601.433/633 Introduction to Algorithms **Lecturer:** Michael Dinitz
Topic: Basic Graph Algorithms **Date:** 10/12/21

13.1 Introduction

Graphs are an incredibly important abstraction in computer science, so there has been a huge amount of work on algorithms for graphs. We'll spend much of the rest of the semester studying graph algorithms. Today we'll begin by talking about a few of the simplest, oldest, and (possibly) most useful algorithms.

Everyone should know what a graph is already, but in case you don't: a graph is a pair $G = (V, E)$ where V is a set of elements (usually called *vertices* or *nodes*) and $E \subseteq \binom{V}{2}$ is a set of edges. The notation $\binom{V}{2}$ means that the edges are unordered or *undirected*: the edge $\{u, v\}$ is the same as the edge $\{v, u\}$. We will sometimes talk about directed graphs, in which $E \subseteq V \times V$ and the edge (u, v) is different from the edge (v, u).

It is reasonably standard to let $n = |V|$ and $m = |E|$, so I'm going to use that notation throughout.

13.2 Graph Representations

The first obvious question is: how do we represent a graph? What's the actual data structure we're going to use? There are many possibilities, but there are two classical answers: adjacency matrices and adjacency lists. Unless otherwise specified, we're going to assume that we use adjacency lists.

Adjacency Matrix: We have an $n \times n$ matrix A. We set $A_{ij} = 1$ if $\{i, j\} \in E$ and set $A_{ij} = 0$ otherwise.

Major benefits: can check whether an edge exists in constant time. It also turns out this matrix has really, really nice properties as a matrix. Obviously it is symmetric, but studying its eigenvalues and eigenvectors can actually tell us a huge amount about the graph. There's a whole area of graph theory called *spectral graph theory* which studies this.

Major drawbacks: takes $\Theta(n^2)$ space even if m is small. Iterating over the edges incident on a vertex v takes time $\Omega(n)$ even if there are only a few such edges (need to scan a whole row or column of the matrix).

Adjacency List: We have an array A of length n, and $A[v]$ is a pointer to a linked list which contains the edges incident on v. In other words, $A[v]$ is a linked list of the vertices that are adjacent to v.

Major benefits: only takes $O(n + m)$ space, can iterate over edges very efficiently.

Major drawbacks: takes a long time to check if $\{u, v\}$ is an edge (time $O(d(u))$ or $O(d(v))$, where $d(u)$ denotes the degree of u).

Adjacency Data Structures: Here's an obvious idea: let's replace the adjacency list $A[v]$ with something smarter than an adjacency list! For example, we could store all of the edges that were in the list in a red-black tree or a hash table (or, if the graph doesn't change, a two-level hash table using perfect hashing). This is basically a good idea, but we're not going to do it: lists are more traditional, and the algorithms that we're going to talk about don't benefit from being stored using fancier data structures. But it's a good option to keep in mind for the future.

13.3 Breadth-First Search

BFS is a way of traversing or searching a graph. It is one of the simplest and most obvious ways of doing this, and has a few nice properties.

Pseudocode is in CLRS – I'd suggest looking at it. But basically we start out from some node v. We mark it as complete, and put all of its neighbors in a queue (first-in first-out), marking them all as "in process". We then pull from the queue, mark that node as complete, and put its neighbors (that are not already in process or complete) in the queue. The key point is that we only put a node in the queue if it is not already in process or complete. This means that we essentially search by levels from v. We first see all of the neighbors of v, then we see all vertices at distance 2 from v, then all vertices at distance 3 from v, etc.

The running time is $O(n + m)$. We need to spend $\Theta(n)$ time on initialization, e.g. setting the "completed" and "in process" field of each node to 0. Once we've started the algorithm, note that the adjacency list of a vertex is scanned only when that vertex is pulled from the queue. Each node is pulled from the queue only once, so each adjacency list is scanned only once, so the time is only $O(m)$. Thus the total running time is $O(n + m)$.

A very nice feature of BFS is that it gives *shortest paths*. This is proved formally in the book, but let's give an informal argument here. Suppose we start at some node v. We claim that for any node u, the path in the BFS tree from v to u is the shortest possible path.

Let's do an informal proof by contradiction. Let $d(v, x)$ denote the length of the shortest path from v to x, for every $x \in V$. Suppose that u is the closest node to v in which the BFS does not return the shortest path. Let w' be the node just before u on the shortest path, and let w be the node just before u on the BFS path. If $w = w'$ then we are done – w' is closer to v than u is, so by assumption the BFS tree has the shortest path to w, and so it also has the shortest path to u. If $w \neq w'$, then we have two cases. If $d(v, w) = d(v, w')$ then we are done, since this means the BFS has a shortest path to u. Otherwise $d(v, w') < d(v, w)$. But this means that the BFS would have found u from w' before it found u from w, giving a contradiction.

13.4 Depth-First Search

DFS is the other obvious way to explore a graph. Instead of going by breadth, we go by depth. Again, pseudocode is in the book, but we're going to spend a little more time on DFS since it will be useful in some of the algorithms we'll talk about later in the lecture. DFS is super easy to implement recursively: when we call DFS on a node v, we iterate through its neighbors and for any unmarked neighbor u we mark it and then call DFS on u. It can also be implemented iteratively using a stack rather than a queue.

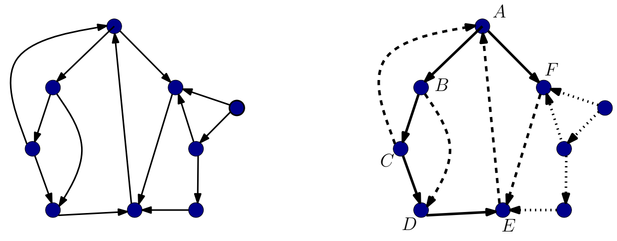
The basic pseudocode would look something like this. Let's assume that G is directed, since it's a more general setting and will be useful for the next few algorithms anyway.

```
Init: for each v in V, mark(v) = False

DFS(v) {
  mark(v) = True
  for each edge (v,u) in A[v] {
    if mark(u) == False then DFS(u)
  }
}
```

DFS also takes time $O(n+m)$. We again have to spend $\Theta(n)$ time initializing the graph, and then since we only visit each vertex once we only scan each adjacency list once, so we spend time $O(m)$ in the main algorithm.

Here's an example. The labeled nodes are the ones visited by calling DFS(A), the the solid edges are traversed by the algorithm, the dashed edges were considered but not traversed, and the dotted edges were not even looked at.



Let's prove a simple property about DFS, which is obvious (and you should know from Data Structures) but is a good warmup proof.

Definition 13.4.1 *A node u is* reachable *from v is there is a path $v = v_0, v_1, \ldots, v_k = u$ such that each (v_i, v_{i+1}) is an edge of G.*

Theorem 13.4.2 *When DFS(v) terminates, it has visited (marked) all of the nodes that are reachable from v.*

Proof: We first prove that DFS(v) terminates. This is obvious because every call to DFS is to an unmarked node, and each such call immediately marks the node. Thus there can be at most n calls to DFS, so it terminates.

Now suppose that u is reachable from v but is not marked when DFS(v) terminates. Since u is reachable from v, there is a path $v = v_0, v_1, \ldots, v_k = u$ from v to u, and since v is marked but u is not there is some node v_i on this path which is the first unmarked node. But this is a contradiction, since when we examined examined the adjacency list of v_{i-1} we would have seen that v_i is unmarked and we would have recursively called DFS(v_i) and marked it. ■

Of course, it might not be possible to reach all nodes in the graph from our starting node, so it's pretty common to simply call DFS on unmarked vertices until everything has been marked:

```
DFS(G) {
  for all v in V, set mark(v) = false
  while there exists an unmarked node v
    DFS(v)
}
```

DFS naturally comes with a notion of timestamps. We can define start(v) to be the "time" at which we began processing v, and we can define finish(v) to be the "time" at which we finished processing v. We can modify our algorithm to keep track of these as follows.

```
DFS(G) {
  i = 0
  for all v in V {
    start(v) = 0
    end(v) = 0
  }

  for all v in V if start(v) == 0 then DFS(v)
}

DFS(v) {
  i = i+1
  start(v) = i
  for each edge (v,u) in A[v] {
    if start(u) == 0 then DFS(u)
  }
  i = i+1
  finish(v) = i
}
```

This is clearly the same algorithm, it just keeps track of start and finish times, which are useful pieces of data when we try to reason about DFS and its uses.

13.4.1 Edge types

Define a *tree edge* to be an edge (v, u) such that DFS(v) calls DFS(u), i.e. the edge is traversed by the algorithm. It is easy to see that the tree edges actually form a collection of trees such that every node is in exactly one tree, i.e. they form a *spanning forest*. This is sometimes called the depth-first forest (or tree) of G. Moreover, every tree is rooted at the node of the tree with minimum start time.

This DFS forest naturally lets us partition the edges into three types:

1. Forward edges are edges (v, u) where u is a descendent of v in this tree. This includes tree edges. Note that in any forward edge, $\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v)$.
2. Back edges are edges (v, u) where u is an ancestor of v. Note that in any back edge, $\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u)$.
3. Cross edges are edges (v, u) where u is neither an ancestor nor a descendent of v. In this case $\text{start}(u) < \text{finish}(u) < \text{start}(v) < \text{finish}(v)$.

A good exercise to do at home is to formally prove these relationships between the start and finishing times, but since they're reasonably intuitive we're just going to trust it for now (but you should know how to prove them!).

13.5 Topological Sort

DFS has a number of nice applications, some of which are discussed in the book. One of its most famous applications is doing a "topological sort" on a directed acyclic graph (DAG). A DAG is a directed graph in which there are no directed cycles, although if we reinterpret edges as undirected there might be cycles. A topological sort of a DAG is an ordering of the vertices $v_1, \ldots, v_n$ so that all edges are of the form (v_i, v_j) where $i < j$. In other words, we can line up all of the nodes so that there are no edges going backwards.

It turns out that we can use DFS to find topological sorts of DAGs. First, let's use DFS to characterize DAGs

Theorem 13.5.1 *A directed graph G is a DAG if and only if DFS(G) has no back edges.*

Proof: One direction is obvious: if there is a back edge then we clearly have a directed cycle. For the other direction, suppose that there is a directed cycle C in G. Then consider the node $u \in C$ with minimum start value. Since all nodes in C are reachable from u, they will be descendants of u and so any $v \in C$ with $v \neq u$ has $\text{finish}(v) < \text{finish}(u)$. But then if v is the predecessor of u on the cycle, the edge (v, u) will be a back edge.

So if G has a back edge then it has a directed cycle and if it has a directed cycle then it has a back edge. This proves the theorem. ■

So now we know that if we call DFS on a DAG, we will never find any back edges. But this automatically gives us a topological sort! When a node finished (i.e. we return from DFS(v)), we just put v at the head of the list. Since there are no back edges, every edge (v, u) is either a forward edge or a cross edge, and thus u has already finished and been put in the list. So no edges go backwards in the list, and we have a topological sort in $O(m + n)$ time.

13.6 Strongly Connected Components

Let's do another application of DFS. This was originally invented by Rao Kosaraju, who you may know or have heard of – he's a professor here, and was one of the founders of the JHU CS department.

Let $G = (V, E)$ be a directed graph. We say that two vertices v and u are *equivalent*, denoted $u \equiv v$, if v is reachable from u and u is reachable from v. It is not hard to see that this is formally an *equivalence relation*, i.e. it satisfies the following three properties.

1. Reflexivity: $v \equiv v$ (obviously).
2. Symmetry: If $v \equiv w$ then $w \equiv v$. This is immediate from the definition.
3. Transitivity: If $v \equiv w$ and $w \equiv u$ then $v \equiv u$. This is also reasonably straightforward: if there is a path from v to w and a path from w to u, then there is a path from v to u. And the same reasoning implies that there is a path from u to v.

Since $\equiv$ satisfies all three properties, it naturally gives us a partition of V into components in which each pair of vertices are equivalent. These are called the *strongly connected components* (or SCCs) of the graph. So C is a SCC if it is a maximal set such that if $u, v \in C$ then $u \equiv v$. We want to design an algorithm to compute the SCCs of G.

A trivial algorithm would be to run DFS or BFS from each node to see what you can reach, and then if two nodes can reach each other we put them in the same SCC. But this requires n different full DFS runs, each of which takes time $O(n + m)$, so the total running time is $O(n^2 + mn)$. This is not so good.

Let's be a little more careful. Before we define the algorithm, let's first set up some notation and get some intuition. Let $\hat{G}$ be the graph of SCCs: there is a vertex $v(C)$ in $\hat{G}$ for each SCC of G, and there is an edge from $v(C)$ to $v(C')$ if there is an edge from some $u \in C$ to some v in C'.

Lemma 13.6.1 *$\hat{G}$ is a DAG.*

Proof: By definition, if two nodes u and v are in the same SCC then they are each reachable from the other. Thus if there is a path in $\hat{G}$ from $v(C)$ to $v(C')$, every node in C' is reachable from every node in C. Thus if there is a directed cycle $v(C_1), v(C_2), \ldots, v(C_k)$ in $\hat{G}$, then $C_1 \cup C_2 \cup \cdots \cup C_k$ would be a SCC in G. This contradicts our definition of $\hat{G}$, and thus $\hat{G}$ cannot have a directed cycle and so is a DAG. ■

Since $\hat{G}$ is a DAG, there is a topological sort of $\hat{G}$. Suppose I knew this topological sort, and suppose that $v(C)$ is a sink $\hat{G}$ (and so has no edges leaving it). Then if we were to run a DFS from a node in C, we would mark *exactly* the nodes in C. In other words, we would find C! We could then remove these nodes from G, and run a DFS from the new final node in the topological sort of $\hat{G}$ (which was previously the next-to-last node), to find the next SCC. We can just keep repeating this until we find all of the SCCs.

Of course, we don't know $\hat{G}$ or else we would already know the SCCs. What Rao figured out is that it's actually easy to find a node in a sink SCC, even though we don't know $\hat{G}$. First, we need to extend our notion of finishing time to SCCs. Let $\text{finish}(C) = \max_{v \in C} \text{finish}(v)$.

Lemma 13.6.2 *Suppose we run a DFS of G. Let C_1 and C_2 be distinct SCCs, and suppose there is an edge $(u, v) \in E$ where $u \in C_1$ and $v \in C_2$. Then finish$(C_1) >$ finish(C_2).*

Proof: Let $x \in C_1 \cup C_2$ be the first node encountered in $C_1 \cup C_2$ by the DFS. We break into two cases. First, suppose that $x \in C_2$. Then the DFS will visit *all* nodes in C_2 before it visits *any* nodes in C_1, so clearly finish$(C_1) >$ finish(C_2). On the other hand, suppose that $x \in C_1$. Then all other nodes in $C_1 \cup C_2$ will be descendants of x and hence the finish time of x will be larger than the finish time of any other node in $C_1 \cup C_2$. Thus finish$(C_1) >$ finish(C_2) ■

Note that this lemma implies that the node with largest finishing time is in a source SCC (an SCC with no incoming edges in $\hat{G}$). Of course, what we wanted was a node in a *sink* SCC, not a node in a *source* SCC. But this is easily fixed: let G^T be the graph we get by reversing every edge of G. Then the SCCs do not change, but clearly the edges of $\hat{G}$ are all reversed. So a node which is in a source SCC of G^T is in a sink SCC of G.

So we can find a node in a sink SCC. Of course, to get overall fast running time we can't do a DFS each time we remove a SCC. But it turns out that we don't need to (see below and the book)! The finishing times of a DFS of G^T are enough. This gives the following overall algorithm.

```
DFS(G^T) to get finishing times
Repeat until G is empty:
  Let v be the vertex in G with largest finishing time
  Runs DFS(v) in G to define an SCC C of all nodes found in this DFS
  Delete C from G
```

Let's first analyze the running time. Flipping the graph is $O(n+m)$. The first DFS call is $O(n+m)$. Then all of the remaining calls combined are only $O(n+m)$, since we only consider each edge once. Thus the total running time is $O(n+m)$. (Note: there are some missing details here. For example, how do we repeatedly find the vertex with largest finishing time without paying a log factor to maintain a heap? Good exercise to do at home: fill in the blanks! Hint: make a list of vertices ordered by finishing time in $O(n + m)$ time (without paying an extra log factor to sort).

For the correctness proof, see the book (I only sketched it in class, and will do only a sketch here).

Theorem 13.6.3 *Kosaraju's SCC algorithm correctly identifies all SCCs.*

Proof: Let $C_1, C_2, \ldots, C_k$ be the sets identified by the algorithm, in order (so the first set deleted is C_1, then C_2, etc). We claim that for all $i \in \{1, \ldots, k\}$, the set C_i is a sink SCC of $G \setminus \cup_{j=1}^{i-1} C_j$. We prove this by induction on i. For the base case, when $i = 1$, we know from Lemma 13.6.2 and the definition of G^T that the vertex v with largest finishing time is in a sink SCC of G. Hence C_1, the set of nodes reachable from v, is exactly the SCC containing v and so is a sink SCC of G as desired.

For the inductive step, suppose that the statement is true for all $j \leq i - 1$, and now we prove it for i. Lemma 13.6.2 implies that the vertex v remaining with largest finishing time must be in an SCC which does not have an edge to any still remaining SCC (or else a node from that SCC would have larger finishing time by Lemma 13.6.2). Hence C_i, the set of nodes reachable from v in $G \setminus \cup_{j=1}^{i-1} C_j$, is indeed a sink SCC of $G \setminus \cup_{j=1}^{i-1} C_j$ as claimed. ■