Main topics are #graph_basics, including #breadth-first_search and #depth-first_search.

Graph Review

- We’re about to begin a few weeks on graphs algorithms, so let’s begin with some review on definitions and graph traversals. I’ll stay light on proofs today, because you likely saw this stuff in your discrete structures courses.
- A graph $G = (V, E)$ is a set of vertices or nodes $V$ and edges $E$. If $G$ is undirected, each edge is a set of vertices, but I’ll write $uv$. If $G$ is directed, each edge is an ordered pair, but I’ll write $u \rightarrow v$.
- This definition does not allow for loops or parallel edges, meaning we must work with simple graphs. Most of the algorithms we talk about extend to multigraph with almost no change.
- If $uv$ is an edge, then $u$ is a neighbor of $v$ and vice versa.
- The degree of a vertex is the number of neighbors. If $u \rightarrow v$ is a directed edge, then $u$ is a predecessor of $v$ and $v$ is a successor of $u$.
- The in-degree of a vertex is the number of predecessors and the out-degree is the number of successors.

[start skipping here]

- A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V'$ subseteq $V$ and $E'$ subseteq $E$.
- A walk is a sequence of edges where each successive pair of edges share a vertex. A path is a walk that visits each vertex at most once.
- An undirected graph is connected if there is a walk between every pair of vertices. The components of a graph are its maximal connected subgraphs.
- A cycle is a walk that only repeats its first / last vertex and has at least one edge. A graph is acyclic or a forest if no subgraph is a cycle. A tree is a connected acyclic graph. A spanning tree of $G$ is a subgraph of $G$ that contains every vertex and is a tree. A spanning forest has one spanning tree per component of $G$.
- A directed graph is strongly connected if there is a directed walk between every ordered pair of vertices. A directed graph is acyclic if there is no directed cycle. I might say DAG to mean directed acyclic graph.
When describing graph algorithms, we may use \( V \) or \( E \) to represent the number of vertices or edges in the input graph, i.e., this algorithm runs in time \( O(V + E) \). Yes, this is weird, and I personally prefer using \( n \) and \( m \) outside the classroom. For some reason, both textbooks do this \( V \) and \( E \) thing, and writing \( |E| \) is tedious, so I guess we’ll go with the flow.

### Computer Representations

- So how do computers represent graphs?
- The two most common methods are the adjacency matrix and the adjacency list. The first lets you look up the existence of edges in constant time, but it uses \( \Theta(V^2) \) space.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<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>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- [skip during lecture] The adjacency matrix of \( G = (V, E) \) is a \( V \times V \) matrix where \( A[i, j] = 1 \) if \((i, j) \in E\) and 0 otherwise. If the graph is undirected, then \( A[i, j] = A[j, i] \) in every case. For simple graphs, meaning no loops, every diagonal entry \( A[i, i] = 0 \).
  - These use \( \Theta(V^2) \) space, so it’s only space-efficient for dense graphs.
  - But, we can decide in \( \Theta(1) \) time if two vertices are adjacent.
  - Listing all neighbors of a vertex means searching its whole row or column in \( \Theta(V) \) time.
- But generally, we’ll use the adjacency list of \( G \) which is an array with one linked-list per vertex \( v \), listing all of its neighbors. If \( G \) is directed, we store only successors of \( v \). So in an undirected graph, each edge \( uv \) appears in the lists for both \( u \) and \( v \). Directed edge \( u \rightarrow v \) appears exactly once and in \( u \)’s list.
  - The space used is only \( \Theta(V + E) \) so its space-efficient even for sparse graphs.
  - Listing the neighbors of vertex \( u \) takes \( O(1 + \deg(u)) \) time since we only need to scan \( v \)’s list.
  - But, we do need \( O(1 + \deg(u)) \) time to decide if edge \( u \rightarrow v \) exists or \( O(1 + \)
min\{\text{deg}(u), \text{deg}(v)\} \text{ time to find edge } uv \text{ if the graph is undirected. In most graph algorithms, we never need to ask if an edge exists, only list edges coming out of certain vertices of interest.}

**BFS**

- A common operation on graphs is to traverse or search its vertices and edges. In particular, we may ask if \( v \) is reachable from \( s \), meaning there is a (directed) path from \( s \) to \( v \).
- Probably the simplest traversal algorithm is the *breadth-first search* or BFS. It works by trying to visit vertices in order of their distance from \( s \). To avoid repeating work, we mark vertices we've seen. Initially all vertices are unmarked.
- Here's one way to implement it based on Erickson. The CLRS implementation is somewhat different but still visits the same vertices.

  - **BFS(s):**
    - put (emptyset, \( s \)) in a queue
    - while the queue is not empty
      - take (\( p, v \)) from the queue
      - if \( v \) is unmarked
        - mark \( v \)
        - parent(\( v \)) ← \( p \)
        - for each edge \( vw \)
          - put (\( v, w \)) into the queue
    - Using induction, we can prove several facts about BFS:
      1. It marks every vertex reachable from \( s \) exactly once.
      2. The set of pairs \((v, \text{parent}(v))\) form a spanning tree on the component of \( G \) containing \( s \), i.e., the set of vertices reachable from \( s \).
      3. The paths in this spanning tree are shortest paths from \( s \) to their endpoints, where every edge has length 1.
- I want to focus a bit more on the analysis of this algorithm, though.
- That for loop is executed once per marked vertex, so at most \( V \) times.
- Therefore, each edge \( vw \) is put into the bag at most twice, once as \((v, w)\) and once as \((w, v)\).
  - So we enqueue at most \( 2E \) times.
- And we can’t take more out of the queue than we put in, so we dequeue at most \( 2E + 1 \) times.
- Queue operations take \( O(1) \) time each, so the total running time is \( O(V + E) \).
- If you’re working with a directed graph, then you loop over edges leaving \( v \) and you get a spanning tree over vertices specifically reachable from \( s \).
- So remember, if you need to compute shortest paths on an unweighted graph, USE BFS; IT’S LINEAR TIME. Don’t use Dijkstra’s algorithm. It’s slower in that case.
Finally, Erickson remarks that this is just a special case of a generic graph search algorithm he calls WhateverFirstSearch. By replacing the queue with other data structures, you get other search algorithms, including Prim’s algorithm for minimum spanning times, Dijkstra’s algorithm for shortest paths, and a non-recursive version of depth-first search. We’ll cover all of those algorithms in the next two weeks, starting with depth-first search today and some applications next Tuesday.

**DFS**

Now, having said that about WhateverFirstSearch, probably the most common way to implement depth-first search or DFS it is to use recursion.

\[
\text{DFS}(v):
\begin{align*}
\text{mark } v \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{PreVisit}(v) \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{for each edge } vw \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{ } \text{if } w \text{ is unmarked} \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{ } \text{parent}(w) \leftarrow v \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{DFS}(w) \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{PostVisit}(v)
\end{align*}
\]

The PreVisit and PostVisit are placeholder functions for doing whatever extra work you want to do before and after fully processing a node.

We can extend this algorithm to mark all vertices in the graph by using a wrapper function.

\[
\text{DFSAll}(G):
\begin{align*}
\text{Preprocess}(G) \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{for all vertices } v \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{unmark } v \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{for all vertices } v \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{if } v \text{ is unmarked} \\
\text{ } & \text{ } \text{ } \text{ } \text{ } \text{DFS}(v)
\end{align*}
\]

Like before, PreProcess lets us do a bit of work before working with G.

We still mark each vertex once and therefore handle each directed edge once, so the running time is \(O(V + E)\).

**Preorder and Postorder**

The applications for DFS all come from the useful order in which it marks vertices.

To see that, let’s pass around a clock variable that increments every time we start or stop visiting a vertex.
We assign \(v.\text{pre}\) just after pushing \(v\) onto the recursion stack and assign \(v.\text{post}\) just before popping it from the stack.

- \(v.\text{pre}\) is often called the *starting time* of \(v\).
- \(v.\text{post}\) is often called the *finishing time* of \(v\).
- and \([v.\text{pre}, v.\text{post}]\) is called the *active interval* of \(v\).

So, because stack timelines are always disjoint or nested, \([u.\text{pre}, u.\text{post}]\) and \([v.\text{pre}, v.\text{post}]\) are either disjoint or nested. In fact, \([u.\text{pre}, u.\text{post}]\) contains \([v.\text{pre}, v.\text{post}]\) if and only if \(\text{DFS}(v)\) is called during the execution of \(\text{DFS}(u)\).

And because we only make recursive calls when there are edges, there must be a directed path from \(u\) to \(v\) in this case. In particular, the set of vertices on the recursion stack form a directed path in \(G\).

Here’s an example of a depth-first search.

![Depth-first search example](image)

Similiar to rooted trees, we can use the \(v.\text{pre}\) labels to get a *preordering* of the vertices “abfgchdlokpeinjm” in that order, and the \(v.\text{post}\) labels to get a *postordering* “dkohlchgbamjnie” in that order.

### Classifying Vertices and Edges

- So let’s say we’re in the middle of running a depth-first search. We can learn a lot about the structure of the graph by using this clock variable.
- Eventually, the algorithm will populate \(v.\text{pre}\) and \(v.\text{post}\) for every vertex \(v\).
- But suppose we’re midway through running DFS. Fix a vertex \(v\) and its eventual pre and post values. But consider the clock at the moment we pause the algorithm. \(v\) is
  - *new* if \(\text{clock} < v.\text{pre}\) (\(\text{DFS}(v)\) has not yet been called)
  - *active* if \(v.\text{pre} \leq \text{clock} < v.\text{post}\) (\(\text{DFS}(v)\) has been called but not yet returned)
  - *finished* if \(v.\text{post} \leq \text{clock}\) (\(\text{DFS}(v)\) has returned)
• Being active corresponds to a vertex being on the recursion stack. That means the active vertices form a directed path in G.

• In turn, using these definitions, we can partition the edges into four classes depending on how they interact with the depth-first search forest. Unlike vertices, these classes apply to an entire run of DFS, not a particular moment in time during the run. Consider edge \( u \rightarrow v \) and the moment when DFS\((u)\) begins.

  • If \( v \) is new, then either we call DFS\((v)\) directly when we iterate over \( u \rightarrow v \), or another intermediate recursive call will mark \( v \) first. Either way, \( u.\text{pre} < v.\text{pre} < v.\text{post} < u.\text{post} \).
    • If DFS\((u)\) calls DFS\((v)\) directly, \( u \rightarrow v \) is called a tree edge.
    • Otherwise, \( u \rightarrow v \) is called a forward edge.

  • If \( v \) is active, then \( v \) is on the stack, so \( v.\text{pre} < u.\text{pre} < u.\text{post} < v.\text{post} \). G has a directed path from \( v \) to \( u \).
    • \( u \rightarrow v \) is called a back edge.

  • If \( v \) is finished, then \( v.\text{post} < u.\text{pre} \).
    • \( u \rightarrow v \) is called a cross edge.

  • Note that \( u.\text{post} < v.\text{pre} \) cannot happen, because we would add \( v \) to the stack before finishing with \( u \).

• Again, this classification of edges depends upon the specific depth-first search tree we get, which depends upon the order in which we iterate over vertices and edges.

• Next time, we look at some applications of depth-first search, including a return to our old friend dynamic programming.