Lecture Notes for CSCI 311: Algorithms

Set 16-Introduction to Graph Algorithms

Professor Talmage

November 11, 2026

---

1 Overview

For most of the rest of the course, we will look at some interesting algorithms in graphs. Graph algorithms is one of the largest fields of algorithmic research, since graphs are ideal for representing connections between data, people, places, ideas, process steps, and many other things. While we have dabbled with graphs a few times already, we will step back and treat them more formally now.

Definition 1. A *graph* G is a pair (V, E) where V is a set of *vertices* or *nodes* and E is a subset of $V \times V$, known as edges. (Each edge is a pair of vertices.)

Notation and further technicalities:

- This actually defines *directed graphs*, where each edge has a direction: edge (a, b) goes from vertex a to vertex b.
- An *undirected graph* is defined identically, except that each $e \in E$ is an unordered pair of elements of V. While it might make sense to write such edges as $\{a, b\}$, by convention we still write (a, b), and just understand that the order of the pair is not meaningful in the graph. We also typically add the constraint that self-loops (edges (a, a)) are not allowed in undirected graphs.
- For now, we will disallow *multiple edges*: (a, b) and (a, b) are the same edge, not two different edges from a to b.
- If $(u, v) \in E$, then we say u and v are *neighbors*, and the edge (u, v) is *incident* on its *endpoints* u and v.
- The *degree* of a vertex is the number of edges incident on it, or equivalently the number of neighbors of that vertex.

We will discuss the following topics:

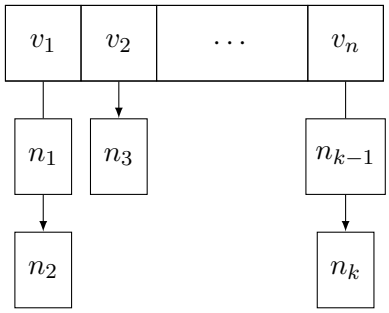
1. Representation: How to store a graph in memory.
2. Searching: How to look through a graph, with some applications.
3. Minimum Spanning Trees: Building a "skeleton" of a graph.
4. Shortest Paths: Single-Source and All-Pairs algorithms.
5. Flows/Cuts: How graphs can represent capacity and how much capacity they have.

2 Representation

There are three primary ways to represent a graph in memory:

1. Dynamic structure. This is a generalization of a linked list. Each vertex will store its connections. This method is good for some things, like moving from node to node, but gives no entire-graph functionality or single point of reference for the graph, so is not generally used.

2. Adjacency list. Store an array of size $|V|$ where each cell holds a pointer to a (linked) list of that node's neighbors.



   Note that in this example, v_1 is connected to both n_1 and n_2, which are not necessarily connected to each other. To determine that, we would need to look up n_1 in the top array, then traverse its neighbor list to see if we find n_2. This is not particularly efficient, so we will have another representation that is much better for checking whether a particular pair of nodes are neighbors. What the adjacency list is good for is walking through all the neighbors of a given node, since we can index to that node and walk down its neighbor list.

3. Adjacency matrix. Store a $|V| \times |V|$ matrix of Booleans. Cell $[i, j]$ holds 1 if $(i, j) \in E$, 0 otherwise. For an undirected graph, this will always be symmetric around the main diagonal.

Example: Consider a graph with 4 nodes and the following representations:

- Adjacency list:

v_1	$[v4]$
v_2	$[v_4, v_3]$
v_3	$[v_2, v_4]$
v_4	$[v_1, v_2, v_3]$

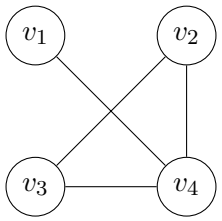
  Note that the order within a node's list of neighbors has no meaning.

- Adjacency matrix:

	v_1	v_2	v_3	v_4
v_1	0	0	0	1
v_2	0	0	1	1
v_3	0	1	0	1
v_4	1	1	1	0

Exercise: Can you tell whether this graph is directed or undirected? Draw the graph.

While the adjacency matrix is symmetric, that does not actually imply that the graph is undirected. The implication is in the other direction.



Comparison: Which representation is best depends on the use case.

> **Exercise:** Fill out the following table with the costs of the two representations for some common questions.

	Space	$(u, v) \in E$?	list u's neighbors
Adjacency List	$\|V\| + 2\|E\|$	$O(degree(u))$	$O(degree(u))$
Adjacency Matrix	$\|V\|^2$	$O(1)$	$\Theta(\|V\|)$

We will generally need both $|V|$ and $|E|$ as parameters for complexity analysis of graph algorithms. $|E| \leq |V|^2$, but since it may be much smaller, it is generally not good practice to simply replace $|E|$ in describing the complexity of a graph algorithm. In fact, if $|E|$ is much less than $|V|^2$, we call the graph *sparse*, and there is an entire field of algorithms for sparse graphs that do interesting things which only work on loosely-connected graph.

3 Searching

We will consider the following version of the search problem, looking at what portion of a graph is reachable from a given node. Different algorithms may solve slightly different versions of the problem, but the fundamental idea is the same.

Input: A graph $G = (V, E)$ and a vertex $s \in V$.

Output: A subset $C \subseteq V$ of vertices which are connected to s.

Definition 2. Two nodes u and v in a graph $G = (V, E)$ are *connected* if there is a path $u \rightsquigarrow v$. A *path* $u \rightsquigarrow v$ is a sequence $(u, x_1), (x_1, x_2), \ldots, (x_k, v)$ of edges in E, where each edge starts where the previous ended.

3.1 Breadth-First Search:

The first algorithm we will consider is Breadth-First Search. This is a generalization of what you have seen in previous classes for trees. In a general graph, we have to handle cycles, where we can find a new path to an already visited vertex, as well as disconnected nodes, which BFS will never reach since they are not connected to the starting node.

> **Exercise:** What extra information does this code compute?

This version of BFS actually computes the minimum distance from s to each other node in the graph. To solve the reachability version of the problem we stated above, loop through the d array and collect the set of vertices with non-infinite distance.

> **Exercise:** Why do we use a queue? Why do we check if $d[v] == \infty$ and only act on v for which the current distance is infinite?

Algorithm 1 Breadth-First Search in a general graph

```
1:  function BFS(s)
2:      create d as an array of size |V|
3:      d[s] = 0
4:      d[v] = ∞ for all v ≠ s
5:      create Q as a empty queue
6:      Q.Enqueue(s)
7:      while Q is not empty do
8:          u = Q.Dequeue()
9:          for all v ∈ Adj(u) do
10:             if d[v] == ∞ then
11:                 d[v] = 1 + d[u]
12:                 Q.Enqueue(v)
13:             end if
14:         end for
15:     end while
16:     return d
17: end function
```

Since this code is breadth-first, it first finds all neighbors of s, which are nodes at distance 1. It stores those for future exploration. When it explores nodes at distance 1 from s, it finds nodes at distance 2, storing them for future exploration. Because the algorithm uses a queue, it will explore all nodes at distance 1 from s before starting on any node at distance 2, then will explore all nodes at distance 2 before any at distance 3, and so on. This guarantees breadth-first exploration. If we ever have an edge to a previously-found nodes, we do not need to update its distance, since the previously-found path to it cannot be longer than the current, by our breadth-first order of exploration. Thus, when we find a second path from s to a node, we can ignore it, as the previous path is at least as good.

Complexity: Recall that an efficient queue algorithm, such as a linked list implementation, has all operations $O(1)$. When using algorithms for previous problems, we can choose the implementation most suited for our use, so we choose an efficient queue. If we use the adjacency list representation of a graph, getting the next neighbor of a given vertex is $O(1)$, which makes our for loop efficient. (If we used an adjacency matrix instead, the total cost of all the for loops could be as high as $|V|^2$, even if there are very few edges.) We place each vertex in the queue at most once, and remove it at most once. We check each edge once (or twice in an undirected graph), so our total cost is $O(|V| + |E|)$.

In general, unless there is a way to solve a problem without fully exploring the graph, $O(|V| + |E|)$ is the best runtime one can expect from a graph algorithm, as that is the cost of looking at every vertex and every edge.

3.2 Depth-First Search:

Next, we consider Depth-First Search. Again, this is a generalization of the algorithm you know from past classes, that works in general graphs instead of just trees. This algorithm is interesting, because it stores seemingly a lot of extra information, but we can use that extra information to solve several other graph problems. One other note is that this algorithm does not take a starting vertex. Instead, it will start at an arbitrary vertex, then when it runs out of reachable nodes, if there are more nodes in the graph, it picks an arbitrary unvisited node and resumes there. Thus, it finds all connected pieces of the graph.

By setting parent pointers, DFS leaves a collection of trees, called a *DFS forest*, each containing all nodes reachable from the vertex chosen as a start vertex in the wrapper function. The code also stores

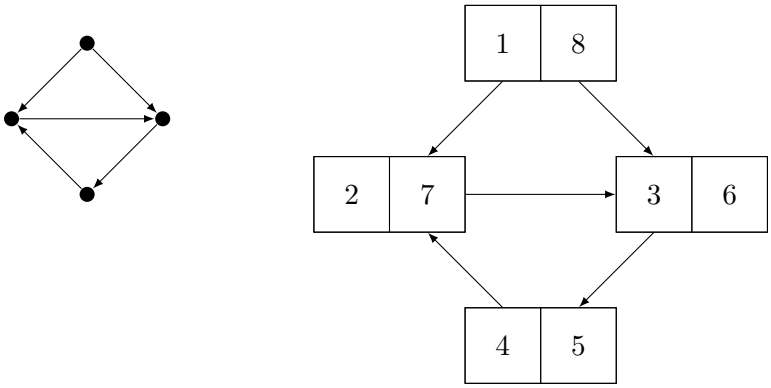
Algorithm 2 Depth-First Search in a general graph

```
1:  function DFS-Wrapper(G = (V, E))
2:      for all u ∈ V do
3:          mark u as unvisited
4:          u.parent = ⊥
5:      end for
6:      for all unvisited vertices s do          ▷ Note that order on these is arbitrary.
7:          DFS-Recursive(s, 0)
8:      end for
9:  end function
10: function DFS-Recursive(s, time)
11:     mark s as visited
12:     time = time + 1
13:     discovered[s] = time
14:     for all undiscovered neighbors u of s do
15:         u.parent = s
16:         DFS − Recursive(u, time)
17:     end for
18:     finished[s] = time
19:     time = time + 1
20: end function
```

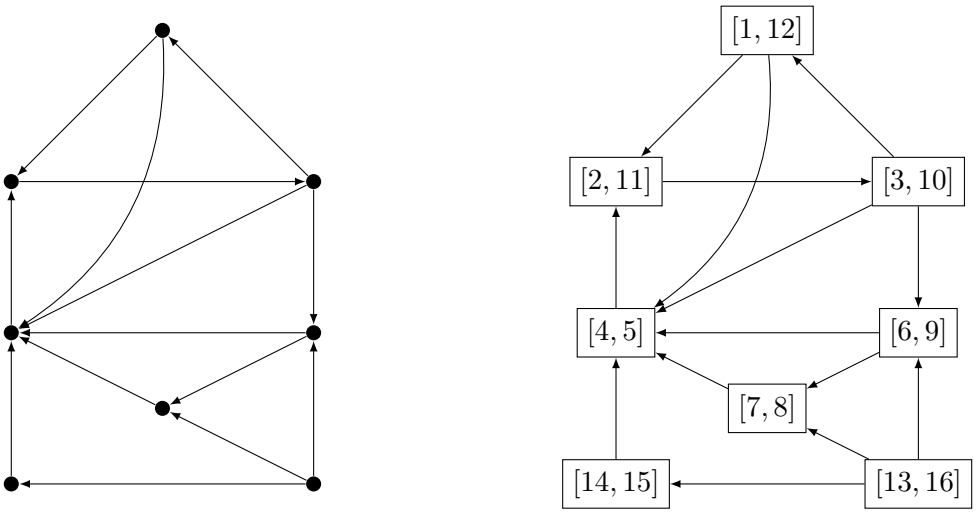
an interval $[discovered[u], finished[u]]$ for each vertex u. These intervals have useful properties, which we will explore shortly.

Complexity: DFS only visits vertices that have not already been visited, and checks each edge at most once, when first visiting a node (though in an undirected graph we may check each edge from each end). Thus, the total cost is linear in both the number of vertices and the number of edges: $O(|V| + |E|)$.

Example: Consider the graph on the left below. Suppose DFS starts from the top vertex. The graph on the right shows the resulting DFS tree, with *discovered* and *finished* time written in each node.



Exercise: Run Depth-First Search on the following graph, computing *discovered* and *finished* times for each vertex.



The graph on the right shows one possible outcome. Different DFS forests are possible, if you make different choices of which nodes to visit next.

3.2.1 DFS Properties

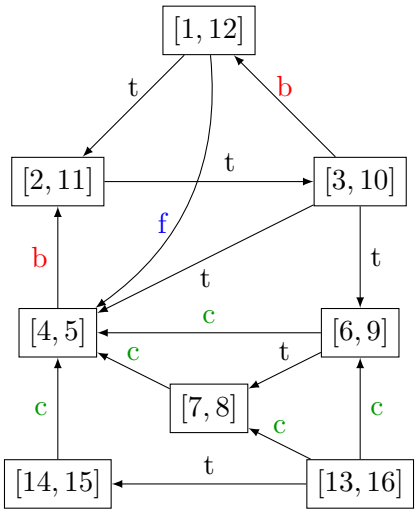
Definition 3. A *tree* is a connected, acyclic graph. A *forest* is a set of trees.

Acyclic means that there are no cycles–a *cycle* is a path from a node to itself.
Running DFS on an arbitrary graph has the following properties:

1. DFS on a graph generates a forest, called a *DFS forest*, using the parent links. There are no cycles because we only add edges to unvisited nodes.
2. We can classify the edges of the original graph into four types. An edge $(u, v) \in E$, w.r.t. a DFS forest is a
   (a) *tree edge* if u is v's parent.
   (b) *back edge* if v is an ancestor of u.
   (c) *forward edge* if v is a (non-child) descendant of u.
   (d) *cross edge* otherwise. This happens when u and v are in different branches or different trees.

   See the figure below which labels each edge in the previous example graph as one of these types.
3. Parenthesis Property: For any two vertices, either one's interval completely contains the other, or the two are disjoint. v is a descendant of u iff u's interval contains v's interval.
4. White-Path Property: w is a descendant of u iff at time $discovered[u]$, there is a path from u to w using only unvisited nodes.

> **Aside:** The name comes from an alternate way of visualizing the DFS algorithm that starts with all nodes white, then partially shades them when first visiting, and fully colors them when they are completed. In that visualization, unvisited nodes are blank ("white"), hence the name.



Lemma 1. *A DFS forest on a digraph (directed graph) has a back edge iff the graph contains a cycle.*

Proof. First, the forward implication. Suppose (w, u) is a back edge. Then u is w's ancestor, so there is a path from u to w in the tree. Then $u \rightsquigarrow w \to u$ is a cycle.

Next, the reverse implication. Let $u_1, u_2, \ldots, u_k$ be a cycle. WLOG, let u_1 be the vertex in the cycle which DFS visits first. u_k will be a descendant of u_1, by the White-Path Property, since $u_2, \ldots, u_k$ are unvisited at $discovered[u_1]$. Thus, (u_k, u_1) is an edge from a descendant to an ancestor, which is the definition of a back edge. □

What this means is that we can use DFS to easily detect cycles in a given graph. This is useful in many applications. For example, if we want to check a dependency graph for validity, we need to check for cycles, as we cannot resolve circular dependencies. With a slight modification to the loop grabbing neighbors of the current node, we can efficiently detect any cycles in the graph.

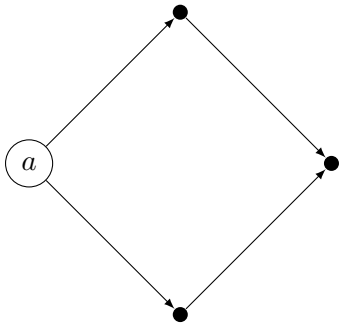
Exercise: Edit the code for `DFS-Recursive` to detect cycles.

4 Topological Sort

Definition 4. A *topological sort* of a DAG (Directed Acyclic Graph) is an ordering of the vertices such that if (u, v) is an edge in the graph, u is ordered before v.

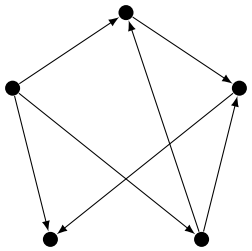
Exercise: Why must a graph be acyclic to even define a topological sort?

Example:



There are two possible topological sorts of this graph: $[a, b, c, d]$ or $[a, c, b, d]$.

Exercise: Give a topological sort of the following graph:



Topological sorts are useful for reducing the complex orderings encoded in a graph down to a single linear order, while respecting the relationships of nodes. For example, as we mentioned earlier this semester, bottom-up dynamic programming computations are just reverse topological sorts of the subproblem graph. We can use DFS to compute a topological sort:

Algorithm 3 Pseudocode for computing a topological sort of a DAG using DFS

```
1: function TopSort(DAG G = (V, E))
2:     Run DFS(G).
3:     When DFS-Recursive finishes for a node, put that node at the front of a linked list.
4:     When DFS completes, return the linked list.
5: end function
```

Complexity: $O(|V| + |E|)$ for DFS, plus $|V|$ linked list prepends, each of which costs $O(1)$, for a total cost of $O(|V| + |E|)$.

Correctness: First, note that the order of nodes this algorithm returns is in decreasing order of finish time, so the last-finished node is first.

Lemma 2. *After running DFS on a DAG $G = (V, E)$, if $(u, w) \in E$, then $finish[u] > finish[w]$.*

Proof. Consider the three possible states of w at time $discovered[u]$:

1. w is unvisited: There is a path from $u \rightsquigarrow w$ using only unvisited nodes (the single edge $u \to w$), so by the White-Path Property, w is a descendant of u. By the Parenthesis Property, every descendant's interval is contained in its ancestors' intervals, so $finish[w] < finish[u]$.

2. w has been discovered but not finished. Then $discovered[w] < discovered[u] < discovered[u] = current < finished[u]$. By the Parenthesis Property, $finish[u] < finish[w]$, since intervals may not partially overlap. Then (u, w) is a back edge, contradicting the fact that a DAG has no cycles.

3. w is finished. That is, $finish[w] < discovered[u]$. $finish[u]$ must be greater than $discovered[u]$, so $finish[u] > finish[w]$.

□

Theorem 1. *This topological sort algorithm is correct.*

Proof. By Lemma 2, every edge in the graph goes from a node with larger finish time to one with smaller finish time, which is the order the algorithm returns, since it puts later-finishing vertices before earlier-finishing ones. □