Recitation 10

Depth-First Search

A breadth-first search discovers vertices reachable from a queried vertex s level-by-level outward from s. A **depth-first search** (DFS) also finds all vertices reachable from s, but does so by searching undiscovered vertices as deep as possible before exploring other branches. Instead of exploring all neighbors of s one after another as in a breadth-first search, depth-first searches as far as possible from the first neighbor of s before searching any other neighbor of s. Just as with breadth-first search, depth-first search returns a set of parent pointers for vertices reachable from s in the order the search discovered them, together forming a **DFS tree**. However, unlike a BFS tree, a DFS tree will not represent shortest paths in an unweighted graph. (Additionally, DFS returns an order on vertices discovered which will be discussed later.) Below is Python code implementing a recursive depth-first search for a graph represented using index-labeled adjacency lists.

```
1  def dfs(Adj, s, parent = None, order = None): # Adj: adjacency list, s: start
2      if parent is None:                        # O(1) initialize parent list
3          parent = [None for v in Adj]          # O(V) (use hash if unlabeled)
4          parent[s] = s                         # O(1) root
5          order = []                            # O(1) initialize order array
6      for v in Adj[s]:                          # O(Adj[s]) loop over neighbors
7          if parent[v] is None:                 # O(1) parent not yet assigned
8              parent[v] = s                     # O(1) assign parent
9              dfs(Adj, v, parent, order)        # Recursive call
10     order.append(s)                           # O(1) amortized
11     return parent, order
```

How fast is depth-first search? A recursive `dfs` call is performed only when a vertex does not have a parent pointer, and is given a parent pointer immediately before the recursive call. Thus `dfs` is called on each vertex at most once. Further, the amount of work done by each recursive search from vertex v is proportional to the out-degree $deg(v)$ of v. Thus, the amount of work done by depth-first search is $O(\sum_{v \in V} deg(v)) = O(|E|)$. Because the parent array returned has length $|V|$, depth-first search runs in $O(|V| + |E|)$ time.

Exercise: Describe a graph on n vertices for which BFS and DFS would first visit vertices in the same order.

Solution: Many possible solutions. Two solutions are a chain of vertices from v, or a star graph with an edge from v to every other vertex.

Full Graph Exploration

Of course not all vertices in a graph may be reachable from a query vertex s. To search all vertices in a graph, one can use depth-first search (or breadth-first search) to explore each connected component in the graph by performing a search from each vertex in the graph that has not yet been discovered by the search. Such a search is conceptually equivalent to adding an auxiliary vertex with an outgoing edge to every vertex in the graph and then running breadth-first or depth-first search from the added vertex. Python code searching an entire graph via depth-first search is given below.

```
def full_dfs(Adj):                          # Adj: adjacency list
    parent = [None for v in Adj]            # O(V) (use hash if unlabeled)
    order = []                              # O(1) initialize order list
    for v in range(len(Adj)):               # O(V) loop over vertices
        if parent[v] is None:               # O(1) parent not yet assigned
            parent[v] = v                   # O(1) assign self as parent (a root)
            dfs(Adj, v, parent, order)      # DFS from v (BFS can also be used)
    return parent, order
```

For historical reasons (primarily for its connection to topological sorting as discussed later) **depth-first search** is often used to refer to both a method to search a graph from a specific vertex, **and** as a method to search an entire (as in `graph_explore`). You may do the same when answering problems in this class.

DFS Edge Classification

To help prove things about depth-first search, it can be useful to classify the edges of a graph in relation to a depth-first search tree. Consider a graph edge from vertex u to v. We call the edge a **tree edge** if the edge is part of the DFS tree (i.e. `parent[v] = u`). Otherwise, the edge from u to v is not a tree edge, and is either a **back edge**, **forward edge**, or **cross edge** depending respectively on whether: u is a descendant of v, v is a descendant of u, or neither are descendants of each other, in the DFS tree.

Exercise: Draw a graph, run DFS from a vertex, and classify each edge relative to the DFS tree. Show that forward and cross edges cannot occur when running DFS on an undirected graph.

Exercise: How can you identify back edges computationally?

Solution: While performing a depth-first search, keep track of the set of ancestors of each vertex in the DFS tree during the search (in a direct access array or a hash table). When processing neighbor v of s in `dfs(Adj, s)`, if v is an ancestor of s, then (s, v) is a back edge, and certifies a cycle in the graph.

Topological Sort

A directed graph containing no directed cycle is called a **directed acyclic graph** or a DAG. A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of the vertices such that for each edge (u, v) in E, vertex u appears before vertex v in the ordering. In the `dfs` function, vertices are added to the `order` list in the order in which their recursive DFS call finishes. If the graph is acyclic, the order returned by `dfs` (or `graph_search`) is the **reverse** of a topological sort order. Proof by cases. One of `dfs(u)` or `dfs(v)` is called first. If `dfs(u)` was called before `dfs(v)`, `dfs(v)` will start and end before `dfs(u)` completes, so v will appear before u in `order`. Alternatively, if `dfs(v)` was called before `dfs(u)`, `dfs(u)` cannot be called before `dfs(v)` completes, or else a path from v to u would exist, contradicting that the graph is acyclic; so v will be added to `order` before vertex u. Reversing the order returned by DFS will then represent a topological sort order on the vertices.

Exercise: A high school contains many student organization, each with its own hierarchical structure. For example, the school's newspaper has an editor-in-chief who oversees all students contributing to the newspaper, including a food-editor who oversees only students writing about school food. The high school's principal needs to line students up to receive diplomas at graduation, and wants to recognize student leaders by giving a diploma to student a before student b whenever a oversees b in any student organization. Help the principal determine an order to give out diplomas that respects student organization hierarchy, or prove to the principal that no such order exists.

Solution: Construct a graph with one vertex per student, and a directed edge from student a to b if student a oversees student b in some student organization. If this graph contains a cycle, the principal is out of luck. Otherwise, a topological sort of the students according to this graph will satisfy the principal's request. Run DFS on the graph (exploring the whole graph as in `graph_explore`) to obtain an order of DFS vertex finishing times in $O(|V| + |E|)$ time. While performing the DFS, keep track of the ancestors of each vertex in the DFS tree, and evaluate if each new edge processed is a back edge. If a back edge is found from vertex u to v, follow parent pointers back to v from u to obtain a directed cycle in the graph to prove to the principal that no such order exists. Otherwise, if no cycle is found, the graph is acyclic and the order returned by DFS is the reverse of a topological sort, which may then be returned to the principal.

We've made a CoffeeScript graph search visualizer which you can find here:
https://codepen.io/mit6006/pen/dqeKEN

6.006 Introduction to Algorithms

Spring 2020