Graphs and Depth-First Search

Many useful real world objects (and objects of purely theoretical interest too!) can be modeled as **graphs**, where a collection of things (vertices) are connected by relationships (edges).

Graphs can be used to represent many things—maps, social networks, and even computations! Since graphs are prevalent in many fields, there are many algorithms which compute useful properties of graphs.

We will start out by reviewing the fundamentals of graphs which you have seen in previous courses. We will then cover graph search algorithms on unweighted graphs, followed by algorithms to find shortest paths on weighted graphs. Our unit on graphs will conclude with a discussion of some parallel graph algorithms centered around graph contraction, including one for minimum spanning trees.

1 Graph Definitions

In previous classes, you have seen the idea of a graph—a network of vertices and edges.

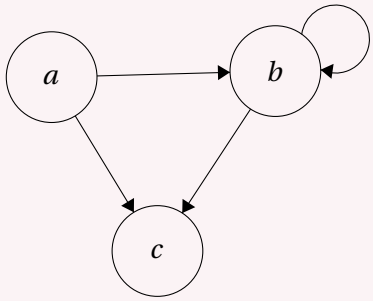
Definition: Graph

Formally, a graph G is defined as a pair of sets, a set of vertices V, and a set of edges E. Each element of E is a pair of elements from V, possibly with an associated weight. The meaning of an edge (u, v) in E can vary based on the kind of graph we are dealing with.

If we have a graph $G = (V, E)$, we will frequently use the variable n to refer to $|V|$, the number of vertices. We will also use m to refer to $|E|$, the number of edges.

Example: A Small Graph

Let's look at how we would mathematically represent the following graph:



This graph has three vertices, so we write $V = \{a, b, c\}$. There are four edges in this graph: one from a to b, one from b to itself, one from b to c, and one from a to c.

In this case, the graph is directed – edges point from one vertex to another. Thus, we'll write the edge $a \mapsto b$ as the tuple (a, b). As such, we get that $E = \{(a, b), (b, b), (b, c), (a, c)\}$. These two sets, together, make up our graph.

Problem: Number of Edges

Let's say we have a directed graph on n vertices, where edges from a vertex to itself are allowed. What is the maximum number of edges that we can create in this graph?

Solution The answer to this depends on whether we allow an edge to appear multiple times in E. If we allow multiple copies of an edge to exist, then our graph on n vertices can have infinitely many edges, since we can duplicate an edge as many times as we want.

If we don't allow multiple copies of an edge to exist, then we can compute the number of edges that could possibly exist within this graph. To create an edge, we need to choose a source vertex and a destination vertex. There are n possible sources and n possible destinations. Thus, the total number of possible edges is n^2.

Problem: Follow-up

If we don't allow multiple copies of an edge to exist, how many possible graphs are there on n vertices?

Solution Let's consider each of the n^2 possible edges that we counted in the previous problem. Each of these edges either exists or does not in any graph on n vertices. Thus, the number of possible graphs is 2^{n^2}, since there are 2 possibilities for each edge.

1.1 Graph Classification

So far, the graphs we have been looking at in our examples have been **directed**, meaning that every edge goes "out of" one vertex and "into" another. However, this does not always reflect what we want in our graphs. For example, let's consider a graph of people, where edges represent that two people are friends. In this case, we probably would not want a directed graph, since this would mean that friendship is a one-way relation.

Thus, we introduce another type of graph, which we call **undirected**. An undirected graph is still a set of vertices V and a set of edges E. However, if I have an edge $(u, v) \in E$, then *it does not matter what order the vertices come in.* In an undirected graph, the presence of edge (u, v) implies that u is connected to v and v is connected to u. As such, E for an undirected graph should not contain both (u, v) and (v, u), since these two tuples represent the same edge.

An edge (such as the one from b to b in the picture above) is called a **self-loop**. A **multi-edge** is an edge that appears more than once in a graph. The term **simple graph** is a useful shorthand for the common case of an undirected graph containing no self-loops or multiple-edges. If we allow the same edge to show up multiple times in E, then we get what is called a **multigraph**—a graph where there can be multiple copies of an edge between two vertices.

1.2 Graph Substructures

Definition: Neighbors and Degrees

A vertex is a **neighbor** of another vertex in an undirected graph if the two vertices are connected by an edge. We define the set $N_G(v)$ for a vertex v in an undirected graph G to be the set of v's neighbors, that is, the set of all vertices u such that $(u, v) \in E$.

In a directed graph, our notion of a neighbor becomes more complex. We say that vertex u is an **in-neighbor** of vertex v if there is an edge from u to v. Likewise, we would call v an **out-neighbor** of u. The set of all in-neighbors of v in graph G is denoted $N_G^-(v)$, while the set of out-neighbors is $N_G^+(v)$. We'll see these neighborhoods come up more as we get into graph algorithms.

The **degree** of a vertex v in an undirected graph G is the number of neighbors it has. Mathematically, we write this as $d_G(v) = |N_G(v)|$. We can similarly define the **in-degree** and **out-degree** of a vertex v in a directed graph G. We denote these as $d_G^-(v)$ and $d_G^+(v)$ respectively.

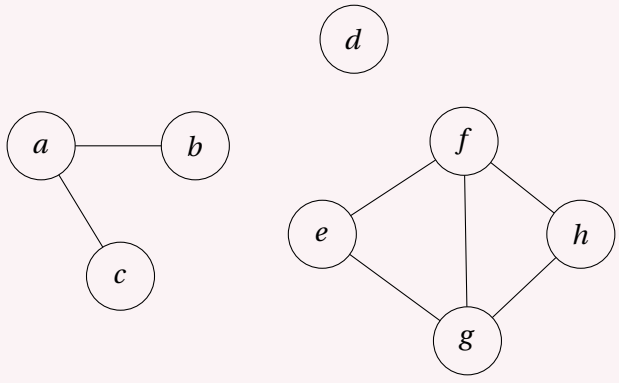
We define a **path** to be a sequence of vertices such that each adjacent pair in the sequence has an edge between them in the graph. The *length* of a path is the number of edges in the path.

There are certain terms we can use to refer to special paths. We define a **simple path** as a path that does not reuse any vertices (and thus it also does not reuse any edges). We also define a **cycle** as a path that starts and ends at the same vertex. Similarly, a **simple cycle** is a cycle which does not reuse any vertices or edges other than the starting/ending vertex.

We also use paths to define the concept of reachability of vertices—that is, checking whether we can get from vertex u to vertex v. We say that vertex v is **reachable** from vertex u if there exists a path from u to v in G.

What about *all* of the vertices that are reachable from a given vertex? In an undirected graph, such a subset is called a **connected component**.

Example: Connected Components



In this graph, the connected components are $\{a, b, c\}$, $\{d\}$, and $\{e, f, g, h\}$.

In a directed graph, things are more complicated since it is possible that v is reachable from u but u is not reachable from v. Directed graphs have a similar notion called *strongly-connected* components, but they are more complicated and require a more sophisticated algorithm.

A special type of a graph is a **forest**, which is an undirected graph which contains no cycles. A **tree** is a forest with one connected component. A **DAG**, or **directed acyclic graph**, is a directed graph which contains no cycles. These graphs have special properties which we'll cover later.

Finally, a **complete graph** is a simple undirected graph with all possible $\binom{n}{2}$ edges.

1.3 Uses of Graphs

Where might graphs be used in the real world, or in computer science? There are a host of applications:

- Utility graphs – We can model utilities, such as electricity, water, and gas, as graphs. The edges in the graph represent connections between two locations, while nodes represent locations where these resources are produced or consumed.
- Dependency graphs – the nodes are tasks, and an edge from a to b means that task b can only be done after task a is completed. It can be used to model computations. An edge from one computation to another signifies that the second computation relies on the first. This will be useful later in the course when we discuss scheduling parallel computations.
- Social network graphs – A social network forms a graph where the vertices represent users. Undirected edges between users can indicate that two users are friends. Some social networks might also form directed graphs. For example, you might follow a celebrity but they don't follow you back, so you have an edge pointing to them, but not the other way around.
- A map of the Internet (or another set of documents) – We can represent the Internet as a graph where the vertices each represent individual webpages, and an edge from one webpage to another indicates that the former webpage links to the latter.

This is only a small subset of the uses for graphs, used as an example. In reality, there are many more things that graphs can be used for!

2 Representing Graphs

When designing graph algorithms, we must choose a representation for storing the graph in memory. One consideration that will affect our representation and the choice of data types is the types of the labels of the vertices. By default, we will often assume that the n vertices of the graph are labeled with the integers 0 to $n-1$. We will call such a graph an **enumerable graph**.

Different representations support different operations more efficiently, so it is useful to understand their trade-offs. Several considerations might affect our choice:

- Is the graph directed or undirected?
- Is the graph weighted or unweighted?

- Is the graph enumerable?
- What kind of operations need to be supported efficiently?

Most graph algorithms, including depth-first search and breadth-first search, primarily need one fundamental operation:

- Given a vertex v, obtain its neighbors.

There are two primary representation strategies used by most algorithms, which are adjacency lists and adjacency matrices. These strategies are widely used because they can work well for both directed or undirected graphs, and weighted or unweighted graphs.

2.1 Adjacency Lists

The most common representation of a graph is the **adjacency list**.

Definition: Adjacency List

For each vertex v, store the collection of edges leaving v.

This definition works equally well for directed and undirected graphs. In an undirected graph, an edge (u, v) will simply be present in both u's list and v's list. This definition also works for both weighted or unweighted; we simply change what data we store in the edges.

Note that this definition is not fully complete—there are many different choices for the type of collection that stores the edges adjacent to v.

Remark: Adjacency "list"

Indeed, this is one of the many situations in computer science where there is not a single agreed-upon definition. Some people use "adjacency list" to specify mean "*linked* lists", while others interpret "list'" more broadly as any kind of sequence, e.g., a dynamic array.

We will avoid using linked-lists to implement adjacency lists since they reduce opportunities for parallelism. Depending on whether the graph is enumerable, we will typically consider two possible representation choices for an adjacency list:

- **Enumerable:** Each vertex stores a **sequence** of its adjacent edges. That is,

  ```
  sequence<sequence<Edge>>
  ```

  Given an adjacency list `adj`, the neighbors of vertex u are stored in `adj[u]`. This allows $O(1)$ access to a vertex's adjacency list and is the most common choice in algorithm design.

- **Otherwise:** If the type of the vertex labels, `V`, is not integers (for example, strings representing words), we can instead store a **dictionary** mapping each vertex to its adjacency list:

  ```
  dictionary<V, sequence<Edge>>
  ```

  Conceptually, this is the same representation: for each vertex, we store its neighbors. The only difference is how we locate the adjacency list for a given vertex.

In an undirected graph, `Edge` could simply contain a vertex label (which would be `int` for an enumerable graph), since all it needs to know is the vertex on the other end. For a weighted graph, we want `Edge` to also store the corresponding weight, so we could represent it a struct, a.k.a., record type, like

```
type Edge = {to : V, weight : W}
```

In this course, we will primarily assume our graphs are enumerable for simplicity, but the algorithms we study apply equally well to graphs whose vertices are arbitrary values.

Claim: Cost of Adjacency Lists

For both the enumerable and non-enumerable version, an adjacency list uses $\Theta(n+m)$ space for a graph with n vertices and m edges. The only difference is the cost of obtaining the neighbours of u. If the graph is enumerable, this is just a sequence/array lookup which takes $O(1)$ time. If the graph is not enumerable, we can either use:

- **A hashtable dictionary**: looking up the neighbours of u takes $O(1)$ expected time,
- **a balanced-BST dictionary**: looking up the neighbours of u takes $O(\log n)$ time.

2.2 Adjacency Matrices

Another representation is the **adjacency matrix**. If the vertices are numbered $0, \ldots, n-1$, we store an $n \times n$ matrix (most naturally represented as a 2D array),

```
sequence<sequence<W>>
```

where entry (u, v) indicates the weight of the edge from u to v. In an unweighted graph, the weights can simply be booleans instead, indicating the presence of lack of the edge (u, v).

Adjacency matrices allow constant-time edge existence queries. However, finding all the neighbors of a vertex requires searching an entire row, which takes $\Theta(n)$ time. The space usage is $\Theta(n^2)$, which is inefficient for sparse graphs, but often the best choice for dense graphs.

3 Depth-First Search

3.1 Introduction

Depth-first search is a very useful technique for analyzing graphs. It can be used to:

- Determine the connected components of a graph.
- Find cycles in a directed or undirected graph.
- Find the biconnected components of an undirected graph.
- Topologically sort a directed graph.
- Determine if a graph is planar, and find an embedding if it is.

- Find the strongly-connected components of a directed graph.

It explores all the edges and vertices in the graph and runs in $O(n+m)$ work and span, i.e., linear in the size of the graph. This means it is a sequential algorithm. In this lecture we will develop some theory about DFS, and show how it can be used for applications such as cycle finding and computing a topological ordering in a DAG.

Since DFS is fundamentally a sequential algorithm, we will be presenting it using more standard imperative data structures (e.g. arrays) and pseudo-code rather than trying to implement it in a purely functional style with immutable and persistent data structures.

We will assume in the remainder of this lecture that the graphs we are processing are enumerable. And we will be using the adjacency list representation of graphs described earlier.

3.2 A Skeleton for Depth-First Search

What does depth-first search do on a graph? As the name suggests, it operates in a depth-first manner, meaning that it goes as deep as it can into the graph before backtracking. It is typically implemented recursively.

Most algorithms that use depth-first search typically follow the same overall structure. We first present a "skeleton" algorithm, which doesn't do anything, but contains all of the essential ingredients of depth-first search.

Algorithm: DFS Skeleton

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun skeleton(adj : adjacency_list):
  n = |adj|
  visited = [False for _ in 0...n-1]

  fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)
```

The skeleton illustrates the key ideas of a depth-first search:

- We keep track of which vertices have been visited so that vertices are never visited multiple times, else the search could take exponential, or even infinite time!
- One call of `dfs` takes a vertex u being searched, and recursively searches all of the neighbours of u that have not been searched yet.

To make this code actually do something, we typically need to decide:

- Which vertex (or vertices) we should initially search from

- What kind of information should we compute and store during the search

3.3 Reachability

Perhaps the simplest example application of depth-first search is **reachability**. Given a start vertex s in a graph, determine the vertices that are reachable from s, i.e., determine the vertices t such that there exists a path from s to t.

To implement reachability using the DFS skeleton above, we don't actually need to store any additional information. Simply knowing which vertices were visited is sufficient, since that's actually precisely the answer we want. To compute the vertices reachable from u, we therefore simply perform a DFS from s, and return the vertices t for which `visited[t]` is True.

Algorithm: Reachability

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun reachable(adj : adjacency_list, s : int):
  n = |adj|
  visited = [False for _ in 0...n-1]

  fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)

  dfs(s)
  return filter(fn u => visited[u], 0...n-1)
```

3.4 Vertex Numbering

We start by showing how DFS can be used on a directed graph to number every vertex with a "starting time" and a "finishing time". Although its not clear why yet, these numbers have useful properties, which we will see later.

Algorithm: DFS Vertex Numbering

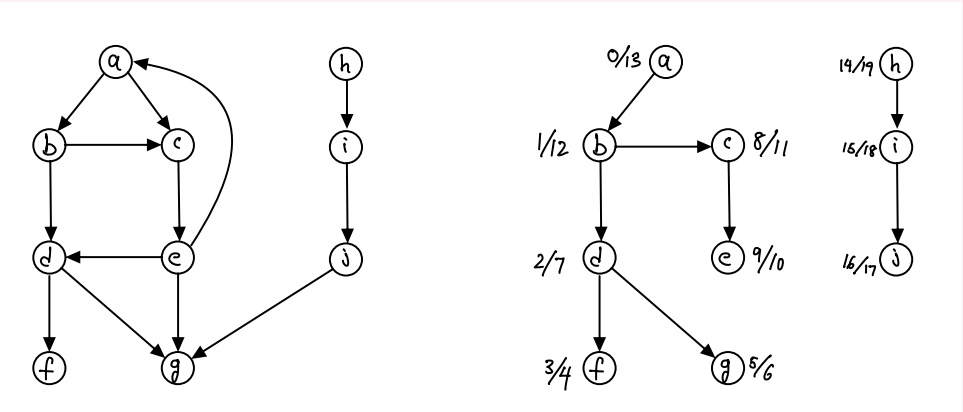
```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun dfs_numbers(adj : adjacency_list)
                -> (sequence<int>, sequence<int>):
  n = |adj|
  visited = [False for _ in 0...n-1]
  start =   [0 for _ in 0...n-1]
  finish =  [0 for _ in 0...n-1]
```

```
i = 0
fun dfs(u : int):
  visited[u] ← True
  start[u] ← i;
  i ← i+1;
  for v in adj[u]:
    if not visited[v]:
      dfs(v)           // (u,v) is a tree edge
  finish[u] ← i
  i ← i+1

for u in 0...n-1:
  if not visited[u]:
    dfs(u)

return (start, finish)
```

Since the algorithm loops over every vertex in the graph and calls `dfs` if the vertex is not visited, it is clear that every vertex is visited, and hence assigned a starting number and a finishing number. And these numbers are all distinct, and range from 0 to $2n-1$.

Let's look at what this algorithm does on an example.

Example: DFS Search and Numbering



Here we see a run of this algorithm on a graph of 10 vertices, shown on the left. The DFS starts from vertex a. The right half of the figure shows the DFS spanning forest, and the numbering that results from the DFS. The numbers d/f shown next to a vertex are the starting and finishing numbers on the vertex.

Note that the choice of which edges are tree edges is dependent on the order in which the vertices are processed at top level, and also on the order of neighbors in the adjacency lists. For example, if the DFS started at h instead of a then g would be a child of j in stead of d in the spanning forest.

Here are some useful properties of the DFS tree and numbers. For any node x, the subtree rooted at x contains exactly the nodes that are visited during the call `dfs(x)`. So if x has numbers a/d and y has numbers b/c then it must be the case that $a < b < c < d$. Conversely if x and y were not related by one being an ancestor of the other then the two intervals $[a,d]$ and $[b,c]$ are disjoint (non-overlapping).

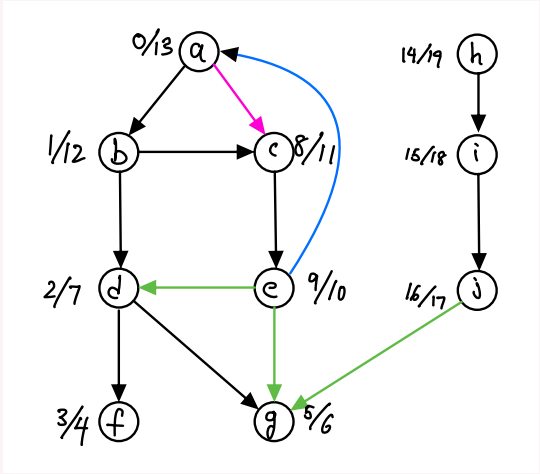
The edges that are not tree edges can be classified into three categories.

Definition: Classification of DFS edges

- There are the **forward edges** which go from a node to one of its descendants in the DFS tree. This is exemplified by edge (a,c) in the example above,
- there are **back edges** that go from a node to one of its ancestors. This is exemplified by edge (e,a),
- and there are the **cross edges** that connect two nodes that are unrelated by the ancestor relation. One example is edge (e,d).

Consider a cross edge (x,y). Since (x,y) is not a tree edge, we know that y must have already been visited before x in the traversal. Therefore, we know that if the numbers of x are a/b and the numbers of y are c/d then $c < d < a < b$.

Example: Edge Types



Tree edges are black, cross edges are green, back edges are blue and forward edges are red. Notice that for non-tree edges the type can be inferred by the vertex numbers.

3.5 Directed Cycle Detection

We claim that the start/finish-time labeling can be used to detect *cycles*.

Theorem: Directed Cycles

A directed graph contains a cycle if and only if its DFS tree contains a back edge.

Proof. If the graph contains a back edge, then it must contain a cycle. More specifically the edge goes from a node to one of its ancestors in the DFS tree. So the cycle is obtained by starting by using the back edge, then moving down the tree to the starting point.

Conversely, if there is no back edge, then there is no cycle. This can be seen by considering the vertex numbers. If you follow a tree or forward edge, the number interval shrinks. Following a cross edge leads to a disjoint interval to the left of where you were. Thus no matter how many tree, forward, or cross edges you follow you cannot get back to where you were before. □

So, an algorithm for testing if a graph is acyclic is to run DFS and see if there are no back edges.

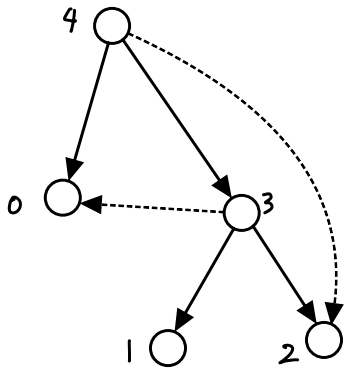
3.6 Computing a Topological Ordering

Suppose the graph is directed and acyclic, i.e., it is a directed acyclic graph (commonly known as a *DAG*). In this case there exists what is called a *topological ordering* of the vertices. This is a list of all the vertices of the graph such that for any edge (u, v) in the graph the vertex u comes before v in the ordering. A reverse topological ordering is one whose reversal is a topological ordering. So for any edge (u, v), v occurs before u in the reverse topological ordering.

Algorithm: Topological Sorting

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun topological_sort(adj : adjacency_list) -> sequence<int>:
  n = |adj|
  visited = [False for _ in 0...n-1]
  answer = [0 for _ in 0...n-1]

  i = 0
  fun dfs(u : int):
    visited[u] ← True
    for v in adj[u]:
      if not visited[v]:
        dfs(v)
    answer[i] ← u
    i ← i+1

  for u in 0...n-1:
    if not visited[u]:
      dfs(u)

  return reverse(answer)
```

This algorithm actually produces a reverse topological ordering, which it then fixes by reversing the answer on the final line. The following is an example of what happens when this algorithm is run on a DAG (before reversing the answer at the end). The vertex numbered 0 is first in the ordering, etc.



Lemma 1

When the above algorithm is run on a DAG it orders the vertices in topological order.

Proof. We will argue that before the final `return` statement, the algorithm produces a list of vertices in *reverse* topological order. It suffices to prove that if there exists an edge (u, v) then the algorithm puts v into `answer` before it puts u. And this result follows if the call to `dfs(v)` returns before the call to `dfs(u)` returns.

Since the graph is acyclic there are no back edges, so (u, v) must be a cross, tree, or forward edge. If it's a cross edge, then the call to `dfs(v)` has completed before `dfs(u)` even started. If it's a tree edge or forward edge, then the sequence of events is that we initiate the call to `dfs(u)`, during which we discover v and call `dfs(v)`, then we complete the call to `dfs(v)` then we complete the call to `dfs(u)`. □