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 widely useful, there are many algorithms for computing 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. We will conclude with *graph contraction*, a technique for parallel graph algorithms, 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.

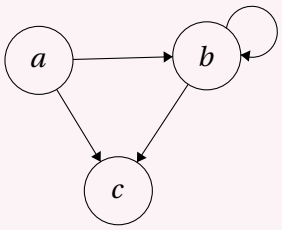
Definition: Graph

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.

Given 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.

1.1 Graph Classification

Directed versus undirected The example graph above is **directed**, meaning that the relationship between the two vertices is *from one to the other* and isn't symmetrical. Sometimes this is what we want, but we may also want to represent a symmetrical relationship. For example, consider a social network such as Facebook in which people can be connected as friends; this relationship is usually symmetric, if I am friends with you then you are friends with me, so we would want an **undirected** graph. On the other hand, a social network in which people can *follow* others often induces a directed graph: Even if I follow Gordon Ramsay on Instagram, he probably doesn't follow me back.

Mathematically, 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. In some books or courses, they use *a two-element set* / unordered pair to represent an undirected edge, $\{u, v\}$, to indicate that the order doesn't matter. We will not adopt this convention.

Self-loops and multi-edges 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.

Definition: Simple Graph

A **simple graph** is an undirected graph containing no self-loops or multi-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.

Remark: What is a "Graph"?

A simple graph is one without self-loops or multi-edges. A multi-graph is a graph that is allowed to contain multi-edges (and sometimes, self-loops, depending on whose textbook you read or course you take). So, what should the plain term "graph" refer to?

Unfortunately there is no universal answer; different textbooks and courses disagree. Sometimes graph means simple and sometimes graph means multigraph. In this course, we will use *graph* as the general term, and specify when we require a simple graph.

1.2 Graph Substructures

Definition: Neighbors

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 $(u, v) \in E$.

Definition: In-Neighbors and Out-Neighbors

In a directed graph, 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.

Definition: Degree

The **degree** of a vertex v in an undirected graph G is the number of edges incident to it (self loops count twice). We denote this by $d_G(v)$. In a simple graph, $d_G(v) = |N_G(v)|$.

Similarly, the **in-degree** and **out-degree** of a vertex in a directed graph count its incoming and outgoing edges, respectively. We denote these by $d_G^-(v)$ and $d_G^+(v)$.

Many interesting questions about graphs involve connections between vertices that are not directly adjacent. For example, can we travel from one vertex to another by following a sequence of edges? This motivates the notion of a **path**.

Definition: Path

A **path** is a sequence of vertices where each consecutive pair is connected by an edge, following edge directions in directed graphs. Its length is the number of edges.

A **simple path** is a path that does not repeat any vertices.

A path may visit the same vertex more than once[1]. In particular, it might eventually return to its starting vertex. Such paths give us another important graph structure: **cycles**.

Definition: Cycle

A **cycle** is a path that starts and ends at the same vertex. It must have at least one edge, and in an undirected graph it must not use the same edge twice in a row.

A **simple cycle** is a cycle that does not repeat any vertices or edges, except its first and last vertices which are the same.

The restrictions in the definition exist to prevent a single vertex on its own with no edges from being considered a cycle, and in an undirected graph to prevent a path that takes the same edge back and forth in both directions from being called a cycle.

We also use paths to define the concept of **reachability**.

Definition: Reachability

We say that vertex v is **reachable** from vertex u if there exists a path from u to v in G.

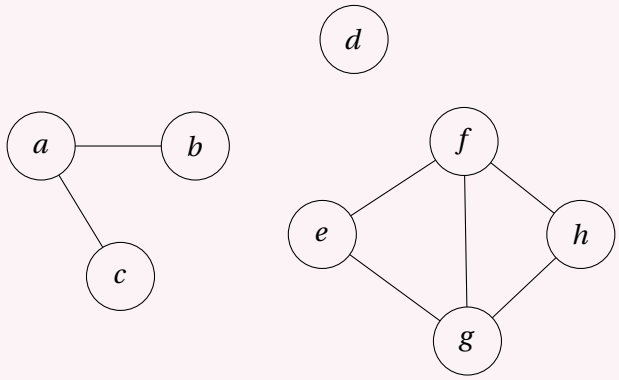
[1]This is another term that is not universally agreed upon. Many textbooks, especially in graph theory, define a path to be a simple path, no repeats allowed. They then define a non-simple path as a *walk*, or a path that can repeat vertices but not repeat edges as a *trail*. For simplicity we select the convention that a path can repeat.

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**.

Definition: Connected Components

A **connected component** of an undirected graph is a maximal subset C of vertices such that every vertex in C is reachable from every other vertex in C.

Example: Connected Components



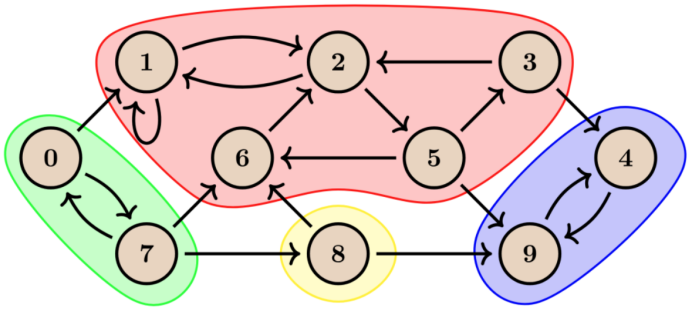
In this graph, the connected components are $\{a, b, c\}$, $\{d\}$, and $\{e, f, g, h\}$.

For directed graphs, reachability is not necessarily symmetric; it is possible that v is reachable from u, but u is not reachable from v. Directed graphs, therefore, use a more complex notion of connectivity.

Definition: Strongly Connected Component

A **strongly connected component** (SCC) of a directed graph is a maximal subset C of vertices such that every pair of vertices in C is mutually reachable: u and v are in the same SCC if and only if u is reachable from v and v is reachable from u.

In the graph below, the strongly connected components are coloured. Note that it is still possible to reach some of the strongly connected components from some of the others, but never to make it back, or they would be part of the same component.



2 Representing Graphs

When designing graph algorithms, an important decision is how to *represent* the graph. Different representations support different operations more efficiently, so our choice of representation depends on what the algorithm needs to do.

One important consideration is the type of the vertex labels. By default, we will focus on **enumerable graphs**, where the n vertices are labeled with the integers 0 through $n-1$. This allows us to use vertex labels directly as sequence indices.

Other considerations include:

- Is the graph directed or undirected?
- Is the graph weighted or unweighted?
- What 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 outgoing edges** (in an undirected graph, its incident edges).

We will consider two common graph representations: **adjacency lists** and **adjacency matrices**. Both can represent directed and undirected graphs, as well as weighted and unweighted graphs, but they have different performance characteristics.

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. In an undirected graph, these are simply the edges to its neighbors.

This representation works for both directed and undirected graphs. In an undirected graph, an edge between u and v is represented in the adjacency lists of both endpoints. It also works for both weighted and unweighted graphs, depending on what information we store for each edge.

Remark: Adjacency "list"

As with much of graph terminology, textbooks disagree on the meaning of "list." Some use *adjacency list* specifically for linked lists, while others use it more broadly for any per-vertex collection of edges.

We adopt the latter definition. In particular, we will typically use sequences rather than linked lists, since sequences provide more opportunities for parallelism.

Enumerable graphs. Recall that in an enumerable graph, the vertices are labeled $0, \ldots, n-1$. We can therefore represent an adjacency list using a sequence of sequences:

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

Given an adjacency list `adj`, the outgoing edges of vertex u are stored in `adj[u]`. Since our sequence interface supports random access, we can obtain this collection in $O(1)$ time.

The type `Edge` depends on the graph:

- **Unweighted graphs:** An edge only needs to identify its destination:

```
type Edge = int
```

- **Weighted graphs:** An edge also stores its weight:

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

In the unweighted case, we will use the following shorthand throughout our graph-search algorithms. It is equally applicable to both directed and undirected graphs:

```
type UnweightedAdjList = sequence<sequence<int>>
```

Other vertex types. If vertex labels have an arbitrary type `V`, such as strings, we can instead use a *dictionary* to locate each vertex's adjacency list:

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

The edge types can then be represented as `type Edge = V` for an unweighted graph, or a record/struct type `type Edge = {to : V, weight : W}` (weighted).

Conceptually, the sequence and dictionary versions are the same representation: each vertex stores its outgoing edges. The only difference is how we locate the corresponding adjacency list. In this course, we will primarily use enumerable graphs for simplicity, although the same algorithmic ideas apply to graphs with arbitrary vertex labels.

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 hash table 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 common representation is the **adjacency matrix**. For an enumerable graph with n vertices, we store an $n \times n$ matrix, typically represented using a two-dimensional sequence:

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

The entry at position (u, v) stores the weight of the edge from u to v. For an unweighted graph, we can simply use booleans to indicate whether an edge exists.

One subtlety is how to represent an *absent edge*. We could use an optional value instead of `W`, but we can often choose a special value depending on the application. For example, we might use ∞ when computing shortest paths, or 0 to represent the absence of capacity in a flow network.

Adjacency matrices support $O(1)$-time queries for the existence or weight of an edge. However, obtaining all the out-neighbors of a vertex requires scanning an entire row, which takes $\Theta(n)$ time. This is very bad for **sparse graphs**, where each vertex might have relatively few edges.

The space usage is $\Theta(n^2)$, which is also inefficient for sparse graphs, but can be useful for dense graphs, particularly when efficient edge-existence queries are important.

3 Depth-First Search

Depth-first search (DFS) is one of the most fundamental techniques for analyzing graphs. As its name suggests, it explores as deeply as possible along a path before backtracking to try other paths. This naturally leads to a recursive algorithm.

The basic idea is simple, but we will see in this lecture that DFS can be adapted to solve many different problems, including but not limited to:

- Finding reachable vertices and connected components.
- Detecting cycles in directed graphs.
- Computing a topological ordering of a DAG.

In addition to this list, other applications which we won't cover in this course include:

- Finding the biconnected components of an undirected graph.
- Determining if a graph is planar, and finding an embedding if it is.
- Finding the strongly-connected components of a directed graph.

Rather than treating these as separate algorithms, we will develop DFS as a *reusable template*, gradually adding information and actions to support different applications.

The standard DFS implementation performs $O(n+m)$ work and has $O(n+m)$ span. Since this version is sequential, we will use ordinary imperative data structures, such as mutable arrays, rather than insisting on a purely functional implementation.

Throughout this section, we assume that graphs are enumerable and represented using an unweighted adjacency-list, `UnweightedAdjList`, as defined earlier.

3.1 Basic DFS Template

The key idea of DFS is to explore each vertex's neighbors recursively, completing the exploration of one neighbor before moving on to the next. We must also remember which vertices have already been visited. Otherwise, a graph containing cycles could cause the algorithm to revisit the same vertices indefinitely.

Algorithm: Basic DFS Template

```
fun dfs_template(adj : UnweightedAdjList):
  n = |adj|
  visited = [False for _ in 0...n-1]

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

  // Call dfs from the appropriate starting vertices
```

The template has two important features:

- Each vertex is visited at most once. When DFS discovers a vertex, it is marked as visited before its neighbors are explored.
- The exploration is depth-first: whenever DFS encounters an unvisited neighbor, it recursively explores that vertex before continuing with the remaining neighbors.

To turn this template into a particular algorithm, we make two decisions:

- **Where to start:** We might call `dfs(s)` for a single source vertex, or iterate over all vertices and start a new search whenever we encounter an unvisited one.
- **What to do when discovering a vertex:** The placeholder `DISCOVER(u)` represents application-specific work performed when u is first visited.

When DFS starts from a vertex s, it explores exactly the vertices reachable from s. If we instead start a new DFS from every vertex that remains unvisited, we eventually explore the entire graph. Both of these will be useful patterns we can apply to problems.

Since each vertex is visited at most once, and each adjacency list is scanned at most once, the total work and span are $O(n+m)$.

3.2 Reachability and Connected Components

Our first application is **reachability**. Given a starting vertex s and target vertex t, the goal is to find whether t is reachable from s, i.e., whether there exists a path from s to t. This problem makes sense in both directed and undirected graphs.

This is almost exactly what our basic DFS template already does. When we start DFS from s, it visits precisely the vertices reachable from s. No additional action on discovery is needed: the `visited` array already contains the answer.

Algorithm: Reachability

```
fun reachable(adj : UnweightedAdjList, s : int, t : int) -> bool:
  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 visited[t]
```

What if we want to explore the entire graph, rather than just the vertices reachable from a single source? For an undirected graph, we can use DFS to identify all the **connected components**, assigning a component number to each vertex when it is discovered.

Algorithm: Connected Components

```
fun connected_components(adj : UnweightedAdjList) -> sequence<int>:
  n = |adj|
  visited = [False for _ in 0...n-1]
  component = [0 for _ in 0...n-1]
  id = 0

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

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

  return component
```

Compared to the first algorithm where we only ran DFS from a single starting vertex, here, we iterated over all vertices, starting a new DFS whenever we encountered one that had not yet been visited. We sometimes call this pattern "DFS All".

The resulting array assigns the same component number to vertices if and only if they belong to the same connected component. The actual numbers depend on the order in which vertices are explored, but the partition into components does not.

Notice that we have made only two changes to the basic DFS template:

- We start DFS from every vertex that remains unvisited, rather than from a single source.
- We use the `DISCOVER(u)` event to record the component number of each newly visited vertex.

Both algorithms perform $O(n+m)$ work and have $O(n+m)$ span: each vertex is visited at most once, and each adjacency list is processed at most once.

3.3 Three-Colour DFS

So far, our applications of DFS have used a boolean `visited` array to remember which vertices have already been discovered. This is sufficient for reachability and connected components, but other applications require more information. In particular, we may want to distinguish vertices whose exploration *is still in progress* from those whose exploration has already finished. To do this, we extend our DFS template to track three possible states instead of two.

Definition: Three-Colour DFS

During DFS, every vertex is in one of three states:

- **`WHITE`**: The vertex has not yet been discovered.
- **`GRAY`**: The vertex has been discovered, but its exploration has not yet finished.
- **`BLACK`**: The vertex has finished exploring all of its outgoing edges.

Initially, every vertex is WHITE. A vertex becomes GRAY when we first discover it, and BLACK when its DFS call finishes. At any point during DFS, the GRAY vertices form a path in the DFS tree, from the root of the current search to the vertex currently being explored.

Algorithm: Three-Colour DFS Template

```
fun dfs_template(adj : UnweightedAdjList):
  n = |adj|
  color = [WHITE for _ in 0...n-1]

  fun dfs(u: int):
    color[u] ← GRAY
    // DISCOVER(u)
    for v in adj[u]:
      // EXAMINE(u, v)
      if color[v] == WHITE:
        dfs(v)
    //FINISH(u)
     color[u] ← BLACK

  // Call dfs from the appropriate starting vertices
```

This version of DFS comes with *three* customizable events:

- **`DISCOVER(u)`**: Record some information each time a new vertex is discovered
- **`EXAMINE(u,v)`**: Record some information each time an edge is processed
- **`FINISH(u)`**: Record some information after a vertex is finished being explored

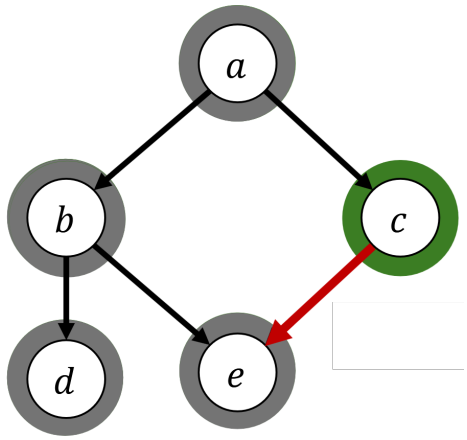
These will allow us to implement more applications and solve more problems that were not possible with just the simpler `visited` / `DISCOVER` pattern. Notice that we examine every outgoing edge, even when its destination has already been discovered. In particular, `EXAMINE(u,v)` can inspect the current colour of v to distinguish an active vertex from a finished one.

The algorithm still performs $O(n+m)$ work and has $O(n+m)$ span, since each vertex is visited at most once and each edge is examined at most once.

3.4 Directed Cycle Detection

Our next application is determining whether a directed graph contains a cycle. Why is the basic DFS template with a boolean `visited` array insufficient?

Suppose we encounter an edge (u, v) where v has already been visited. This does not necessarily mean there is a cycle: the exploration of v might have finished long before we discovered u.



This is precisely where three-colour DFS becomes useful. Recall that the GRAY vertices are the vertices that are still being explored. This means that *there is a path from them to the current vertex*. Therefore, if we encounter an edge (u, v) where v is GRAY, then the path in the DFS tree from v to u, together with the edge (u, v), forms a directed cycle.

Theorem: Directed Cycle Detection

A directed graph contains a cycle if and only if DFS examines an edge (u, v) and v is GRAY.

Proof. We have already established one direction: an edge to a GRAY vertex produces a cycle.

Conversely, suppose the graph contains a directed cycle. Let u be the first vertex on that cycle discovered by DFS. At the moment u is discovered, all other vertices on the cycle are still WHITE.

Since all vertices on the cycle are reachable from u, they must all be explored before u finishes. In particular, consider the vertex v immediately preceding u on the cycle. When DFS examines the edge (v, u), vertex u is still GRAY. Thus, DFS must encounter an edge to a GRAY vertex. □

This gives us an algorithm: whenever DFS examines an edge (u, v), check whether v is GRAY.

Algorithm: Directed Cycle Detection

```
fun has_cycle(adj : UnweightedAdjList) -> bool:
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  found_cycle = False

  fun dfs(u: int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == GRAY:    // EXAMINE(u,v)
        found_cycle ← True
      else if color[v] == WHITE:
        dfs(v)
    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

  return found_cycle
```

Notice that we use the "DFS All" pattern, since a cycle might exist in a part of the graph that is not reachable from a particular starting vertex. The only customization of the three-colour DFS template is at `EXAMINE(u,v)`: we check whether the destination is GRAY and record if we find such an edge. As before, the algorithm performs $O(n+m)$ work and has $O(n+m)$ span.

Remark: Undirected Graphs

In an undirected graph, an edge to a GRAY vertex does not necessarily indicate a cycle, since each DFS tree edge can also be traversed in the reverse direction. To detect cycles in an undirected graph, we must exclude the edge back to the parent.

3.5 Topological Sorting

Our next application is **topological sorting**, which is useful for scheduling tasks with dependencies. As an example, consider the problem of choosing an order to complete your CMU courses in. You must take 15-122 and 15-150 before you take 15-210, and you must take 15-151 before you take 15-150, etc. So, how do we express this as a graph problem?

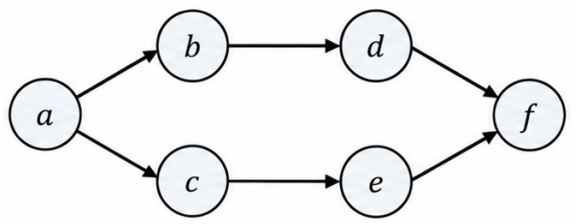
Suppose we have a directed graph where an edge (u, v) means that task u must be completed before task v. We want to find an ordering of the vertices that respects all these dependencies.

Definition: Topological Ordering

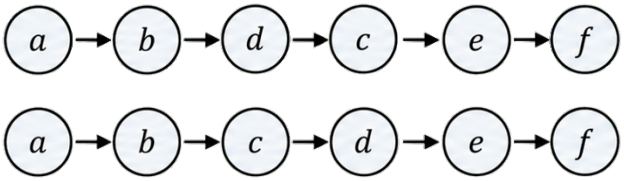
A **topological ordering** of a directed graph is an ordering of all its vertices such that, for every edge (u, v), vertex u appears before vertex v.

The topological sorting problem is to produce a topological ordering from a given graph. However, not every directed graph has a topological ordering. In particular, a directed cycle would require each vertex to come before the next, eventually requiring a vertex to come before itself. Thus, topological sorting is only possible for **directed acyclic graphs** (DAGs).

For example, consider the following directed graph:



It has several valid topological orderings; they are not necessarily unique. In particular, the chains $b \to d$ and $c \to e$ can be interleaved or done in either order, as long as both are fully completed before f. Two such topological orderings are shown below.



Using DFS to find an ordering. A natural first idea to try is to order vertices by when DFS discovers them, since DFS will traverse from a prerequisite to one of its post-requisites. Unfortunately, this does not work in general, because (a) we could start at a vertex without visiting any of its prerequisites and (b) it is not guaranteed that it will discover a vertex after discovering *all* of its prerequisites, just one of them.

This is where we need to pay closer attention to, and take advantage of, the *depth-first* nature of DFS. When a vertex is discovered, we have no way to know whether all of its prerequisites have been discovered, however, consider what we know when a vertex *finishes*.

A depth-first search from a vertex u, by design, explores every undiscovered vertex reachable from u before finishing u. Therefore, when u finishes, all of its *post-requisites* have finished, and hence it is valid to put u *before* all of those vertices in a topological order. So, a DFS can be used to build a topological order in *reverse order*.

Let's consider this in our three-colour DFS framework: Suppose we examine an edge (u, v):

- If v is WHITE, DFS recursively explores v, so v finishes before u.
- If v is GRAY, we found a directed cycle, hence the graph does not have a topological order.
- If v is BLACK, then v has already finished before u.

Therefore, for every edge (u, v), vertex v finishes before u. In other words, the finishing order is exactly the *reverse* of a topological ordering. Our algorithm therefore simply records each vertex when it finishes, then reverses the resulting sequence.

Algorithm: Topological Sorting

```
fun topological_sort(adj : UnweightedAdjList) -> sequence<int>:
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  answer = []   // mutable dynamic array

  fun dfs(u: int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == GRAY:
        raise "Not acyclic" // error: not a DAG
      if color[v] == WHITE:
        dfs(v)
    answer.append(u)     // FINISH(u)
    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

  return reverse(answer)
```

The algorithm uses the "DFS All" pattern to ensure that every vertex appears in the result, even if the graph is disconnected.

The sequence `answer` initially contains the vertices in reverse topological order. Reversing it produces a topological ordering.

Lemma 1

When the above algorithm is run on a DAG, it returns a valid topological ordering.

Proof. Every vertex is visited exactly once and appended to `answer` when it finishes, so the result contains every vertex exactly once.

Consider any edge (u, v). As argued above, when DFS examines this edge, v is either WHITE or BLACK, since a GRAY destination would imply a directed cycle. If v is WHITE, its recursive exploration finishes before u finishes. If v is BLACK, it has already finished. In either case, v is appended to `answer` before u. Consequently, after reversing `answer`, every edge (u, v) has u appearing before v, which is precisely a topological ordering. □

The algorithm performs $O(n+m)$ work and has $O(n+m)$ span. Each vertex and edge is processed at most once, appending to a dynamic array takes $O(1)$ amortized work, and reversing the resulting sequence takes $O(n)$ work.

3.6 DFS Timestamps

Our three-colour DFS template tells us which vertices have not yet been discovered, which are currently being explored, and which have finished. However, these colours only describe the *current state* of the traversal. They do not record the order in which vertices were discovered and finished. We can obtain more information about the structure of a DFS traversal by recording two **timestamps** for each vertex:

- The **start time** of u, denoted s_u, records when DFS first discovers u.
- The **finish time** of u, denoted f_u, records when DFS finishes exploring u.

We assign these timestamps using a counter that increments whenever a vertex is discovered or finishes. The counter starts at 0, so the resulting timestamps are precisely the integers $0, \ldots, 2n-1$, each appearing exactly once.

Notice that this scheme strictly generalizes the three-colour DFS:

- **WHITE**: `start[u] == NONE`
- **GRAY**: `start[u] != NONE` and `finish[u] == NONE`
- **BLACK**: `finish[u] != NONE`

In our pseudocode we will use `NONE`[2] to indicate that a timestamp has not yet been assigned. Although the sequences temporarily contain optional values, the "DFS All" pattern guarantees that every vertex receives both timestamps. Thus, the returned sequences contain only integers.

Algorithm: DFS Timestamps

```
fun dfs_times(adj: UnweightedAdjList)->(sequence<int>,sequence<int>):
  n = |adj|
  start = [NONE for _ in 0...n-1]
  finish = [NONE for _ in 0...n-1]
  time = 0

  fun dfs(u: int):
    start[u] ← time++       // DISCOVER(u)
    for v in adj[u]:
      // EXAMINE(u,v)
      if start[v] == NONE:
        dfs(v)
        // RETURN(u,v)
    finish[u] ← time++      // FINISH(u)

  for u in 0...n-1:
    if start[u] == NONE:
      dfs(u)

  return (start, finish)
```

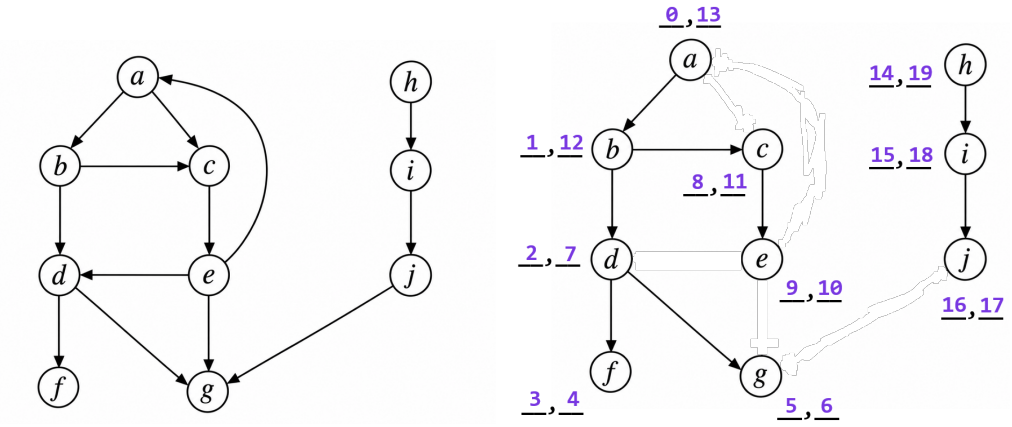
[2] In an actual programming language you might choose to implement this using a sentinel value like `-1`, or as `option<int>` values which are then unwrapped before returning them.

This gives us our most powerful and general variant of DFS, which computes and returns a history of its traversal order in the form of timestamps and has four customizable events:

- **`DISCOVER(u)`**: Record some information each time a new vertex is discovered
- **`EXAMINE(u,v)`**: Record some information each time an edge is processed
- **`RETURN(u,v)`**: Record some information after a vertex explores one of its neighbours
- **`FINISH(u)`**: Record some information after a vertex is finished being explored

As before, the algorithm performs $O(n+m)$ work and has $O(n+m)$ span.

Let's demonstrate this with an example.



The graph on the left is the original, while the graph on the right is a **DFS spanning forest**. Each edge in this forest corresponds to a recursive call from a vertex to a previously undiscovered neighbor. The forest contains one tree for each new DFS started by the "DFS All" pattern.

Each vertex u is labeled with its start and finish times, s_u and f_u. Notice that these numbers depend on the order in which vertices and their neighbors are explored.

Properties of DFS timestamps. Because DFS fully explores each vertex before returning to its parent, the start and finish times reflect the structure of the DFS forest. In particular, the exploration of a descendant must occur entirely within the exploration of its ancestors, while the explorations of unrelated vertices cannot overlap.

Claim: DFS Timestamp Intervals

For any two distinct vertices u and v, the intervals $[s_u, f_u]$ and $[s_v, f_v]$ are either *nested* or *disjoint*. Furthermore, u is an ancestor of v in the DFS forest if and only if

$$s_u < s_v < f_v < f_u.$$

This property allows us to determine whether one vertex is an ancestor of another in the DFS forest using only $O(1)$ time, after the initial DFS traversal.

3.7 Classification of DFS Edges

Recall that the edges along which DFS recursively discovers new vertices form a spanning forest. However, the original graph may contain many other edges. We can classify the edges of a directed graph into categories, depending on the relationship between their endpoints.

Definition: Classification of DFS Edges

Consider a directed edge (u, v) examined during DFS:

- A **tree edge** is an edge used to discover a new vertex v.
- A **back edge** connects a vertex u to one of its ancestors v in the DFS forest (or to itself)
- A **forward edge** connects a vertex u to one of its descendants v, but is not a tree edge.
- A **cross edge** connects two vertices that are not ancestors or descendants of each other.

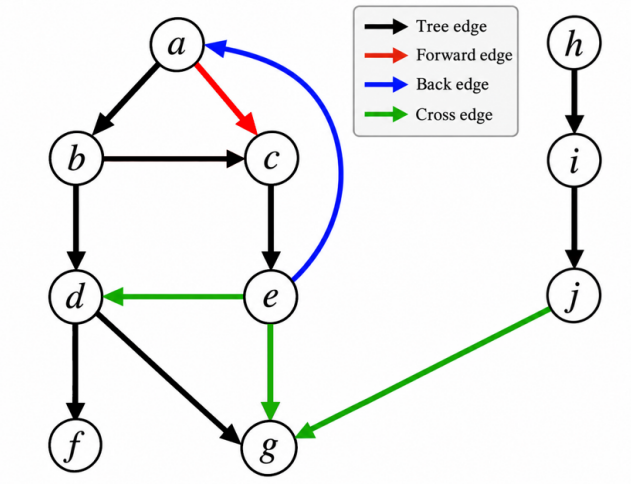
Theorem: Edge Classification via Timestamps

For an edge (u, v) with $u \neq v$, the possible timestamp relationships are:

- **Tree or forward:** $s_u < s_v < f_v < f_u$,
- **Back:** $s_v < s_u < f_u < f_v$,
- **Cross:** $s_v < f_v < s_u < f_u$

The first two cases follow from the ancestor relationships we established earlier. For a cross edge (u, v), the exploration of v must have finished before u was discovered. Otherwise, DFS would have discovered v while exploring u, making it a descendant instead.

Notice that tree and forward edges have identical timestamp relationships. To distinguish them, we must also know whether the edge was actually used to discover v. The classification is therefore not unique for a given graph; it depends on the particular DFS traversal, including the order in which vertices and their neighbors are explored.



In this example, the tree edges are shown in black, forward edges in red, back edges in blue, and cross edges in green. For instance, (a, c) is a forward edge, (e, a) is a back edge, and (e, d) is a cross edge. Note that cross edges may occur between vertices in the same component of the forest or between different components.

Finally, recall that our directed cycle detection algorithm identified cycles by finding edges to GRAY vertices. We can now restate that result in terms of our edge classification:

Theorem: Directed Cycles and Back Edges

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

This is precisely the result we proved earlier using three-colour DFS.