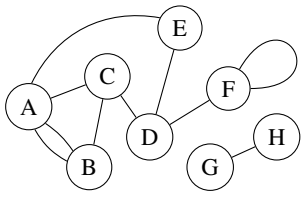
Chapter 19

Graphs and Elementary Graph Algorithms

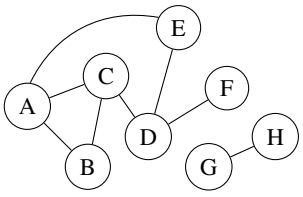
19.1 Introduction to Graphs

In the previous chapter, we discussed the concept of a tree. However, trees actually fall into a broader category of structures known as graphs. A **graph** $G = (V, E)$ is defined as a set of vertices $V = \{v_1, v_2, \dots\}$ and a set of edges $E = \{e_1, e_2, \dots\}$ that connect pairs of vertices. We often describe edges in a graph using pairs of vertices, where the notation $e_m = (v_s, v_t)$ indicates that edge m connects vertices s and t.

Compared to a tree, the definition of a graph is not as strict, as any structure constructed using a collection of vertices and edges can be considered a graph. Because of this, graphs can be categorized into many different groups based on their individual characteristics. One such category, as we have seen, is the *tree*: an acyclic, connected graph comprised of a collection of nodes (vertices) that are connected by edges. However, graphs can take on many more forms, beyond the stringent rules that define a tree. For example, it is entirely possible for a graph to have *parallel edges* (two edges connecting the same pair of vertices) or *self loops* (an edge that connects a vertex with itself). The following graph is an example that incorporates these two features:

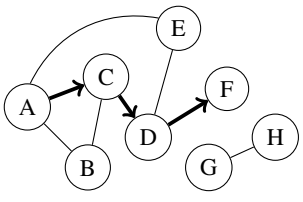


The simplest type of graph, as its name implies, is the **simple graph**. A simple graph is a graph without any parallel edges or self loops. The graph above is not a simple graph, because there is a parallel edge connecting vertices A and B, as well as a self loop on vertex F. However, if we remove these features, then the graph would become a simple graph.

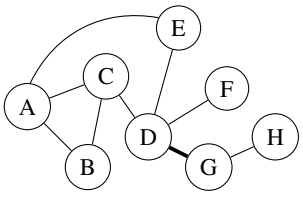


Notice that a graph need not be fully connected for it to be considered a simple graph. Vertices G and H in the previous graph are disconnected from the other vertices, but the graph is still a simple graph since it has no parallel loops or self loops. For the rest of this chapter (and for this class in general), you may assume that a graph is simple unless stated otherwise.

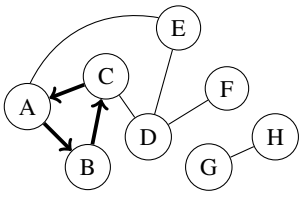
A **simple path** in a graph is a sequence of edges leading from one vertex to another with no vertex appearing twice. For example, the following is a simple path from vertex A to vertex F:



We consider a graph to be a **connected graph** if a simple path exists between any pair of vertices in the graph. A connected graph does *not* imply that every vertex is connected to every other vertex (such a graph would be a *complete graph*, which we will cover later); it only implies that there exists at least one path that can get you from any vertex to any other vertex in the graph. For instance, the graph above is *not* a connected graph because there is no way to reach vertices G or H from any of the other vertices. In order for the graph to be connected, we would have to connect either vertex G or vertex H with any of the other six vertices, as shown below:

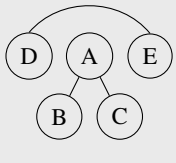


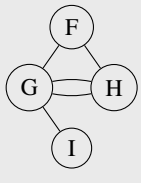
A **cycle** is a path with distinct edges in the graph where the starting and ending vertices are the same. For example, the path $A \to B \to C \to A$ is a cycle in the following graph, since the path starts and ends on vertex A.

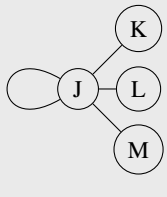


If there exist no cycles in a graph, then that graph is considered **acyclic**.

Example 19.1 Consider the following three graphs. Which ones are simple graphs? Which ones are connected graphs? Which ones are acyclic graphs?



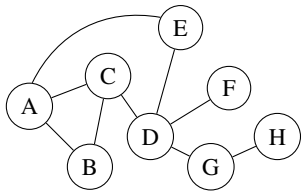




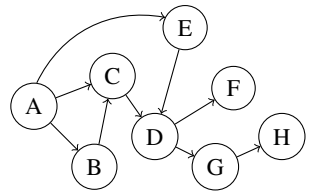
The first graph is simple and acyclic, but not connected (since there is no path from vertices D or E to vertices A, B, and C). The second graph is connected, but not simple nor acyclic (since there are parallel edges, and $F \to G \to H \to F$ is a cycle in the graph). The third graph is also connected, but not simple nor acyclic (since there is a self loop, which trivially implies that there exists a cycle).

We can also categorize graphs based on the nature of their edges. A **directed graph** (or *digraph*) is a graph whose edges have direction. Edges in a directed graph are often denoted using an ordered pair of vertices, where the notation $e_m = (v_s, v_t)$ indicates that there is an edge *from* v_s *to* v_t, but not necessarily the other way around. On the other hand, an **undirected graph** is a graph whose edges do not have direction. In an undirected graph, the order of vertices in an edge does not matter, and a connection from vertex v_s to vertex v_t implies that there is also a connection from vertex v_t to vertex v_s.

Undirected Graph

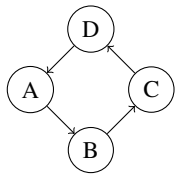


Directed Graph

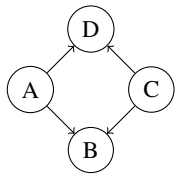


The concept of edge direction throws a wrench into our definition of a connected graph. Even though there are no disjoint vertices in the directed graph on the previous page, the direction of the edges makes it impossible to travel between all pairs of vertices. For instance, even though there exists an edge between vertex *A* and vertex *C*, there is no way to travel from vertex *C* to vertex *A* since the edge only supports one direction. In such a case, would we still consider the graph as connected?

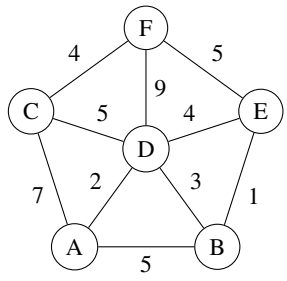
To handle this issue, "connected" directed graphs are split into two categories. A **strongly connected graph** is a directed graph where there exists a valid path between every pair of vertices, even after edge direction is taken into account. For example, the following graph is strongly connected, as there exists a valid, directional path between all pairs of vertices.



On the other hand, a **weakly connected graph** is a directed graph that does not guarantee a valid path between every pair of vertices (due to edge direction), but would be connected if all of its directed edges were replaced with undirected edges. For example, the following graph is weakly connected, because there is no path from vertices *B* or *D* to any of the other vertices, even though the graph would be connected without considering edge direction.

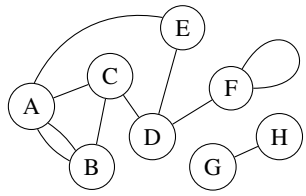


Another way we categorize graphs is by looking at whether their edges are weighted or unweighted. In a **weighted graph**, edges may be assigned a "weight" value that represents the distance or cost associated with traversing that edge. An example of a weighted graph is shown below:

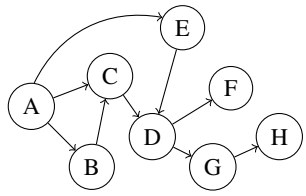


On the other hand, the edges in an **unweighted graph** do not have any weight value associated with them. As a result, you can think of an unweighted graph as a graph where all the edges have the same distance or cost. The distinction between weighted and unweighted edges is important, as it could determine the graph algorithm that is best suited to solve a problem. Some algorithms, such as the standard breadth-first search and depth-first search algorithms, are designed for unweighted graphs, as they simply search for whether a path exists between two vertices. Other algorithms, such as Dijkstra's algorithm and the Floyd-Warshall algorithm, are designed for weighted graphs, as they optimize for the lowest *weighted* path between two vertices (these "shortest path" graph algorithms will be covered in chapter 25).

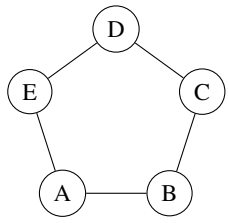
Another common term you may see with graphs is the term "degree". The **degree** of a vertex in a graph is the number of edges that are incident to that vertex, where self loops are counted twice. For example, the degree of vertex C in the following graph is three, since there are three edges that touch the vertex. The degree of vertex F is also three, since the self loop is counted twice, once for each end.



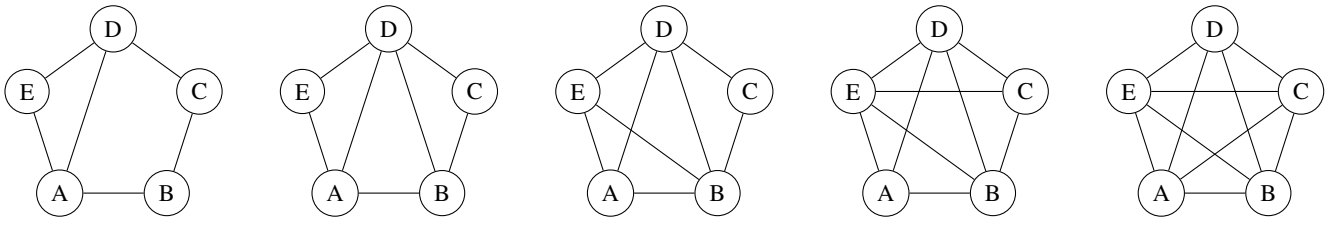
If a graph is directed, we also consider the direction of each edge when calculating degree. To do so, we split the concept of degree into in-degree and out-degree. The **in-degree** of a vertex is the total number of incoming edges that are directed toward it, while the **out-degree** of a vertex is the total number of outgoing edges that are directed away from it. For example, the in-degree of vertex C in the following directed graph is two, since there are two edges that are directed toward it (from vertices A and B). The out-degree of vertex C is one, since only one edge is directed away from it (toward vertex D).



Another useful characteristic of a graph is the *density* of its edges. For example, the following graph is rather sparse, since its vertices are not connected using many edges.



However, we can add edges to this graph to increase its density:



We can use the density of a graph to categorize graphs into two groups: sparse graphs and dense graphs.[1] The formal definitions of these two types of graphs are described below:

- A **sparse graph** is a graph where the number of edges $|E|$ is on the order of the number of vertices $|V|$ (i.e., $|E|$ is $O(|V|)$).
- A **dense graph** is a graph where the number of edges $|E|$ is on the order of the number of vertices squared (i.e., $|E|$ is $\Theta(|V|^2)$).

The distinction between sparse and dense plays a important role in determining how a graph should be represented in memory. We will discuss this in the next section.

A **complete graph** is a type of dense graph where every pair of distinct vertices is connected by a unique edge. You can think of a complete graph as the "densest" graph possible, where all pairs of vertices have a direct connection. To calculate the number of edges required to construct a complete graph with n vertices, we can use the equation

$$|E_{\text{connected}}| = \binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

This equation works because a complete graph has an edge between every pair of vertices, and there are $\binom{n}{2}$ distinct pairs of vertices that can be connected using an edge.

[1] The idea of sparse vs. dense is useful for selecting graph data structures or algorithms, but the distinction can be vague at times, especially if the graph at hand is substantially small. Because of this, we will typically refer to graphs that are *not* trivially small when discussing the concept of graph density, as to reduce the ambiguity between the two categorizations.

A good strategy for determining whether a graph is sparse or dense is to look at its degree as its size increases. If the average degree of the graph stays relatively constant as the number of vertices increases, then the graph is sparse. However, if the average degree of the graph grows with the number of vertices, then the graph is dense. A few examples are covered below:

Example 19.2 Consider a graph G that fits the definition of a tree. Is G a sparse or dense graph?

By definition, a tree is an acyclic, connected graph. Because of this, we can conclude that a tree with $|V|$ nodes must have $|V|-1$ edges. Thus, $|E|$ is $O(|V|)$, so G must be sparse.

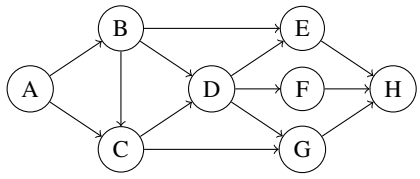
Example 19.3 Consider a graph G that represents the internet, where each vertex represents a web page, and each edge represents a hyperlink that connects two web pages. Is G a sparse or dense graph?

The number of links on each web page in the graph is relatively constant and does not depend on the total number of pages on the internet, so $|E|$ is $O(|V|)$. Thus, G must be sparse. (For G to be dense, each web page would have to link to pretty much every other web page on the internet!)

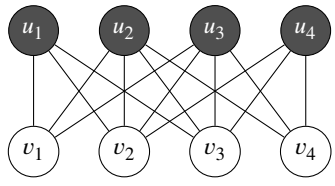
Example 19.4 Consider a graph G that represents a dictionary, where each vertex represents a word in the dictionary, and each edge connects two words that share the same letter (e.g., the word "apple" in this graph would have a direct connection with every other word that starts with "a"). Is G a sparse or dense graph?

In this case, there are 26 disjoint sets within the graph (one for each letter). On average, each of these disjoint sets must have size $|V|/26$, where $|V|$ represents the total number of words in the dictionary. Since each of these disjoint sets is also *complete*, the average number of edges in each disjoint set is $\binom{|V|/26}{2} = \Theta(|V|^2)$. Thus, $|E|$ is $\Theta(|V|^2)$, and G must be dense. You can also obtain this result if you use the degree approach: the average degree of any word in the graph is $|V|/26$, which grows with the size of the graph. This indicates that G is dense, which matches our conclusion from before.

There are two other categories of graphs that you may encounter when working with graph problems. One such graph is the **directed acyclic graph (DAG)**, which is a directed graph with no cycles. This type of graph is fairly common in many computer science problems, as directed acyclic graphs are able to elegantly characterize problems that involve dependencies (for example, Git version control can be represented using a DAG). An example of a directed acyclic graph is shown below. One important algorithm for solving problems involving these graphs is *topological sort*, which will be covered in a later section.



Another special type of graph you may encounter is the **bipartite graph**. A bipartite graph is a graph whose vertices can be split into two independent groups U and V, such that every edge connects a vertex in U to a vertex in V. For a graph to be bipartite, you must be able to color it using two colors such that every edge connects two nodes of *different* colors. This invariant also implies that bipartite graphs cannot have any cycles of odd length. An example of a bipartite graph is shown below:



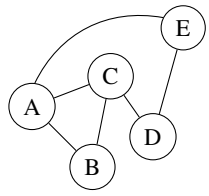
Bipartite graphs serve as a foundation for many important graph algorithms, such as network flow and graph matching.

19.2 Graph Representations

Two important graph ADTs that can be used to represent a graph in memory are the *adjacency matrix* and *adjacency list*. The benefits of choosing one implementation over the other depend on the type of graph problem you are working on.

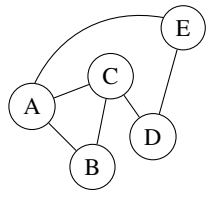
※ 19.2.1 *Adjacency Matrix*

An **adjacency matrix** is a matrix m where the cell $m[i][j]$ represents whether an edge exists from vertex i to vertex j. In an unweighted graph, the element at row i, column j of the adjacency matrix is 1 if and only if the edge (v_i, v_j) exists in the set of edges E, and 0 otherwise. For *undirected* graphs, an adjacency matrix is always symmetric across the diagonal (since a connection between i and j also implies a connection between j and i). An example of an adjacency matrix for an undirected graph is shown below:



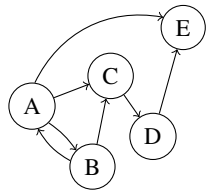
	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Because the adjacency matrix of an undirected graph is always symmetric, we can save time and space by only filling out half of the matrix. If we do this, we must make sure to access the rows and columns in the correct order (such as making sure that the row index is never larger than the column index).



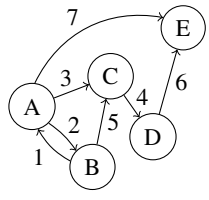
	A	B	C	D	E
A	0	1	1	0	1
B	-	0	1	0	0
C	-	-	0	1	0
D	-	-	-	0	1
E	-	-	-	-	0

However, in a *directed* graph, we will have to fill out the entire matrix. This is because the existence of a path from vertex A to B does not mean that there is one the other way around. An example of an adjacency matrix for a directed graph is shown below:



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

An adjacency matrix can also be used to represent *weighted* graphs. If a graph is weighted, the cell $m[i][j]$ would store the weight of the edge from vertex i to vertex j. If no edge exists between vertex i and vertex j, then the cell $m[i][j]$ is assigned an edge weight of infinity.[2] An example is shown below:



	A	B	C	D	E
A	∞	2	3	∞	7
B	1	∞	5	∞	∞
C	∞	∞	∞	4	∞
D	∞	∞	∞	∞	6
E	∞	∞	∞	∞	∞

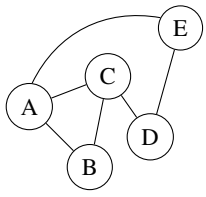
Remark: To initialize a value to ∞, you can use `std::numeric_limits<double>::infinity()` from the `<limits>` library.

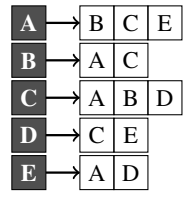
Given a graph with $|V|$ vertices, the time complexity to build an adjacency matrix is $\Theta(|V|^2)$, since you have to construct a $|V| \times |V|$ matrix. Because adjacency matrices store a value for every possible edge in the graph, the time complexity of looking up an edge is $\Theta(1)$, since you can directly index its cell in the matrix. However, adjacency matrices also have their disadvantages. The time required to iterate over all *adjacent* vertices of a vertex in an adjacency matrix is $\Theta(|V|)$, and the time required to iterate over *all* edges is $\Theta(|V|^2)$, regardless of how dense the graph is (this is because, if you wanted to find which vertices are adjacent to any vertex, you would have to iterate over the entire row of that vertex to see where there exists an edge). This is fine for dense graphs, since each vertex may have many edges, but not for sparse graphs, where most vertices do not have direct connections with each other. Thus, an adjacency matrix is *not* the best way to represent a sparse graph in memory.

[2] Depending on the problem you are trying to solve, you could assign non-existent edges with different values. In our examples, we will assign non-existent edges with a weight of ∞ because we are treating edge weight as a cost that you want to minimize. This is standard behavior for most problems we will see in this class, since weighted graphs are often used to solve minimization problems.

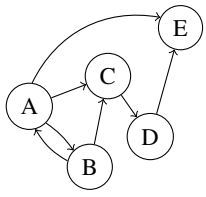
※ 19.2.2 *Adjacency List*

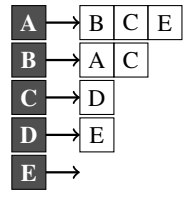
Instead, if a graph is sparse, a better alternative is to use an **adjacency list** as the underlying representation. In an adjacency list, each vertex v is mapped to a list of edges that originate from v. For example, the following depicts an adjacency list for an undirected graph, where the presence of $A \to [B, C, E]$ indicates that vertex A has a connection to vertices B, C, and E.



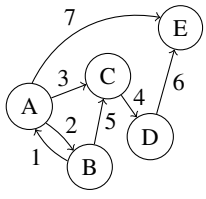


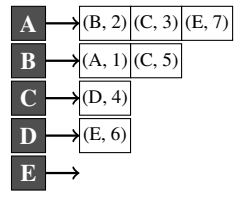
Adjacency lists also work with directed graphs and weighted graphs. The following depicts an adjacency list for a directed graph (which is similar to the undirected representation, with the exception that $X \to Y$ does not necessarily guarantee $Y \to X$).





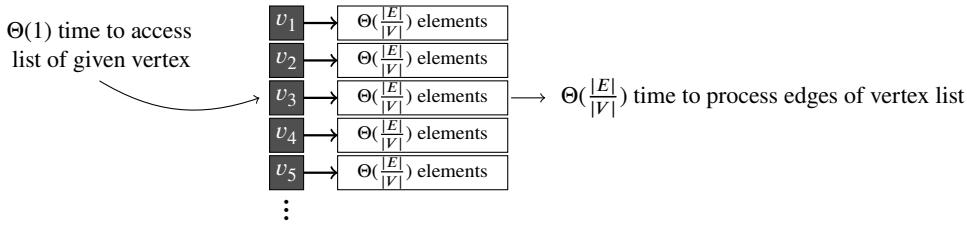
The following depicts an adjacency list for a weighted graph. When dealing with weighted graphs, each edge in the adjacency list is paired with its weight (e.g., $A \to (B, 2)$ indicates that there is an edge from A to B with weight 2).





Given a graph with $|V|$ vertices, the time complexity to build an adjacency list is $\Theta(|E|)$, and the space complexity is $\Theta(|V| + |E|)$. The time complexity is $\Theta(|E|)$ because you have to iterate through each edge and insert it into the adjacency list, where each insertion takes $\Theta(1)$ time (assuming the adjacency list is stored in a 2-D vector or a hash table). The space complexity is $\Theta(|V| + |E|)$ because storing all the vertices requires $\Theta(|V|)$ space, and storing all the edges in the lists requires $\Theta(|E|)$ space.

What is the time complexity of processing all the edges of a *single vertex* in an adjacency list? If edges are randomly distributed, each vertex list should have a length of $\Theta(|E|/|V|)$ on average (since $|E|$ edges are distributed across $|V|$ vertices). Thus, in the average case, the time it takes to process all edges of a given vertex is $\Theta(1 + |E|/|V|)$, where the $\Theta(1)$ component comes from accessing the list of the vertex, and the $\Theta(|E|/|V|)$ component comes from iterating through the entire vertex list.



Because the average cost of processing the edges of a single vertex is $\Theta(1 + |E|/|V|)$, the average cost of processing *all* vertices in the adjacency list is $\Theta(|V| \times (1 + |E|/|V|)) = \Theta(|V| + |E|)$, since a $\Theta(1 + |E|/|V|)$ operation is performed $|V|$ times. This is great for sparse graphs, since $|E| = O(|V|)$, which implies that the time complexity of iterating through all edges in an adjacency list is $\Theta(|V| + |E|) = \Theta(|V| + O(|V|)) = \Theta(|V|)$. However, for dense graphs, $|E| = \Theta(|V|^2)$, and the time complexity of iterating through all edges becomes $\Theta(|V| + |E|) = \Theta(|V| + |V|^2) = \Theta(|V|^2)$. This is the same time complexity as that of an adjacency matrix! Thus, adjacency lists do not provide a performance improvement if the underlying graph is dense (and could be arguably worse since adjacency lists do not support $\Theta(1)$ time edge lookup).

※ 19.2.3 *Summary of Graph Representations*

A summary of the two graph representations is shown in the following table:

Adjacency Matrix	Adjacency List
• Implemented using a $\|V\| \times \|V\|$ matrix. Uses $\Theta(\|V\|^2)$ space.	• Typically implemented using either a vector/list of edges or a hash table that maps each vertex to a vector/list of adjacent edges. Uses $\Theta(\|V\| + \|E\|)$ space ($\|V\|$ vertices and $\Theta(\|E\|)$ elements across all the vertex lists).
• Should be used to represent **dense** graphs.	• Should be used to represent **sparse** graphs.
• Given two vertices v_i and v_j, you can check if an edge exists between these two vertices in $\Theta(1)$ time (by accessing the value at row i, column j of the matrix).	• Given two vertices v_i and v_j, you can check if an edge exists between these two vertices in worst-case $\Theta(\|V\|)$ time (if v_i is connected to every other vertex, and v_j is at the back of its vertex list), and average-case $\Theta(\|E\|/\|V\|)$ time (assuming edges are randomly distributed).
• The time complexity of iterating over all edges in the graph is worst-case $\Theta(\|V\|^2)$.	• The time complexity of iterating over all edges in the graph is worst-case $\Theta(\|V\| + \|E\|)$.
• Adding an edge to the graph takes $\Theta(1)$ time, since you can just update the value at the correct index of the matrix.	• Adding an edge to the graph takes $\Theta(1)$ time, since you can just append the new edge to the back of the correct vertex list.
• Removing an edge from the graph takes $\Theta(1)$ time, since you can just update the value at the correct index of the matrix.	• Removing an edge from the graph takes worst-case $\Theta(\|E\|)$ time (if you end up searching all edges in the list to find the one you want to remove), and average-case $\Theta(\|E\|/\|V\|)$ time (assuming edges are randomly distributed).
• Supports both directed and undirected graphs. – If directed, the value at row i, column j indicates if an edge exists from vertex v_i to vertex v_j. – If undirected, the adjacency matrix is symmetrical across the diagonal, and thus only $\|V\|^2/2$ cells of the matrix need to be filled out (e.g., if you know that $(v_i, v_j) = 1$, you also know that $(v_j, v_i) = 1$).	• Supports both directed and undirected graphs. – If directed, the adjacency list contains each edge once in the edge set, where $v_i \to [\ldots, v_j, \ldots]$ indicates the existence of an edge from v_i to v_j. – If undirected, the adjacency list contains each edge twice in the edge set, where $v_i \to [\ldots, v_j, \ldots]$ implies that $v_j \to [\ldots, v_i, \ldots]$.
• Supports both weighted and unweighted graphs. – If weighted, row i, column j of the matrix stores the weight of the edge from vertex v_i to v_j if an edge exists, or ∞ if no edge exists. – If unweighted, row i, column j of the matrix stores 1 if an edge exists from vertex v_i to v_j, or 0 if no edge exists.	• Supports both weighted and unweighted graphs. – If weighted, each edge in the vertex list is paired with its weight, where $v_i \to [\ldots, (v_j, k), \ldots]$ indicates that the edge from v_i to v_j has weight k. – If unweighted, each edge in the vertex list appears on its own, with no associated weight value.

Example 19.5 You are given a file containing airport data. Each line of this file includes information on the distance between two connecting airports in the following format:

```
<origin airport ID> <destination airport ID> <distance in miles>
```

For example, the following data indicates that there is a connection between SFO and LAX with distance 337 miles, there is a connection between JFK and ORD with distance 740 miles, and so on.

```
SFO LAX 337
JFK ORD 740
MIA DFW 1121
SFO BOS 2704
ORD DFW 802
...
```

Implement the `AdjMatrix` and `AdjList` classes, which read in this data and store it in the form of an adjacency matrix or adjacency list, respectively. Each of these classes should support an `init()` function, which reads in the contents of the data file using input redirection (you may assume that the data file is well-formatted); an `add_edge()` and `remove_edge()` function, which adds or removes the specified edge from the graph; and an `edge_weight()` function, which returns the weight of an edge between two given airports. You may assume the graph is *directed*, where "`SFO LAX 337`" implies an edge from SFO to LAX, but not necessarily the other way around.

Adjacency matrix class:

```
class AdjMatrix {
  // use this "airports" hash table to index into the adjacency matrix
  // e.g., if "SFO" maps to 3, then "SFO" is assigned index 3 of matrix
  std::unordered_map<std::string, int32_t> airports;
  // underlying adjacency matrix
  std::vector<std::vector<double>> adj_matrix;
public:
  // this function initializes the "airports" hash table,
  // assume this has already been implemented for you
  void init_airports();
  // TODO: reads in data and initializes adjacency matrix
  void init();
  // TODO: adds edge connecting airport a1 with airport a2
  void add_edge(const std::string& a1, const std::string& a2, double weight);
  // TODO: removes edge connecting airport a1 with airport a2
  void remove_edge(const std::string& a1, const std::string& a2);
  // TODO: returns weight of edge between airports a1 and a2,
  // and infinity if the edge does not exist
  double edge_weight(const std::string& a1, const std::string& a2);
};
```

Adjacency list class:

```
struct Edge {
  std::string dest;
  double weight;
};

class AdjList {
  // underlying adjacency list
  std::unordered_map<std::string, std::vector<Edge>> adj_list;
public:
  // TODO: reads in data and initializes adjacency list
  void init();
  // TODO: adds edge connecting airport a1 with airport a2
  void add_edge(const std::string& a1, const std::string& a2, double weight);
  // TODO: removes edge connecting airport a1 with airport a2
  void remove_edge(const std::string& a1, const std::string& a2);
  // TODO: returns weight of edge between airports a1 and a2,
  // and infinity if the edge does not exist
  double edge_weight(const std::string& a1, const std::string& a2);
};
```

First, let's start with the adjacency matrix implementation. The `init()` function reads in the data from the file using input redirection and uses it to initialize the contents of the matrix. Since we have an `airports` hash table that stores a mapping from all the airports to an index, we can use it to initialize all the cells of the matrix with a starting value of ∞ (we are initializing to infinity instead of zero because the graph is weighted). Then, we read in the contents of the file line by line and add edges using the `add_edge()` function. The code is shown below:

```
void AdjMatrix::init() {
  adj_matrix.resize(airports.size(), std::vector<double>(airports.size(),
    std::numeric_limits<double>::infinity()));
  std::string orig, dest;
  double weight;
  while (std::cin >> orig >> dest >> weight) {
    add_edge(orig, dest, weight);
  } // while
} // init()
```

To implement the `add_edge()` function, we simply look at the corresponding index of the matrix based on the origin and destination airports and update its value. The code is shown below:

```
void AdjMatrix::add_edge(const std::string& a1, const std::string& a2, double weight) {
  adj_matrix[airports[a1]][airports[a2]] = weight;
} // add_edge()
```

Removing an edge follows a similar process: we check the correct index of the matrix based on the given input strings and update its value to infinity. The code is shown below:

```
void AdjMatrix::remove_edge(const std::string& a1, const std::string& a2) {
  adj_matrix[airports[a1]][airports[a2]] = std::numeric_limits<double>::infinity();
} // remove_edge()
```

To return the edge weight, we just need to return its weight value in the matrix.

```
double AdjMatrix::edge_weight(const std::string& a1, const std::string& a2) {
  return adj_matrix[airports[a1]][airports[a2]];
} // edge_weight()
```

Our implementation of the adjacency matrix is now complete. Next, we will look at the adjacency list implementation. To initialize the list with the contents of the file, we would go through each line and push the destination airport into the vector associated with the origin airport, along with its weight. The code is shown below:

```
1  void AdjList::init() {
2    std::string orig, dest;
3    double weight;
4    while (std::cin >> orig >> dest >> weight) {
5      add_edge(orig, dest, weight);
6    } // while
7  } // init()
8
9  void AdjList::add_edge(const std::string& a1, const std::string& a2, double weight) {
10   adj_list[a1].push_back(Edge{a2, weight});
11 } // add_edge()
```

To remove an edge from the adjacency list, we go through the vertex list of the origin airport and remove the edge associated with the destination airport. This is shown below, using the STL `std::remove_if()` function:

```
1  void AdjList::remove_edge(const std::string& a1, const std::string& a2) {
2    std::vector<Edge>& dest = adj_list[a1];
3    auto it = std::remove_if(dest.begin(), dest.end(), [&a2](const Edge& e) {
4      return e.dest == a2;
5    });
6    dest.erase(it, dest.end());
7  } // remove_edge()
```

To return the weight of a given edge, we would look through the vertex list of the origin airport and return the weight value associated with the destination airport. The code is shown below, using the STL `std::find_if()` function:

```
1  double AdjList::edge_weight(const std::string& a1, const std::string& a2) {
2    std::vector<Edge>& dest = adj_list[a1];
3    auto it = std::find_if(dest.begin(), dest.end(), [&a2](const Edge& e) {
4      return e.dest == a2;
5    });
6    return (it == dest.end()) ? std::numeric_limits<double>::infinity() : it->weight;
7  } // edge_weight()
```

Our implementations of both classes are now complete.

Example 19.6 You are given an adjacency *matrix* that represents a graph of airports. What are the best, worst, and average-case time complexities of determining if a direct flight exists between two given airports X and Y?

For an adjacency matrix, you can determine if an edge exists by just indexing into the correct position of the matrix, $m[X][Y]$. This takes $\Theta(1)$ time in all cases.

Example 19.7 You are given an adjacency *list* that represents a graph of airports. What are the best, worst, and average-case time complexities of determining if a direct flight exists between two given airports X and Y?

In the best case, airport Y could be at the front of airport X's vertex list, or the list of airport X could be empty; this allows you to determine whether an edge exists in $\Theta(1)$ time. In the worst case, the list of airport X has length $\Theta(|V|)$ (if it is connected to almost all other airports), and you end up having to traverse the entire list before you discover if airport Y exists as a connection; this would take $\Theta(|V|)$ time. In the average case, the list of airport X has length $\Theta(|E|/|V|)$, and you would need to traverse $\Theta(|E|/|V|)$ elements in an adjacency list before you can determine if a direct flight exists between two airports; this takes $\Theta(|E|/|V|)$ time.

Example 19.8 You are given an adjacency *matrix* that represents a graph of airports. What are the best, worst, and average-case time complexities of determining the closest other airport to a given airport X?

For an adjacency matrix, you have look at all the values in the row of airport X, which has length $\Theta(|V|)$. There is no way to do better, since you have to look at all of airport X's connections to determine which connection has the lowest weight, and the only way to look at all connections in an adjacency matrix is to iterate over all the vertices. Thus, the time complexity of this task is $\Theta(|V|)$ in all cases.

Example 19.9 You are given an adjacency *list* that represents a graph of airports. What are the best, worst, and average-case time complexities of determining the closest other airport to a given airport X? Assume the lists are not sorted by weight.

In the best case, the vertex list of airport X has one element, allowing you to discover the closest connection in $\Theta(1)$ time. In the worst case, airport X is connected to all other vertices, and you will end up searching all $\Theta(|V|)$ airports to find out which one is the closest. In the average case, the list of airport X has length $\Theta(|E|/|V|)$, allowing you to find the closest airport in average-case $\Theta(|E|/|V|)$ time.

Example 19.10 You are given an adjacency *matrix* that represents a graph of airports. What are the best and worst-case time complexities of determining if *any* flights depart from airport X?

To determine if any flights depart from airport X in an adjacency matrix, we will have to traverse the row of airport X to see where there exists a connection. As soon as we encounter the first edge that does not have a value of infinity, we can conclude that airport X has an outgoing flight. Otherwise, if all the values in the row of airport X have a value of infinity, no flights depart from airport X. In the best case, the first edge in the row is not infinity, allowing us to conclude that a departing flight exists in $\Theta(1)$ time. In the worst case, only the last edge in the row has a value that is not infinity (or all the edges have a value of infinity, when happens if airport X has no outgoing connections at all). When this happens, the entire row needs to be traversed before we can determine if a departing flight exists, which results in a runtime of $\Theta(|V|)$. (We will not discuss the average-case analysis here, since it depends on how dense or sparse the underlying graph is, but we consider it as an $\Theta(|V|)$ operation since, on average, we have to visit a fraction of all $|V|$ vertices before we find the first vertex with a connection.)

Example 19.11 You are given an adjacency *list* that represents a graph of airports. What are the best and worst-case time complexities of determining if *any* flights depart from airport X?

To determine if any flights depart from airport X in an adjacency list, we just need to check if its vertex list is empty. If the vertex list of a given airport is empty, then that airport does not have departing connections; otherwise, it does have departing connections. Checking the length of a vertex list always takes $\Theta(1)$ time.

The adjacency matrix and adjacency list are two ways we can represent the internal contents of a graph in memory. Once we represent a graph in memory, we can apply different graph algorithms on it to solve many types of important graph problems. For example, given a graph of airports and their connections, we can use graph algorithms to find the lowest-cost path between any two airports, or the path with the fewest intermediary layovers, just to name a few.

Graphs can be argued as one of the most important structures in the field of computer science, as they have enormous practical applications in many of the things we use on a daily basis. Map applications such as Google Maps use graph algorithms to calculate optimal routes between two locations. Social media sites use graphs to represent friendships and connections, and they utilize graph algorithms to help come up with friend suggestions or suggested posts. E-commerce sites like Amazon use graph algorithms to provide users with recommendations based on what they have viewed or bought before. Search engines like Google use a graph of the web to rank search results. In your own computer's operating system, graphs are used to manage the resources they are being used by running processes to ensure that deadlocks do not occur. The internet itself is also built on the foundation of graphs, and graph algorithms play a crucial role in allowing us to communicate with each other, all across the world. In fact, graph algorithms played a pivotal role in allowing professors to seamlessly livestream lectures to students all across the world, back when classes were held online amidst the global pandemic in 2020 and 2021.

Because the material on graphs is so vast, its contents will be split across several chapters (and even then, we will not be able to cover everything). In the previous chapter, we covered a restricted form of a graph known as a tree, as well as algorithms that can be used to solve tree problems. In this chapter, we covered the general implementation of a graph, and we will cover important elementary graph algorithms such as depth-first and breadth-first search. In the next chapter, we will cover a special type of graph known as a spanning tree, as well as algorithms that can be used to build spanning trees that minimize total edge weight. Lastly, in chapter 25, we will cover shortest path algorithms, which can be used to identify a path between two vertices that minimizes total distance.

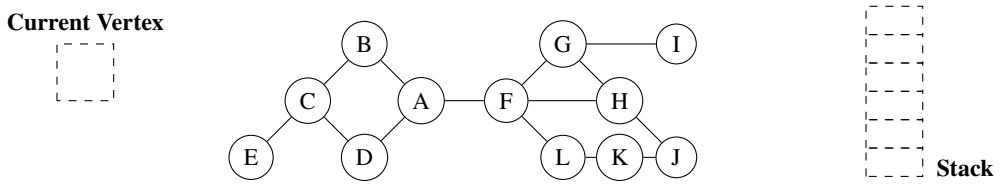
19.3 Depth-First Search

The **depth-first search (DFS)** algorithm is a core graph traversal algorithm that can be used to systematically explore the vertices and edges of a graph that are reachable from a given source vertex. In a depth-first search, the algorithm starts at a source vertex and explores each branch of the graph *as far as possible* until there are no more undiscovered vertices along its path, at which point it backtracks and continues along another branch. The implementation of a depth-first search generally relies on recursion or the `std::stack<>` data structure. An outline of the DFS algorithm is as follows:

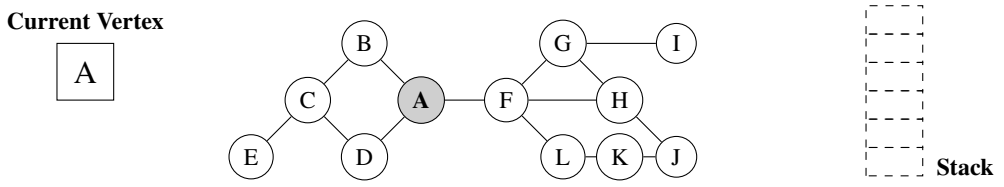
1. Mark the source vertex as *visited*, and then push its neighbors (i.e., adjacent vertices) into a stack. This can be done by pushing all neighbors into an actual `std::stack<>`, or by performing a recursive call on every neighbor (which pushes neighbors onto the program call stack).
2. Pop the vertex at the top of the stack and set it as the current vertex (this behavior is automatically done by a recursive call, if a recursive approach is used).
3. Push the unvisited neighbors of the current vertex into the stack (or perform a recursive call on each neighbor, if a recursive approach is used) and mark them as visited.
4. Repeat steps 2 and 3 until the stack is empty.

※ 19.3.1 *Iterative Depth-First Search*

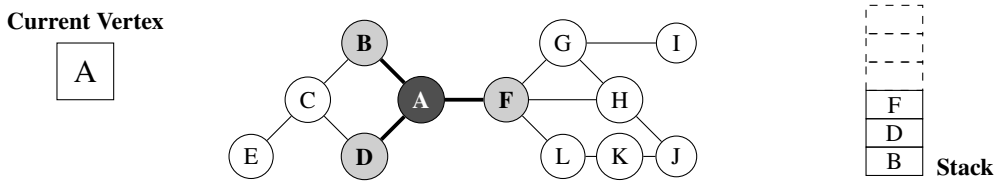
A depth-first search can be done both iteratively (using a `std::stack<>`) or recursively (using the program stack). For example, consider the following graph, which we will traverse using an *iterative* depth-first search starting at vertex A. Here, we will classify visited nodes into two groups: *discovered* and *processed*. A *discovered* vertex is a vertex that has been encountered and pushed into the stack; discovered vertices will be represented with a light gray shading. A *processed* vertex is a vertex that has already been taken out of the stack and had its adjacency list fully examined; processed vertices will be represented with a dark gray shading. We start by initializing a `std::stack<>` to track vertices yet to be explored.



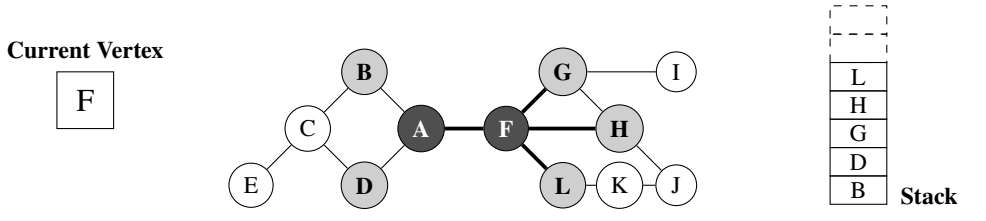
First, we will mark the starting vertex as visited. In this case, our starting vertex is vertex *A*, so we will mark *A* as visited.



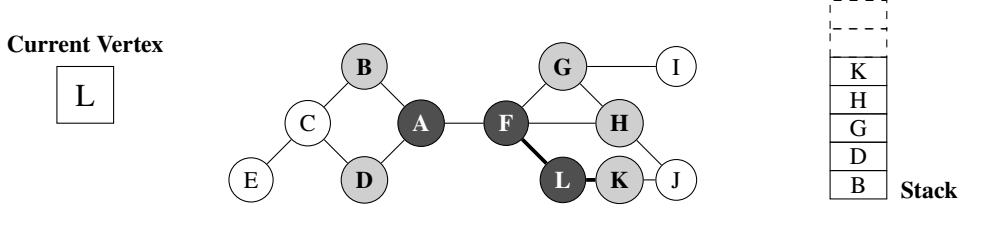
Next, we look at the vertices that vertex *A* is connected to, mark them as visited, and add them to the stack. The order we add these vertices doesn't matter, but for our example, we will add them in alphabetical order (i.e., *B* is inserted first, then *D*, then *F*).



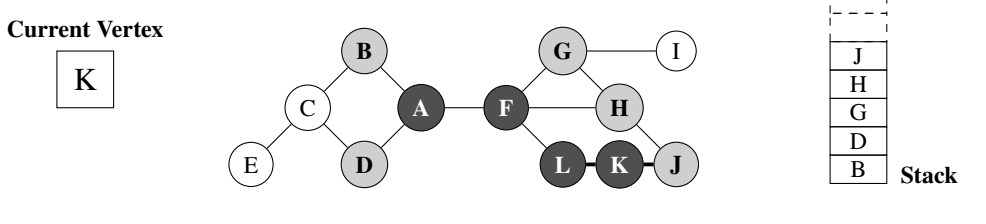
Next, we take out the vertex at the top of the stack and set it as our current vertex. Here, vertex *F* is at the top of the stack, so we pop it and set it as our current vertex. We then push all of *F*'s unvisited neighbors into the stack. Vertex *F* is connected to vertices *A*, *G*, *H*, and *L*, but only *G*, *H*, and *L* are unvisited. Thus, vertices *G*, *H*, and *L* are pushed into the stack and marked as visited.



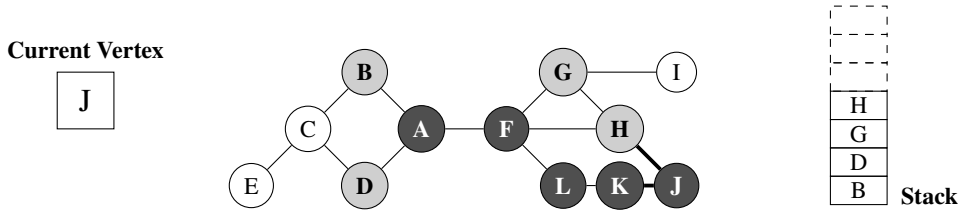
Vertex *L* is now at the top of the stack, so we pop it and set it as our current vertex. We then push all of vertex *L*'s unvisited neighbors into the stack. Vertex *L* is connected to vertices *F* and *K*, but only *K* is unvisited. Thus, only vertex *K* is pushed into the stack and marked as visited.



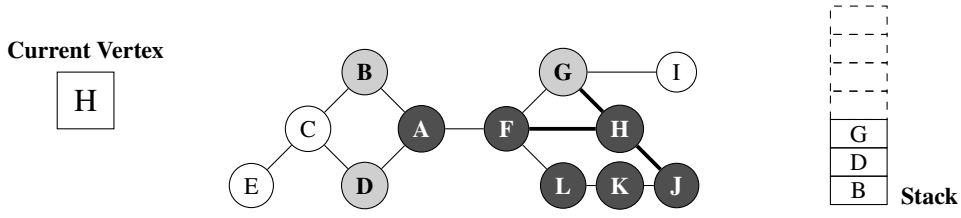
Vertex *K* is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex *K*'s unvisited neighbors into the stack. Vertex *K* is connected to vertices *L* and *J*, but only *J* is unvisited. Thus, only vertex *J* is pushed into the stack and marked as visited.



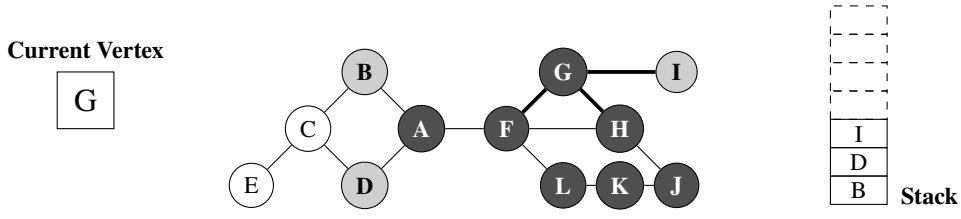
Vertex J is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex J's unvisited neighbors into the stack. Vertex J is connected to vertices H and K — however, both H and K have already been marked as visited. Thus, nothing is pushed into the stack at this step.



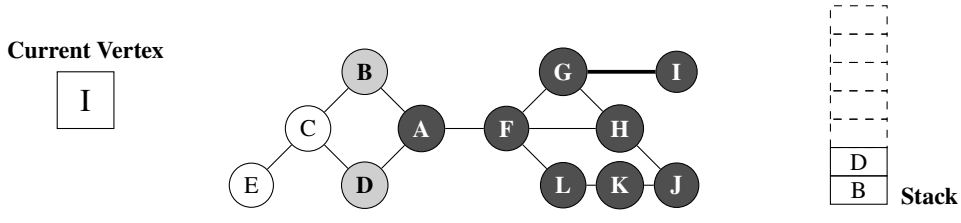
Vertex H is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex H's unvisited neighbors into the stack. Vertex H is connected to vertices F, G, and J, but similar to before, all three of these vertices have been marked as visited. Thus, nothing is pushed into the stack at this step.



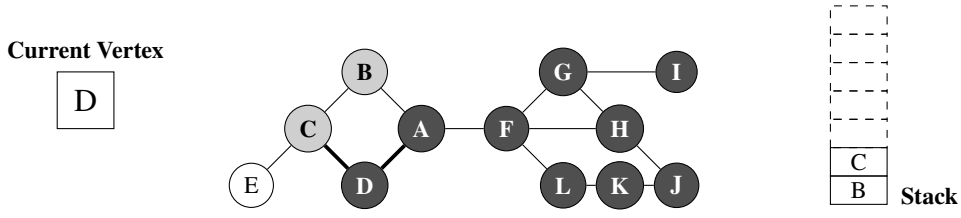
Vertex G is at the top of the stack, so we pop it and set it as our current vertex. We then push all of vertex G's unvisited neighbors into the stack. Vertex G is connected to vertices F, H, and I, but only vertex I is unvisited. Thus, vertex I is pushed into the stack and marked as visited.



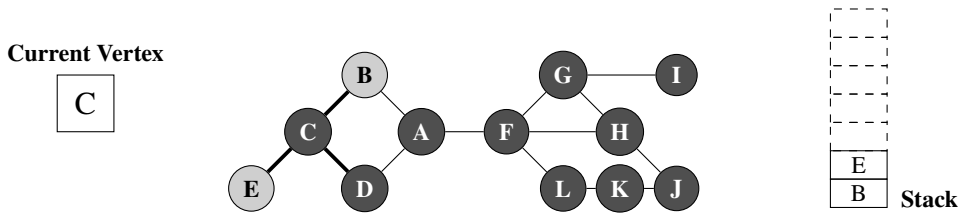
Vertex I is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex I's unvisited neighbors into the stack. Vertex I is connected to vertex G, but vertex G has already been marked as visited. Thus, nothing is pushed into the stack at this step.



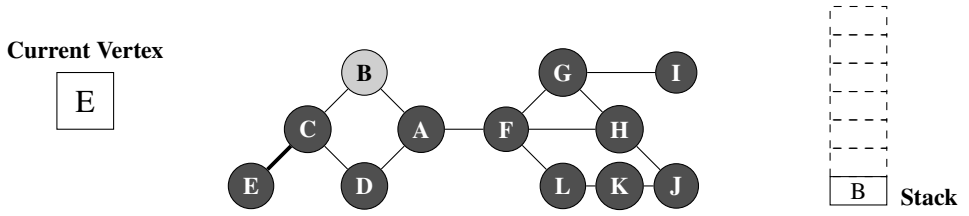
Vertex D is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex D's unvisited neighbors into the stack. Vertex D is connected to vertices A and C, but vertex A has already been marked as visited. Thus, only vertex C is pushed into the stack and marked as visited.



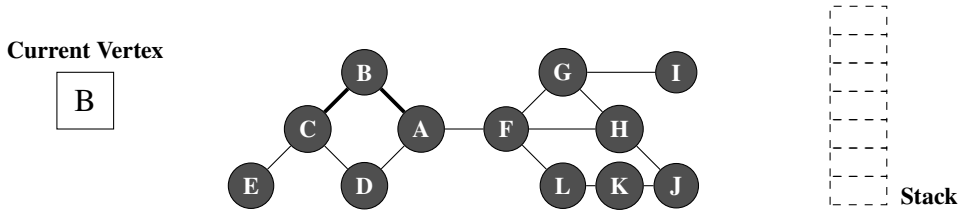
Vertex C is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex C's unvisited neighbors into the stack. Vertex C is connected to vertices B, D, and E, but only vertex E has not yet been visited. Thus, vertex E is pushed into the stack and marked as visited.



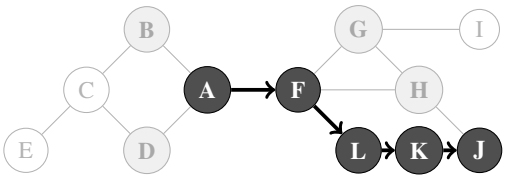
Vertex E is now at the top of the stack, so we pop it and set it as our current vertex. We then push all of vertex E's unvisited neighbors into the stack. Vertex E is connected to vertex C, but vertex C has already been marked as visited. Thus, nothing gets pushed into the stack at this step.



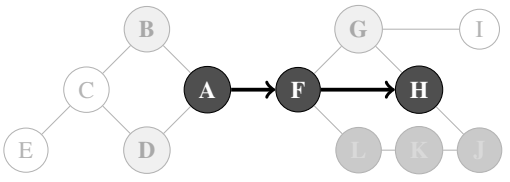
Vertex B is now at the top of the stack, so we pop it off and set it as our current vertex. We then push all of vertex B's unvisited neighbors into the stack. Vertex B is connected to vertices A and C, but both of these vertices have already been marked as visited. Thus, nothing gets pushed into the stack at this step.



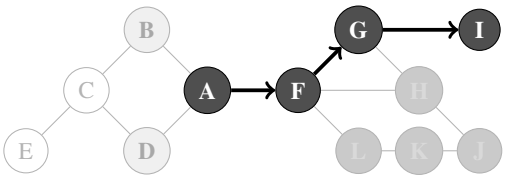
The stack is now empty, so our depth-first search is complete, and we have processed every vertex reachable from vertex A. Notice that, because a depth-first search relies on a stack to process the nodes in a graph, the algorithm explores and processes vertices in the graph one branch at a time in LIFO order. Here, the depth-first search first processed the branch $A \to F \to L \to K \to J$:



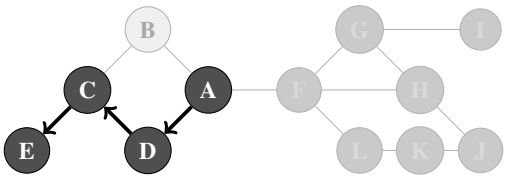
After processing these five vertices, the search then backtracks to vertex F and explores the new branch $A \to F \to H$, which adds vertex H to the list of processed nodes.



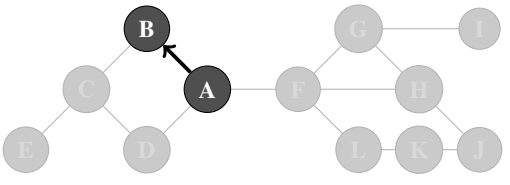
The search then backtracks to vertex F and explores the branch $A \to F \to G \to I$, which adds vertices G and I to the list of processed nodes.



The search then backtracks to vertex A and explores the branch $A \to D \to C \to E$, adding vertices D, C, and E to the list of processed nodes.



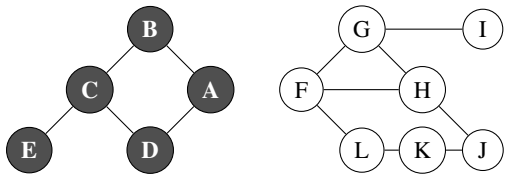
The search then backtracks to vertex A and explores the final branch $A \to B$, which adds vertex B to the list of processed nodes.



Why do depth-first searches exhibit this "branch by branch" exploration behavior? This is a consequence of how stacks work. Because elements in a stack can only be cleared from top to bottom, as long as a branch of the graph still has unvisited vertices remaining in its path, these vertices will get pushed to the top of the stack and will thus be processed before other vertices lower in the stack. As a result, if a depth-first search starts exploring a branch of the graph, it will continue exploring that branch until it runs out of unvisited elements to push into the stack.

The order in which branches are visited and the specific vertices that end up in each branch depend on the order in which vertices are pushed into the stack. In our iterative DFS example, because vertex F was pushed into the stack *after* vertices B and C, branches starting with $A \rightarrow F \rightarrow \ldots$ were explored first because vertex F ended up at the top of the stack. Furthermore, because vertex D being pushed into the stack *after* vertex B, we ended up exploring the branch $A \rightarrow D \rightarrow C \rightarrow E$ instead of the branch $A \rightarrow B \rightarrow C \rightarrow E$ (note: if a recursive depth-first search were performed instead, the behavior would be slightly different, since a recursive call on one neighbor runs to completion before a recursive call is made on another neighbor; we will cover the recursive implementation on the next page).

Because a depth-first search visits all vertices that are accessible from the starting vertex, it can be used to determine if a simple path exists between any two vertices in a graph. If it is possible to reach a vertex Y from a vertex X, then vertex Y must be encountered at some point during the depth-first search starting at vertex X. On the other hand, if vertex Y is not accessible from vertex X (e.g., it is in a separate disconnected component), then the depth-first search starting at X will never visit Y before completion. For example, if we disconnected the edge between vertices A and F, then a depth-first search starting at vertex A will only process vertices A, B, C, D, and E, as the remaining elements will never be pushed into the stack.

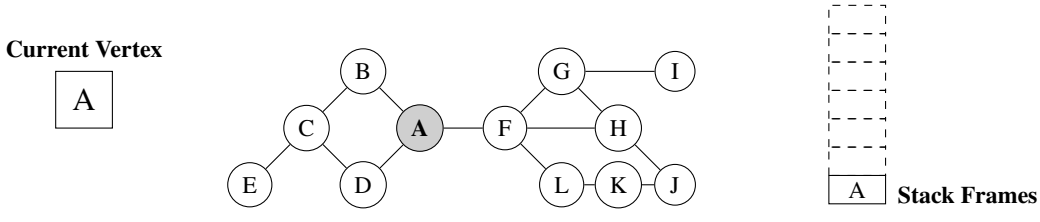


The pseudocode for determining if a path exists between two vertices using a depth-first search is shown below:

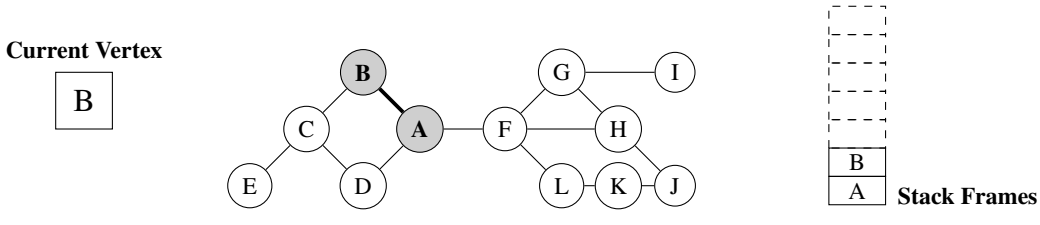
```
1   Algorithm DFS(source, destination):
2     mark source as visited
3     push source into stack
4     while stack is not empty:
5       get/pop candidate from top of stack
6       for each neighbor of candidate:
7         if neighbor is unvisited:
8           mark neighbor as visited
9           push neighbor to top of stack
10          if neighbor is destination:
11            return true (success)
12    return false (failure)
```

※ 19.3.2 *Recursive Depth-First Search*

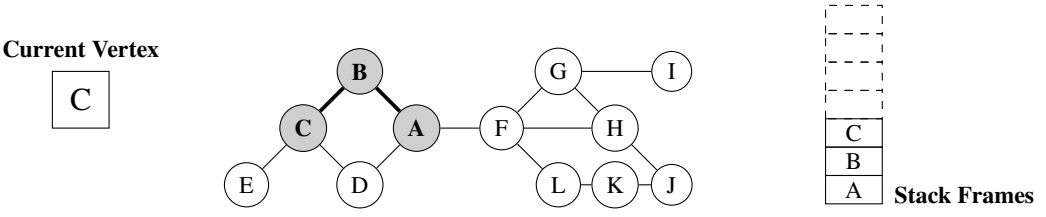
A depth-first search can also be performed recursively rather than iteratively. In the recursive approach, we will push vertices onto the program stack via recursive calls. By using recursive stack frames to keep track of vertices yet to be explored, we bypass the need to explicitly declare a separate container, which we needed with the iterative approach. Consider the same graph as before, but this time, we will perform a *recursive* depth-first search starting at vertex A.



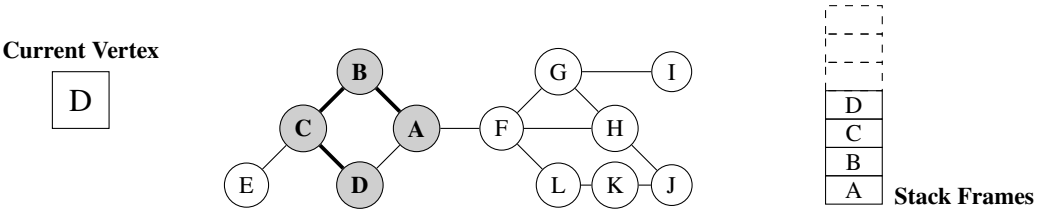
Previously, in an iterative depth-first search, we would push all of vertex A's neighbors into a stack. However, in the recursive approach, we can accomplish this behavior by *performing a recursive call* on each of vertex A's unvisited neighbors. Similar to before, we will conduct recursive calls in alphabetical order, so we will recursively search vertex B first. The recursive call pushes vertex B to the top of the call stack and sets it as our current vertex.



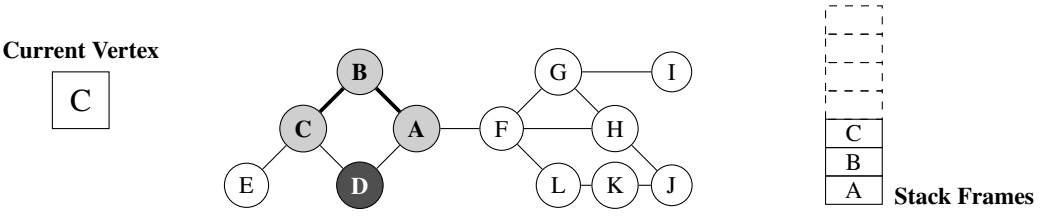
We then make recursive calls on all of vertex B's unvisited neighbors. Vertex B has two neighbors, A and C, but A has already been visited. Thus, we will make a recursive call on vertex C, which becomes our current vertex.



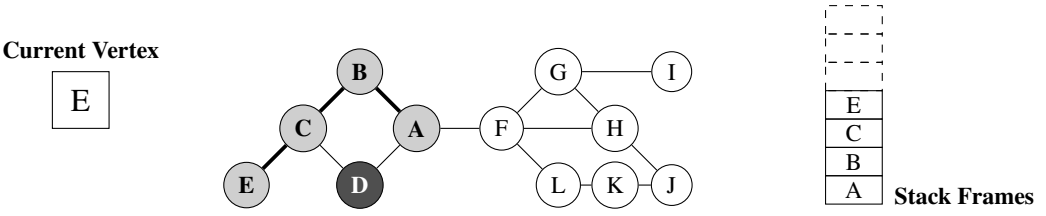
Next, we will make recursive calls on all of C's unvisited neighbors. Vertex C has three neighbors: B, D, and E. Vertex B has already been visited, so we will make a recursive call on vertex D, which becomes our current vertex.



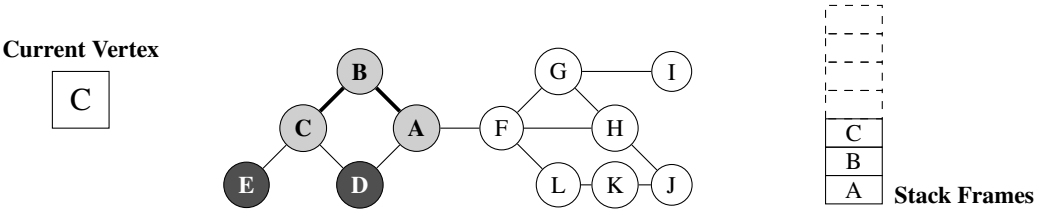
We then make recursive calls on all of vertex D's unvisited neighbors. Since all of D's neighbors have been visited, no recursive calls are made, and the recursion unrolls back to vertex C.



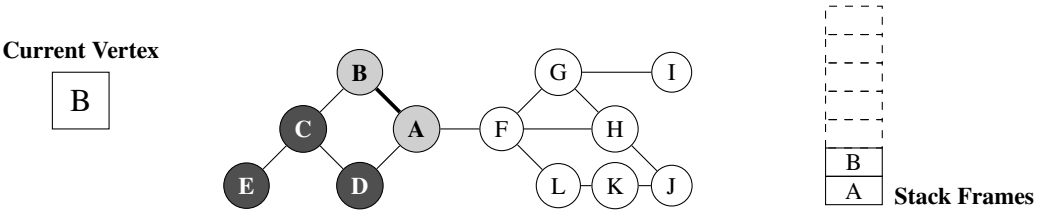
Next, we will make a recursive call on vertex C's remaining unvisited neighbor, vertex E.



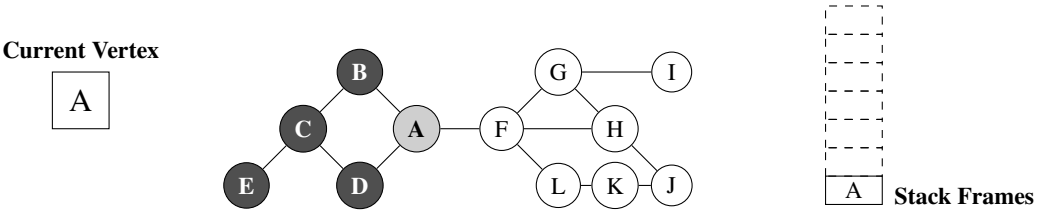
Vertex E has no unvisited neighbors, so the recursion unrolls back to vertex C.



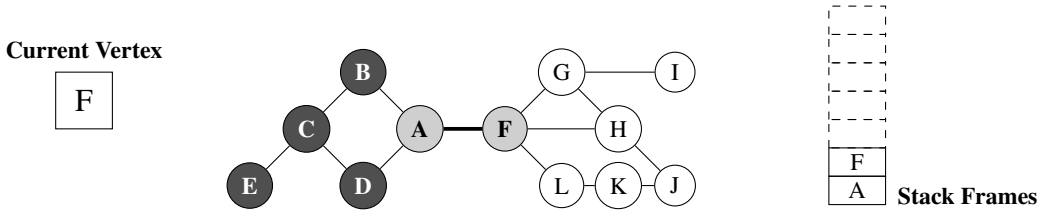
All of vertex C's neighbors have been visited, so the recursion unrolls back to vertex B.



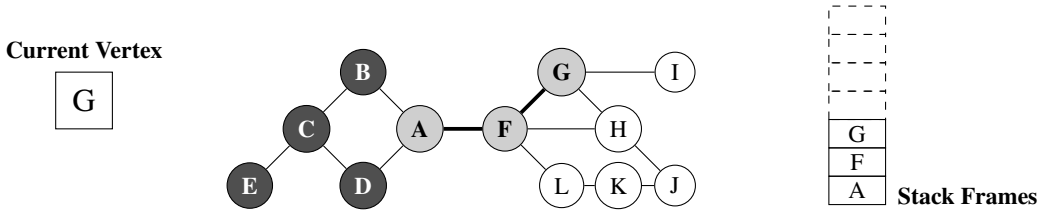
All of vertex B's neighbors have been visited, so the recursion unrolls back to vertex A.



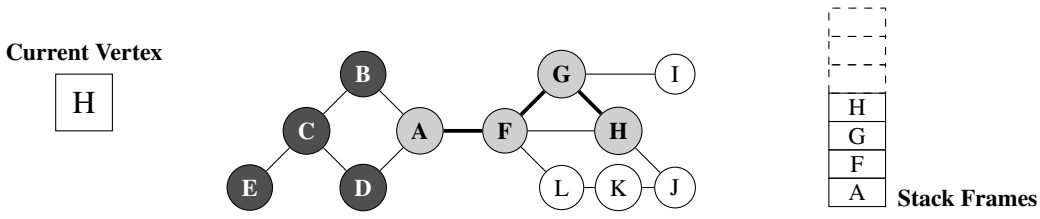
Now, we will make a recursive call on vertex F, which is vertex A's only remaining unvisited vertex. Note that vertex D was unvisited before the recursive call to B, but the recursive call to B ended up visiting D, so we don't need to visit it again.



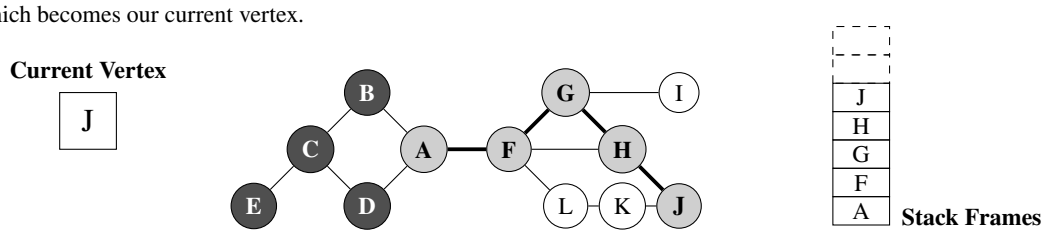
Next, we will make a recursive call on each of vertex F's unvisited neighbors. Vertex G is the first neighbor alphabetically, so we will make a recursive call on G, which becomes our current vertex.



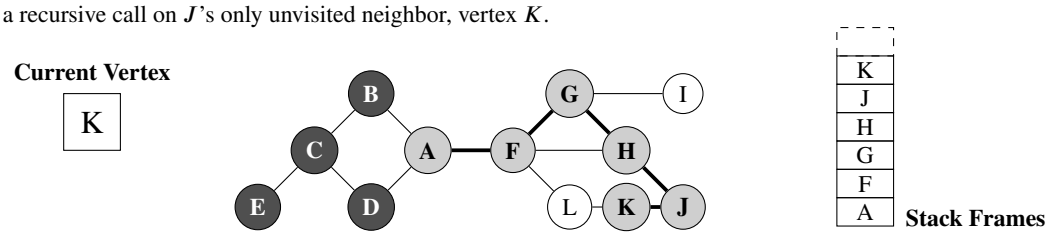
We then make recursive calls on all of G's unvisited neighbors. Vertex G has three neighbors, vertices F, H, and I. Vertex F has already been visited, so we will make a recursive call on vertex H.



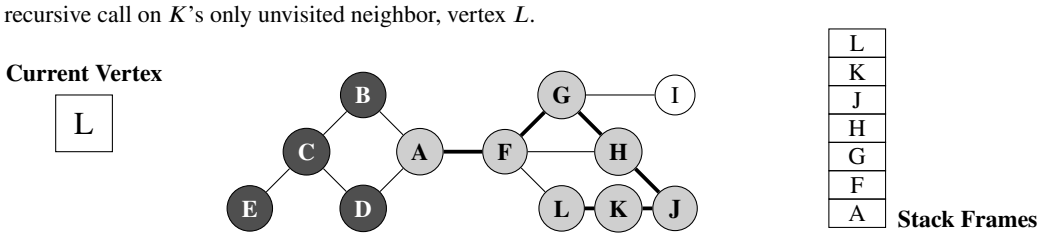
We then make a recursive call to all of H's unvisited neighbors. Vertex J is the only unvisited neighbor of H, so we will make a recursive call on vertex J, which becomes our current vertex.



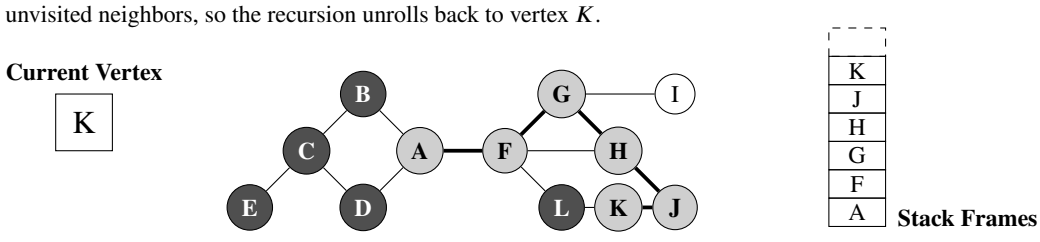
Next, we make a recursive call on J's only unvisited neighbor, vertex K.



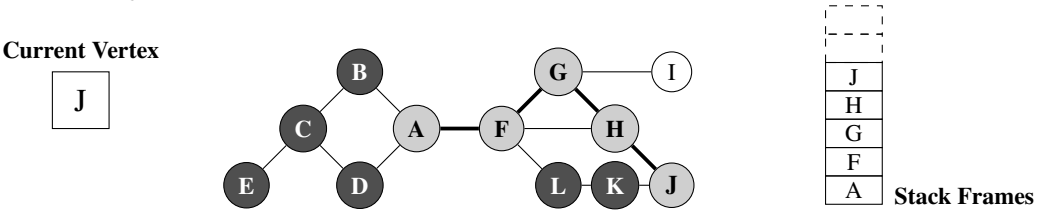
We then make a recursive call on K's only unvisited neighbor, vertex L.



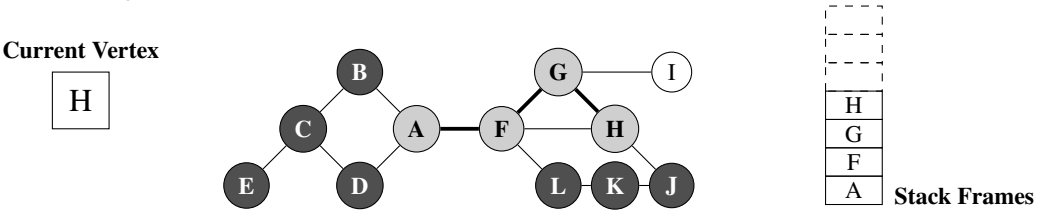
Vertex L has no unvisited neighbors, so the recursion unrolls back to vertex K.



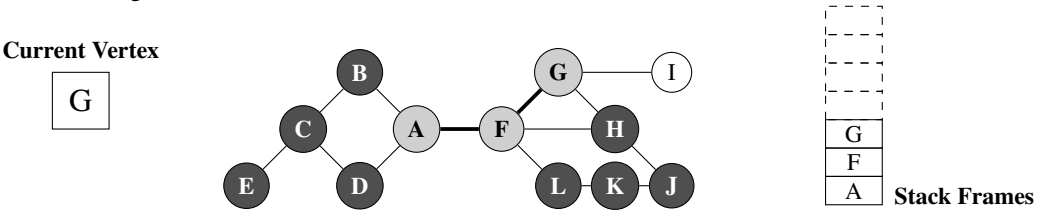
Vertex K has no unvisited neighbors, so the recursion unrolls back to vertex J.



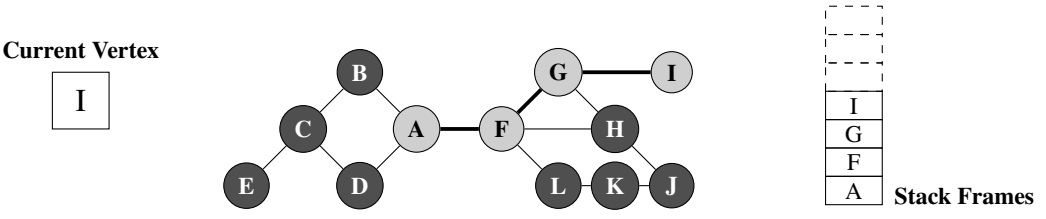
Vertex J has no unvisited neighbors, so the recursion unrolls back to vertex H.



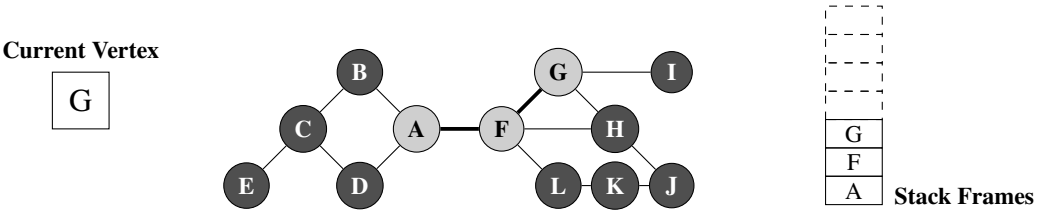
Vertex H has no unvisited neighbors, so the recursion unrolls back to vertex G.



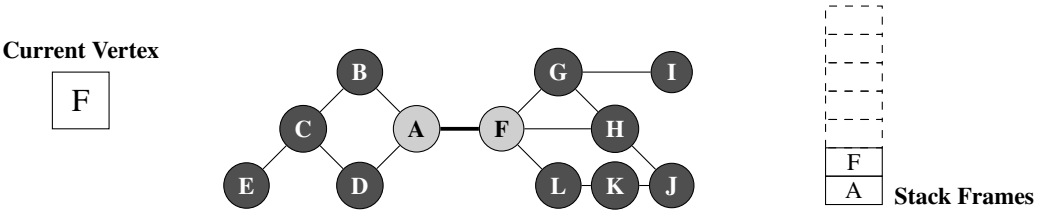
Now, we will make a recursive call on vertex I, which is vertex G's only remaining unvisited neighbor.



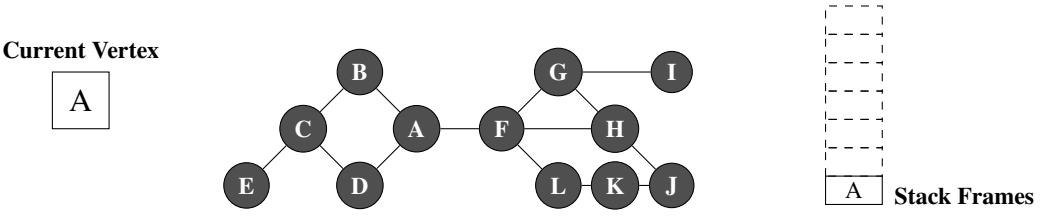
Vertex I has no unvisited neighbors, so no recursive call is made. The recursion unrolls back to vertex G.



All of vertex G's neighbors have now been visited, so the recursion unrolls back to vertex F.



Vertex F's neighbors have also all been visited, so the recursion unrolls back to vertex A.



All of vertex A's neighbors have now been visited. Since A was the source node, the recursive depth-first search starting at vertex A is now complete, and we have successfully processed all vertices in the graph that are reachable from vertex A. The order of branches we explored during the recursive search was different from the branches we explored during the iterative search: we first explored the branch $A \to B \to C \to D$, then we backtracked to vertex C and explored the branch $A \to B \to C \to E$, then we backtracked to vertex A and explored the branch $A \to F \to G \to H \to J \to K \to L$, then we backtracked to vertex G and explored the branch $A \to F \to G \to I$. This is because a recursive call on a node automatically sets it as our current vertex.

※ 19.3.3 *Depth-First Search Time Complexity*

What is the time complexity of a depth-first search? This depends on whether the underlying graph is represented as an adjacency list or an adjacency matrix. In an adjacency *list*, each vertex is only visited once, and we only have to traverse the *direct* neighbors of each vertex when we iterate through its vertex list. As a result, each edge in an adjacency list is visited at most once in a directed graph, and at most twice in an undirected graph (an undirected edge connecting two vertices A and B is checked twice during the depth-first search: once when checking the neighbors of A, and once when checking the neighbors of B). Because each vertex and each edge is visited a constant number of times during a depth-first search, the overall time complexity of performing DFS on an adjacency list is worst-case $\Theta(|V|+|E|)$. On the other hand, traversing through the edges of a vertex in an adjacency *matrix* requires us to iterate through *all* $|V|$ vertices in its row. Since this $\Theta(|V|)$ process is done $|V|$ times, once for each vertex, the overall time complexity of performing DFS on an adjacency matrix is worst-case $\Theta(|V|^2)$.

In summary, depth-first searches provide a method for traversing over all of the vertices in a graph reachable from a given source vertex. If you recall from the previous chapter, trees themselves are a special type of graph. Thus, the preorder, inorder, and postorder traversals are actually variants of a depth-first search on a tree, since they rely on a recursive stack to explore the nodes of a tree. The idea of depth-first search actually appears in many different types of problems, and it will show up again in the future when we discuss algorithm families such as backtracking and branch and bound.

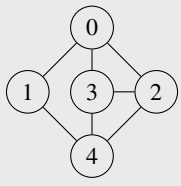
Graph Representation	Adjacency List	Adjacency Matrix
DFS Time Complexity	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$

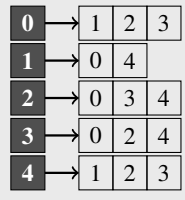
※ 19.3.4 *Solving Problems Using Depth-First Search*

Example 19.12 You are given an unweighted, undirected graph in the form of an adjacency list (where n vertices are labeled from 0 to $n-1$) and two vertices: a source and a destination. Write a function that returns all possible paths from the source vertex to the destination that visits a vertex at most once (i.e., no cycles), in any order. The function header is shown below:

```
std::vector<std::vector<int32_t>> find_all_paths(const std::vector<std::vector<int32_t>>& graph,
                                                 int32_t source, int32_t dest);
```

Example: Given the following graph:

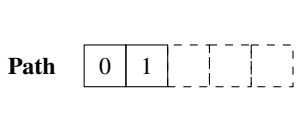


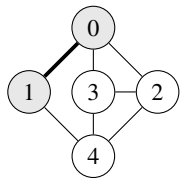


you should return the following paths, in any order:

```
[[0, 1, 4, 2, 3], [0, 1, 4, 3], [0, 2, 3], [0, 2, 4, 3], [0, 3]]
```

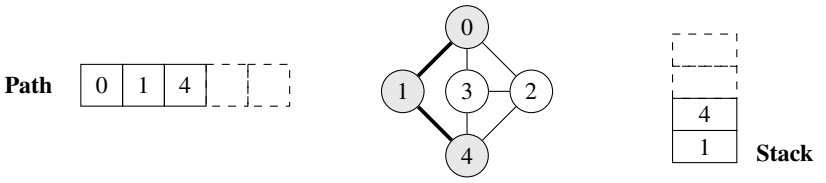
This is a graph search problem, so we can solve it by performing a depth-first search. We will start with the source node and begin our search, keeping track of our current path in a vector. If the destination is ever reached, we push the contents of this path into our solution vector. Since the graph is undirected and we want to avoid cycles, we must mark vertices in our current path as *visited* so that we do not end up in a cycle. An illustration of this process is shown below. We start at vertex 0, our source vertex, and make a recursive call on each of its adjacent vertices (1, 2, and 3). We will start by recursing on vertex 1, which pushes it into the call stack. We then mark vertex 1 as visited and add it to our path.



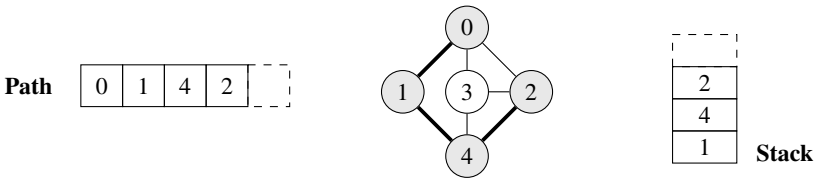


Solutions:

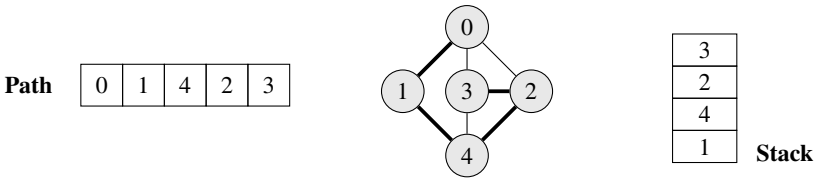
We then make a recursive call on each of 1's unvisited adjacent vertices. In this case, the only unvisited neighbor is 4, so we make a recursive call on 4, which pushes it into the call stack. We then mark vertex 4 as visited and add it to our path.



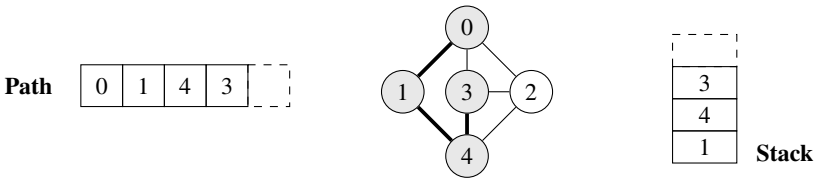
We then make a recursive call on each of 4's unvisited vertices. In this case, the unvisited neighbors are 2 and 3. We will first make a recursive call on 2, which pushes it into the call stack. We then mark vertex 2 as visited and add it to our path.



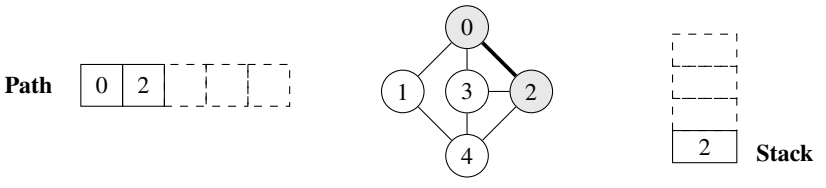
We then make a recursive call on each of 2's unvisited vertices. In this case, the only unvisited neighbor is 3, which is our destination vertex. As a result, we can add 3 to our path and append it to our list of solutions.



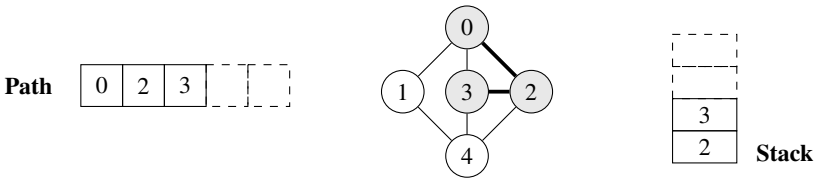
We have finished processing all of 2's unvisited neighbors, so the recursion unrolls back to vertex 4. We then consider 4's other unvisited neighbor, which happens to be 3, our destination vertex. As a result, we can also this path to our list of solutions.



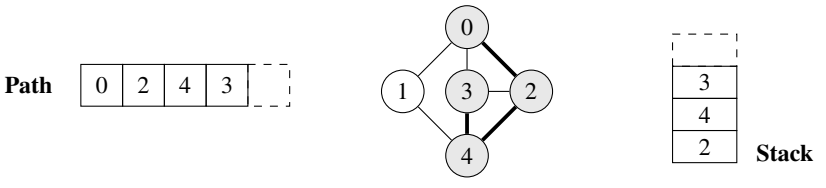
We have finished processing all of 4's unvisited neighbors, so the recursion unrolls back to vertex 1. Additionally, since 4 was 1's only unvisited neighbor, the recursion unrolls back to vertex 0. We now make a recursive call on 0's next unvisited neighbor of 2, which pushes it into the call stack. We then mark vertex 2 as visited and add it to our path.



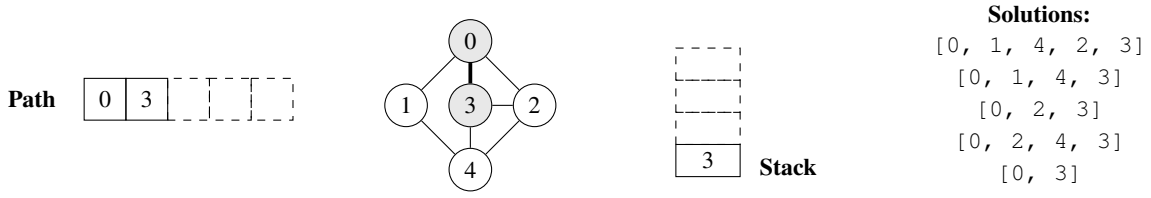
We then make a recursive call on each of 2's unvisited vertices, which are 3 and 4. Since 3 is our destination, we can add this path to our solution.



We then process 2's unvisited vertex of 4. Recursing into vertex 4 gives us the following path, which we add to our solution (note that we also recurse into vertex 1 after recursing into vertex 4, but that search does not lead to our destination vertex, so nothing gets added to the solution along that search path).



The recursion then unrolls back to vertex 0. The last neighbor of 0 we have yet to consider is vertex 3, which is our destination vertex. We therefore add this path to our solution. Since we have searched every neighbor of the source vertex, our algorithm is now complete, and our solutions vector holds all the valid paths from the source to the destination.



An implementation of this solution is shown below:

```
1   void find_all_paths_helper(const std::vector<std::vector<int32_t>>& graph,
2                              int32_t curr, int32_t dest, std::vector<bool>& visited,
3                              std::vector<int32_t>& current_path,
4                              std::vector<std::vector<int32_t>>& solution) {
5     visited[curr] = true;
6     current_path.push_back(curr);
7
8     if (curr == dest) {
9       // we have reached the destination vertex, so append the current path to the solution
10      solution.push_back(current_path);
11    } // if
12    else {
13      // recurse into all adjacent vertices that are not visited
14      const std::vector<int32_t>& neighbors = graph[curr];
15      for (int32_t neighbor : neighbors) {
16        if (!visited[neighbor]) {
17          find_all_paths_helper(graph, neighbor, dest, visited, current_path, solution);
18        } // if
19      } // for i
20    } // else
21
22    // once you are done processing current vertex, remove it from path and mark it as unvisited
23    current_path.pop_back();
24    visited[curr] = false;
25  } // find_all_paths_helper()
26
27  std::vector<std::vector<int32_t>> find_all_paths(const std::vector<std::vector<int32_t>>& graph,
28                                                   int32_t source, int32_t dest) {
29    std::vector<bool> visited(graph.size(), false);
30    std::vector<int32_t> current_path;
31    std::vector<std::vector<int32_t>> solution;
32    find_all_paths_helper(graph, source, dest, visited, current_path, solution);
33    return solution;
34  } // find_all_paths()
```

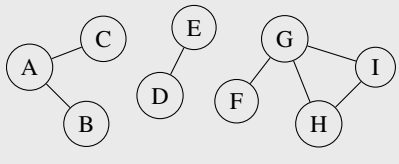
Example 19.13 You are given n devices. Some of these devices are connected, while some are not. If device X is directly connected to device Y, and device Y is directly connected to device Z, then device X is *indirectly* connected to device Z.

A **network** is a group of directly or indirectly connected devices. Two devices are a part of different networks if they are not connected in any manner, neither directly nor indirectly.

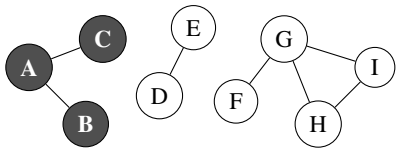
You are given an $n \times n$ adjacency matrix `devices`, where `devices[i][j] = true` if the i^{th} and j^{th} devices are directly connected, and `devices[i][j] = false` otherwise. You may assume that `devices[i][i] = false` without self loops. Implement the `num_devices()` function, which returns the total number of networks that exist among the devices.

```
int32_t num_devices(std::vector<std::vector<bool>>& devices);
```

Example: Given the following devices, the `num_devices()` function would return 3, since there exist three networks among these devices (*A*-*C*, *D*-*E*, and *F*-*I*).



One way to solve this problem is to use a depth-first search. A depth-first search can be used to identify all devices that are reachable from a given device, which allows you to determine the groups of devices that form each network. For example, if you start a depth-first search on device A, you will learn that devices B and C are part of the same network.



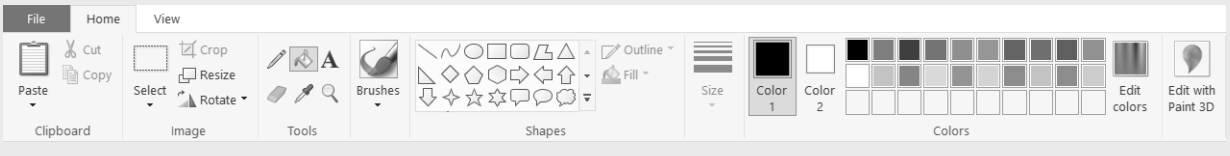
Therefore, to solve this problem, we can iterate over all devices and conduct a depth-first search on a device as long as it has not been visited. The number of searches we need is equal to the number of networks among the devices, since each search visits all devices within each network. In our example, a depth-first search on device A ends up visiting devices B and C, a depth-first search on device D ends up visiting device E, and a depth-first search on device F ends up visiting devices G, H, and I. A total of three searches are needed to visit all the devices, so the total number of networks must be three. The problem can be solved recursively or iteratively. A recursive depth-first search solution is shown below:

```
void dfs_helper(std::vector<std::vector<bool>>& adj_matrix,
                std::vector<bool>& visited, size_t i) {
  visited[i] = true;
  for (size_t j = 0; j < visited.size(); ++j) {
    if (adj_matrix[i][j] && !visited[j]) {
      dfs_helper(adj_matrix, visited, j);
    } // if
  } // for j
} // dfs_helper()

int32_t num_devices(std::vector<std::vector<bool>>& adj_matrix) {
  std::vector<bool> visited(adj_matrix.size(), false);
  int32_t network_count = 0;
  for (size_t i = 0; i < visited.size(); ++i) {
    if (!visited[i]) {
      ++network_count;
      dfs_helper(adj_matrix, visited, i);
    } // if
  } // for i
  return network_count;
} // num_devices()
```

Alternatively, an iterative depth-first search using a `std::stack<>` can also be used:

```
int32_t num_devices(std::vector<std::vector<bool>>& adj_matrix) {
  std::vector<bool> visited(adj_matrix.size(), false);
  std::stack<size_t> dfs;
  int32_t network_count = 0;
  for (size_t i = 0; i < visited.size(); ++i) {
    if (!visited[i]) {
      dfs.push(i);
      while (!dfs.empty()) {
        size_t curr = dfs.top();
        dfs.pop();
        for (size_t j = 0; j < adj_matrix[curr].size(); ++j) {
          if (adj_matrix[curr][j] && !visited[j]) {
            visited[j] = true;
            dfs.push(j);
          } // if
        } // for j
      } // while
      ++network_count;
    } // if
  } // for i
  return network_count;
} // num_devices()
```

Example 19.14 If you have worked with any image editing software before, you are probably familiar with the "fill" tool, which can be used to apply a color to a specific region of an image. In Microsoft Paint, for example, the fill tool is represented using a paint bucket, which is selected in the screenshot below. In this problem, we will implement a function that emulates the behavior of the fill tool on a given image.



You are given a vector of vectors `image` with *m* rows and *n* columns, where `image[i][j]` represents the color of a pixel in the image. You are also given three integers, `row`, `column`, and `color`. Write a function that completes a fill of the image when the color `color` is applied on the pixel `image[row][column]`. To perform a fill, all pixels that are 4-directionally adjacent to the starting pixel with the same color (as well as any pixels of the same color that are adjacent to those pixels) are replaced with the new color. You may assume that the provided row and column are not out of bounds. The function header is shown below:

```
void apply_fill(std::vector<std::vector<int32_t>>& image, int32_t row, int32_t column, int32_t color);
```

Example: Given the following image:

1	1	1	2	2
2	1	1	1	2
1	1	0	0	0
2	1	2	1	0
2	2	1	1	1

If a fill is applied with a color of 3 on the pixel at row 1, column 2 (which currently has a color of 1, shown by the bolded cell below), then all adjacent pixels with a color of 1 are changed to have a color of 3.

1	1	1	2	2
2	1	**1**	1	2
1	1	0	0	0
2	1	2	1	0
2	2	1	1	1

3	3	3	2	2
2	3	**3**	3	2
3	3	0	0	0
2	3	2	1	0
2	2	1	1	1

This problem can be solved using a graph searching algorithm. We begin at the starting pixel (whose color is to be changed) and search for all pixels that are reachable from this starting pixel that share the same color. This is done by changing the color of a pixel, and then checking its four neighboring pixels to see if they share the same original color. If any pixel does share the same color, we update its color and search its neighboring pixels as well using the same process. This continues until no pixel reachable from the starting pixel contains the original color that was changed. A recursive depth-first search solution is shown below:

```
1  void dfs_helper(std::vector<std::vector<int32_t>>& image, int32_t curr_row, int32_t curr_column,
2                  int32_t starting_color, int32_t new_color) {
3    // out of bounds check + check if color should be applied (if not, return)
4    if (curr_row < 0 || curr_row >= image.size() || curr_column < 0 ||
5        curr_column >= image[curr_row].size() || image[curr_row][curr_column] == new_color ||
6        image[curr_row][curr_column] != starting_color) {
7      return;
8    } // if
9    // if the code made it here, image[curr_row][curr_column] should be changed to a new color
10   image[curr_row][curr_column] = new_color;
11   // perform a DFS to search adjacent pixels and update their colors if necessary
12   dfs_helper(image, curr_row - 1, curr_column, starting_color, new_color);
13   dfs_helper(image, curr_row + 1, curr_column, starting_color, new_color);
14   dfs_helper(image, curr_row, curr_column - 1, starting_color, new_color);
15   dfs_helper(image, curr_row, curr_column + 1, starting_color, new_color);
16 } // dfs_helper()
17
18 void apply_fill(std::vector<std::vector<int32_t>>& image, int32_t row,
19                 int32_t column, int32_t color) {
20   int32_t starting_color = image[row][column];
21   dfs_helper(image, row, column, starting_color, color);
22 } // apply_fill()
```

This problem can also be solved iteratively. An iterative DFS solution is provided below:

```
void apply_fill(std::vector<std::vector<int32_t>>& image, int32_t row,
                int32_t column, int32_t color) {
  // if pixel already has the new color, no need to change anything
  int32_t starting_color = image[row][column];
  if (starting_color == color) {
    return;
  } // if

  int32_t num_rows = image.size();
  int32_t num_columns = image[0].size();
  std::stack<std::pair<int32_t, int32_t>> dfs;

  // this allows us to easily visit the positions adjacent to the current pixel
  std::vector<std::pair<int32_t, int32_t>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

  // apply new color to pixel (this also marks it as "visited" for our search)
  image[row][column] = color;

  // perform a DFS of adjacent pixels
  dfs.emplace(row, column);
  while (!dfs.empty()) {
    auto [curr_row, curr_column] = dfs.top();
    dfs.pop();
    for (auto [x_increment, y_increment] : dirs) {
      int32_t new_row = curr_row + x_increment;
      int32_t new_column = curr_column + y_increment;
      // out of bounds check
      if (new_row < 0 || new_column < 0 || new_row >= image.size() || new_column >= image[0].size()) {
        continue;
      } // if
      // if pixel has same color as starting color, update to new color and push into search stack
      if (image[new_row][new_column] == starting_color) {
        image[new_row][new_column] = color;
        dfs.emplace(new_row, new_column);
      } // if
    } // for [x_increment, y_increment]
  } // while
} // apply_fill()
```

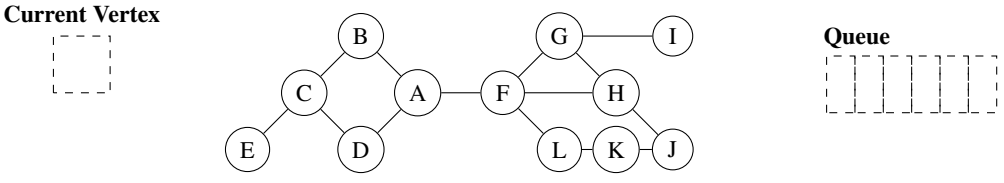
19.4 Breadth-First Search

※ 19.4.1 *Breadth-First Search*

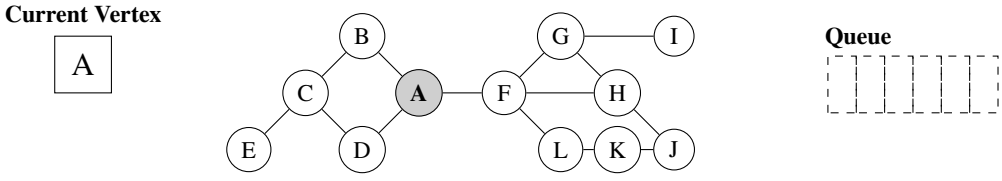
The **breadth-first search (BFS)** algorithm is another graph traversal algorithm that can be used to explore the vertices and edges of a graph that are reachable from a given source vertex. In a breadth-first search, the algorithm starts at a source vertex and explores vertices of the graph in a breadthward fashion, *in order of increasing edge distance from the source node*. The implementation of a breadth-first search relies on the `std::queue<>` data structure. An outline of the algorithm is as follows:

1. Mark the source vertex as visited, and then push its neighbors (i.e., adjacent vertices) into a queue.
2. Pop the vertex at the front of the queue and set it as the current vertex.
3. Push the unvisited neighbors of the current vertex into the queue and mark them as visited.
4. Repeat steps 2 and 3 until the queue is empty.

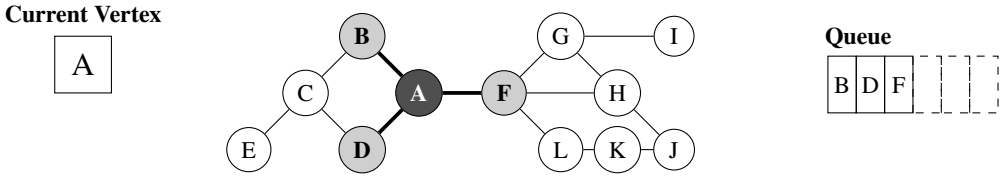
As an example, consider the following graph, which we will traverse using a breadth-first search starting at vertex *A*. Similar to the depth-first search example, discovered vertices will be given a light gray shading, while processed vertices will be given a dark gray shading. We start the breadth-first search process by creating a `std::queue<>` that will be used to keep track of the vertices that we still need to explore.



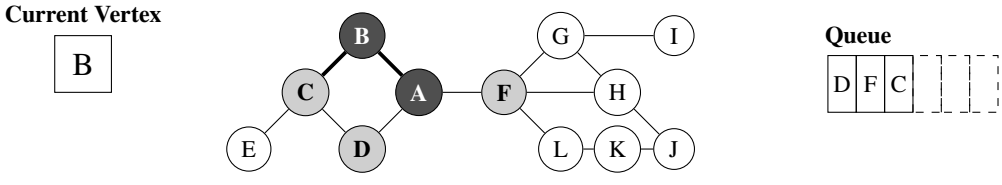
The first step is to mark the starting vertex as visited. In this case, our starting vertex is vertex A, so we will mark A as visited.



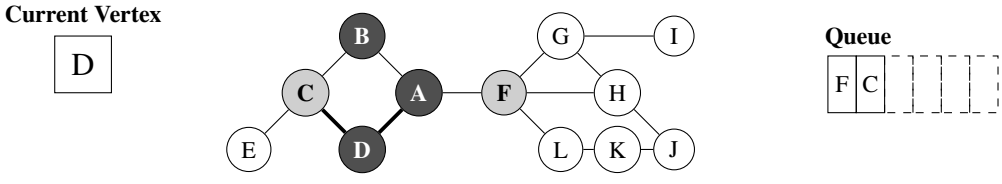
Next, we look at the vertices that vertex A is connected to, mark them as visited, and add them to the queue. The order we add these vertices does not matter, but for our example, we will add them in alphabetical order (i.e., B is inserted first, then D, then F).



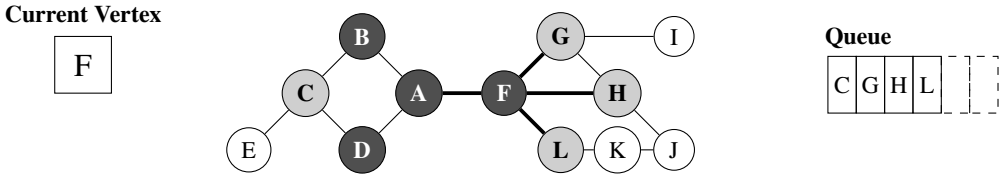
We then take out the vertex at the front of the queue, vertex B, and set it as our current vertex. Since vertex B is our current vertex, we push all of its unvisited neighbors into the queue. Vertex B is connected to vertices A and C, but only vertex C is marked as unvisited, so vertex C gets pushed into the queue and marked as visited.



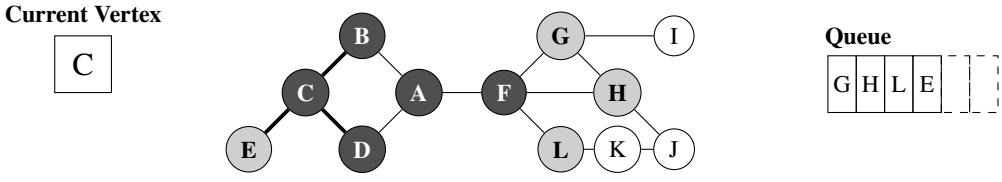
Vertex D is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex D's unvisited neighbors into the queue. Vertex D is connected to vertices A and C, but both A and C have been marked as visited. Thus, nothing is pushed into the queue at this step.



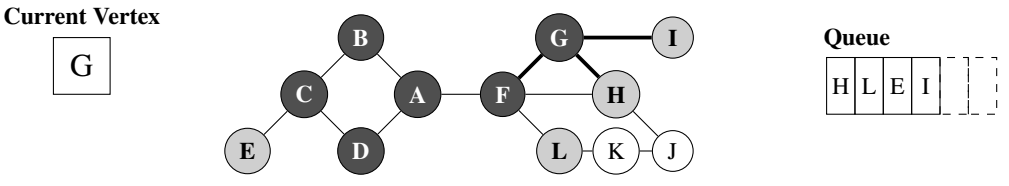
Vertex F is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex F's unvisited neighbors into the queue. Vertex F is connected to vertices A, G, H, and L, but only vertex A has been marked as visited. Thus, vertices G, H, and L are pushed into the queue and marked as visited.



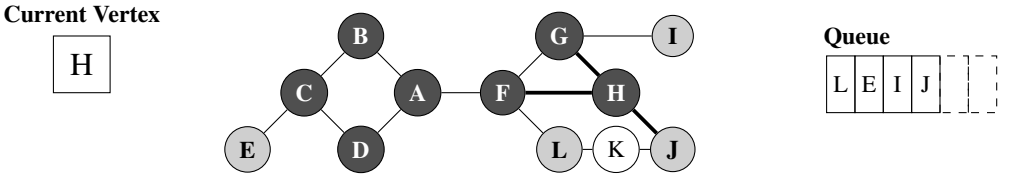
Vertex C is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex C's unvisited neighbors into the queue. Vertex C is connected to vertices B, D, and E, but only vertex E has not yet been marked as visited. Thus, vertex E is pushed into the queue and marked as visited.



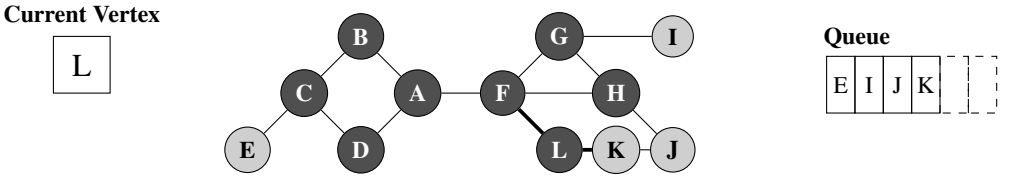
Vertex G is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex G's unvisited neighbors into the queue. Vertex G is connected to vertices F, H, and I, but only vertex I has not yet been marked as visited. Thus, vertex I is pushed into the queue and marked as visited.



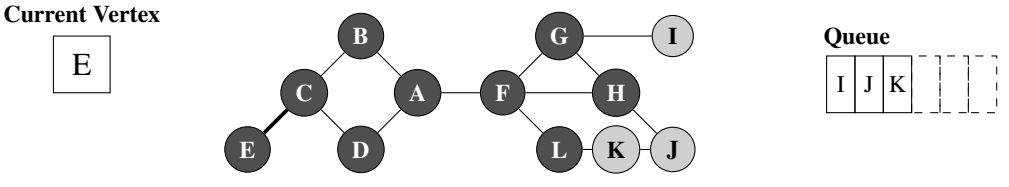
Vertex H is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex H's unvisited neighbors into the queue. Vertex H is connected to vertices F, G, and J, but only vertex J has not yet been marked as visited. Thus, vertex J is pushed into the queue and marked as visited.



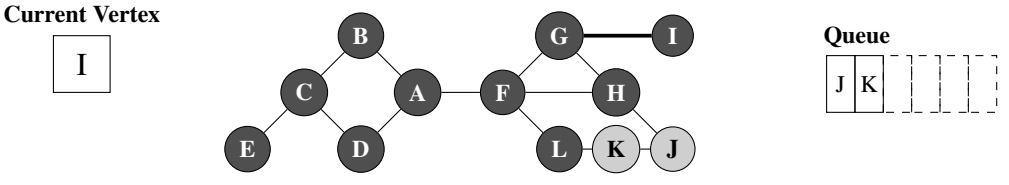
Vertex L is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex L's unvisited neighbors into the queue. Vertex L is connected to vertices F and K, but only vertex K has not yet been marked as visited. Thus, vertex K is pushed into the queue and marked as visited.



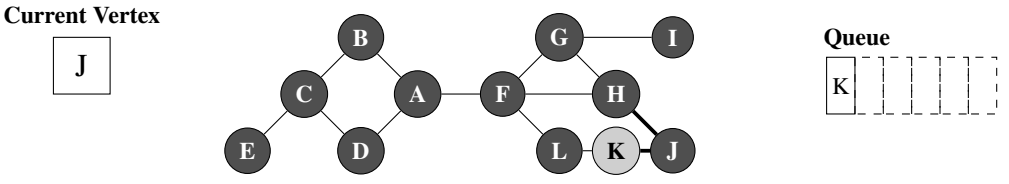
Vertex E is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex E's unvisited neighbors into the queue. Vertex E is connected to vertex C, which has already been marked as visited. Thus, nothing is pushed into the queue at this step.



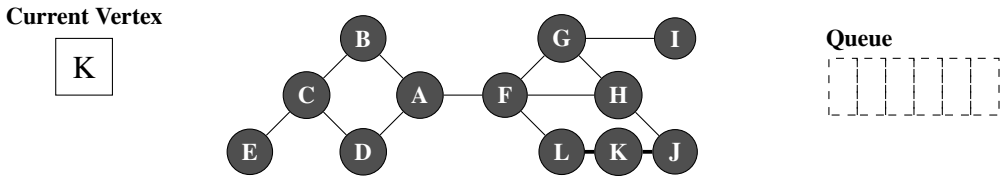
Vertex I is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex I's unvisited neighbors into the queue. Vertex I is connected to vertex G, which has already been marked as visited. Thus, nothing is pushed into the queue at this step.



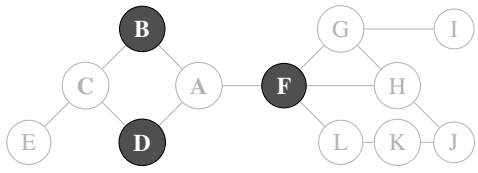
Vertex J is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex J's unvisited neighbors into the queue. Vertex J is connected to vertices H and K, but both have already been visited. Thus, nothing is pushed into the queue at this step.



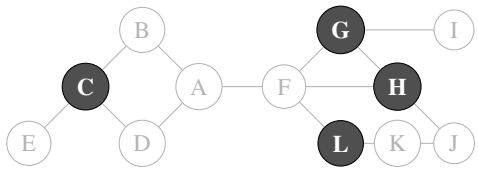
Vertex K is now at the front of the queue, so we pop it off and set it as our current vertex. We then push all of vertex K's unvisited neighbors into the queue. Vertex K is connected to vertices J and L, but both have already been visited. Thus, nothing is pushed into the queue at this step.



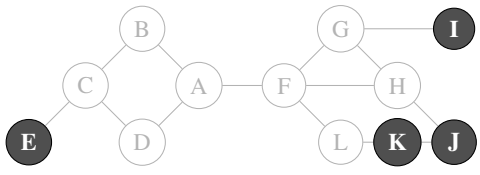
The queue is now empty, so our breadth-first search is complete, and we have successfully processed every vertex in the graph. Because a breadth-first search relies on a queue to process nodes in a graph, the algorithm processes vertices in the graph in FIFO order. As a result, a breadth-first search ends up visiting vertices in order of increasing edge distance from the source vertex: vertices that are closer to the source vertex are pushed into the queue *before* vertices that are farther away, and are thus taken out of the queue and processed earlier as well! In our example, the breadth-first search first processed vertices that were one edge away from the source vertex A: vertices B, D, and F (in this order, since we pushed them into the queue in alphabetical order).



Then, the breadth-first search processed vertices that were two edges away from the source vertex A:



Then, the breadth-first search processed vertices that were three edges away from the source vertex A:



Had the graph been larger, the breadth-first search would then have processed vertices four edges away, then five edges away, and so on. Because breadth-first searches exhibit this behavior, they can always be used to find the *shortest path between any two vertices in an unweighted graph, or a graph where all edge weights are the same.* As a result, breadth-first searches are often useful for solving shortest path graph problems, particularly for graphs that are unweighted.[3]

Similar to a depth-first search, a breadth-first search also visits all vertices that are accessible from any starting vertex, and thus can be used to determine if a simple path exists between any two vertices in a graph. If a vertex is never discovered during a breadth-first search before it completes (when the queue becomes empty), then that vertex is not accessible from the starting node. The pseudocode for determining if a path exists between two vertices using a breadth-first search is shown below:

```
1   Algorithm BFS(source, destination):
2     mark source as visited
3     push source into queue
4     while queue is not empty:
5       get/pop candidate from front of stack
6       for each neighbor of candidate:
7         if neighbor is unvisited:
8           mark neighbor as visited
9           push neighbor to top of queue
10          if neighbor is destination:
11            return true (success)
12    return false (failure)
```

[3] If a graph is weighted, a breadth-first search will no longer guarantee the shortest path between any two vertices. Instead, we will have to use a shortest path algorithm such as *Dijkstra's algorithm* instead of BFS. We will cover Dijkstra's in a later chapter.

※ 19.4.2 *Breadth-First Search Time Complexity*

Similar to a depth-first search, the time complexity of a breadth-first search is worst-case $\Theta(|V|+|E|)$ on an adjacency list, and $\Theta(|V|^2)$ on an adjacency matrix. In an adjacency list, each vertex and edge in the graph is visited a constant number of times. However, in an adjacency matrix, iterating over all the edges of a vertex would require you to iterate over all $|V|$ vertices in the graph to check where an edge exists, which results in a $\Theta(|V|^2)$ time complexity, as a $\Theta(|V|)$ is done $|V|$ times.

In summary, breadth-first searches provide a way to traverse over all of the vertices in a graph, and they can also be used to discover the shortest path between two vertices in an unweighted graph. If you think back to tree traversals, the level-order traversal is a breadth-first search on a tree, since it uses a queue to determine the order in which nodes are processed. This is why a level-order traversal is able to visit nodes of a tree level by level; by performing a breadth-first search, it will always visit nodes that are closer to the root before nodes that are farther away!

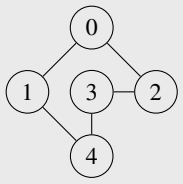
Graph Representation	Adjacency List	Adjacency Matrix
BFS Time Complexity	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$

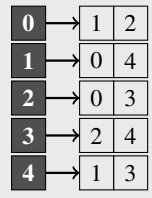
※ 19.4.3 *Solving Problems Using Breadth-First Search*

Example 19.15 You are given an unweighted, undirected graph in the form of an adjacency list (where n vertices are labeled from 0 to $n-1$) and two vertices: a source and a destination. Write a function that returns the *shortest* path from the source vertex. If there are multiple shortest paths, you can return any. If there is no path, return an empty vector. The function header is shown below:

```
std::vector<int32_t> find_shortest_path(const std::vector<std::vector<int32_t>>& graph,
                                        int32_t source, int32_t dest);
```

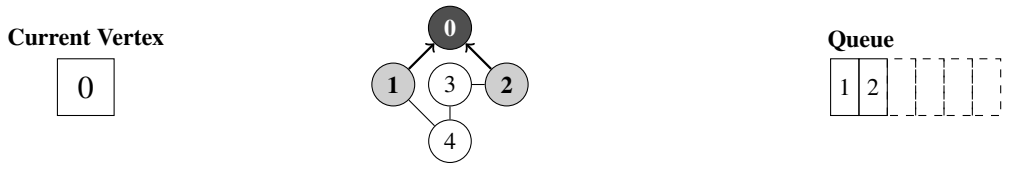
Example: Given the following graph, a source of 0, and a destination of 3, you should return `[0, 2, 3]`.



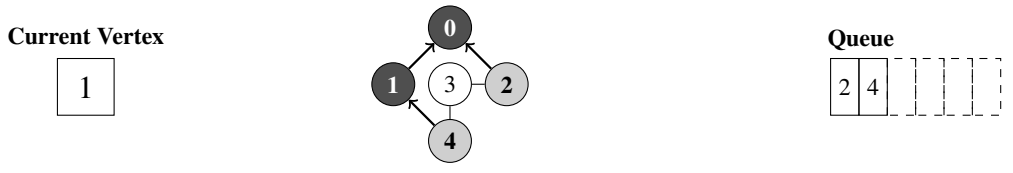


This is a graph search problem, but unlike the previous example, we want the shortest path rather than any path between two vertices. Since the graph is unweighted, the shortest path is the one that traverses the fewest nodes between the starting vertex and the destination. Thus, we would want to use a breadth-first search, as this method visits vertices in order of distance from the source node (via a FIFO queue). Once you encounter the destination vertex along your breadth-first search, you have found the shortest path!

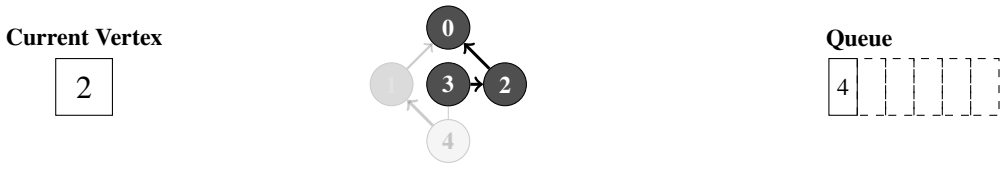
To fully implement a solution, you will need to *backtrace* and reproduce the path that brought you to the destination vertex along your search. For instance, if you wanted to find the shortest path from vertex 0 to vertex 3, you would need to identify the sequence of vertices that you followed to get to vertex 3. This can be done by tracking the *predecessor* of each vertex encountered, starting from the source. As an example, let us complete a breadth-first search from vertex 0 to vertex 3 using the graph above. The first vertices we push into the queue are 1 and 2, since they are the unvisited vertices that are directly reachable from 0. Since 1 and 2 are pushed into the queue while examining vertex 0, we will set vertex 0 as the *predecessor* of vertices 1 and 2 (represented using the directional edge arrows below).



Vertex 1 is at the front of the queue, so we take it out and set it as our current vertex. The only other unvisited vertex directly connected to vertex 1 is vertex 4, which we push into the queue. We will also set vertex 1 as the predecessor of 4.



Vertex 2 is at the front of the queue, so we take it out and set it as our current vertex. Notice that our destination, vertex 3, is directly reachable from vertex 2. Since a breadth-first search explores nodes in increasing distance from the source node, we have therefore found the shortest path from source to destination. After setting vertex 2 as the predecessor of 3, we can use the predecessors to reconstruct the path that was taken (by looking at 3's predecessor, and then the predecessor's predecessor, and so on until we reach the source vertex). In this case, we can see that the shortest path is $0 \rightarrow 2 \rightarrow 3$ by following the predecessors from 3 back to 0.



To avoid running into cycles, we need a way to keep track of whether a vertex has been visited or not. However, notice that our above solution does not have an explicit container that keeps track of whether a vertex has been visited or not. This is because, by keeping track of the predecessors of each vertex along our search, we can use this information to identify whether a vertex has been visited, as only visited vertices can have a predecessor! An implementation of this solution is shown below:

```
1   std::vector<int32_t> find_shortest_path(const std::vector<std::vector<int32_t>>& graph,
2                                           int32_t source, int32_t dest) {
3     // this map stores the predecessor of each vertex discovered along our BFS
4     // e.g., if backtrace[2] = 1, then vertex 1 directly precedes 2 along path
5     std::unordered_map<int32_t, int32_t> backtrace;
6     backtrace[source] = -1;  // source vertex has no predecessor
7
8     // perform BFS
9     std::queue<int32_t> bfs;
10    bfs.push(source);
11    bool path_found = false;
12    while (!path_found && !bfs.empty()) {
13      int32_t curr = bfs.front();
14      bfs.pop();
15      // iterate over vertices directly connected with current vertex
16      // if any are unvisited (i.e., has no predecessor), push into queue
17      for (int32_t neighbor : graph[curr]) {
18        if (backtrace.find(neighbor) == backtrace.end()) {
19          backtrace[neighbor] = curr;
20          // if neighbor is solution, break out of loop
21          if (neighbor == dest) {
22            path_found = true;
23            break;
24          } // if
25          bfs.push(neighbor);
26        } // if
27      } // for neighbor
28    } // while
29
30    // backtrace from dest back to source
31    // this is done by following the predecessors from dest to source
32    std::vector<int32_t> path;
33    int32_t prev_vertex = dest;
34    if (path_found) {
35      path.push_back(prev_vertex);
36      while (backtrace[prev_vertex] != -1) {
37        path.push_back(backtrace[prev_vertex]);
38        prev_vertex = backtrace[prev_vertex];
39      } // while
40      std::reverse(path.begin(), path.end());
41    } // if
42
43    return path;
44  } // find_shortest_path()
```

Example 19.16 You have a job interview tomorrow, but your company just notified you today! As a result, you have to schedule a flight immediately. Given a function that takes in a starting airport (`origin`), a destination airport (`dest`), and a vector of available flights (`flights`, where `flights[i][0]` represents the origin of flight `i`, `flights[i][1]` represents the destination of flight `i`, and `flights[i][2]` represents the flight number), write a function that prints an itinerary for the sequence of flights that *minimizes* total layovers. Each line of the output should be of the form:

```
[<NUMBER>] <ORIGIN> -> <DEST>
```

where `<NUMBER>` represents the flight number, `<ORIGIN>` represents the origin of that flight, and `<DEST>` represents the destination of that flight. For example, if you have to take flight 281 to fly from airport `DTW` to airport `ORD`, then the output for that flight would be:

```
[281] DTW -> ORD
```

If there are multiple sequences of flights that minimize layovers, you should prioritize flights with *higher flight numbers* at each intermediate airport along your trip (e.g., if both flight 280 and flight 281 can be taken to yield a path that minimizes layovers, choose flight 281). If there is no solution (i.e., `dest` is not reachable from `origin`), print out all reachable airports from `origin` in the following format (the reachable airports can be printed out in any order):

```
No solution, <N> airports reachable
<REACHABLE AIRPORT 1>
<REACHABLE AIRPORT 2>
<REACHABLE AIRPORT 3>
<REACHABLE AIRPORT 4>
<REACHABLE AIRPORT 5>
...
<REACHABLE AIRPORT N>
```

The function header is as follows:

```
void print_itinerary(std::vector<std::vector<std::string>>& flights,
                     const std::string& origin, const std::string& dest);
```

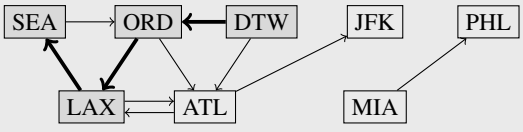
Example 1: Given `origin = "DTW"` and `dest = "SEA"`, and the following connections in the `flights` vector:

```
flights = {
  {"ATL", "JFK", 123},
  {"ORD", "LAX", 734},
  {"MIA", "PHL", 2152},
  {"SEA", "ORD", 203},
  {"DTW", "ORD", 281},
  {"DTW", "ATL", 230},
  {"ORD", "ATL", 12},
  {"ATL", "LAX", 280},
  {"LAX", "SEA", 3943},
  {"LAX", "ATL", 10}
};
```

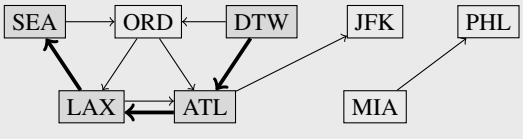
the output of the function should be

```
[281] DTW -> ORD
[734] ORD -> LAX
[3943] LAX -> SEA
```

This is because the path `DTW->ORD->LAX->SEA` results in the fewest layovers to get from `DTW` to `SEA`.



Note that `DTW->ATL->LAX->SEA` is also a valid path that minimizes layovers, but `DTW->ORD` was selected instead of `DTW->ATL` because its flight number was larger (281 > 280).



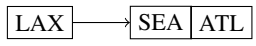
Example 2: Given the same graph as above, but with `dest = PHL`, the output of the function would be

```
No solution, 6 airports reachable
SEA
ORD
DTW
ATL
LAX
JFK
```

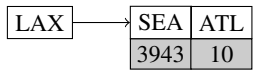
This is because there is no valid path from `DTW` to `PHL` using the flights given. The order of airports printed out is arbitrary.

This problem has several components, so let's break it down step by step. The first thing to notice is that this is a graph search problem. Thus, we should first convert the `flights` vector into either an adjacency list or adjacency matrix to facilitate the searching process. For this problem, we will be using an adjacency list (since they are a bit easier to work with for this problem — if you are ever given an exam problem where you have to pick between the two, look at the time and space complexity constraints of the problem).

In an adjacency list, each vertex of the graph should be mapped to its outgoing connections. For example, the `LAX` vertex should be mapped to the airports `SEA` and `ATL`, since these are the airports that are reachable from `LAX`.



However, in this problem, we also have to keep track of flight numbers, since the number of a flight may be used to determine which path to take in the presence of a tie. Thus, we will also store the flight number of each edge in the adjacency list.

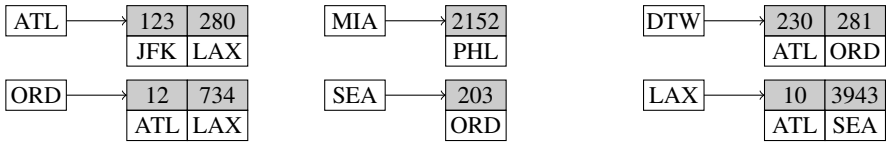


One way to represent this adjacency list is by using a hash table that maps each airport to a vector of pairs, where each pair stores an outgoing airport with its flight number. The process of initializing such an adjacency list is shown in the code below:

```
1  std::unordered_map<std::string, std::vector<std::pair<std::string, int32_t>>> adj_list;
2  for (auto& flight : flights) {
3    adj_list[flight[0]].emplace_back(flight[1], stoi(flight[2]));
4  } // for flight
```

However, in this problem, we need to be able to access the flight numbers in *sorted* order. Because of this, a vector of pairs is not the best way to store information about each flight, since data values in a vector are not sorted. Instead, we should store the flight information in a sorted container such as a `std::map<>`. This change is shown below (we want the flight number to be the key since maps are sorted by key):

```
1  std::unordered_map<std::string, std::map<int32_t, std::string>> adj_list;
2  for (auto& flight : flights) {
3    adj_list[flight[0]].emplace(std::stoi(flight[2]), flight[1]);
4  } // for flight
```



Now that we have our graph representation, we want to find the path between `origin` and `dest` that goes through the fewest vertices. This is a shortest-path problem, and since the graph is unweighted, we can solve this using a breadth-first search. However, there are two details we have to pay attention to when implementing this problem: we want to print out the *entire* sequence of airports we visit along the shortest path from our origin to our destination, and we want to prioritize flights with higher flight numbers in the case of a tie. How can address these two issues within our breadth-first search?

To print out every intermediary airport in our shortest path, we will need to keep track of the "predecessor" of each vertex whenever we discover it. For example, if we are iterating through the connections of `LAX` and push `SEA` into the queue, we have to remember that we reached `SEA` from `LAX`. This will allow us to "backtrack" after the completion of the algorithm to discover the path we took to reach the destination. Since this container should map each airport to the airport we came from (and its flight number), we can use an unordered map to represent it.

```
// maps airport to previous airport and flight number
std::unordered_map<std::string, std::pair<std::string, int32_t>> backtrace;
```

Next, we want to ensure that flights with higher flight numbers are chosen over flights with lower flight numbers in the case of a tie. This may seem complicated to address at first, but it's actually quite simple once you notice that breadth-first searches process vertices using a queue, which exhibits first-in, first-out behavior. As a result, to ensure that higher flight numbers are processed before lower ones, we just need to push flights with higher flight numbers into the queue *before* those with lower flight numbers! For instance, consider the previous example, where `DTW->ORD->LAX->SEA` and `DTW->ATL->LAX->SEA` were both valid paths that minimized layovers. If we wanted to select the path with `DTW->ORD` instead of the one with `DTW->ATL`, we just need to push `ORD` into the queue *before* `ATL` when processing the connections of `DTW`. That way, the edge `DTW->ORD` will be popped off the queue and processed before the edge `DTW->ATL` is. This behavior can be accomplished by simply iterating over the vertex list of each airport in descending flight number order.

This is shown in the code below:

```
std::queue<std::string> bfs;
bfs.push(origin);
backtrace[origin] = {"", -1};  // also serves as "visited" container
while (!bfs.empty()) {
  std::string depart = bfs.front();
  bfs.pop();
  auto conn_it = adj_list.find(depart);
  if (conn_it != adj_list.end()) {
    std::map<int32_t, std::string>& curr = conn_it->second;
    // add the flights into queue in DESCENDING flight number order (this uses reverse iterators)
    for (auto it = curr.rbegin(); it != curr.rend(); ++it) {
      const std::string& arrive = it->second;
      if (backtrace.find(arrive) == backtrace.end()) {
        backtrace[arrive] = {depart, it->first};
        bfs.push(arrive);
        ...
      } // if
    } // for it
  } // if
} // while
```

Lastly, we want to print out the shortest path we encountered from `origin` to `dest`. This can be done by starting at the destination airport and iteratively computing the previous airport (using the stored predecessors) until the starting airport is reached.

```
std::vector<std::pair<std::string, int32_t>> path = {{dest, 0}};
std::string prev_flight = dest;
while (prev_flight != origin) {
  path.push_back(backtrace[prev_flight]);
  prev_flight = path.back().first;
} // while
for (size_t i = path.size() - 1; i > 0; --i) {
  std::cout << "[" << path[i].second << "] " << path[i].first << " -> " << path[i - 1].first << '\n';
} // for i
```

In the no solution case, we want to print out all reachable airports. This can be done by printing out all the airports currently in the `backtrace` map, since each airport is inserted into this map as soon as it is discovered to be reachable.

```
std::cout << "No solution, " << backtrace.size() << " airports reachable\n";
for (auto it = backtrace.begin(); it != backtrace.end(); ++it) {
  std::cout << it->first << '\n';
} // for
```

Putting this all together, we have the following completed function:

```
void print_itinerary(std::vector<std::vector<std::string>>& flights,
                     const std::string& origin, const std::string& dest) {
  std::unordered_map<std::string, std::map<int32_t, std::string>> adj_list;
  std::unordered_map<std::string, std::pair<std::string, int32_t>> backtrace;
  for (auto& flight : flights) {
    adj_list[flight[0]].insert({std::stoi(flight[2]), flight[1]});
  } // for

  std::queue<std::string> bfs;
  bfs.push(origin);
  backtrace[origin] = {"", -1};
  bool finished = false;
  while (!finished && !bfs.empty()) {
    std::string depart = bfs.front();
    bfs.pop();
    auto conn_it = adj_list.find(depart);
    if (conn_it != adj_list.end()) {
      std::map<int32_t, std::string>& curr = conn_it->second;
      for (auto it = curr.rbegin(); it != curr.rend(); ++it) {
        const std::string& arrive = it->second;
        if (backtrace.find(arrive) == backtrace.end()) {
          backtrace[arrive] = {depart, it->first};
          if (arrive == dest) {
            finished = true;
            break;
          } // if
          bfs.push(arrive);
        } // if
      } // for it
    } // if
  } // while

  /* ... continued on next page ... */
```

```
    if (!finished) {  // no solution
      std::cout << "No solution, " << backtrace.size() << " airports reachable\n";
      for (auto it = backtrace.begin(); it != backtrace.end(); ++it) {
        std::cout << it->first << '\n';
      } // for it
    } // if
    else {  // backtrace from dest back to origin
      std::vector<std::pair<std::string, int32_t>> path = {{dest, 0}};
      std::string prev_flight = dest;
      while (prev_flight != origin) {
        path.push_back(backtrace[prev_flight]);
        prev_flight = path.back().first;
      } // while
      for (size_t i = path.size() - 1; i > 0; --i) {
        std::cout << "[" << path[i].second << "] " << path[i].first << " -> "
                  << path[i - 1].first << '\n';
      } // for i
    } // else
  } // print_itinerary()
```

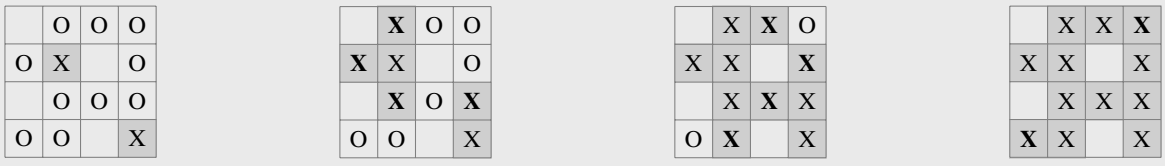
Example 19.17 You are given an $m \times n$ grid of PotatoBots, where each cell can have one of three enum values:

```
enum class Status {
  Empty,
  Functional,
  Corrupted
};
```

There is a deadly virus currently spreading among these PotatoBots, and any PotatoBot who catches this virus will become corrupted (and subsequently give terrible answers to Piazza questions). Every minute, any functional PotatoBot that is four-directionally adjacent (north, east, south, west) to a corrupted PotatoBot will also become corrupted. Implement the `time_to_corrupt()` function, which takes in the $m \times n$ grid and returns the total number of minutes that must elapse before all PotatoBots are corrupted (or `-1` if any PotatoBots can never be corrupted). The function header is as follows:

```
int32_t time_to_corrupt(std::vector<std::vector<Status>>& grid);
```

Example: Given the grid on the left (where positions marked with an "O" indicate a functional PotatoBot, and positions marked with an "X" indicate a corrupted PotatoBot), you would return `3`, since 3 minutes will elapse before all the PotatoBots are corrupted:



This problem can be solved using a breadth-first search starting with the corrupted PotatoBots, since a BFS visits elements outwards from a given starting vertex. This allows us to iteratively identify which PotatoBots are infected at each point in time. To start, we will first iterate over the entire grid and do two things:

- Count the number of functional PotatoBots (using a variable counter): this allows us to detect if any PotatoBots are still unaffected at the end of the algorithm.
- Push the positions of all of the corrupted PotatoBots into a BFS queue.

Then, as long as there remain corrupted PotatoBots in the queue, we take each corrupted PotatoBot out of the queue, examine its adjacent cells for any functional PotatoBots, corrupt them, and then push these newly corrupted PotatoBots into the queue. However, we also have to keep track of the time at which each PotatoBot is corrupted, so that we can return the total time elapsed at the end of the function. This can be handled by processing the PotatoBots on a time-by-time basis, where we first process PotatoBots that are corrupted at time 0, then PotatoBots that are corrupted at time 1, and so on. This is done by using an additional inner loop within the outer `while` loop that runs based on the size of the queue (similar to the one used in example 18.8, when we wanted to find the maximum level sum of a tree). This structure makes it easy for us to correctly keep track of total time using a simple counter.

A solution to the problem is shown below:

```
int32_t time_to_corrupt(std::vector<std::vector<Status>>& grid) {
  std::queue<std::pair<int32_t, int32_t>> bfs;
  std::vector<std::pair<int32_t, int32_t>> dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
  int32_t functional_bots = 0;
  int32_t time_elapsed = -1;

  // iterate over grid and push all initially corrupted PotatoBots into BFS queue
  for (size_t i = 0; i < grid.size(); ++i) {
    for (size_t j = 0; j < grid[i].size(); ++j) {
      if (grid[i][j] == Status::Corrupted) {
        bfs.push({i, j});
      } // if
      else if (grid[i][j] == Status::Functional) {
        ++functional_bots;
      } // else if
    } // for j
  } // for i

  // perform a BFS, using the queue size to distinguish between minutes
  while (!bfs.empty()) {
    ++time_elapsed;
    size_t sz = bfs.size();
    for (size_t k = 0; k < sz; ++k) {
      auto [curr_row, curr_col] = bfs.front();
      bfs.pop();
      for (auto [x_increment, y_increment] : dirs) {
        int32_t new_row = curr_row + x_increment;
        int32_t new_col = curr_col + y_increment;
        if (new_row < 0 || new_col < 0 || new_row >= grid.size() ||
            new_col >= grid[0].size() || grid[new_row][new_col] != Status::Functional) {
          continue;
        } // if
        grid[new_row][new_col] = Status::Corrupted;
        bfs.push({new_row, new_col});
        --functional_bots;
      } // for [x_increment, y_increment]
    } // for k
  } // while

  return functional_bots == 0 ? std::max(0, time_elapsed) : -1;
} // time_to_corrupt()
```

Example 19.18 Consider the following provided helper function, which returns a set of all prime numbers under one million, where each number is returned in the form of a string:

```
std::vector<std::string> get_primes() {
  constexpr uint64_t million = 1000000;
  std::vector<bool> is_prime(million, true);
  is_prime[0] = is_prime[1] = false;
  for (uint64_t n = 2; n * n < million; ++n) {
    if (is_prime[n]) {
      for (uint64_t multiple = n * n; multiple < million; multiple += n) {
        is_prime[multiple] = false;
      } // for multiple
    } // if
  } // for n

  std::vector<std::string> prime_strs;
  for (size_t i = 0; i < is_prime.size(); ++i) {
    if (is_prime[i]) {
      prime_strs.push_back(std::to_string(i));
    } // if
  } // for i

  return prime_strs;
} // get_primes()
```

Given two prime numbers, implement a function that returns the length of the shortest path between these numbers, provided that one of the following rules must be applied when changing between two prime numbers:

- A single digit is altered (i.e., changing from the prime 281 to the prime 2**7**1).
- A single digit is added to the number (i.e., changing from the prime 281 to the prime **2**281).
- A single digit is removed from the number (i.e., changing from the prime 281**9** to the prime 281).

The prime number changed to must be less than one million (i.e., it must be in the set returned by the `get_primes()` function), and leading zeros are not valid (e.g., `"0281"` is not a valid string that can be encountered along the path). If there is no solution, return `-1`. You may assume that the input values are both valid prime numbers. The function header is shown below:

```
int32_t shortest_path_between_primes(int32_t source, int32_t dest);
```

Example 1: If you are asked to convert the prime number 281 to the prime number 280183, you would return 5, since the shortest path between these two prime numbers is 5 if restricted to the rules above:

$$281 \rightarrow 283 \rightarrow 2083 \rightarrow 20183 \rightarrow 280183$$

Example 2: If you are asked to convert the prime number 281 to the prime number 619, you would return 5, since the shortest path between these two prime numbers is 5 if restricted to the rules above:

$$281 \rightarrow 181 \rightarrow 11 \rightarrow 19 \rightarrow 619$$

Example 3: If you are asked to convert the prime number 281 to the prime number 340723, you would return 9, since the shortest path between these two prime numbers is 9 if restricted to the rules above:

$$281 \rightarrow 241 \rightarrow 641 \rightarrow 643 \rightarrow 6473 \rightarrow 64793 \rightarrow 640793 \rightarrow 340793 \rightarrow 340723$$

Example 4: If you are asked to convert the prime number 999979 to the prime number 2, you would return 8, since the shortest path between these two prime numbers is 8 if restricted to the rules above:

$$999979 \rightarrow 399979 \rightarrow 39979 \rightarrow 39929 \rightarrow 3929 \rightarrow 929 \rightarrow 29 \rightarrow 2$$

To solve this problem, we can think of the problem's setup as a graph, where vertices are represented by the prime numbers that we can visit, and edges are represented by whether it is possible to change from one number to another. For example, the prime numbers reachable from 281 using the given rules are 181, 211, 241, 251, 271, 283, 881, 2081, 2281, 2381, 2801, 2819, 2851, 2861, 5281, and 9281. Thus, 281 would be connected to these other numbers in our graph. We would then be able to perform a breadth-first search to find the shortest way to change from one prime number to another.

To build our graph, we will need to know if it is possible to change from one prime number to another. This can be done by writing a separate helper function that takes in two prime numbers and determines if it is valid to change from one to another. An implementation of this function is shown below:

```
bool is_valid_change(const std::string& prime1, const std::string& prime2) {
  // if the two numbers are off by more than one, it is not a valid change
  int32_t length_diff = std::fabs(prime1.length() - prime2.length());
  if (length_diff > 1) {
    return false;
  } // if

  // if the two numbers have the same length, a digit must be altered
  int32_t digit_diff = 0;
  if (length_diff == 0) {
    for (size_t i = 0; i < prime1.length(); ++i) {
      if (prime1[i] != prime2[i]) {
        // if more than one digit is different, it is not a valid change
        if (++digit_diff > 1) {
          return false;
        } // if
      } // if
    } // for i
    return digit_diff == 1;
  } // if

  // if the two numbers differ in length by 1, a digit must be inserted or removed
  const std::string& longer_prime = prime1.length() > prime2.length() ? prime1 : prime2;
  const std::string& shorter_prime = prime1.length() > prime2.length() ? prime2 : prime1;
  for (size_t i = 0, j = 0; i < longer_prime.length(); ++i) {
    // only increment i and j together if the digits match, else only increment for longer prime
    if (longer_prime[i] == shorter_prime[j]) {
      ++j;
    } // if
    // there is a mismatch, so increment the diff counter and return false if it goes above 1
    else if (++digit_diff > 1) {
      return false;
    } // else if
  } // for i, j

  return true;
} // is_valid_change()
```

We could use this method to construct an adjacency list, as shown:

```
std::unordered_map<std::string, std::vector<std::string>> adj_list;
std::vector<std::string> all_primes = get_primes();
for (size_t i = 0; i < all_primes.size(); ++i) {
  for (size_t j = i + 1; j < all_primes.size(); ++j) {
    const std::string& prime1 = all_primes[i];
    const std::string& prime2 = all_primes[j];
    if (is_valid_change(prime1, prime2)) {
      // using the rules, if it is possible to change from prime1 to prime2,
      // it must also be possible to change from prime2 back to prime1
      // this is also why it is okay to start j at i + 1, since the graph is undirected
      adj_list[prime1].push_back(prime2);
      adj_list[prime2].push_back(prime1);
    } // if
  } // for j
} // for i
```

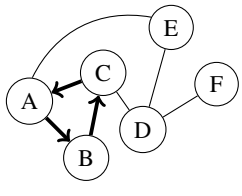
However, this is more work than necessary, since it is unlikely for the search to consider all combinations of primes. Instead, it is more efficient to only consider whether a change is valid for the prime numbers you encounter along your search, rather than for every prime number under one million (which can get quite computationally intensive).

Since we only need to return the length of the shortest path between two primes, we can store the distance of each number from the source number, rather than the value of its predecessor. For example, if we change 281 to 881 along the search, we only need to know that 881 is one change away from the source number of 281, and not that its predecessor is 281. An implementation of the solution is shown below:

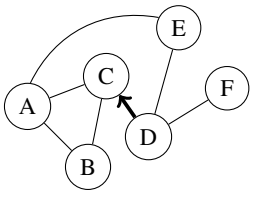
```
int32_t shortest_path_between_primes(int32_t source, int32_t dest) {
  std::vector<std::string> all_primes = get_primes();

  std::unordered_map<std::string, int32_t> dist_from_source;
  std::string source_str = std::to_string(source);
  std::string dest_str = std::to_string(dest);
  dist_from_source[source_str] = 1;

  std::queue<std::string> bfs;
  bfs.push(source_str);
  while (!bfs.empty()) {
    std::string curr = bfs.front();
    bfs.pop();
    for (const std::string& prime : all_primes) {
      if (dist_from_source.find(prime) == dist_from_source.end() && is_valid_change(curr, prime)) {
        dist_from_source[prime] = dist_from_source[curr] + 1;
        if (prime == dest_str) {
          return dist_from_source[prime];
        } // if
        bfs.push(prime);
      } // if
    } // for neighbor
  } // while

  return -1;  // no solution
} // shortest_path_between_primes()
```

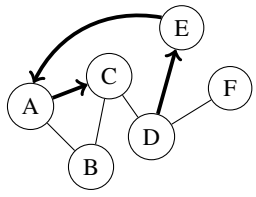
19.5 Cycle Detection

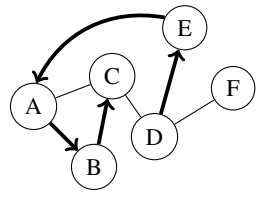
In this section, we will discuss approaches that can be used to detect cycles in a graph. To recap, a **cycle** is a path with distinct edges in a graph where the starting and ending vertices are the same:



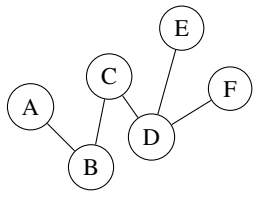
By definition, if there exists a cycle in a graph, then there must be a way to access a node in the graph via more than one unique path. For example, consider vertex C in the graph above. For any arbitrary vertex in the graph other than C, there are multiple paths we can follow to reach vertex C. For example, if we start from vertex D, we can reach vertex C along the path $D \to C$, $D \to E \to A \to C$, or $D \to E \to A \to B \to C$.







However, if a graph were acyclic, then there exists only one way to reach each vertex in the graph from any starting vertex.



We can take advantage of this fact when implementing a cycle detection algorithm. Recall that DFS and BFS mark each node as visited as soon as it is discovered. However, because there exists only one way to reach each vertex in an acyclic graph, the search algorithm can only discover each vertex *once* if a graph has no cycles! Thus, if the algorithm rediscovers a vertex after it has already been marked as visited, then there must exist more than one way to access that node from the starting vertex, and a cycle must exist in the graph.

Example 19.19 You are given an undirected, connected graph in the form of an adjacency list. Write a function that returns `true` if the graph has a cycle, and `false` if it does not. Implement your cycle detection algorithm using a depth-first search.

```
bool is_graph_cyclic(const std::vector<std::vector<int32_t>>& adj_list);
```

Since the graph is undirected, if an edge exists between vertices `u` and `v`, then `adj_list[u]` will contain `v`, and `adj_list[v]` will contain `u`. You may assume that `u` will not appear in `adj_list[u]`.

To solve this problem using a depth-first search, we will select an arbitrary node (in this case, we can just select `adj_list[0]`) and begin a depth-first search, marking each vertex as visited as soon as it is discovered. Since the graph is connected, we know that the depth-first search will eventually visit every vertex in the graph. If we ever come across a vertex that has already been visited, then there must be more than one path that leads to that vertex from the starting vertex, which indicates the existence of a cycle in the graph. A recursive depth-first search implementation is shown below:

```
1  bool dfs_helper(const std::vector<std::vector<int32_t>>& adj_list,
2                  std::vector<bool>& visited, int32_t curr, int32_t prev) {
3    visited[curr] = true;
4    for (int32_t neighbor : adj_list[curr]) {
5      if (neighbor != prev) {
6        if (visited[neighbor]) {
7          return true;  // already seen vertex already, so cycle exists
8        } // if
9        if (dfs_helper(adj_list, visited, neighbor, curr)) {
10         return true;
11       } // if
12     } // if
13   } // for neighbor
14   // completed DFS without discovering same vertex twice, so no cycle exists
15   return false;
16 } // dfs_helper()
17
18 bool is_graph_cyclic(const std::vector<std::vector<int32_t>>& adj_list) {
19   std::vector<bool> visited(adj_list.size(), false);
20   return !adj_list.empty() && dfs_helper(adj_list, visited, 0, -1);
21 } // is_graph_cyclic()
```

Note that we check whether `curr` is equal to `prev` on line 5. This is important, because our algorithm would otherwise treat the path $A \to B \to A$ as a cycle, when it is just an edge. To fix this, we will keep track of the previous node at every vertex to ensure that we do not revisit the node we came from and incorrectly assume that there exists a cycle in the graph.

An iterative depth-first search implemention is shown below:

```
bool is_graph_cyclic(const std::vector<std::vector<int32_t>>& adj_list) {
  if (adj_list.empty()) {
    return false;
  } // if
  std::vector<int32_t> parent(adj_list.size(), -1);
  std::stack<int32_t> dfs;
  parent[0] = -2;  // not -1 since -1 indicates unvisited (can use std::optional<> instead in C++17)
  dfs.push(0);
  while (!dfs.empty()) {
    int32_t current = dfs.top();
    dfs.pop();
    for (int32_t neighbor : adj_list[current]) {
      if (neighbor != parent[current]) {
        if (parent[neighbor] != -1) {
          return true;
        } // if
        parent[neighbor] = current;
        dfs.push(neighbor);
      } // if
    } // for neighbor
  } // while
  return false;
} // is_graph_cyclic()
```

In this implementation, the `parent` vector keeps track of each vertex's previous node, so that we don't mistakenly revisit the vertex we came from and immediately conclude that there exists a cycle.

Example 19.20 Implement the same function in the previous example, but using a breadth-first search instead.

The breadth-first search solution is fairly similar to the iterative depth-first search approach, but using a queue instead of a stack. We pick an arbitrary vertex in the graph and begin a breadth-first search. If we ever discover the same vertex more than once, then there must be a cycle; otherwise, there is not. The code is shown below.

```
bool is_graph_cyclic(const std::vector<std::vector<int32_t>>& adj_list) {
  if (adj_list.empty()) {
    return false;
  } // if
  std::vector<int32_t> parent(adj_list.size(), -1);
  std::queue<int32_t> bfs;
  parent[0] = -2;
  bfs.push(0);
  while (!bfs.empty()) {
    int32_t current = bfs.top();
    bfs.pop();
    for (int32_t neighbor : adj_list[current]) {
      if (neighbor != parent[current]) {
        if (parent[neighbor] != -1) {
          return true;
        } // if
        parent[neighbor] = current;
        bfs.push(neighbor);
      } // if
    } // for neighbor
  } // while
  return false;
} // is_graph_cyclic()
```

In addition to BFS and DFS, there is actually a shortcut that can be used to detect cycles in a graph, *provided that the graph is connected.* Given a connected graph with $|V|$ vertices, the number of edges in the graph must be equal to $|V|-1$ for the graph to be acyclic. If there are fewer than $|V|-1$ edges, then the graph would not be connected, as every vertex in the graph needs to touch an edge that connects it to the remaining elements. However, the addition of an any edge beyond $|V|-1$ would end up creating a cycle, since it would then be possible to reach some vertex in the graph along more than one unique path. As a result, we can detect the existence of a cycle in a connected graph by simply counting up the edges and checking if it is equal to $|V|-1$.

Example 19.21 Implement the same function as the previous two examples, but solve the problem by counting edges in the graph instead of conducting a graph search.

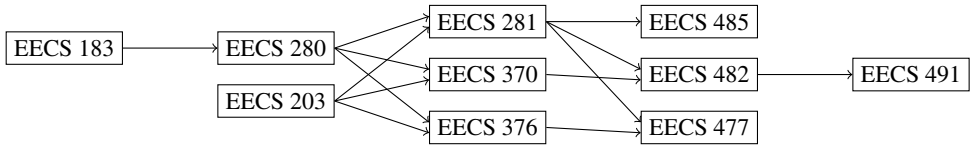
Since the graph is undirected and connected, each edge appears twice in the adjacency list. Thus, if the number of edges in the graph is $2(|V|-1)$, then the graph has no cycles. If the number of edges exceeds $2(|V|-1)$, then there must exist a cycle in the graph.

```
1  bool is_graph_cyclic(const std::vector<std::vector<int32_t>>& adj_list) {
2    size_t num_vertices = adj_list.size();
3    size_t num_edges = 0;
4    for (size_t i = 0; i < num_vertices; ++i) {
5      num_edges += adj_list[i].size();
6    } // for i
7    return !adj_list.empty() && (num_edges / 2) > (num_vertices - 1);
8  } // is_graph_cyclic()
```

19.6 Topological Sort (⁎)

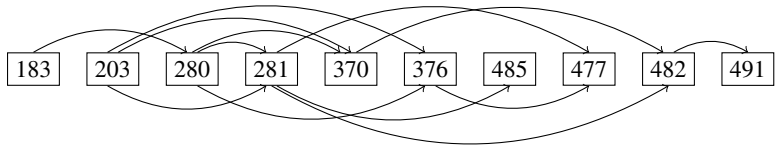
※ 19.6.1 *Topological Orderings* (⁎)

DFS and BFS can also be applied to sort the vertices of a directed graph with no cycles. This is quite useful, since many problems in real life can be modeled as directed graphs, where certain events must happen before others. For instance, consider the problem of class prerequisites: if you wanted to take EECS 281, you would have to take all of its prerequisites first (EECS 183, EECS 203, EECS 280, etc.). We can represent these prerequisites in the form of a graph, as shown below:

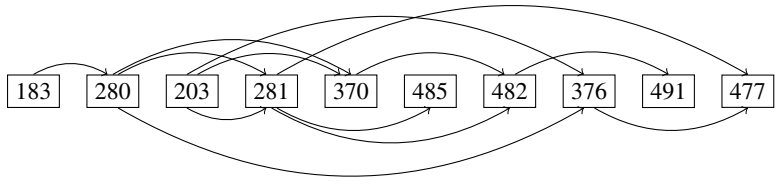


In this graph, there exists an *ordering* to the vertices, such that certain vertices must come before others. For example, you can't go and take EECS 491 without first taking EECS 370 and EECS 482 (assuming that you have taken EECS 281 already).

In a directed graph, a valid ordering of its vertices is known as a **topological ordering**. In a topological ordering, the existence of a directed edge from vertex X to vertex Y indicates that X comes before Y in the ordering of vertices. Topological orderings only exist in directed graphs that do *not* contain cycles, known as directed acyclic graphs (DAGs). This is because it is impossible to order vertices in a directed cycle, as the last vertex in any ordering would always point back to a vertex before it. The following is a valid topological ordering of the classes in the previous graph:

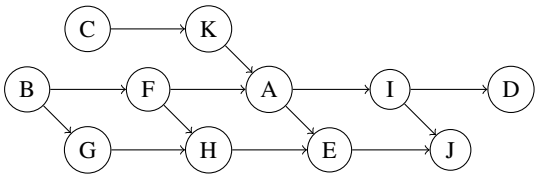


The above ordering is valid because all of the directed edges point to the right. Topological orderings need *not* be unique, and there may be multiple topological orderings that exist within a single graph. For instance, the following is another valid topological ordering of the classes:

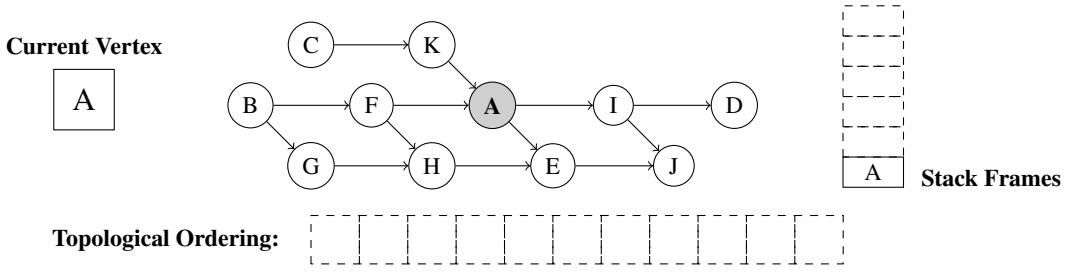


※ 19.6.2 *Depth-First Search Topological Sort* (✻)

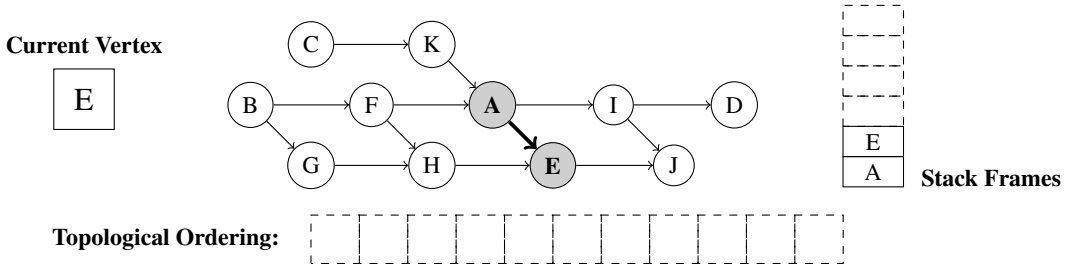
To find a valid topological ordering of a directed acyclic graph, we can use an algorithm known as **topological sort**. One common implementation of topological sort relies on a *depth-first search*. In this approach, an unvisited vertex is arbitrarily selected in the graph, and a recursive depth-first search is performed to explore all vertices that are reachable from that vertex. Every time a recursive call unrolls along the search, the current vertex is added to the last available position of the current topological ordering. This process is repeated as long as there exist vertices that remain unvisited. As an example, suppose we wanted to find a topological ordering of the following DAG:



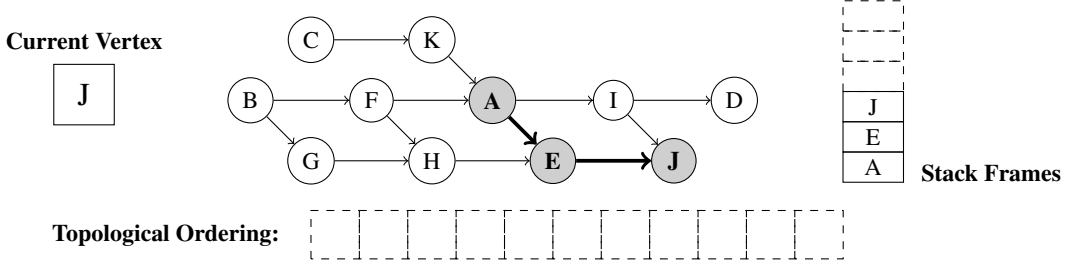
To topologically sort this graph using the depth-first search approach, we first select an arbitrary vertex and begin a recursive depth-first search. In this case, we will select vertex A.



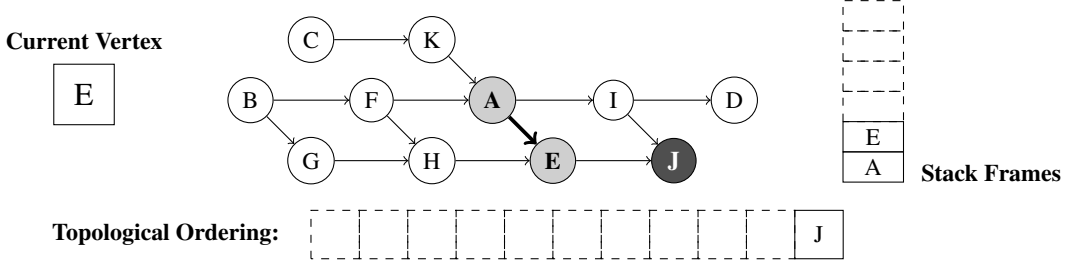
To perform the depth-first search, we will make recursive calls on each of vertex A's unvisited neighbors. To keep things simple, we will make these recursive calls in alphabetical order. In this case, we will first make a recursive call on vertex E.



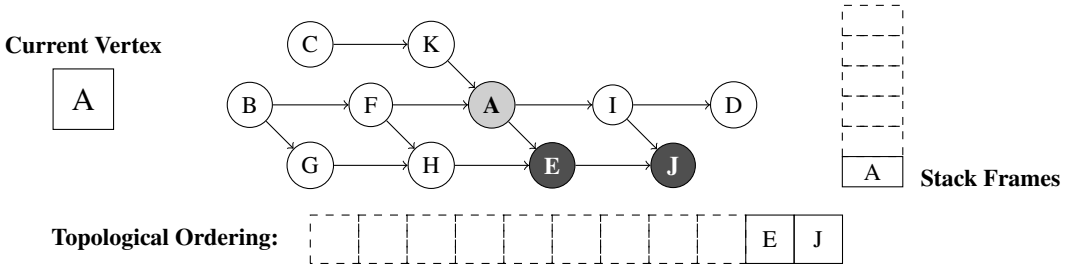
Vertex E is now our current vertex. We then make a recursive call on vertex J, which is vertex E's only unvisited neighbor.



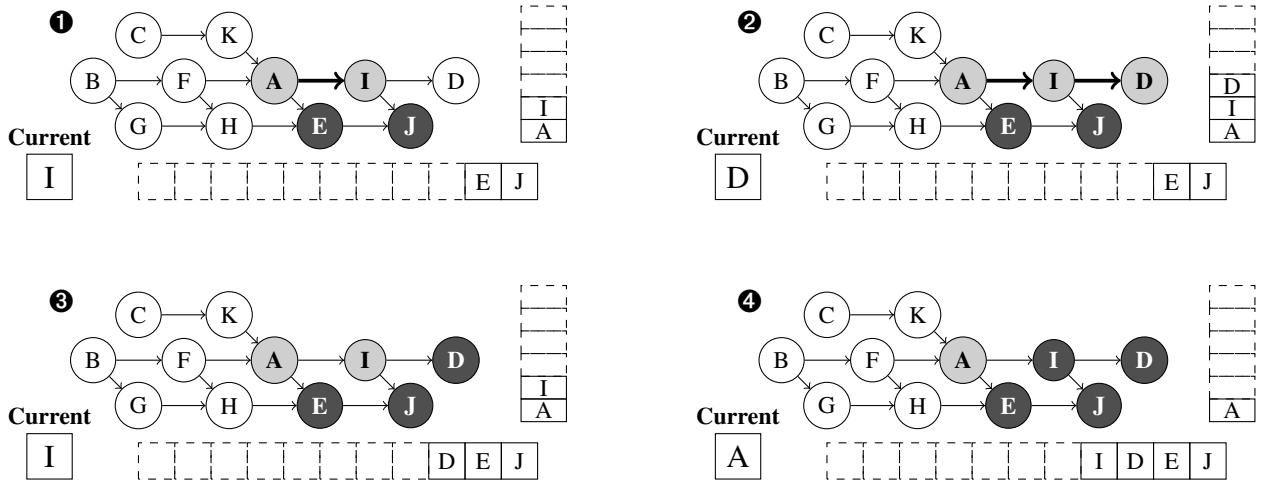
Vertex J has no unvisited neighbors, so there are no recursive calls that need to be done. Thus, we are done processing vertex J, so we can unroll the recursion back to vertex E. Before we return from the recursive call, we will add J to the last available position of our current topological ordering (since we know that no unvisited vertices depend on vertex J).



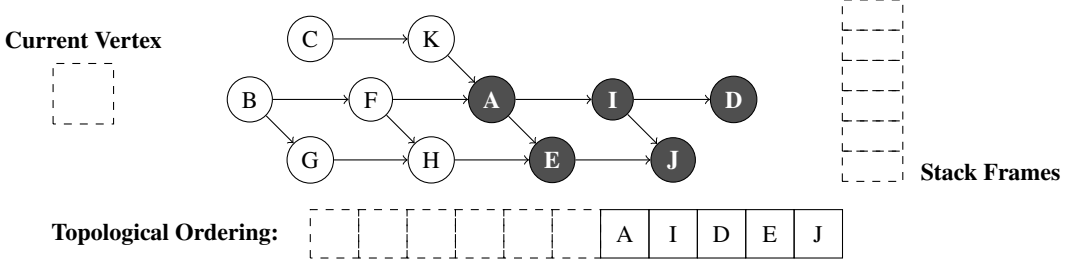
The recursion unrolls back to vertex E. Vertex E now has no more unvisited neighbors, so we are done processing it. Before the recursion returns to vertex A, we will add E to the last available position of our current topological ordering.



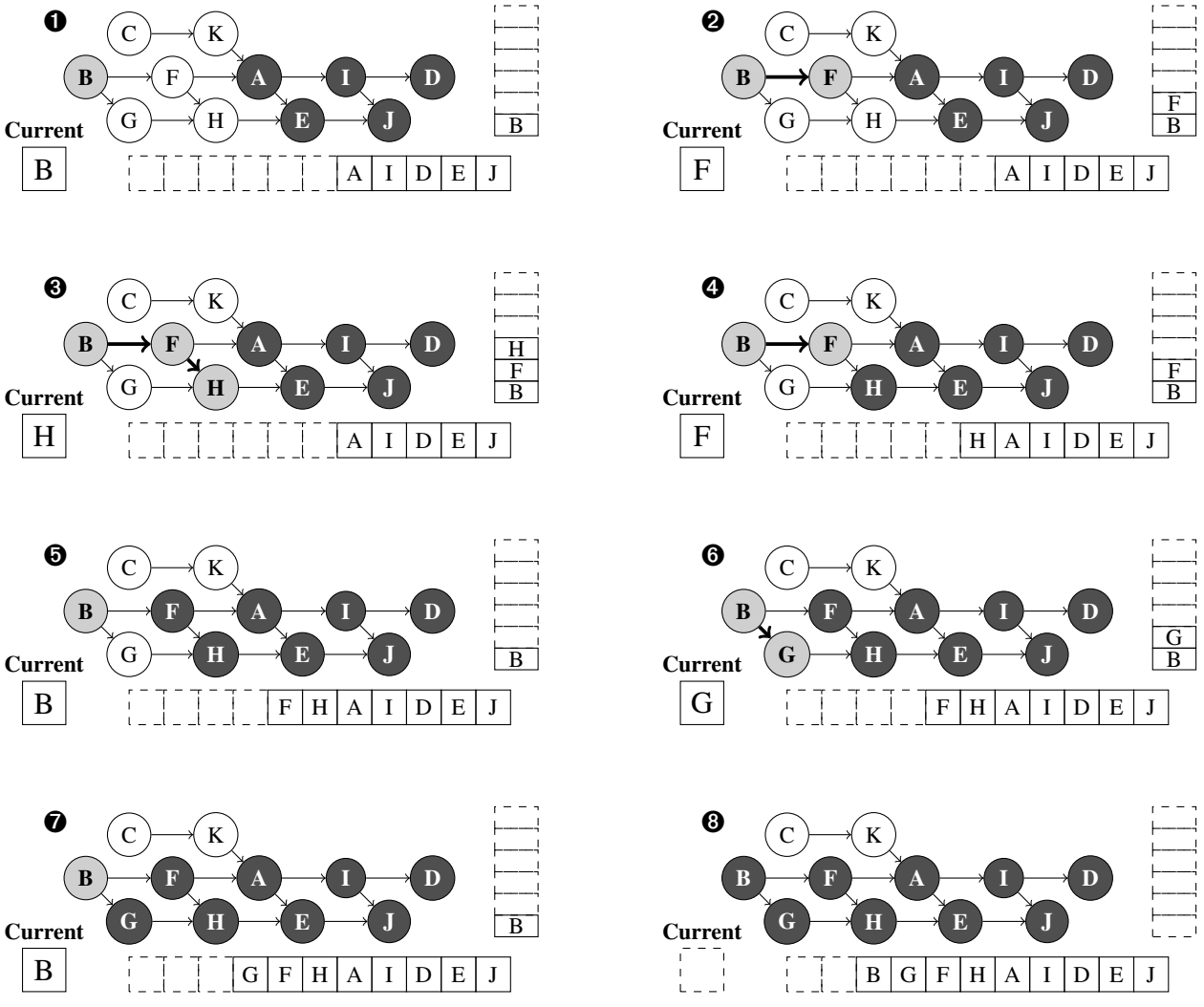
The recursion unrolls back to vertex A. Now that the recursive call on vertex E has been completed, we will now make a recursive call on the remaining unvisited neighbor, vertex I. We follow the same procedure as before: after making a recursive call on vertex I, we then make a recursive call on vertex D. Since vertex D has no unvisited neighbors, the recursion unrolls from D back to I, and we add D to the last available position of our current topological ordering. Similarly, after processing vertex D, all of vertex I's neighbors have been visited, so we add I to the last available position of our current topological ordering.



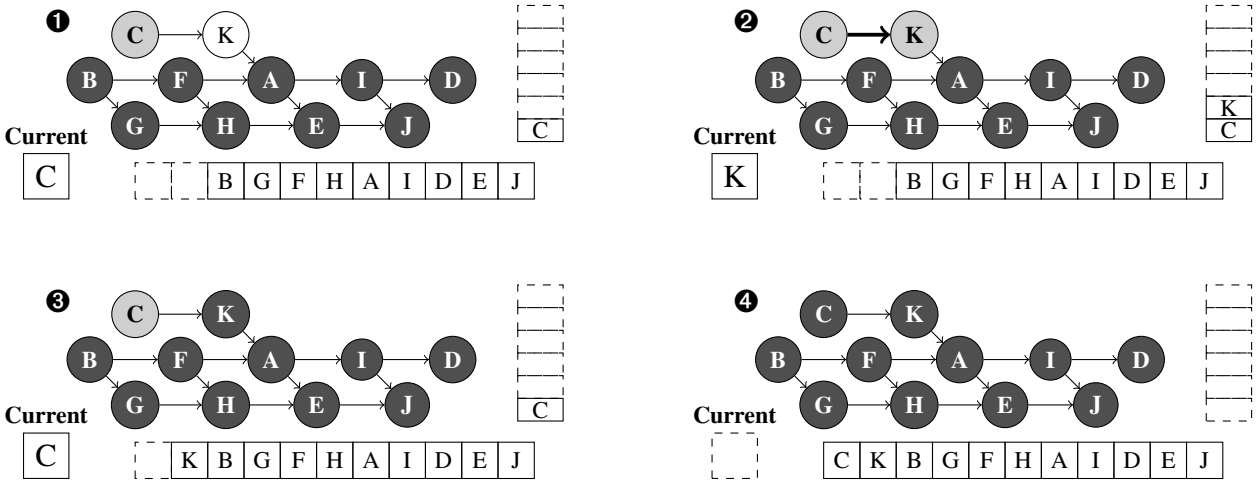
All of vertex A's neighbors have been processed, so we add A to the front of our current topological ordering. The stack frame of vertex A unrolls, and our depth-first search on vertex A is complete.



Our topological sort, however, is not complete, since we still have vertices that still need to be visited. Therefore, we will need to select another arbitrary unvisited vertex and begin a depth-first search. For our example, we will select vertex B next. During this depth-first search on vertex B, we add vertices H, then F, then G, then B to the last available position of our topological ordering.



We still have vertices that need to be visited, so we will need to select another unvisited vertex to perform a depth-first search on. For this example, we will select vertex C. During our depth-first search on vertex C, we first add vertex K, and then add vertex C to the last available position of our topological ordering.



All of the vertices in the graph have been added to the ordering, so our topological sort is complete. The result we get is a valid topological ordering of the given graph. An implementation of the depth-first search approach to topological sort is shown below:

```
void dfs_helper(const std::vector<std::vector<int32_t>>& adj_list,
                std::unordered_set<int32_t>& visited, std::deque<int32_t>& order, int32_t vertex) {
  visited.insert(vertex);
  for (int32_t neighbor : adj_list[vertex]) {
    if (visited.find(neighbor) == visited.end()) {
      dfs_helper(adj_list, visited, order, neighbor);
    } // if
  } // for neighbor
  order.push_front(vertex);
} // dfs_helper()

std::deque<int32_t> topological_sort(const std::vector<std::vector<int32_t>>& adj_list) {
  std::deque<int32_t> order;
  std::unordered_set<int32_t> visited;
  for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
    if (visited.find(vertex) == visited.end()) {
      dfs_helper(adj_list, visited, order, vertex);
    } // if
  } // for vertex
  return order;
} // topological_sort()
```

Example 19.22 There are a total of n classes you have to take to graduate, each labeled with an integer from 0 to $n-1$. Some courses may have prerequisites. You are given a vector of all prerequisite pairs, `prereqs`, where `prereq[i] = [a, b]` indicates that you must take course `b` *before* course `a` (i.e., `b` is a prerequisite of `a`). Given the total number of courses `n` and a vector of all prerequisite pairs, return an ordering of courses you should take to finish all courses. If there are multiple valid orderings, return any of them. You are guaranteed that there exists at least one valid ordering.

```
std::vector<int> schedule_courses(int n, std::vector<std::vector<int>>& prereqs);
```

Given `n = 4` and `prereqs = [[1, 0], [2, 0], [3, 1], [3, 2]]`, you would output `[0, 1, 2, 3]` OR `[0, 2, 1, 3]`. This is because course 3 can only be taken after finishing both 1 and 2, and courses 1 and 2 can only be taken after finishing course 0.

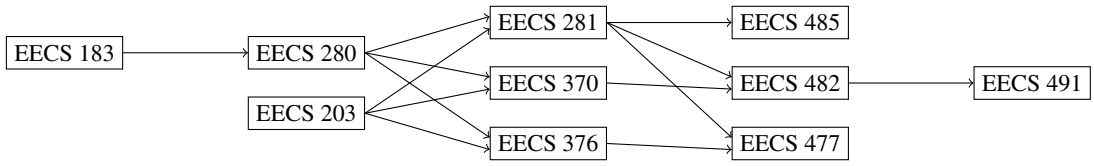
To solve this problem, we will use a topological sort. Using the depth-first search approach, we get the following:

```
std::vector<int32_t> schedule_courses(int32_t n, const std::vector<std::vector<int32_t>>& prereqs) {
  std::vector<std::vector<int32_t>> adj_list(n);
  for (auto& vec : prereqs) {
    adj_list[vec[1]].push_back(vec[0]);
  } // for vec
  std::vector<int32_t> order;
  std::vector<bool> visited(n, false);
  for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
    if (!visited[vertex]) {
      dfs_helper(adj_list, visited, order, vertex);
    } // if
  } // for vertex
  std::reverse(order.begin(), order.end());
  return order;
} // schedule_courses()

void dfs_helper(const std::vector<std::vector<int32_t>>& adj_list,
                std::vector<bool>& visited, std::vector<int32_t>& order, int32_t vertex) {
  visited[vertex] = true;
  for (int32_t neighbor : adj_list[vertex]) {
    if (!visited[neighbor]) {
      dfs_helper(adj_list, visited, order, neighbor);
    } // if
  } // for neighbor
  order.push_back(vertex);
} // dfs_helper()
```

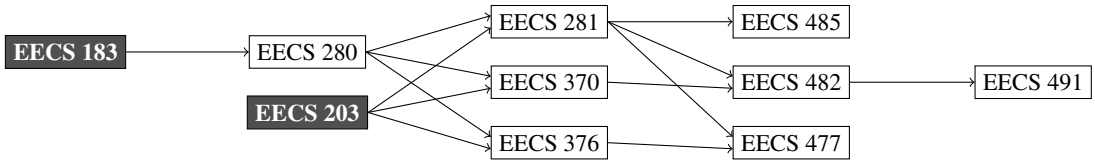
The time complexity of the topological sorting algorithm using depth-first search is $\Theta(|V|+|E|)$ on an adjacency list, since each vertex and edge is examined a constant number of times. The time complexity of this algorithm on an adjacency matrix is $\Theta(|V|^2)$, since finding all the connections of a vertex takes $\Theta(|V|)$ time, which is done a total of $|V|$ times.

※ 19.6.3 *Kahn's Algorithm* (⁎)

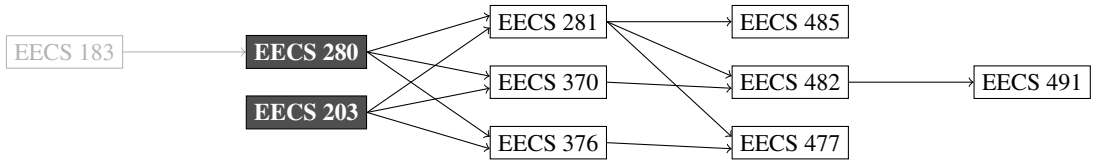
A topological sort can also be done iteratively rather than recursively. A common iterative implementation of topological sort is **Kahn's algorithm**, which relies on a *breadth-first search* and takes advantage of a vertex's *in-degree* to find a topological ordering. Kahn's algorithm is actually rather intuitive, as it reflects how events are ordered in real life. Consider the graph of classes introduced previously:



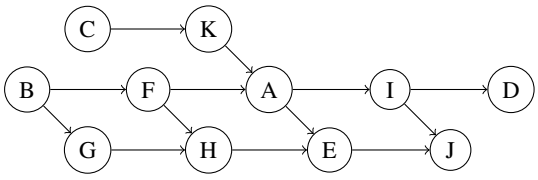
If we wanted to build a schedule of classes that followed prerequisite rules, the logical decision would be to first look for classes that do not have any existing prerequisites (i.e., vertices with an in-degree of zero) and complete those first — in this case, we are able to take EECS 183 and EECS 203, so one of those classes should be first in the topological ordering.



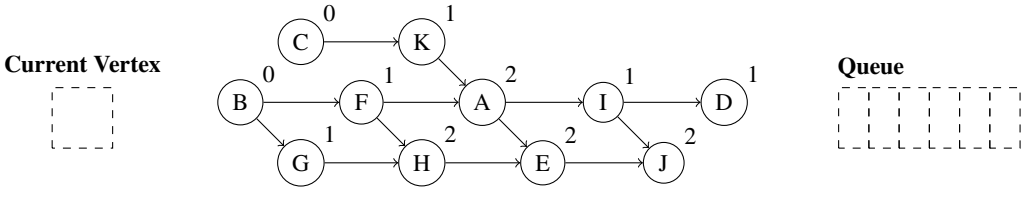
Suppose we choose EECS 183 as the first class in our topological ordering. After taking EECS 183, we would then be able to take EECS 280, since we have satisfied all of its prerequisites. This is akin to "removing" EECS 183 from consideration in our prerequisite graph.



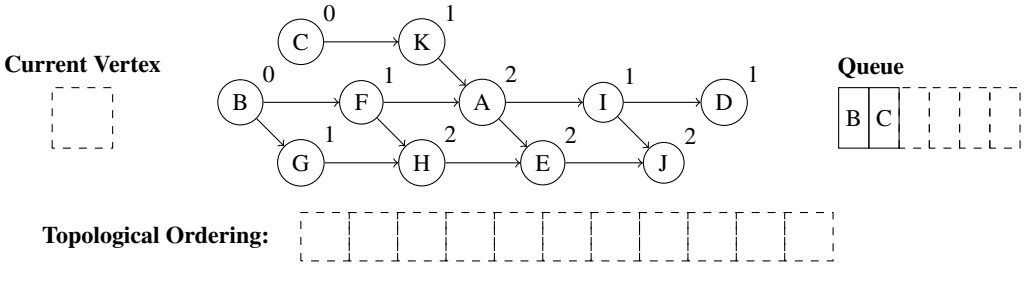
Kahn's algorithm follows this idea to topologically sort a graph. At the beginning, all vertices with an in-degree of zero are pushed into a queue. Then, the vertex at the front of the queue is taken out, added to the topological ordering, and then removed from the graph. The neighbors of this removed vertex are then updated, and any vertex with an updated in-degree of zero is pushed into the queue. This procedure is repeated until all the vertices are processed. For example, suppose we wanted to topologically sort the following graph using Kahn's algorithm.



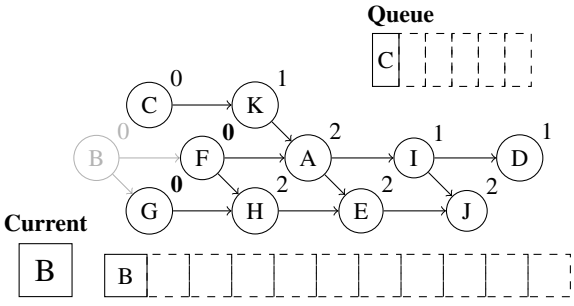
First, we will calculate the in-degree of all of the nodes.

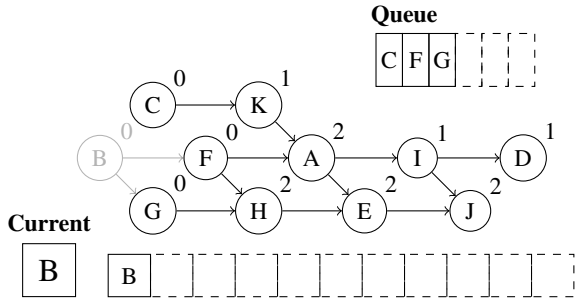


Next, we will push all of the vertices with an in-degree of zero into the queue. Similar to before, we will push vertices into the queue in alphabetical order, so vertex *B* will be pushed in before vertex *C*.

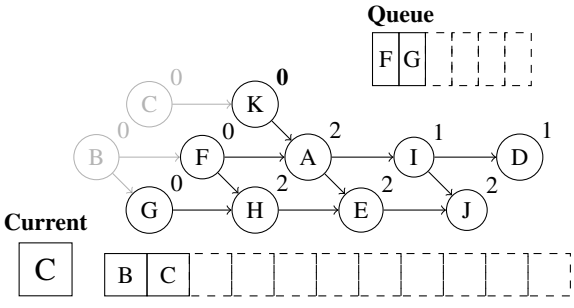


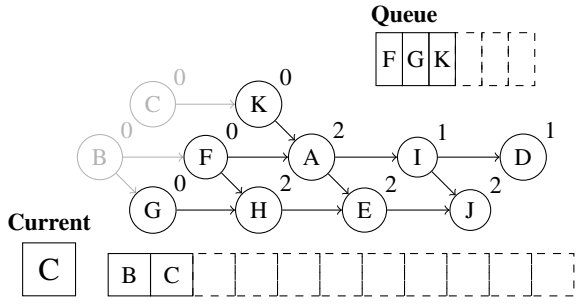
We take vertex B out of the front of the queue and add it to the topological ordering. Then, we remove vertex B from the graph and update the in-degree of its neighbors. Vertices F and G now have an in-degree of zero, so we push them into the queue.



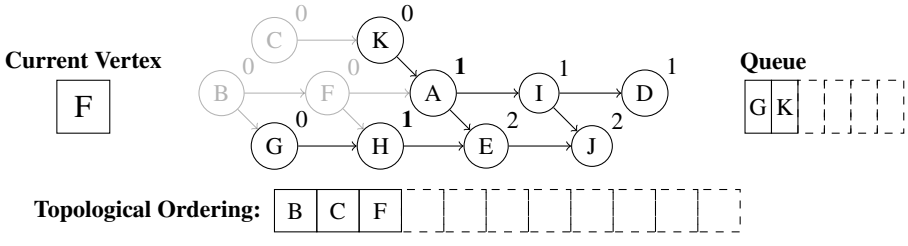


Vertex C is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex C from the graph and update the in-degree of its neighbor, vertex K. Vertex K now has an in-degree of zero, so we push K into the queue.

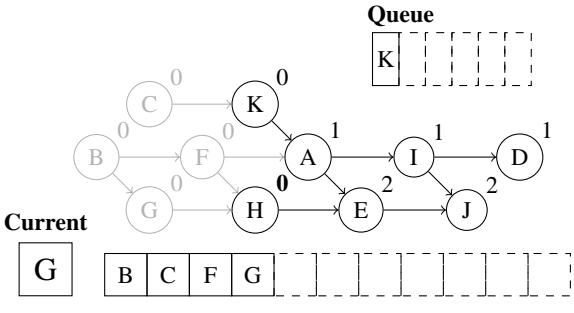


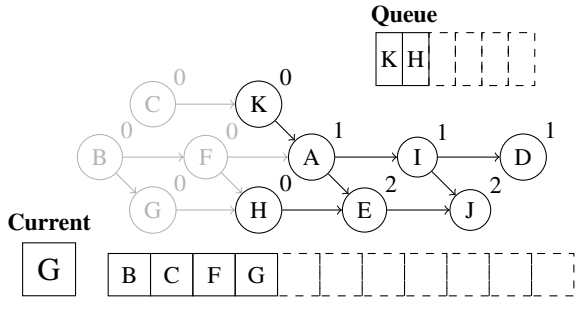


Vertex F is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex F from the graph and update the in-degree of its neighbors, vertices A and H.

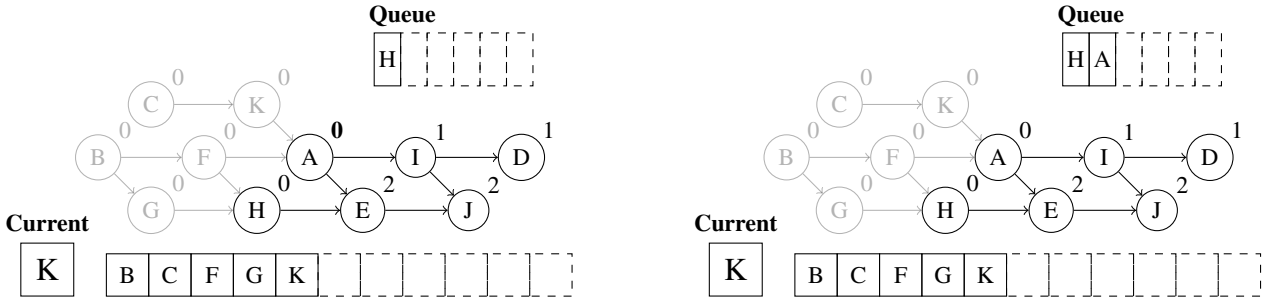


Neither vertex A nor vertex H has an updated in-degree of zero, so nothing gets added to the queue just yet. Next, we take out vertex G from the front of the queue and add it to our topological ordering. We then remove vertex G from the graph and update the in-degree of its neighbor, vertex H. Vertex H's in-degree is now zero, so we push H into the queue.

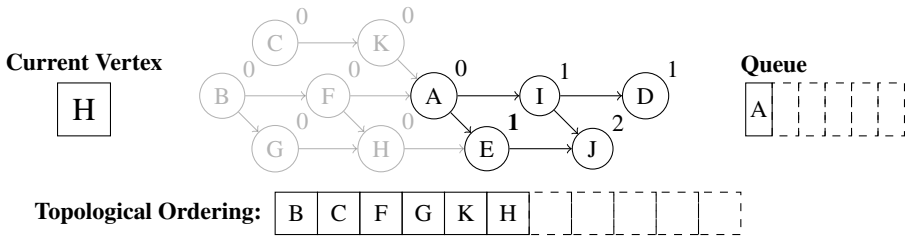




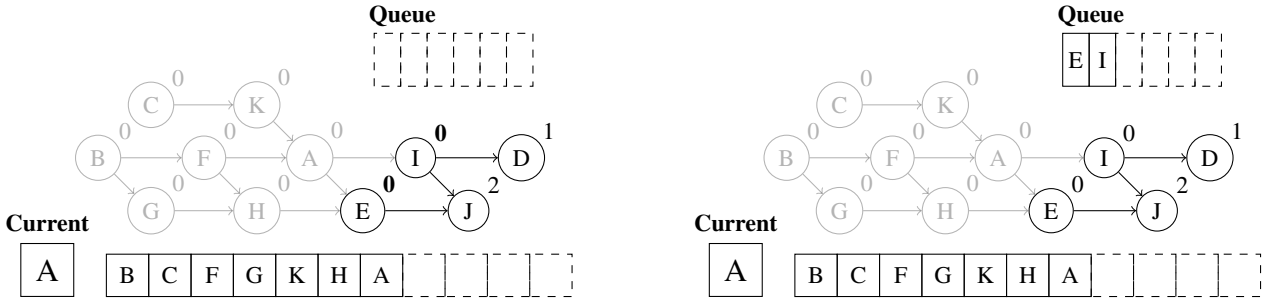
Vertex K is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex K from the graph and update the in-degree of its neighbor, vertex A. Vertex A's in-degree is now zero, so we push A into the queue.



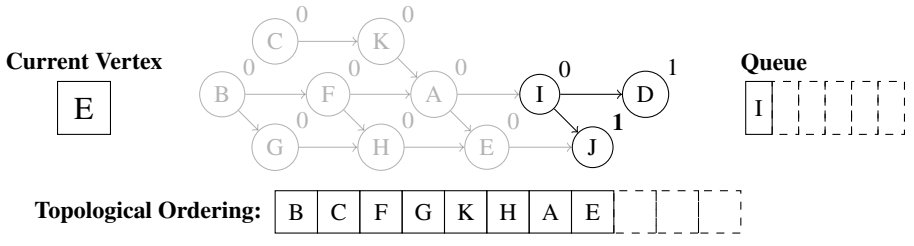
Vertex H is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex H from the graph and update the in-degree of its neighbor, vertex E.



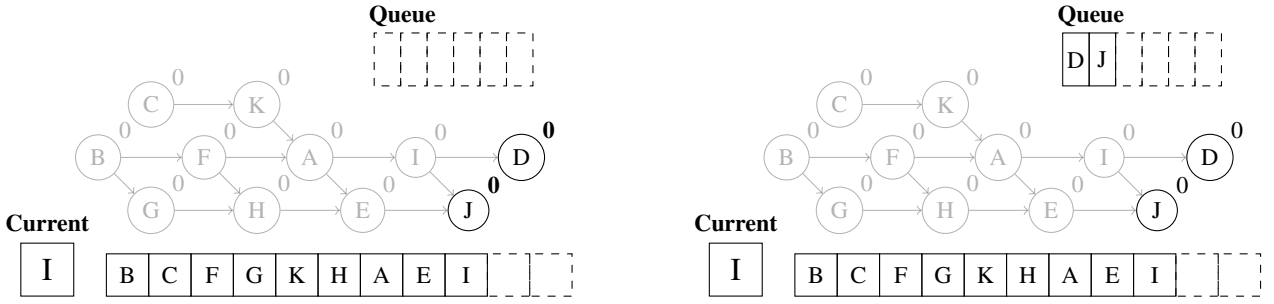
Vertex E's in-degree is not yet zero, so we do not push it into the queue just yet. Next, we take out vertex A, which is at the top of the queue, and add it to the topological ordering. We then remove vertex A from the graph and update the in-degrees of its neighbors, vertices E and I. Both E and I now have in-degrees of zero, so we push them into the queue.



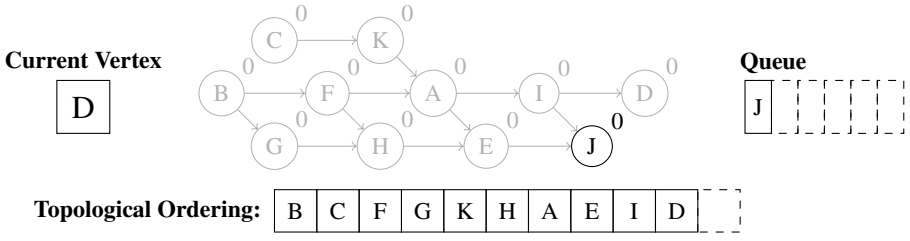
Vertex E is at the front of the queue, so we take it out and add it to our topological ordering. We then remove vertex E from the graph and update the in-degree of its neighbor, vertex J.



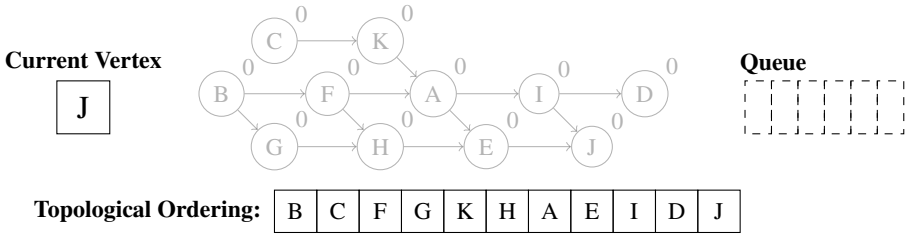
Vertex J's updated in-degree is not yet zero, so we do not push it into the queue. Next, we take out vertex I, which is at the front of the queue, and add it to our topological ordering. We then remove vertex I from the graph and update the in-degrees of its neighbors, vertices D and J. Both D and J now have in-degrees of zero, so we push them into the queue.



Vertex D is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex D from the graph. Since D has no neighbors, nothing is pushed into the queue.



Vertex J is next in the queue, so we take it out and add it to our topological ordering. We then remove vertex J from the graph. Similar to vertex D, vertex J has no neighbors, so nothing is pushed into the queue. All the vertices in the graph have been processed, so the algorithm completes, and we have our final topological ordering.



The code for Kahn's algorithm is shown below. First, we calculate the in-degree of every vertex in the graph. Then, we explore vertices with an in-degree of zero using a breadth-first search, adding them to the topological ordering and removing them from the graph along the way. The time complexity of Kahn's algorithm on an adjacency list is $\Theta(|V|+|E|)$, since most of the work comes from the breadth-first search. Similarly, the time complexity of Kahn's algorithm is $\Theta(|V|^2)$ on an adjacency matrix.

```
1   std::vector<int32_t> topological_sort(const std::vector<std::vector<int32_t>>& adj_list) {
2     std::vector<int32_t> in_degrees(adj_list.size(), 0);
3     for (auto& row : adj_list) {
4       for (int32_t vertex : row) {
5         ++in_degrees[vertex];
6       } // for vertex
7     } // for row
8     std::vector<int32_t> result;
9     std::queue<int32_t> bfs;
10    for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
11      if (in_degrees[vertex] == 0) {
12        bfs.push(vertex);
13      } // if
14    } // for vertex
15    while (!bfs.empty()) {
16      int32_t curr = bfs.front();
17      bfs.pop();
18      result.push_back(curr);
19      for (int32_t neighbor : adj_list[curr]) {
20        if (--in_degrees[neighbor] == 0) {
21          bfs.push(neighbor);
22        } // if
23      } // for
24    } // while
25    return result;
26  } // topological_sort()
```

Example 19.23 There are a total of n classes you have to take to graduate, each labeled with an integer from 0 to $n-1$. Some courses may have prerequisites. You are given a vector of all prerequisite pairs, `prereqs`, where `prereq[i] = [a, b]` indicates that you must take course `b` *before* course `a` (i.e., `b` is a prerequisite of `a`). Given the total number of courses `n` and a vector of all prerequisite pairs, return an ordering of courses you should take to finish all courses. If there are multiple valid orderings, return any of them. You are guaranteed that there exists at least one valid ordering. *This is the same problem as example 19.22, but use Kahn's algorithm to topologically sort the graph.*

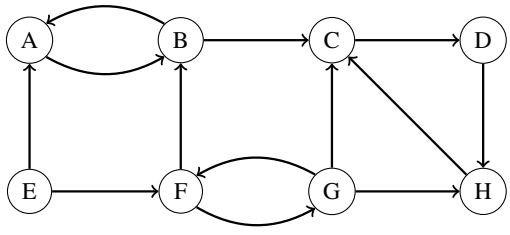
Using the structure of Kahn's algorithm, we can implement the function as follows:

```
std::vector<int32_t> schedule_courses(int32_t n, std::vector<std::vector<int32_t>>& prereqs) {
  std::vector<std::vector<int32_t>> adj_list(n);
  std::vector<int32_t> in_degrees(adj_list.size(), 0);
  for (auto& prereq : prereqs) {
    adj_list[prereq[1]].push_back(prereq[0]);
    ++in_degrees[prereq[0]];
  } // for prereq
  std::vector<int32_t> result;
  std::queue<int32_t> bfs;
  for (int32_t course = 0; course < n; ++course) {
    if (in_degrees[course] == 0) {
      bfs.push(course);
    } // if
  } // for course
  while (!bfs.empty()) {
    int32_t curr = bfs.front();
    bfs.pop();
    result.push_back(curr);
    for (int32_t neighbor : adj_list[curr]) {
      if (--in_degrees[neighbor] == 0) {
        bfs.push(neighbor);
      } // if
    } // for neighbor
  } // while
  return result;
} // schedule_courses()
```

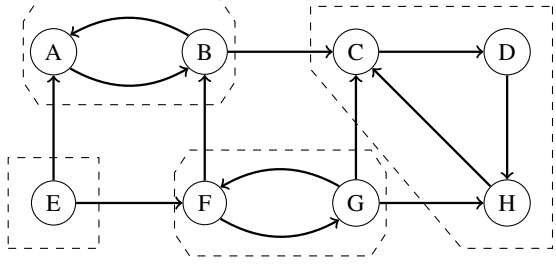
19.7 Algorithms for Strongly Connected Components (⁎)

※ 19.7.1 *Strongly Connected Components* (⁎)

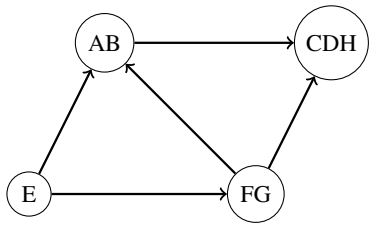
In a directed graph, a **strongly connected component** (SCC) is a subgraph such that, for every pair of vertices u and v in the subgraph, there exists a directed path from u to v and from v to u. That is, in a strongly connected component, it is possible to reach any vertex in the SCC from any other vertex in the same SCC using the directed edges of the graph. As an example, consider the following graph:



There are four strongly connected components in this graph, as shown:



Each of the vertices in a strongly connected component can reach all other vertices in the exact same strongly connected component. One particular trait of strongly connected components is their ability to be condensed into a directed acyclic graph (DAG) by contracting each component into a single vertex. For instance, if we take the above graph and treat each SCC as its own vertex, we would get the following DAG:



Notice that this is always true because it is impossible to create a cycle between two distinct strongly connected components — otherwise, they would be a part of the exact same strongly connected component!

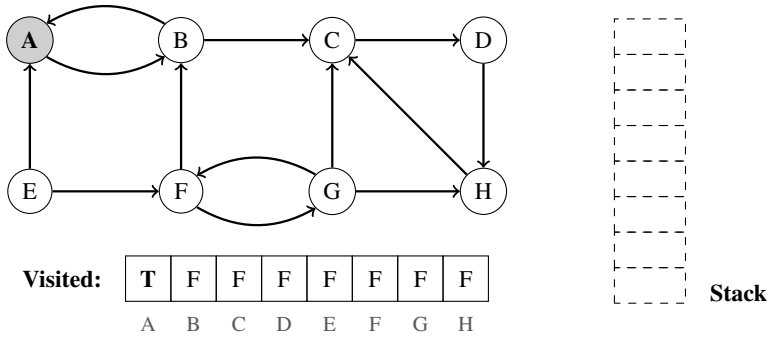
Strongly connected components show up in a lot of real-life examples, so they have several practical applications (for example, you could use SCCs to identify groups of people on a social media page with similar interests). In this section, we will discuss a common algorithm that can be used to solve strongly connected component problems: *Kosaraju's algorithm*.

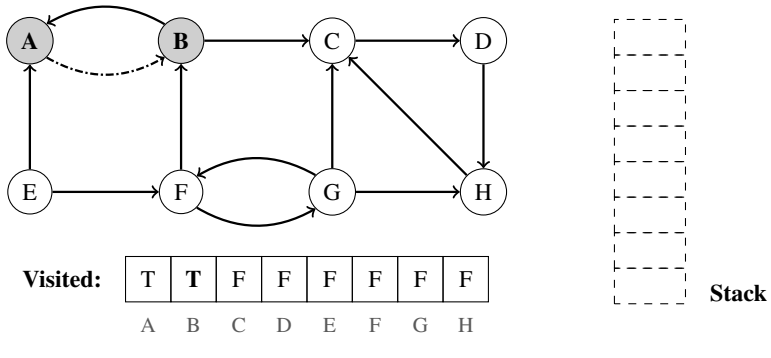
※ 19.7.2 *Kosaraju's Algorithm* (✻)

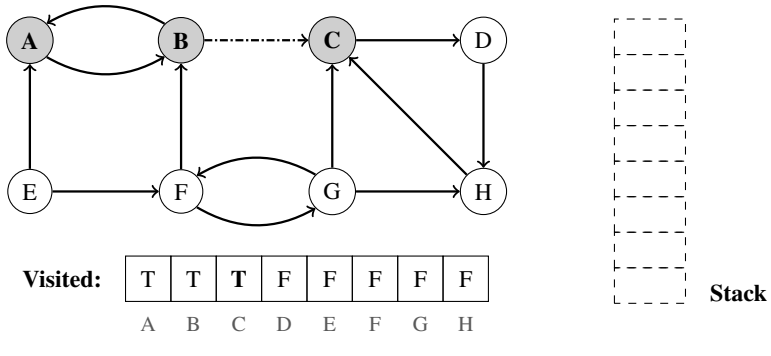
Kosaraju's algorithm is an algorithm that can be used to find the strongly connected components of a graph. This algorithm relies on two passes of depth-first search: the first pass on the original graph, and the second pass on a *transpose* or the original graph (which is the same graph, but with the direction of every edge reversed). The steps of Kosaraju's algorithm are as follows:

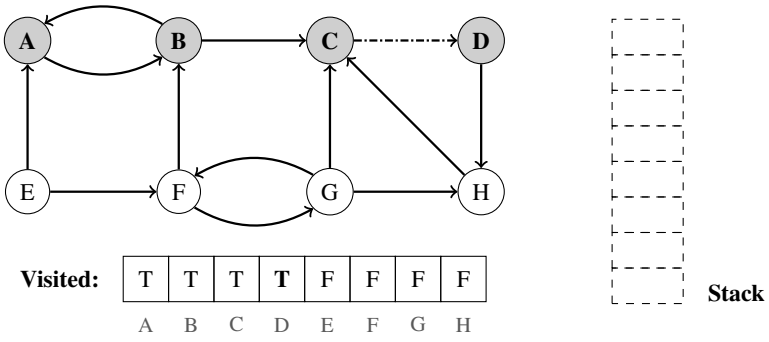
1. Iterate over all the vertices of the original graph. If a vertex is unvisited, perform a recursive depth-first search starting from that vertex, and mark any vertex discovered during the search as visited. After all of a vertex's neighbors are visited, push the vertex onto a stack.
2. Reverse all of the edges of the graph to get its transpose.
3. Continuously pop out vertices from the stack and perform a DFS on the transpose graph starting from each vertex if it is unvisited. All the vertices encountered during each iteration of DFS forms a strongly connected component. Repeat this until all vertices are visited.

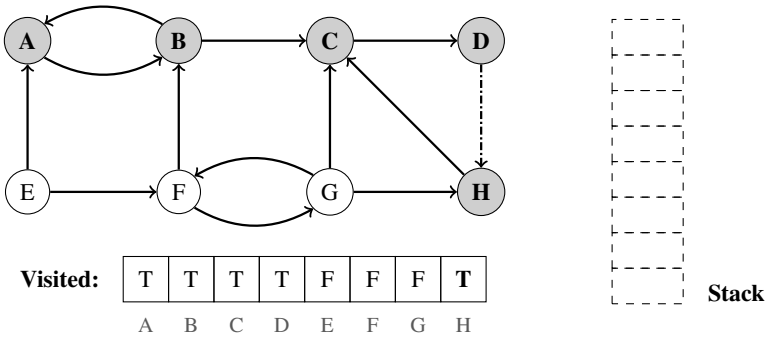
To illustrate this process, consider the graph provided previously. We begin by performing a DFS starting from vertex A, marking each vertex we encounter as visited. Once all of a vertex's neighbors are visited, we push it onto a stack.

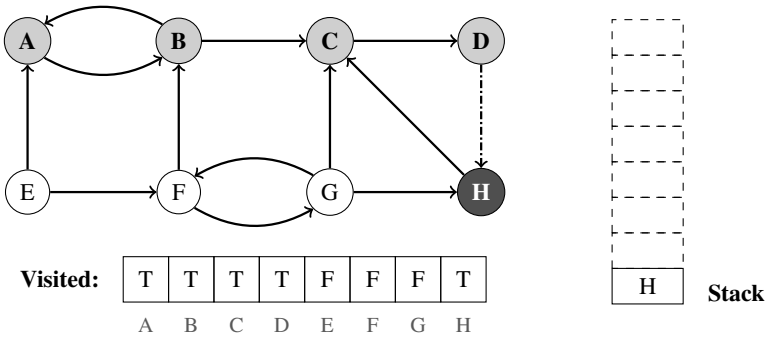


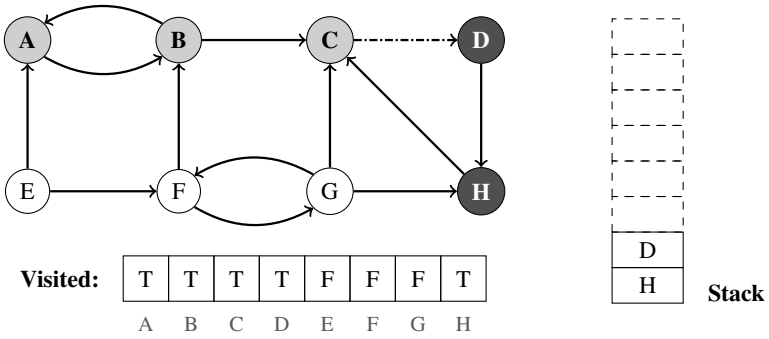


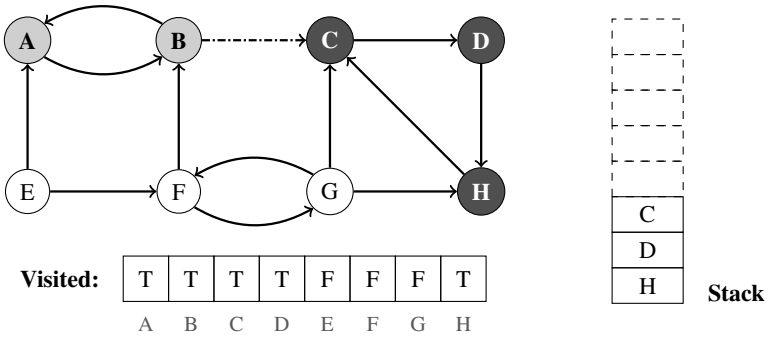


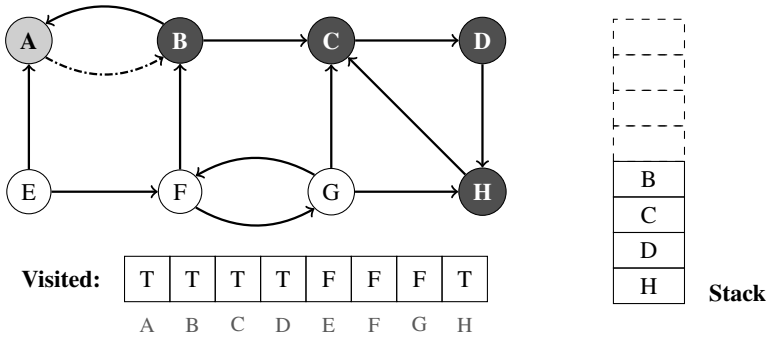

A
B
C
D
E
F
G
H
Visited:
T T T T F F F F
A B C D E F G H
Stack

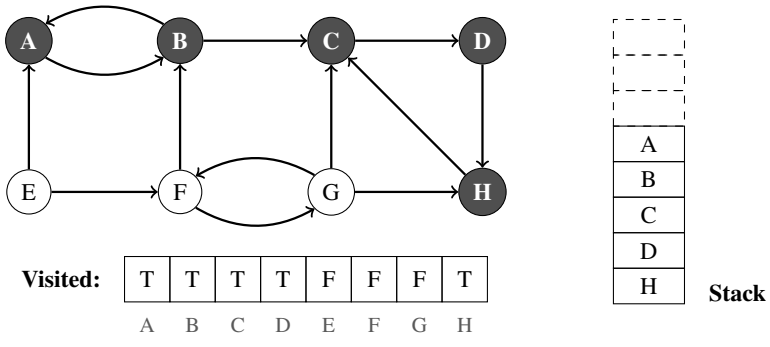

A
B
C
D
E
F
G
H
Visited:
T T T T F F F T
A B C D E F G H
Stack


A
B
C
D
E
F
G
H
Visited:
T T T T F F F T
A B C D E F G H
H
Stack

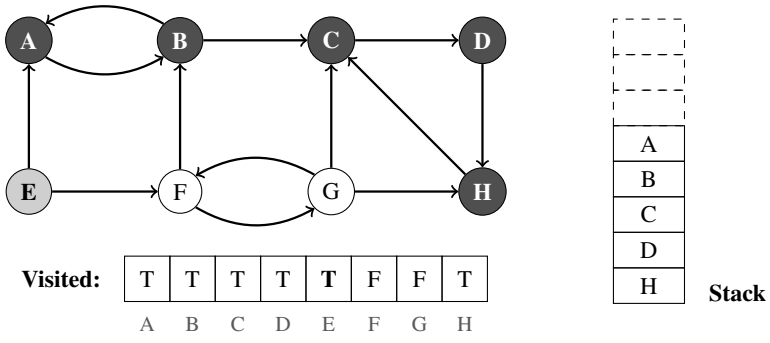

A
B
C
D
E
F
G
H
Visited:
T T T T F F F T
A B C D E F G H
D
H
Stack

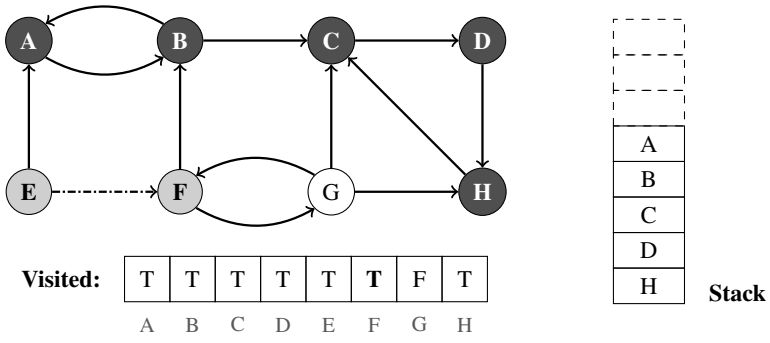

A
B
C
D
E
F
G
H
Visited:
T T T T F F F T
A B C D E F G H
C
D
H
Stack

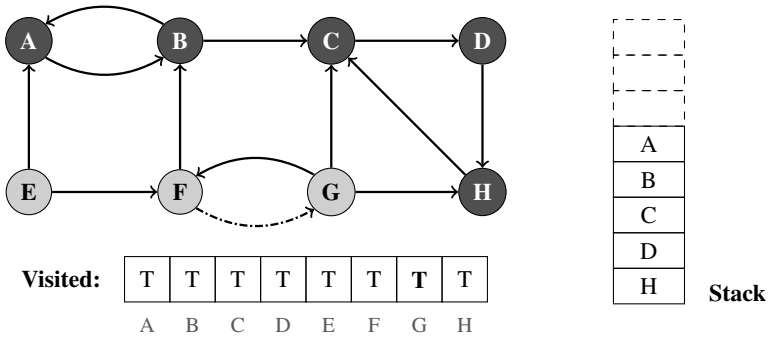


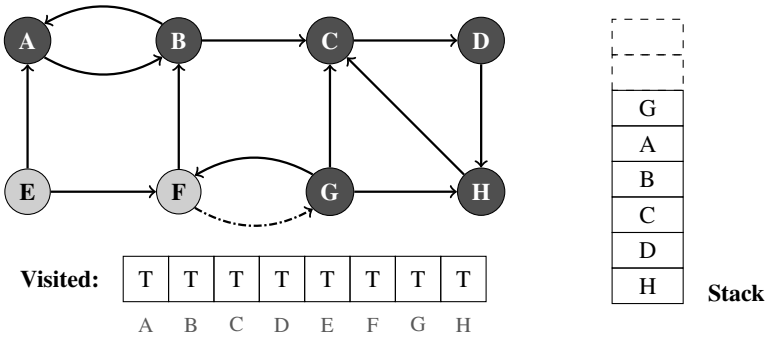


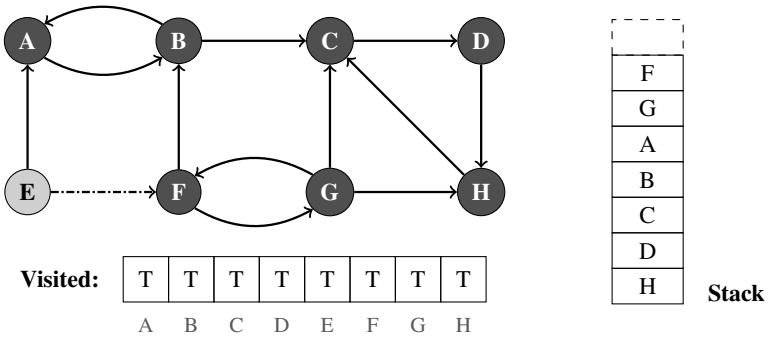
Our DFS from vertex A is now complete. Next, we look at vertex B. Vertex B is already visited, so no work needs to be done there. The same applies to vertices C and D. The next unvisited vertex we encounter is vertex E, so we perform a DFS on vertex E, adding vertices to the stack after we are done visiting all of its neighbors.

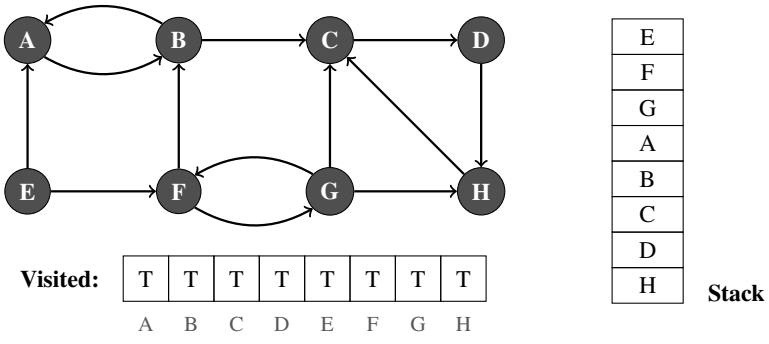




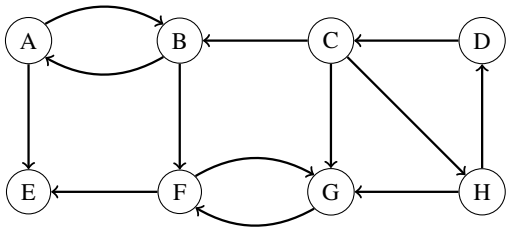




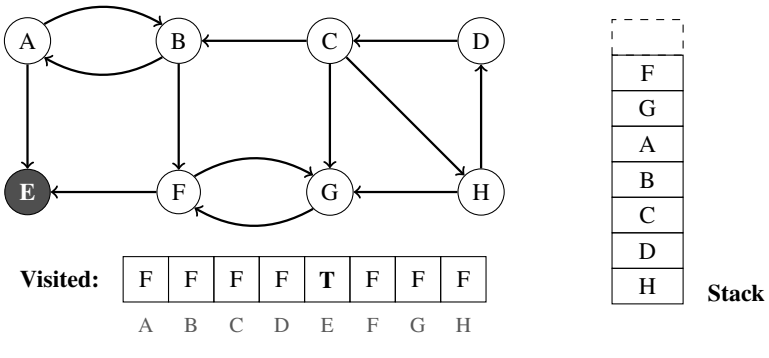




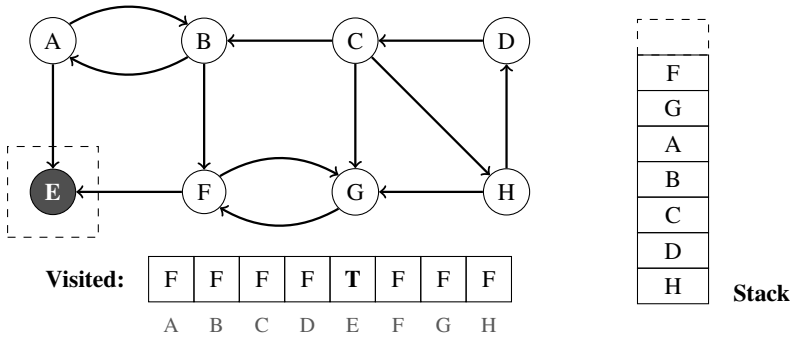
All the vertices have been visited and pushed into the stack, so we have completed our first pass of DFS over the graph. Our next step is to generate a transpose of the graph by reversing the directions of all the edges. The new transpose graph is shown below:



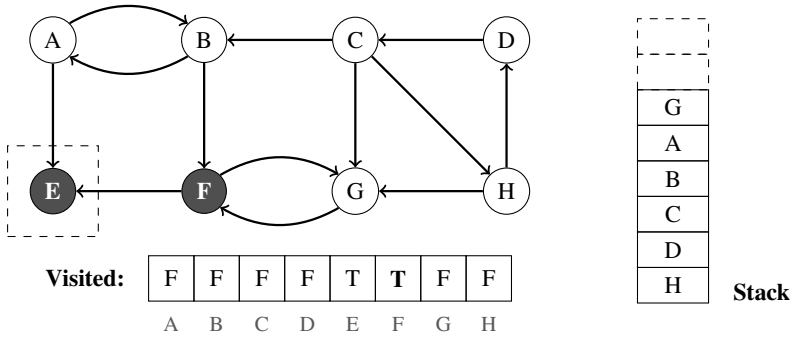
Next, we begin another pass of DFS, this time on the transpose graph. During this second pass, we consider vertices in the order in which they are popped from the stack. In this case, E is at the top of the stack, so we start with a DFS beginning at vertex E.

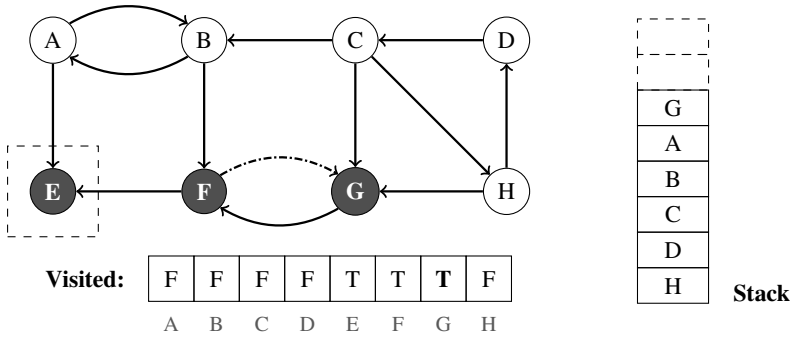


There are no vertices that are reachable from E in this transpose graph, so our DFS completes. We can therefore conclude that E is part of its own strongly connected component.

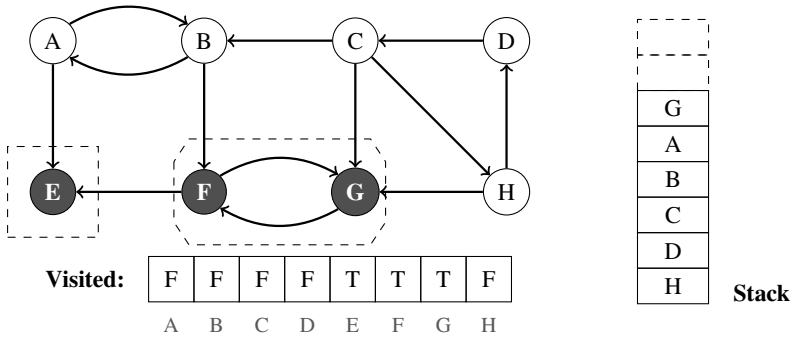


The next vertex at the top of the stack is F. F is unvisited, so we begin a DFS starting from vertex F.

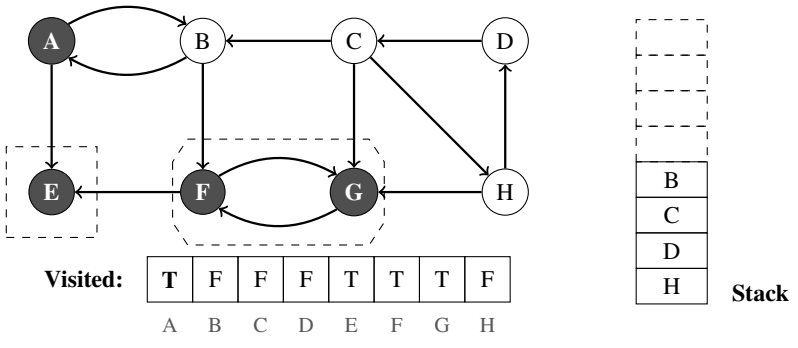


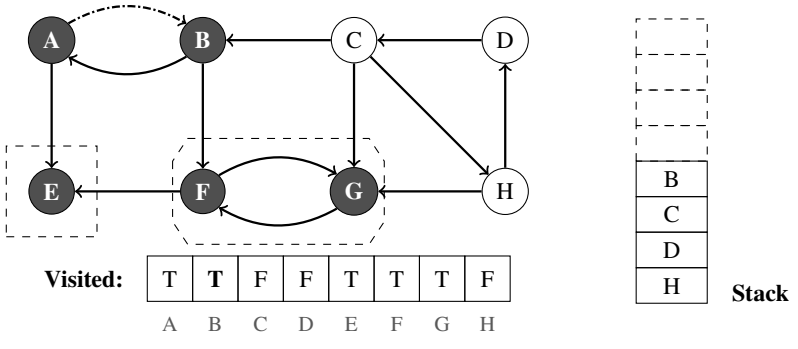


There are no more unvisited vertices that are reachable starting from vertex F, so this iteration of DFS completes. The two vertices that we visited during this DFS, vertices F and G, must therefore be a part of the same strongly connected component.

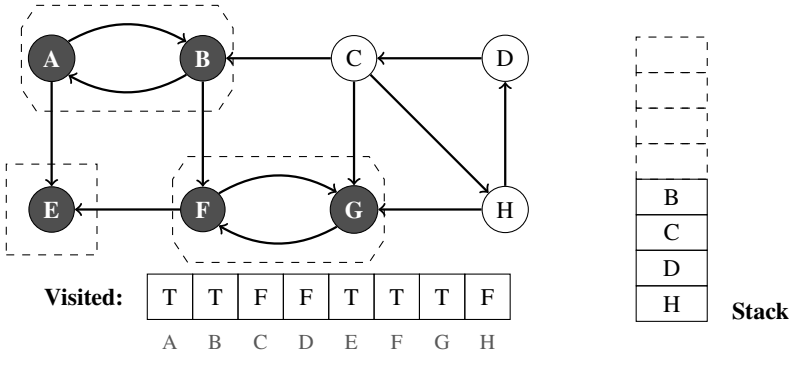


The next vertex we pop off the stack is vertex G, but G has already been visited, so we do not need to do any additional work here. The vertex after that is vertex A. Vertex A is unvisited, so we start a DFS from vertex A.

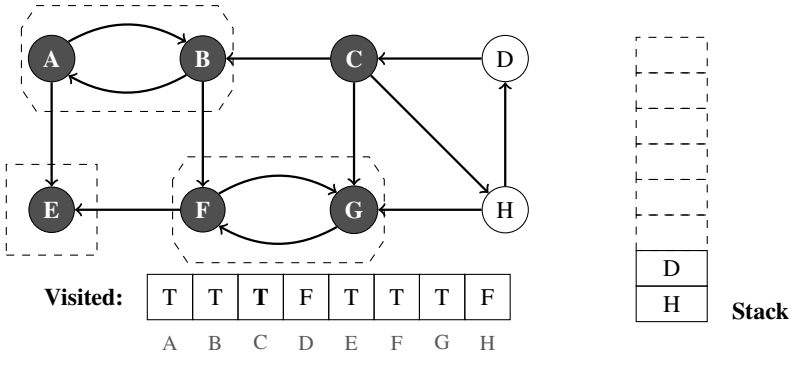


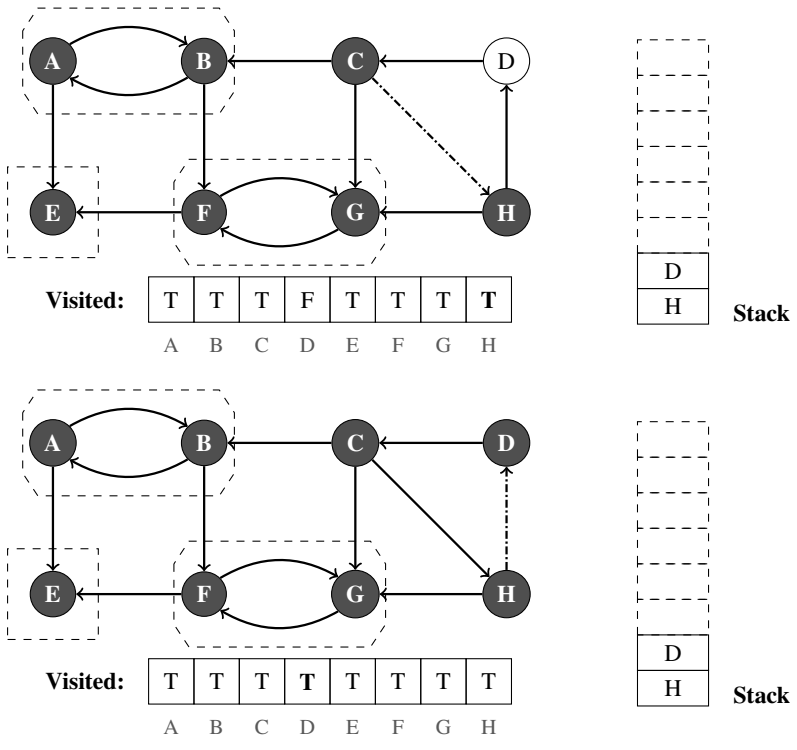


There are no more unvisited vertices that are reachable starting from vertex A, so this iteration of DFS completes. The two vertices that we visited during this DFS, vertices A and B, must therefore be a part of the same strongly connected component.

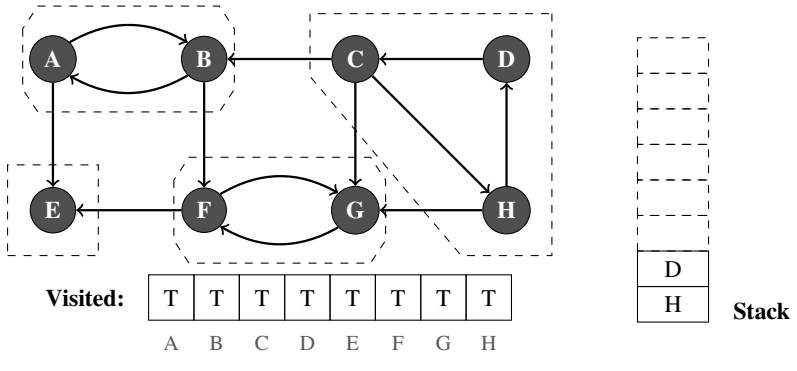


The next vertex we pop off the stack is vertex B, but B has already been visited, so we do not need to do any additional work here. The vertex after that is vertex C. Vertex C is unvisited, so we start a DFS from vertex C.





There are no more unvisited vertices that are reachable starting from vertex C, so this iteration of DFS completes. The three vertices that we visited during this DFS, vertices C, D, and H, must therefore be a part of the same strongly connected component.



The next two vertices in the stack, D and H, have already been visited, so we can pop them off without doing any additional work. The stack is now empty, meaning that our algorithm is complete, and we have found all the strongly connected components in the original graph.

An implementation of Kosaraju's algorithm is shown below (this implementation prints the SCCs to the console):

```
void dfs_helper(const std::vector<std::vector<int32_t>>& adj_list, int32_t vertex,
                std::vector<bool>& visited, std::stack<int32_t>& visit_order) {
  visited[vertex] = true;
  // iterate over all neighbors and begin a DFS if not visited
  for (int32_t neighbor : adj_list[vertex]) {
    if (!visited[neighbor]) {
      dfs_helper(adj_list, neighbor, visited, visit_order);
    } // if
  } // for
  // push the vertices visited into the stack
  visit_order.push(vertex);
} // dfs_helper()

std::vector<std::vector<int32_t>> get_transpose(const std::vector<std::vector<int32_t>>& adj_list) {
  std::vector<std::vector<int32_t>> transpose_graph(adj_list.size());
  for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
    for (int32_t neighbor : adj_list[vertex]) {
      transpose_graph[neighbor].push_back(vertex);
    } // for neighbor
  } // for vertex
  return transpose_graph;
} // get_transpose()

void dfs_helper_transpose(const std::vector<std::vector<int32_t>>& transpose_graph, int32_t vertex,
                          std::vector<bool>& visited) {
  visited[vertex] = true;
  // print out vertex (to identify which vertices belong to each strongly connected component)
  std::cout << vertex << " ";
  // iterate over all neighbors and begin a DFS if not visited
  for (int32_t neighbor : transpose_graph[vertex]) {
    if (!visited[neighbor]) {
      dfs_helper_transpose(transpose_graph, neighbor, visited);
    } // if
  } // for
} // dfs_helper_transpose()

void kosaraju(const std::vector<std::vector<int32_t>>& adj_list) {
  std::stack<int32_t> visit_order;
  std::vector<bool> visited(adj_list.size(), false);
  // first pass of DFS
  for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
    if (!visited[vertex]) {
      dfs_helper(adj_list, vertex, visited, visit_order);
    } // if
  } // for vertex
  // get transpose of graph
  std::vector<std::vector<int32_t>> transpose_graph = get_transpose(adj_list);
  // reset all visited values to false before second pass of DFS
  std::fill(visited.begin(), visited.end(), false);
  // second pass of DFS, visit vertices in the order popped from the stack
  while (!visit_order.empty()) {
    int32_t next_vertex = visit_order.top();
    visit_order.pop();
    if (!visited[next_vertex]) {
      dfs_helper_transpose(transpose_graph, next_vertex, visited);
      std::cout << std::endl; // separates SCCs using a newline
    } // if
  } // while
} // kosaraju()
```

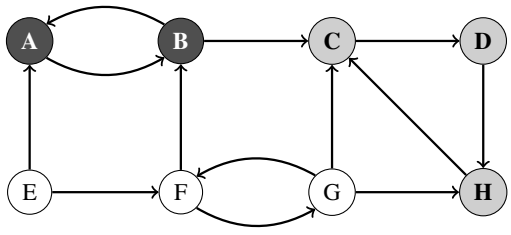
If the above implementation were run using our example graph (where vertex $A = 0$, $B = 1$, …, $H = 7$), we would get the following output that identifies the four strongly connected components in the graph:

```
4

5 6

0 1

2 7 3
```

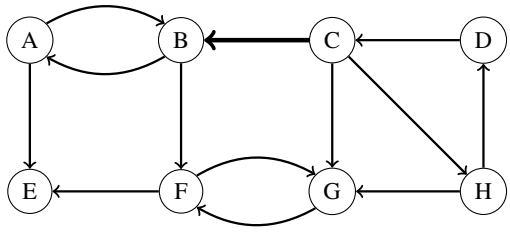
Because Kosaraju's algorithm performs two passes of DFS on the given graph, the time complexity of the algorithm is also determined by the time complexity of a DFS; namely, $\Theta(|V| + |E|)$ on an adjacency list and $\Theta(|V|^2)$ on an adjacency matrix. Although the DFS is performed on two different graphs, the complexity of the first and second passes are identical since the transpose of any graph still contains the same number of vertices and edges.

Kosaraju's Algorithm	Adjacency List	Adjacency Matrix
Time Complexity	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\|^2)$

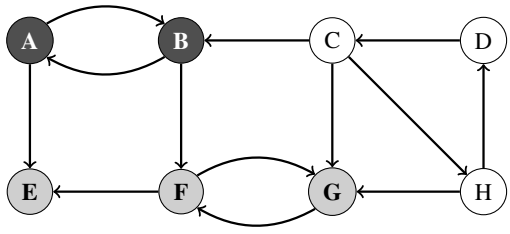
Why does Kosaraju's algorithm work? To understand the intuition behind the algorithm, consider what happens when we perform a DFS on the original graph for the SCC containing vertices A and B. This DFS ends up visiting A and B, but it also visits several other vertices that are not in the same component (C, D, and H). Because of this, the first DFS pass alone is not enough to determine the SCCs of a graph.



However, when we reverse the edges of the graph, we are able to filter out these extraneous vertices. In the transpose graph, the edge between B and C is flipped, making it impossible for us to visit the component containing C, D, and H from the component containing A and B.



However, reversing the graph brings out a new issue: a DFS on the transpose graph starting from the SCC containing A and B also visits vertices we did not in the original graph, namely E, F, and G. However, E, F, and G are not in the same SCC as A and B.



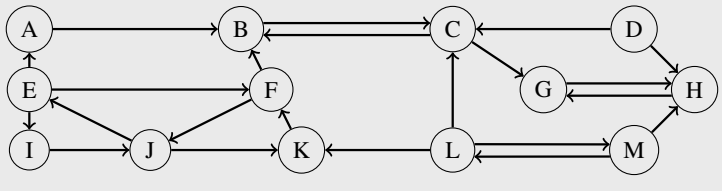
This issue is addressed using the order in which we visit vertices in the transpose graph. Recall that we push each vertex onto a stack *after* we have visited all of its neighbors in the original graph. This means that, for any pair of vertices u and v in the original graph such that there is a path from u to v but not from v to u, we must have pushed u into the stack *after* v. This also means that u is visited *before* v in the transpose graph since the vertices are visited in last in, first out (LIFO) order via the stack. Thus, when we begin a DFS on v in the transpose graph, we do not consider u as part of the same SCC even though there is a path from v to u: this is because u must have already been marked as visited before the DFS on v (and thus is ignored during the DFS on v). As a result, this implies that each iteration of DFS in the transpose graph identifies a distinct SCC in the graph, which in turn also identifies a distinct SCC in the original graph (since any graph has the exact same strongly connected components as its transpose).

In our example above, the ordering of the vertices in the stack ensures that vertices E, F, and G are already visited before we process vertices A and B. Thus, when we process vertices A and B in the transpose graph, we do not end up considering E, F, or G as part of the same SCC; instead, all the vertices encountered during the same iteration of DFS must also be part of the same SCC.

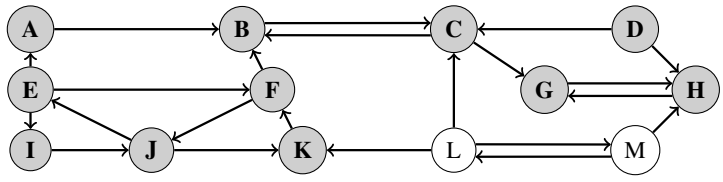
In summary, for any vertex u in the original graph, any vertex v that *is reachable* from u in the original graph but not the other way around is filtered out due to the reversed edges in the transpose graph, and any vertex v that *is able to reach* u in the original graph but not the other way around is filtered out due to the LIFO ordering used to visit vertices in the transpose graph. As a result, if u is able to visit v in both the original graph and the transpose graph, then the two vertices must be part of the same strongly connected component.

Example 19.24 You are running a promotion for an ad campaign, and you want to share a promotion message to a specific group of users on a social media page. Assume that, if a user shares a message, then every one of that user's followers will also share the message to their own followers. Given an adjacency list of users and the people that follow them, implement a function that returns the minimum number of people that you need to reach out to so that your promotion message can be spread across the entire network of users.

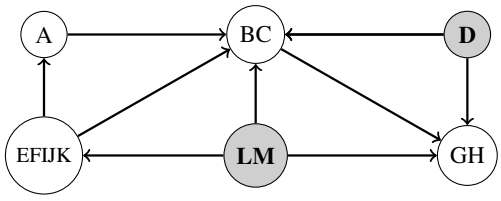
Example: Given the following graph of users, you would return 2, since you would only need to request two users to send your promotion message for the entire network of users to receive it (user D, and one of either user L or M):



At first glance, it may seem that the solution is to send a message to all the users whose in-degree is zero (i.e., vertices in the graph that have no edges directed at it), since these are the users that cannot be reached via a direct connection with another user they follow. However, this approach would fail in the provided example, as user D is the only vertex with an in-degree of zero. If a message were only sent to user D, then users L and M would not be able to get it through the network.



Even though users L and M do not have an in-degree of zero, they are not able to get the message from user D because they are a part a self-contained cycle that is not reachable from any of the other vertices. As a result, we cannot simply consider each user in the graph individually, as users with an in-degree greater than zero may still be disjoint from other users in the network (as was the case with L and M). Instead, a better solution would be to treat the graph as a collection of strongly connected components. Why is this so? Notice that if a single vertex has an in-degree greater than zero, we *cannot* conclude that the vertex is reachable from another arbitrary vertex since the positive in-degree may be caused by the presence of self-contained cycles. However, if a strongly connected component has an in-degree greater than zero, then we *can* conclude that is reachable from another strongly connected component. This is because a graph whose strongly connected components are each condensed into a single vertex cannot have any cycles!



In this condensed graph, only two vertices have an in-degree of zero: D and LM. Since the graph is also a directed acyclic graph, all the other vertices must be reachable from some other strongly connected component in the graph. Therefore, you would only need to request user D and one of either user L or M to send your promotion message for it to eventually reach everyone in the network. From this, we can see that the solution to the problem is to simply condense the users into individual SCCs, and then count the number of SCCs with an in-degree of zero. This can be done using Kosaraju's algorithm, where a possible implementation is shown below:

```
1  void dfs_helper(const std::vector<std::vector<int32_t>>& adj_list, int32_t vertex,
2                  std::vector<bool>& visited, std::stack<int32_t>& visit_order) {
3    visited[vertex] = true;
4    // iterate over all neighbors and begin a DFS if not visited
5    for (int32_t neighbor : adj_list[vertex]) {
6      if (!visited[neighbor]) {
7        dfs_helper(adj_list, neighbor, visited, visit_order);
8      } // if
9    } // for
10   // push the vertices visited into the stack
11   visit_order.push(vertex);
12 } // dfs_helper()
13
14 std::vector<std::vector<int32_t>> get_transpose(const std::vector<std::vector<int32_t>>& adj_list) {
15   std::vector<std::vector<int32_t>> transpose_graph(adj_list.size());
16   for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
17     for (int32_t neighbor : adj_list[vertex]) {
18       transpose_graph[neighbor].push_back(vertex);
19     } // for neighbor
20   } // for vertex
21   return transpose_graph;
22 } // get_transpose()
23
24 void dfs_helper_transpose(const std::vector<std::vector<int32_t>>& transpose_graph, int32_t vertex,
25                           std::vector<int32_t>& representatives, int32_t rep_vertex) {
26   representatives[vertex] = rep_vertex;
27   // iterate over all neighbors and begin a DFS if not visited; since this recurses on all vertices
28   // in the same SCC as the passed in 'vertex', we can set all of them to have a rep of 'vertex'
29   for (int32_t neighbor : transpose_graph[vertex]) {
30     if (representatives[neighbor] == -1) {
31       dfs_helper_transpose(transpose_graph, neighbor, representatives, rep_vertex);
32     } // if
33   } // for
34 } // dfs_helper_transpose()
```

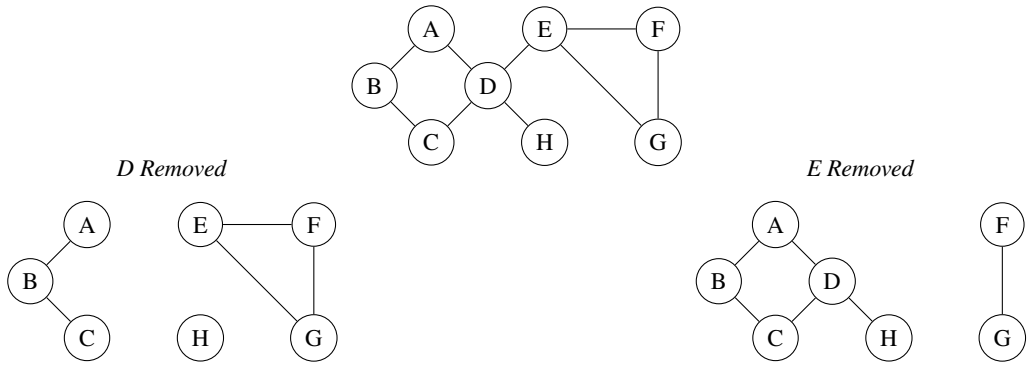
```cpp
int32_t min_users_to_spread_message(const std::vector<std::vector<int32_t>>& adj_list) {
  std::stack<int32_t> visit_order;
  std::vector<bool> visited(adj_list.size(), false);

  // first pass of DFS
  for (size_t vertex = 0; vertex < adj_list.size(); ++vertex) {
    if (!visited[vertex]) {
      dfs_helper(adj_list, vertex, visited, visit_order);
    } // if
  } // for vertex

  // get transpose of graph
  std::vector<std::vector<int32_t>> transpose_graph = get_transpose(adj_list);

  // create vector of representatives to keep track of distinct SCCs
  std::vector<int32_t> representatives(adj_list.size(), -1);

  // second pass of DFS, visit vertices in the order popped from the stack
  while (!visit_order.empty()) {
    int32_t next_vertex = visit_order.top();
    visit_order.pop();
    if (representatives[next_vertex] == -1) {
      // 'next_vertex' is part of an SCC we have not visited yet, so set 'next_vertex' as the
      // representative of every vertex we visit during this recursive call
      dfs_helper_transpose(transpose_graph, next_vertex, representatives, next_vertex);
    } // if
  } // while

  // count in-degrees of condensed SCC graph, then return number of reps with an in-degree of zero
  std::vector<int32_t> in_degrees(adj_list.size(), 0);
  for (int32_t i = 0; i < adj_list.size(); ++i) {
    for (int32_t j : adj_list[i]) {
      if (representatives[i] != representatives[j]) {
        // directed edge from SCC of i to SCC of j, so increment in-degree of SCC of j
        ++in_degrees[representatives[j]];
      } // if
    } // for j
  } // for i

  int32_t count = 0;
  for (size_t k = 0; k < in_degrees.size(); ++k) {
    if (representatives[k] == k && in_degrees[k] == 0) {
      ++count;
    } // if
  } // for k

  return count;
} // min_users_to_spread_message()
```

This solution runs one iteration of Kosaraju's algorithm (which takes $\Theta(|V|+|E|)$ time on an adjacency list), and then loops over the edges to count the in-degrees (which takes $\Theta(|E|)$ time), and then loops over the vertices to determine which components have an in-degree of zero (which takes $\Theta(|V|)$ time). Since the Kosaraju's step has the dominant time complexity, the overall time complexity is $\Theta(|V|+|E|)$.

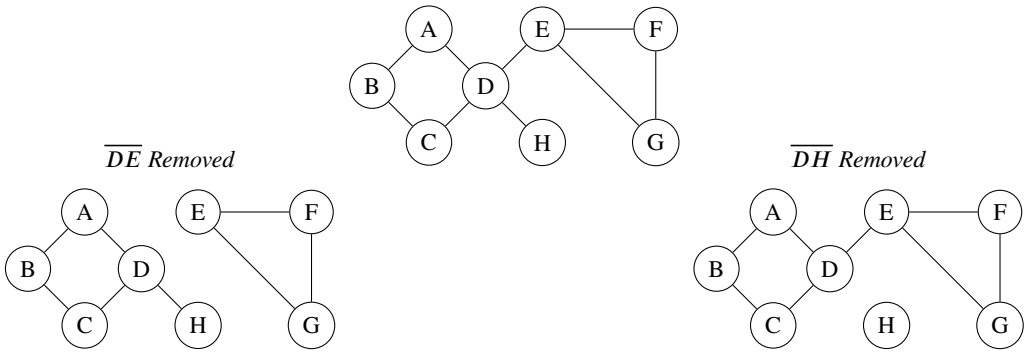
19.8 Algorithms for Articulation Points and Bridges (⁎)

※ 19.8.1 *Articulation Points and Bridges* (⁎)

Articulation points and bridges are two additional concepts that are important when analyzing graphs. An **articulation point** (or *cut vertex*) in a graph is a vertex whose removal would increase the number of connected components in the graph. For instance, vertices D and E are articulation points in the following graph, since removing either of these vertices would disconnect the graph.



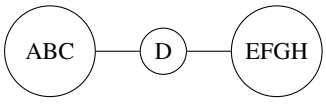
Similarly, a **bridge** (or *cut edge*) in a graph is an edge whose removal would increase the number of connected components in the graph. In the following graph, edges $\overline{DE}$ and $\overline{DH}$ are bridges, since removing either of these edges would disconnect the graph.



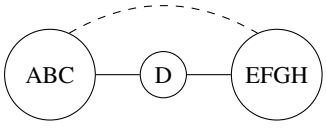
Algorithms that work with articulation points and bridges play an important role in graph theory and have several practical applications in computer science. For example, the ability to identify articulation points and bridges in a network will allow you to determine which vertices and connections are critical to the function of the network and may be vulnerable to failure or disruption. In this section, we will discuss one algorithm that can be used to find articulation points and bridges: *Tarjan's bridge-finding algorithm*.

※ 19.8.2 *Tarjan's Bridge-Finding Algorithm* (✻)

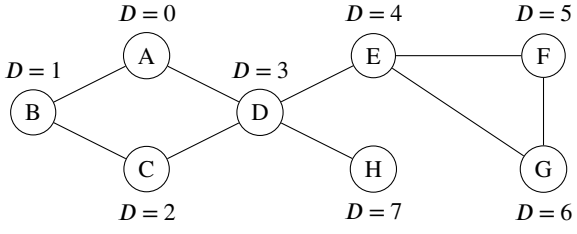
Tarjan's bridge-finding algorithm is an algorithm that can be used to identify articulation points and bridges in a graph in linear time. To understand how Tarjan's bridge-finding algorithm works, let us first look at how we can determine whether a vertex is an articulation point or not. Consider vertex D in our previous example, which we will illustrate as a connection between two disjoint subgraphs: one containing vertices A to C, and one containing vertices E to H:



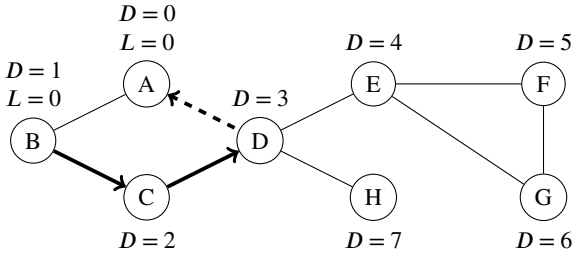
We know that D is an articulation point because there is no edge that connects the subgraph ABC to the subgraph $EFGH$. Had there been an edge between these two subgraphs, then D would not be an articulation point, as the remaining subgraphs would still be connected via that edge.



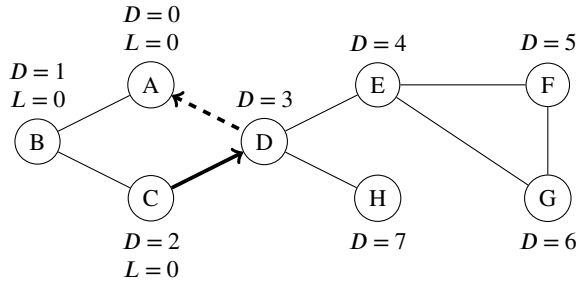
Thus, to identify articulation points, we would need a way to efficiently determine if this condition is met for every vertex in the graph. This forms the basis of Tarjan's bridge-finding algorithm. To start, the algorithm picks an arbitrary vertex (the *root*) and performs a depth-first search on the graph, marking each vertex with a number that indicates the order in which it was discovered (we will denote this number as a vertex's *discovery time*). An example is shown below, where a DFS is initiated at vertex A, and adjacent vertices are visited in alphabetical order.



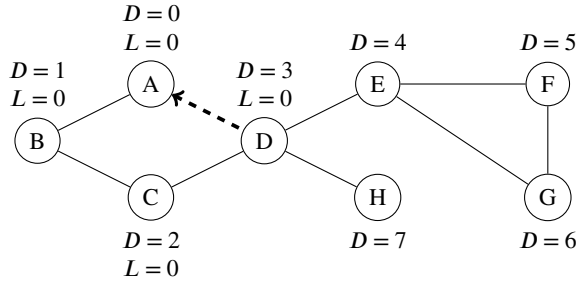
During the DFS, we also keep track of the *low time* of each vertex, which is lowest discovery time that is reachable from that vertex by only traversing at most one *back edge* in the graph. A back edge is an edge in the graph that travels from one vertex to another vertex with a smaller discovery time (with the exception of the direct parent of the vertex, i.e., if you discover vertex B from vertex A during the DFS, then the edge from B directly back to A cannot be considered). In the graph above, the low time of vertex A is 0, since that is trivially the lowest discovery time that is reachable from vertex A (which has a discovery time of 0 itself). The low time of vertex B is also 0, since that is the lowest discovery time reachable from B using only one back edge (depicted using the dashed line).



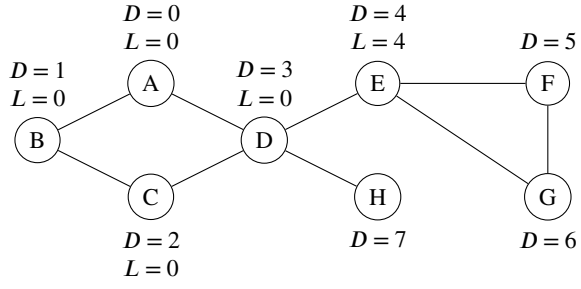
The low time of vertex C is 0, since that is the lowest discovery time reachable from C using only one back edge.



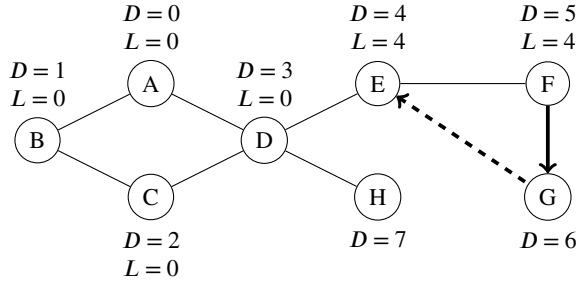
The low time of vertex D is 0, since that is the lowest discovery time reachable from D using only one back edge.



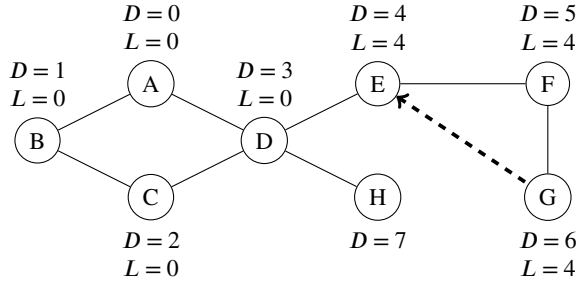
The low time of vertex E is 4, since that is the lowest discovery time reachable from E (note that we cannot travel directly back to D, since D was the direct parent of E during the DFS, and thus $\overline{ED}$ is not considered to be a back edge).



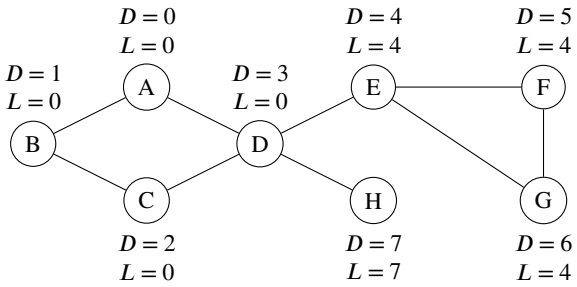
The low time of vertex F is 4, since that is the lowest discovery time reachable from F using only one back edge.



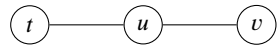
The low time of vertex G is 4, since that is the lowest discovery time reachable from G using only one back edge.



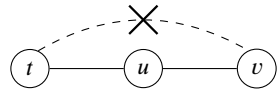
Lastly, the low time of vertex H is 7, since that is the lowest discovery time reachable from H (again, since we discovered H from D during our initial DFS, we cannot count the edge from H back to D as a back edge).



With this information, we are now able to determine the articulation points of the graph. Observe that, for any two adjacent vertices u and v in the graph such that u has a lower discovery time than v (except the special case where u is the root, which we will cover later), then u must be an articulation point if v has a low time that is greater than or equal to u (i.e., $L_v \geq D_u$). We can see why using the same visualization we used earlier. Assuming u is not the root of a DFS, we know that u is connected to two subgraphs — one with vertices with discovery times less than u (denoted as t), and one with vertices with discovery times greater than u (denoted as v):

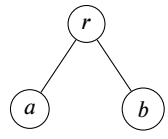


If there exists a vertex in v with a low time that is *not lesser* than the discovery time of u, that means it is impossible for that vertex to reach any vertex in t via a back edge (otherwise, the low time of v would be less than the discovery time of u, since v would be able to reach a vertex with a discovery time less than that of u using that back edge). As a result, we know that vertex u must be an articulation point.



If $L_v \geq D_u$, then there cannot be a back edge from v to t, so u must be an articulation point.

There is one separate case that is not covered by this explanation: what if the vertex u were the root vertex of our depth-first search? Luckily, determining whether the root is an articulation point is relatively straightforward: we just need to check if the root is connected to at least two separate components that are disconnected from each other. If it is, then the root must be an articulation point. For instance, the root vertex r is connected to two subgraphs a and b in the illustration below. If a and b are disjoint from each other, then r must be an articulation point since its removal would disconnect the graph. To keep track of the number of disjoint components connected to the root, we can use a counter that is incremented whenever we encounter a vertex adjacent to the root that has not been visited within any previous component explored during the DFS (see the solution to example 19.12 if you want to review how this would work).



In our initial graph, vertex D has direct connections with E and H, which are both discovered after D during our DFS. If we look at the low times of E and H, we can see that they are both greater than or equal to the discovery time of D. As a result, we can conclude that D is an articulation point, since there must not exist a back edge that connects E and H to any vertex that was discovered before D during our DFS. The same logic applies to vertex E — the adjacent vertices F and G are discovered after E and both have a low time greater than or equal to the discovery time of E, so E is also an articulation point, as there must be no back edge that connects F and G with vertices discovered before E.

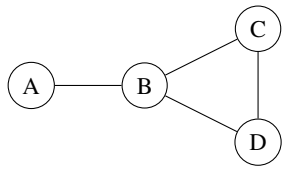
To summarize, we can use Tarjan's bridge-finding algorithm to identify articulation points in a graph by following these steps:

1. Select an arbitrary vertex (which becomes the root) and perform a depth-first search starting from this root. Keep track of a counter that indicates the number of disjoint components that are connected to the root.
2. During the depth-first search, assign each visited vertex with the following information:
   - A *discovery time*, which indicates the order in which the vertices of the graph are visited during the depth-first search.
   - A *low time*, which indicates the lowest discovery time that is reachable from that vertex using at most one back edge.
3. If a vertex u is encountered such that there is a vertex v with a larger discovery time where the low time of v is greater than or equal to the discovery time of u, then u must be an articulation point.
4. After the DFS is complete, check if the root is connected to more than one disjoint component. If so, then the root is also an articulation point in the graph. At this point, all articulation points will have been discovered.

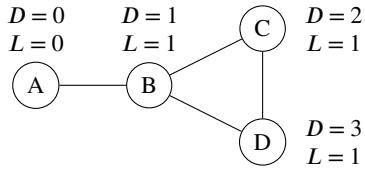
What is the time complexity of this algorithm? Notice that we are simply performing a depth-first search with additional steps to keep track of discovery times and low times (both of which can be computed in constant time during the DFS). Because of this, the time complexity of Tarjan's bridge-finding algorithm for finding articulation points is the same as that of a depth-first search: $\Theta(|V| + |E|)$ on an adjacency list, and $\Theta(|V|^2)$ on an adjacency matrix.

Now, what if you want to find bridges instead of articulation points? The algorithm would be pretty much the same, with one notable exception: an edge $\overline{uv}$ is a bridge only if the low time of v is *strictly* greater than the discovery time of u (i.e., $L_v > D_u$).

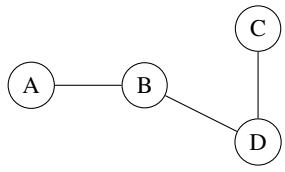
Why does the equal case no longer apply? If the low time of v were equal to the discovery time of u, and we were working with articulation points, we could ensure that removing u would also break the the graph into additional components since v cannot reach anything before u. However, when working with bridges, removing $\overline{uv}$ is not guaranteed to do the same if the edge were part of a cycle. Consider the graph below:



If we perform our depth-first search starting from vertex A, we would get the following discovery times and low times:

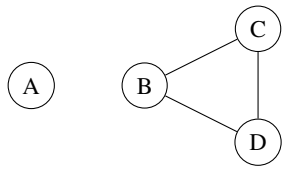


Consider the edge $\overline{BC}$. In this case, $L_C = D_B$, which indicates that vertex B is an articulation point (since the condition to satisfy is $L_C \geq D_B$). However, $\overline{BC}$ is not a bridge, which would require L_C to be strictly greater than D_B.



Removing $\overline{BC}$ does not disconnect the graph

Now, consider the edge $\overline{AB}$. In this case, $L_B > D_A$, which indicates that $\overline{AB}$ is a bridge in the graph.



Removing $\overline{AB}$ disconnects the graph

To summarize, we can use Tarjan's bridge-finding algorithm to identify bridges in a graph by following these steps:

1. Select an arbitrary vertex (which becomes the root) and perform a depth-first search starting from this root. (Since we want to identify bridges, we do not need to count the number of disjoint components connected to the root, since the edges connected to the root are already handled by the following procedure.)
2. During the depth-first search, assign each visited vertex with the following information:
   - A *discovery time*, which indicates the order in which the vertices of the graph are visited during the depth-first search.
   - A *low time*, which indicates the lowest discovery time that is reachable from that vertex using at most one back edge.
3. If an edge $\overline{uv}$ is encountered such that the low time of v is strictly greater than the discovery time of u, then $\overline{uv}$ must be a bridge.
4. After the DFS is complete, all bridges will have been discovered.

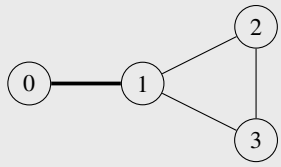
Like before, the time complexity of this algorithm is the same as that of a depth-first search: $\Theta(|V| + |E|)$ on an adjacency list, and $\Theta(|V|^2)$ on an adjacency matrix.

Example 19.25 There are a total of n servers in a room, numbered from 0 to $n-1$. These servers are connected by undirected server-to-server connections that form a network; any server can reach another server in the room directly or indirectly through the network.

You are given a vector of connections, such that `[a, b]` represents a connection from server a to server b. Return all the *critical connections* in the network, or connections that, when removed, would prevent some servers from reaching other servers in the original network.

```
std::vector<std::vector<int32_t>> critical_connections(int32_t n, const std::vector<std::vector<int32_t>>& connections);
```

Example: Given $n = 4$ servers and the following connections: `[[0, 1], [1, 2], [2, 3], [1, 3]]`, you would return `[[0, 1]]`, since the edge from server 0 to server 1 is a critical connection.



This is a bridge-finding problem, so we can use Tarjan's bridge-finding algorithm. First, we will need to convert our input into an adjacency list. Then, we will apply the algorithm discussed above, adding each encountered bridge to a container that is returned. One possible implementation of the solution is shown below:

```
1  void dfs(int32_t curr, int32_t parent, const std::vector<std::vector<int32_t>>& adj_list,
2           std::vector<int32_t>& discovery_times, std::vector<int32_t>& low_times, int32_t& time,
3           std::vector<std::vector<int32_t>>& bridges) {
4    low_times[curr] = discovery_times[curr] = ++time;
5    for (int32_t neighbor : adj_list[curr]) {
6      if (neighbor == parent) {
7        continue;
8      } // if
9      if (discovery_times[neighbor] == 0) {
10       dfs(neighbor, curr, adj_list, discovery_times, low_times, time, bridges);
11       if (low_times[neighbor] > discovery_times[curr]) {
12         bridges.push_back({curr, neighbor});
13       } // if
14       low_times[curr] = std::min(low_times[curr], low_times[neighbor]);
15     } // if
16     else {
17       low_times[curr] = std::min(low_times[curr], discovery_times[neighbor]);
18     } // else
19   } // for neighbor
20 } // dfs()
21
22 std::vector<std::vector<int32_t>> critical_connections(
23     int32_t n, const std::vector<std::vector<int32_t>>& connections) {
24   std::vector<std::vector<int32_t>> adj_list(n);
25   for (const auto& connection : connections) {
26     int32_t u = connection[0];
27     int32_t v = connection[1];
28     adj_list[u].push_back(v);
29     adj_list[v].push_back(u);
30   } // for connection
31
32   int32_t time = 0;
33   std::vector<int32_t> discovery_times(n, 0);
34   std::vector<int32_t> low_times(n, 0);
35   std::vector<std::vector<int32_t>> bridges;
36
37   for (int32_t i = 0; i < n; ++i) {
38     if (discovery_times[i] == 0) {
39       dfs(i, i, adj_list, discovery_times, low_times, time, bridges);
40     } // if
41   } // for i
42
43   return bridges;
44 } // critical_connections()
```

The time complexity of this solution is $\Theta(|V| + |E|)$, for the number of vertices $|V|$ and edges $|E|$ in the input graph.