.
- The nodes on the third level are all 2 hops away from $V_2$. Thus we are finding how far, in hops, the various nodes are from $V_2$. 
The BFS tree gives the shortest paths from the root to all other vertices in unweighted graphs.

To print the shortest path from root to $x$:

- Start from $x$
- Follow the parent link by recursion, and print the vertex itself after recursion comes back
- Stop recursion when there is no parent (that is, the root)
A graph is *connected* if there is a path between any two vertices.

A *connected component* of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices.

To find connected components, do BFS and obtain one component, and then repeat with the remaining vertices until all vertices appear in a component.
Depth-First Search

- We start from a specific node (determined as a parameter) and go as deep as we can. When we can’t go any deeper, we backtrack and continue traversing the rest of the graph depth-first.
- Replace the queue (FIFO) in BFS by a stack (LIFO)
- Instead of using a real stack, we can just use the recursive calls where the runtime system maintains the call stacks
- An edge of the graph is either a DFS *tree edge* or a *back edge* linking to an *ancestor* vertex
- That is, an edge is either a forward edge or a backward edge
- \( O(n + m) \) runtime
Pseudocode for DFS

Set up Set of ‘‘unvisited’’ vertices, initially containing all the vertices.

for each vertex v
  if (v is in unvisited)
    dfs(v)

dfs(vertex v)
  remove v from unvisited set
  for each vertex n adjacent to v
    if (n is in unvisited)
      dfs(n)
Depth-First Search (DFS)

- DFS starting from $V_2$, etc. where we use the lower numbered adjacent vertex first. (We need some way of deciding which edge to visit first.)
- In implementations, we simply follow the order of elements on the adjacency list for the vertex.)
- See how we can follow out-edges down and down:

$V_2 \rightarrow V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6 \rightarrow V_5$
- We’re stuck with no place to go at $V_5$.
- We can backtrack back to $V_6$, back to $V_4$, back to $V_3$, see another adjacent vertex, $V_6$, but already visited, back to $V_1$, see another adj vertex, $V_4$, but already visited, back to $V_0$, see another adj vertex, $V_3$, but already visited, back to $V_2$, but $V_0$ and $V_5$ were already visited. Done.
- Notice that we don’t necessarily have to start from $V_2$. 

Depth-First Search (DFS)
Let’s keep a clock during DFS.

The clock ticks each time we enter or exit a vertex.

Each vertex has two additional data members: entry time and exit time.

If $u$ is an ancestor of $v$, the entry and exit times of $u$ properly encompass those of $v$.
  - Entering $u$ earlier than $v$.
  - Exiting $u$ later than $v$.

Half of the difference between the entry and exit times is the number of descendents in the DFS subtree.
We mark every node with two numbers. Whenever we make a step in the DFS we advance a clock by one step. The number on the left is the time step when we first encounter the node (entry time). The number on the right is the last time we encounter it (exit time).
Another DFS Example

For a simpler graph, consider the following:

```
A   D
\  /  \\
B  C
```

- All the edges point downward.
- The DFS starting from A visits A, then B, then backtracks and visits C, backtracks back to A and is done for one tree.
- It starts over at D, but finds no unvisited vertices, so D is alone as the second tree.
- So the DFS visitation order is A, B, C, D.
- Notify vertices in DFS visitation order
When going from $u$ to $v$

1. If $v$ is undiscovered, the edge $(u, v)$ becomes a tree edge
   - We make a recursive DFS call
2. If $v$ is discovered and the parent of $u$, we would be following the tree edge backwards to where we just came from
   - So don’t go there – just move on to the next edge in the adjacency list
3. If $v$ is discovered but not the parent of $u$, we have found a cycle
   - We can print the cycle using the parent links

The second case is a spurious two-vertex cycle $(v, u, v)$
During DFS on a directed graph, an edge \((u, v)\) is
1. Tree edge: parent\([v]\) is \(u\)
2. Back edge: discovered\([v]\) && !processed\([v]\)
3. Forward edge: processed\([v]\) && entryTime\([v]\) > entryTime\([u]\)
4. Cross edge: processed\([v]\) && entryTime\([v]\) < entryTime\([u]\)

During DFS of an undirected graph, every edge in the graph is either a tree edge or a back edge to an ancestor in the DFS tree.

The same DFS algorithm works for both undirected and directed graphs.

The graph has a cycle ⇔ DFS has a back edge.
A cycle in a directed graph is a path that returns to its starting vertex.

An acyclic directed graph is also called a DAG.

These graphs show up in lots of applications.

- Example, the graph of course prerequisites.

It is a DAG, since a cycle in prerequisites would be ridiculous.

Diagram:

```
CS110 ➔ CS210 ➔ CS310
  
CS240
```
A DAG induces a *partial order* on the nodes
- Not all element pairs have an order, but some do, and the ones that do must be consistent
- So CS110 < CS210 < CS310, and so CS110 < CS310, but CS210 and CS240 have no order between them
- Suppose a student takes only one course per term in CS
- A sequence that satisfies the partial order requirements, for example, is CS110, CS210, CS240, CS310
- Another possible sequence is CS110, CS240, CS210, CS310
- One of these fully ordered sequences that satisfy a partial order (DAG) is called a *topological sort* of the DAG
A topological sort orders the nodes such that if there is a path from $u$ to $v$, then $u$ will appear before $v$.

The vertices of the graph are ordered on a straight line such that all edges point from left to right.

Each DAG has at least one topological sort.

A topological sort gives us an ordering to process each vertex before any of its successors.
Topological Sort Example

CLRS 22.7
Use DFS to Find a Topological Sort

- Start DFS from a vertex with in-degree 0
- A back edge during DFS indicates that there is a loop
  - The graph is not DAG
- Ordering the vertices in the reverse order that they are processed (reverse order of the exit time) is a topological sort
- Consider the edge \((u, v)\) when we process the vertex \(u\)
  1. If \(v\) is undiscovered, we will start a recursive DFS from \(v\). So \(v\) will be completely processed before \(u\). So \(u\) will appear before \(v\) in the listing, as it must.
  2. If \(v\) is processed, and because \(u\) is not yet completely processed, \(u\) will appear before \(v\) in the listing, as it must.
  3. If \(v\) is discovered but not yet completely processed, then \((u, v)\) is a back edge, which is forbidden in a DAG.
DFS Example

There is only one topological sort of this graph: \((G, A, B, C, F, E, D)\)
Another Method to Find a Topological Sort

- The textbook presents a non-recursive algorithm for finding a topological sort of a DAG, checking that it really has no cycles.
- The first step of this algorithm is to determine the in-degree of all vertices in the graph.
- The *in-degree* of a vertex is the number of edges in the graph with this vertex as the destination-vertex.
- Once we have all the in-degrees for the vertices, we look for a vertex with in-degree 0.
- Because it has no incoming edges, it can be the vertex at the start of a topological sort, like CS110.
Finding a Topological Sort

- Note that there must be a node with in-degree 0
  - If there weren’t, then we could start a path anywhere, extend backwards along some in-edge from another vertex and from there to another, etc
  - Eventually we would have to start repeating vertices
- For example, if we have managed to avoid repeating vertices and have visited all the vertices, then the last vertex still has an in-edge not yet used, and it goes to another vertex, completing a cycle
- Thus the lack of an in-degree-0 vertex is a sure sign of a cycle and a DAG doesn’t have any cycles
- Now we have the very first vertex, but what about the rest? Think recursively!
Topological Sort Example

1. Graph with nodes v0, v1, v2, v3, v4, v5, v6.
The topological order is: \( V_2, V_0, V_1, V_3, V_4, V_6, V_5 \)
### Topological Sorting Using an Array of In-degree Values

<table>
<thead>
<tr>
<th></th>
<th>V₀</th>
<th>V₁</th>
<th>V₂</th>
<th>V₃</th>
<th>V₄</th>
<th>V₅</th>
<th>V₆</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>V₂</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>V₀</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>V₁</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>V₃</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>V₄</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>V₆</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V₅</td>
</tr>
</tbody>
</table>

**Diagram:**

```
V₀ → V₁ → V₂ → V₃ → V₄
V₂ → V₃ → V₅ → V₆
V₃ → V₄
```

Mohammad Hadian
Advanced Data Structures and Algorithms
July 17, 2018 37 / 42
1. Create a queue q and enqueue all vertices of in-degree 0
2. Create an empty list t for topologically-sorted vertices
3. Loop while q is not empty
   1. Dequeue from q a vertex u and append it to list t
   2. Loop over vertices v adjacent to u
   3. Decrement in-degree of v
   4. If v’s in-degree becomes 0, enqueue v in q
4. Return topologically-sorted list t
What Happens If There is a Cycle?

- The presence of a loop means in-degree contributions of one for all members of the cycle, so the cycle protects the whole group from being put on the queue.
- For example, consider $A \rightarrow B \rightarrow C \rightarrow A$ and see in-degree $= 1$ for all nodes.
- Each pass of the loop dequeues an element from the queue, so all that can happen is that the queue goes empty with the cycle members still un-enqueued.
- Use a counting trick to add cycle-detection to this algorithm.
Pseudocode for Topological Sort with Cycle Detection

1. Create a queue $q$ and enqueue all vertices of in-degree 0
2. Create an empty list $t$ for topologically-sorted vertices
3. Set $\text{loopcount}$ to 0
4. Loop while $q$ is not empty
   1. Increment $\text{loopcount}$
   2. Dequeue from $q$ a vertex $u$ and append it to list $t$
   3. Loop over vertices $v$ adjacent to $u$
   4. Decrement in-degree of $v$
   5. If $v$'s in-degree becomes 0, enqueue $v$ in $q$
5. If $\text{loopcount} < \text{number of nodes in graph}$, return cycle-in-graph
6. Return topologically-sorted list $t$
DFS – Some Comments

- We can do DFS of any directed (or undirected) graph, and if it’s acyclic, DFS yields a topological sort.
- If there is a cycle in the graph, it doesn’t cause an infinite loop because DFS doesn’t revisit a vertex.
- In general DFS works in phases, finding trees, so the whole thing finds a forest.
- The ability of DFS to turn a graph into a forest of trees is useful in many algorithms.
- Trees are a lot easier to work with than general graphs.
DFS versus BFS

- DFS is like preorder tree traversal, plunging further and away from the source node until we can’t go any further, then back
  - Can detect cycles
  - Can do topological sort of an acyclic graph
- BFS: explore nodes adjacent to the source node, then nodes adjacent to those (that haven’t been visited yet), and so on
  - Good for finding all neighbors, all neighbors of neighbors, etc. – hop counts