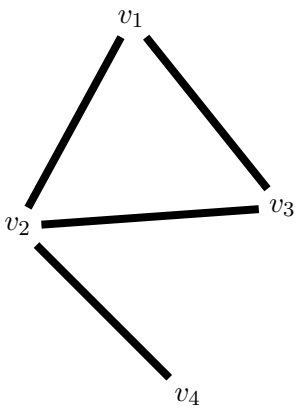
CS 3510 Algorithms Last updated: September 22, 2023

Lecture 6: Depth-First Search and Strongly Connected Components

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

1 Graphs

A graph is defined as a set of vertices and a set of edges donated by $G = (V, E)$. For instance, on the graph on the right, there are four vertices $V = v_1, v_2, v_3, v_4$. There are also edges $E = (1, 2), (1, 3), (2, 3), (2, 4)$; edges can be directed and undirected. In an undirected graph, each edge connects edges both ways, while in a directed graph, the direction of the edge matters; for instance, we could have an edge from v_1 to v_2, but not from v_2 to v_1. [a] Both can be useful in developing algorithms and solving real-world problems.



[a] Trivially you can represent any undirected graph as a directed graph by generating two edges for each edge in the undirected graph going in either directions.

1.1 Representation

We have two main choices in how to represent graphs like these. The first is called an adjacency matrix; this is a square matrix of size $|V| \times |V|$. We associate each element of the matrix as an edge in the graph. The above graph could be represented as the following matrix

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

where an entry $M_{i,j} = 1$ if v_i is neighbors with v_j, and 0 otherwise.

Notice here, that there are a lot of zeros. This format can be wasteful for graphs without a lot of edges. Graphs without many edges are considered sparse and are opposed to dense graphs which have many edges. We say a graph is dense if $|E| \approx |V|^2$, and that a graph is sparse if it is not dense. The following is an example of a dense graph

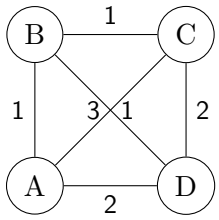


Figure 1: A weighted graph.

This would be represented by the following matrix

$$M = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 3 \\ 1 & 1 & 0 & 2 \\ 2 & 3 & 2 & 0 \end{pmatrix}$$

This matrix introduces the notion that weights of edges can be used in the adjacency matrix.

Let us come back to this issue of representation for sparse graphs. we could simply store the edges instead of all possible vertex pair combinations using lists. To do this let's collect each vertex with a list of all the vertices where there is an edge from some vertex. Look at the below for instance

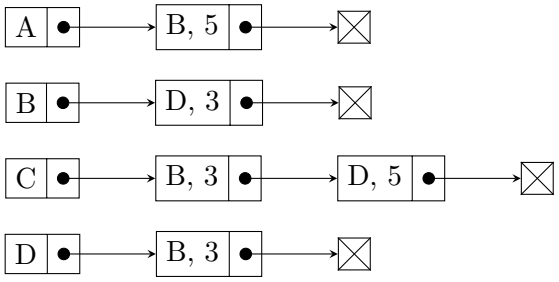


Figure 2: The adjacency list representation of a graph.

With the above representation we have a slightly slower lookup, but our space is exactly $\mathcal{O}(|V| + |E|)$. We can actually make this runtime better though by using maps with $\Omega(1)$ lookups, then we can get lookup to be $\Omega(1)$ instead of linear time.

2 Graph Algorithms

Sometimes we may want to ask questions about the graph, such as

1. Is the graph connected?
2. What is the shortest path between two vertices?
   (a) Shortest path by number of vertices?
   (b) Shortest path by lowest sum of weights of edges?

(c) Shortest path that contains all vertices (traveling salesman)?

3. Maximum sized matching in a graph?

A natural question to ask about graph algorithms is to ask which of these problems can be solved in poly(n) time. If these is, how fast can we make the algorithm? And if not, how well can we approximate the solution?

2.1 Traversal

We'll start with the easiest problem: graph traversal. Say we're at one vertex in a graph, and we want to know all the graphs we can reach. For instance, we can ask which vertices can and can not reached in the following graph:

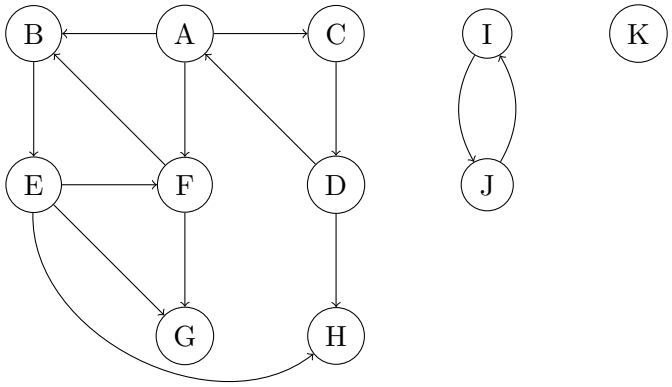


Figure 3: An example graph for explore.

To solve this problem, we will define a function called "explore(v)", which will use recursion to take advantage of the call stack. Our algorithm relies on some simple logic. Specifically, if you're at a vertex v, and there's an edge to vertex u, then all nodes that u can reach must also be reachable from v by simply prepending (v, u) to the path. Then, our subproblems will be the vertices that some vertex u is connected to. We'll use an auxillary data structure marked to keep track of which vertices we've been to as to not create an cyclic recurssion. Then we come to the following explore algorithm.

Now if you run this algorithm on vertex A, you'll see all vertices except vertices I, J, and K. If you want to see all the vertices you could simply loop over all vertices and run explore on each of them except the ones that have been seen by a previous explore call.

For a non-recursive implementation of explore, we can consider using a stack to determine which vertex to visit next. First, push the node that you would like to visit first onto the stack. While the stack is not empty, pop off the first element in the stack; call this v. Push all of the neighbors of v that are unmarked into the stack (remember, this pushes them to the top of the stack, in a LIFO order); mark these neighbors as having been seen. Repeat this process as necessary until the stack is empty, implying that you have seen all nodes possible.

Algorithm 1 Explore

```
1: procedure EXPLORE(G, v)
2:     marked[v] = true
3:     pre(v)
4:     for edge(v, u) in G connected to v do
5:         if not marked[u] then
6:             explore(G, u)
7:         end if
8:     end for
9:     post(v)
10: end procedure
```

2.2 Depth-First Search

We can also use "Depth-First Search", or DFS, to traverse a graph. Consider the following algorithm, which calls explore on each component of the graph:

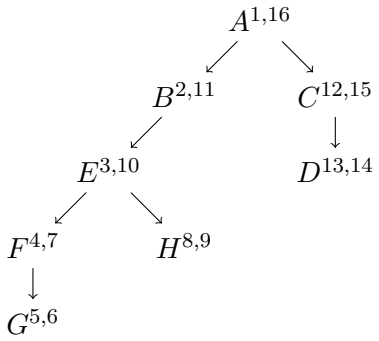
Algorithm 2 Depth-First Search

```
1: procedure DFS(G = (E, V))
2:     for v in V do
3:         if not marked[u] then
4:             explore(G, u)
5:         end if
6:     end for
7: end procedure
```

In the process of running DFS, we can also create a DFS tree which allows us to describe the relationship of vertices in the call stack. Recall our non-recursive definition of explore. Each time a vertex u is popped, some (if already marked) or all of the neighbors of u are added to the stack. If v was added into the stack by u, then we can say that v was visited by u. Since each vertex, if visited, must have been visited from exactly 1 other vertex, we can map this relationship in a tree for the main component in Figure 3, as shown below.

We have also added two numbers corresponding to the "pre" and "post" labels of each vertex ("pre" meaning before the visit, "post" meaning after the visit). The idea here is that a counter is incremented each time a vertex is first added to the stack ("seen"), and incremented again when the vertex is removed from the stack ("visited"). This allows us to get an idea of the relative ordering of visitation on the vertices.

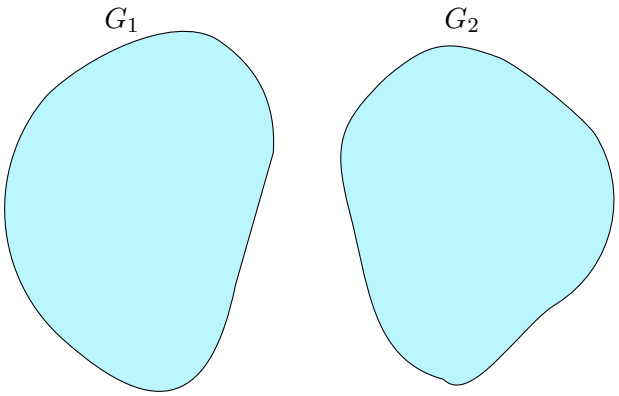
It turns out that these pre- and post- labels can be very useful as a building block for many algorithms. In linear time on $|E| + |V|$, information about these labels can guarantee us certain desirable properties of nodes.



Let's analyze the running time of DFS. We claim that DFS has a running time of $\mathcal{O}(m+n)$ where n is the number of vertices and m is the number of edges.

To see why, we see that each vertex is visited at most one time, and there are only n vertices (a vertex may not be visited if the graph is not connected). Similarly, each edge can be visited at most twice (once from each endpoint), and there are only m vertices. Thus, it is easy to see that DFS has a running time of $\mathcal{O}(m+n)$.

2.3 Connected Components



In this example on the left, we have two components–maximally connected subgraphs, and we may be asked to find the number of connected components and the number of vertices in each component.

If we use the above graph traversal algorithm in the context of the connected components problem, we can see that we can use this to determine whether or not the graph is connected. Say that we have a graph G with some $v_0 \in V(G)$. If we call $\text{DFS}(G, v_0)$, then if the returned array is all true, then G is connected.

Proof sketch. ($\Rightarrow$) If the returned array is all true, then all nodes are reachable from v_0, so it is connected by definition. ($\Leftarrow$) If G is connected, by definition, there must exist a path between every two pairs of vertices in G. Thus it must be reachable from v_0.

Another way to prove this is by contradiction. Suppose there is a node u which is reachable from v, but the DFS algorithm returned false. Why is this not possible?

To see why, construct a path $v, x_1, x_2, \ldots, x_n, u$ from v to u. There must exist some "transition" edge such that x_i is true but x_{i+1} is false. By DFS, when x_i is called, x_{i+1} will not be called if and only if it is already visited and marked as true. But x_{i+1} is false, which is a contradiction.