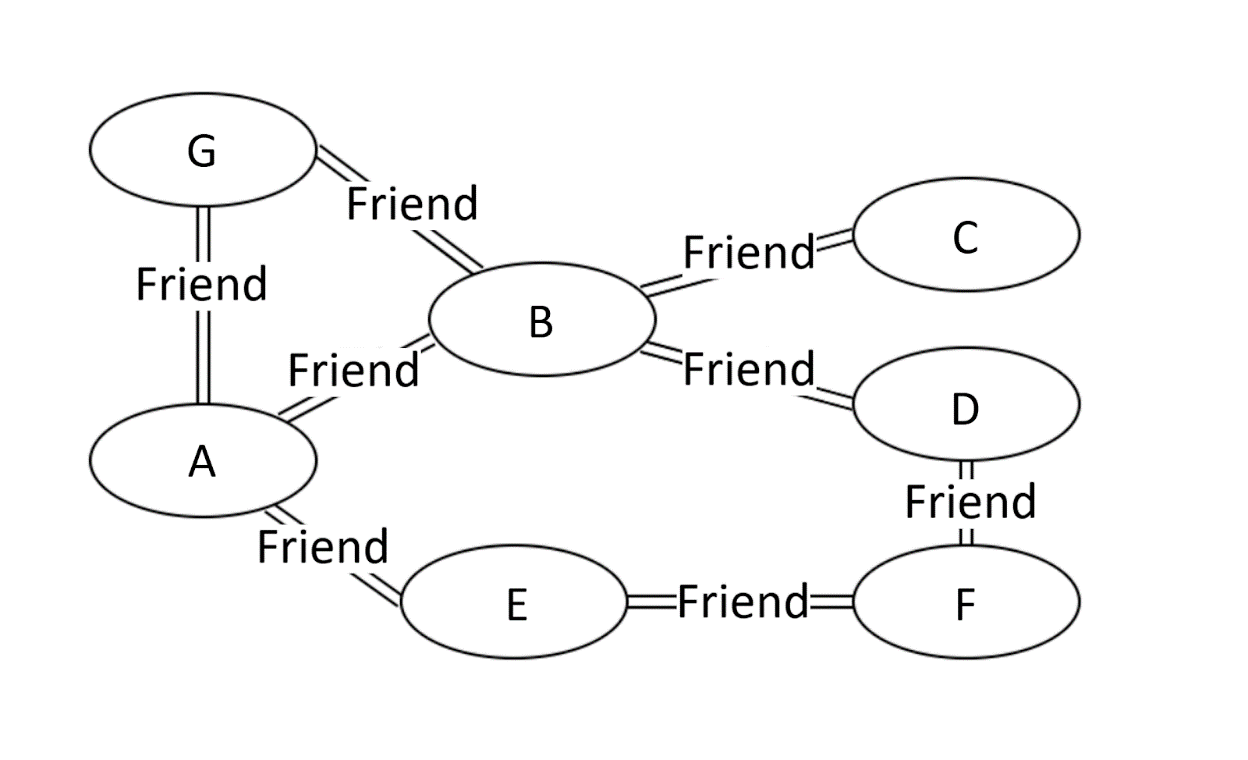
Graphs

# **Introduction**

The content on this card is exclusive to our premium users. If you like the content after previewing the first chapter for free, please subscribe to LeetCode premium here.

Graph is probably the data structure that has the closest resemblance to our daily life. There are many types of graphs describing the relationships in real life. For instance, our friend circle is a huge “graph”.



**Figure 1. An example of a undirected graph.**

In Figure 1 above, we can see that person G, B, and E are all direct friends of A, while person C, D, and F are indirect friends of A. This example is a social graph of friendship. So, what is the “graph” data structure?

## **Types of “graphs”**

There are many types of “graphs”. In this Explore Card, we will introduce three types of graphs: undirected graphs, directed graphs, and weighted graphs.

### **Undirected graphs**

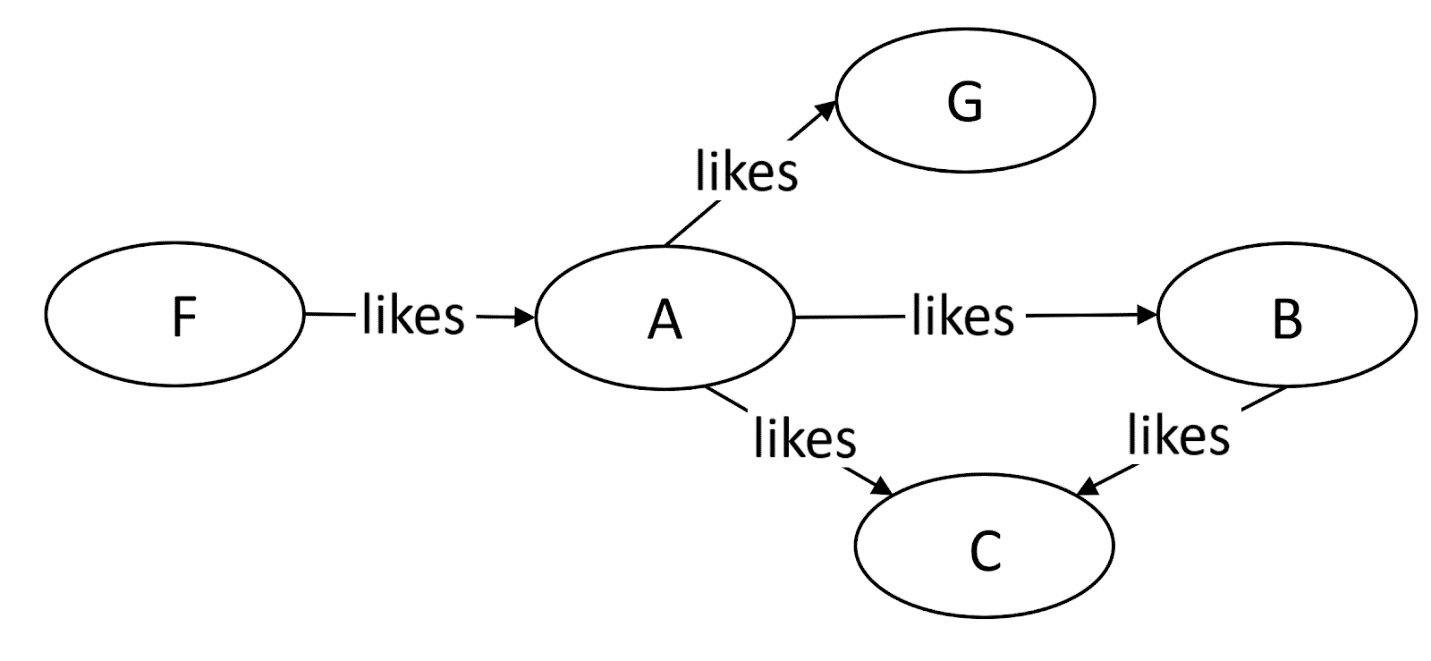
The edges between any two vertices in an “undirected graph” do not have a direction, indicating a two-way relationship.

Figure 1 is an example of an undirected graph.

### **Directed graphs**

The edges between any two vertices in a “directed graph” graph are directional.

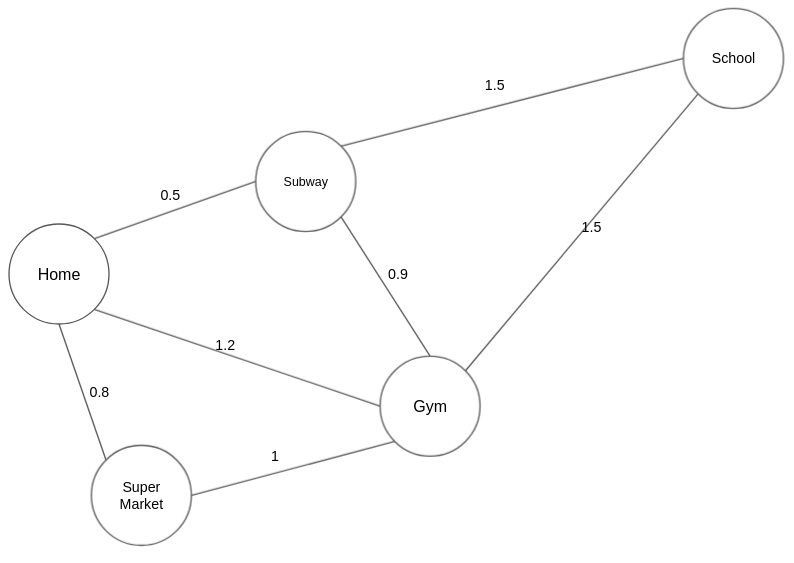
Figure 2 is an example of a directed graph.



**Figure 2. An example of a directed graph.**

### **Weighted graphs**

Each edge in a “weighted graph” has an associated weight. The weight can be of any metric, such as time, distance, size, etc. The most commonly seen “weighted map” in our daily life might be a city map. In Figure 3, each edge is marked with the distance, which can be regarded as the weight of that edge.



**Figure 3. An example of a weighted graph.**

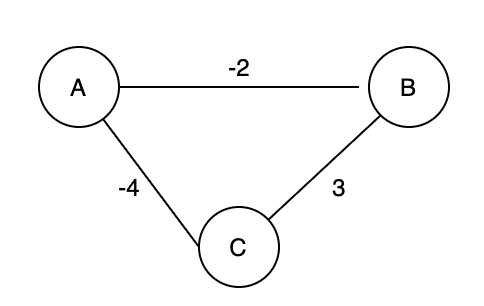
### **The Definition of “graph” and Terminologies**

“Graph” is a non-linear data structure consisting of vertices and edges. There are a lot of terminologies to describe a graph. If you encounter an unfamiliar term in the following Explore Card, you may look up the definition below.

* Vertex: In Figure 1, nodes such as A, B, and C are called vertices of the graph.
* Edge: The connection between two vertices are the edges of the graph. In Figure 1, the connection between person A and B is an edge of the graph.
* Path: the sequence of vertices to go through from one vertex to another. In Figure 1, a path from A to C is [A, B, C], or [A, G, B, C], or [A, E, F, D, B, C].

\*\*Note\*\*: there can be multiple paths between two vertices.

* Path Length: the number of edges in a path. In Figure 1, the path lengths from person A to C are 2, 3, and 5, respectively.
* Cycle: a path where the starting point and endpoint are the same vertex. In Figure 1, [A, B, D, F, E] forms a cycle. Similarly, [A, G, B] forms another cycle.
* Negative Weight Cycle: In a “weighted graph”, if the sum of the weights of all edges of a cycle is a negative value, it is a negative weight cycle. In Figure 4, the sum of weights is -3.
* Connectivity: if there exists at least one path between two vertices, these two vertices are connected. In Figure 1, A and C are connected because there is at least one path connecting them.
* Degree of a Vertex: the term “degree” applies to unweighted graphs. The degree of a vertex is the number of edges connecting the vertex. In Figure 1, the degree of vertex A is 3 because three edges are connecting it.
* In-Degree: “in-degree” is a concept in directed graphs. If the in-degree of a vertex is d, there are d directional edges incident to the vertex. In Figure 2, A’s indegree is 1, i.e., the edge from F to A.
* Out-Degree: “out-degree” is a concept in directed graphs. If the out-degree of a vertex is d, there are d edges incident from the vertex. In Figure 2, A’s outdegree is 3, i,e, the edges A to B, A to C, and A to G.



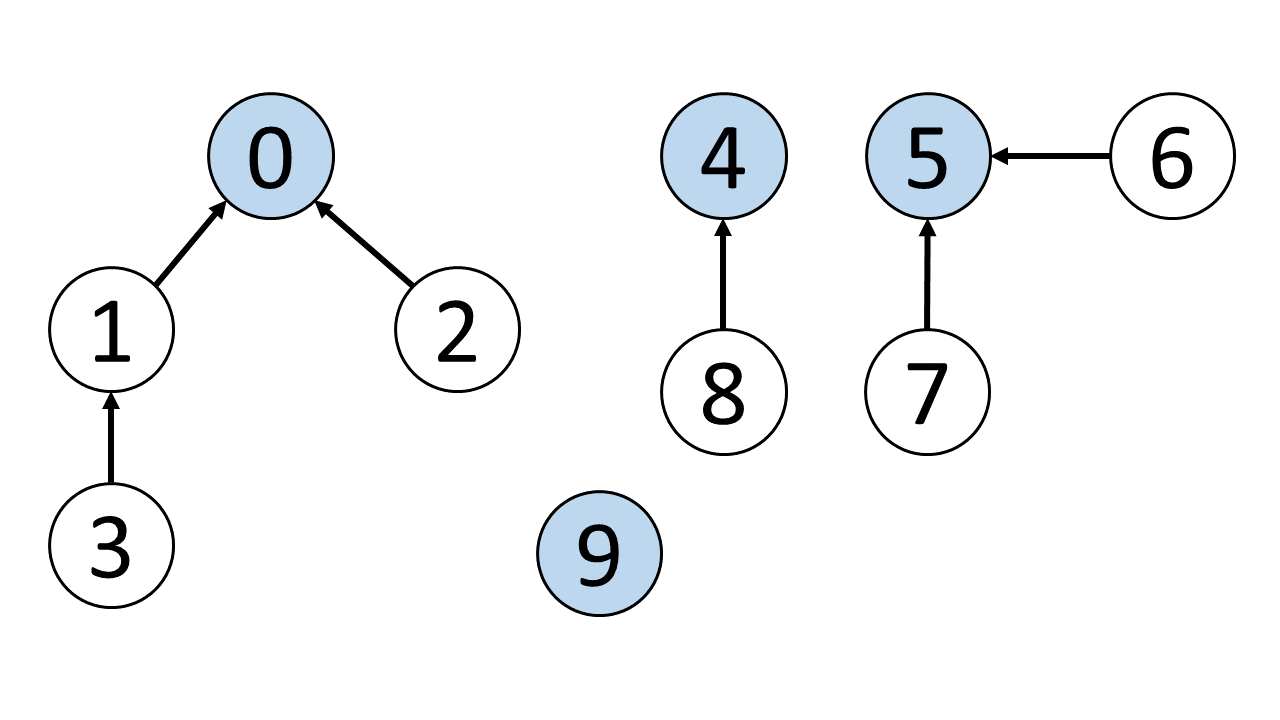
**Figure 4. An example of a negative weight cycle.**

After understanding the basics of “graph”, let’s start our journey on learning data structures and algorithms related to “graph”.

# **Disjoint Set**

## **Overview of Disjoint Set**

Given the vertices and edges between them, how could we quickly check whether two vertices are connected? For example, Figure 5 shows the edges between vertices, so how can we efficiently check if 0 is connected to 3, 1 is connected to 5, or 7 is connected to 8? We can do so by using the “disjoint set” data structure, also known as the “union-find” data structure. Note that others might refer to it as an algorithm. In this Explore Card, the term “disjoint set” refers to a data structure.



**Figure 5. Each graph consists of vertices and edges. The root vertices are in blue.**

The primary use of disjoint sets is to address the connectivity between the components of a network. The “network“ here can be a computer network or a social network. For instance, we can use a disjoint set to determine if two people share a common ancestor.

### **Terminologies**

* Parent node: the direct parent node of a vertex. For example, in Figure 5, the parent node of vertex 3 is 1, the parent node of vertex 2 is 0, and the parent node of vertex 9 is 9.
* Root node: a node without a parent node; it can be viewed as the parent node of itself. For example, in Figure 5, the root node of vertices 3 and 2 is 0. As for 0, it is its own root node and parent node. Likewise, the root node and parent node of vertex 9 is 9 itself. Sometimes the root node is referred to as the head node.

**The two important functions of a “disjoint set.”**

* The find function finds the root node of a given vertex. For example, in Figure 5, the output of the find function for vertex 3 is 0.
* The union function unions two vertices and makes their root nodes the same. In Figure 5, if we union vertex 4 and vertex 5, their root node will become the same, which means the union function will modify the root node of vertex 4 or vertex 5 to the same root node.

### **There are two ways to implement a “disjoint set”.**

* Implementation with Quick Find: in this case, the time complexity of the find function will be O(1). However, the union function will take more time with the time complexity of O(N).
* Implementation with Quick Union: compared with the Quick Find implementation, the time complexity of the union function is better. Meanwhile, the find function will take more time in this case.

Next, we will learn these two implementations and two common strategies to optimize a disjoint set.

## **Quick Find - Disjoint Set**

### **Algorithm**

***# UnionFind class***

**class** **UnionFind**:

**def** \_\_init\_\_(self, size):

        self.root = [i for i in range(size)]

**def** **find**(self, x):

        return self.root[x]

**def** **union**(self, x, y):

        rootX = self.find(x)

        rootY = self.find(y)

        if rootX != rootY:

            for i in range(len(self.root)):

                if self.root[i] == rootY:

                    self.root[i] = rootX

**def** **connected**(self, x, y):

        return self.find(x) == self.find(y)

***# Test Case***

uf = UnionFind(**10**)

***# 1-2-5-6-7 3-8-9 4***

uf.union(**1**, **2**)

uf.union(**2**, **5**)

uf.union(**5**, **6**)

uf.union(**6**, **7**)

uf.union(**3**, **8**)

uf.union(**8**, **9**)

print(uf.connected(**1**, **5**))  ***# true***

print(uf.connected(**5**, **7**))  ***# true***

print(uf.connected(**4**, **9**))  ***# false***

***# 1-2-5-6-7 3-8-9-4***

uf.union(**9**, **4**)

print(uf.connected(**4**, **9**))  ***# true***

### **Time Complexity**

|  | **Union-find Constructor** | **Find** | **Union** | **Connected** |
| --- | --- | --- | --- | --- |
| **Time Complexity** | *O*(*N*) | *O*(1) | *O*(*N*) | *O*(1) |

Note: **N** is the number of vertices in the graph.

* When initializing a union-find constructor, we need to create an array of size NN with the values equal to the corresponding array indices; this requires linear time.
* Each call to find will require O(1) time since we are just accessing an element of the array at the given index.
* Each call to union will require O(N) time because we need to traverse through the entire array and update the root vertices for all the vertices of the set that is going to be merged into another set.
* The connected operation takes O(1) time since it involves the two find calls and the equality check operation.

### **Space Complexity**

We need O(N) space to store the array of size N.

## **Quick Union - Disjoint Set**

### **Algorithm**

**class** **UnionFind**:

**def** \_\_init\_\_(self, size):

        self.root = [i for i in range(size)]

**def** **find**(self, x):

        while x != self.root[x]:

            x = self.root[x]

        return x

**def** **union**(self, x, y):

        rootX = self.find(x)

        rootY = self.find(y)

        if rootX != rootY:

            self.root[rootY] = rootX

**def** **connected**(self, x, y):

        return self.find(x) == self.find(y)

***# Test Case***

uf = UnionFind(**10**)

***# 1-2-5-6-7 3-8-9 4***

uf.union(**1**, **2**)

uf.union(**2**, **5**)

uf.union(**3**, **8**)

uf.union(**8**, **9**)

print(uf.connected(**1**, **5**))  ***# true***

### **Time Complexity**

|  | **Union-find Constructor** | **Find** | **Union** | **Connected** |
| --- | --- | --- | --- | --- |
| **Time Complexity** | O(N) | O(N) | O(N) | O(N) |

**Note**: N is the number of vertices in the graph. In the worst-case scenario, the number of operations to get the root vertex will be H where H is the height of the tree. Because this implementation does not always point the root of the shorter tree to the root of the taller tree, H can be at most N when the tree forms a linked list.

* The same as in the quick find implementation, when initializing a union-find constructor, we need to create an array of size NN with the values equal to the corresponding array indices; this requires linear time.
* For the find operation, in the worst-case scenario, we need to traverse every vertex to find the root for the input vertex. The maximum number of operations to get the root vertex would be no more than the tree's height, so it will take O(N) time.
* The union operation consists of two find operations which (only in the worst-case) will take O(N) time, and two constant time operations, including the equality check and updating the array value at a given index. Therefore, the union operation also costs O(N) in the worst-case.
* The connected operation also takes O(N) time in the worst-case since it involves two find calls.

### **Space Complexity**

We need O(N) space to store the array of size N.

## **Optimized “disjoint set” with Path Compression and Union by Rank**

### **Implementation**

***# UnionFind class***

**class** **UnionFind**:

**def** \_\_init\_\_(self, size):

        self.root = [i for i in range(size)]

***# Use a rank array to record the height of each vertex, i.e., the "rank" of each vertex.***

***# The initial "rank" of each vertex is 1, because each of them is***

***# a standalone vertex with no connection to other vertices.***

        self.rank = [**1**] \* size

***# The find function here is the same as that in the disjoint set with path compression.***

**def** **find**(self, x):

        if x == self.root[x]:

            return x

        self.root[x] = self.find(self.root[x])

        return self.root[x]

***# The union function with union by rank***

**def** **union**(self, x, y):

        rootX = self.find(x)

        rootY = self.find(y)

        if rootX != rootY:

            if self.rank[rootX] > self.rank[rootY]:

                self.root[rootY] = rootX

            elif self.rank[rootX] < self.rank[rootY]:

                self.root[rootX] = rootY

            else:

                self.root[rootY] = rootX

                self.rank[rootX] += **1**

**def** **connected**(self, x, y):

        return self.find(x) == self.find(y)

***# Test Case***

uf = UnionFind(**10**)

***# 1-2-5-6-7 3-8-9 4***

uf.union(**1**, **2**)

uf.union(**2**, **5**)

uf.union(**5**, **6**)

uf.union(**6**, **7**)

uf.union(**3**, **8**)

uf.union(**8**, **9**)

print(uf.connected(**1**, **5**))  ***# true***

print(uf.connected(**5**, **7**))  ***# true***

print(uf.connected(**4**, **9**))  ***# false***

***# 1-2-5-6-7 3-8-9-4***

uf.union(**9**, **4**)

print(uf.connected(**4**, **9**))  ***# true***

### **Time Complexity**

|  | **Union-find Constructor** | **Find** | **Union** | **Connected** |
| --- | --- | --- | --- | --- |
| **Time Complexity** | O(N) | *O*(*α*(*N*)) | *O*(*α*(*N*)) | *O*(*α*(*N*)) |

**Note**: **N** is the number of vertices in the graph. **α** refers to the Inverse Ackermann function. In practice, we assume it's a constant. In other words, O(α(N)) is regarded as O(1) on average.

For the union-find constructor, we need to create two arrays of size NN each.

When using the combination of union by rank and the path compression optimization, the find operation will take O(α(N)) time on average. Since union and connected both make calls to find and all other operations require constant time, union and connected functions will also take O(α(N)) time on average.

### **Space Complexity**

We need O(N) space to store the array of size N.

## **Summary of the “disjoint set” data structure**

The main idea of a “disjoint set” is to have all connected vertices have the same parent node or root node, whether directly or indirectly connected. To check if two vertices are connected, we only need to check if they have the same root node.

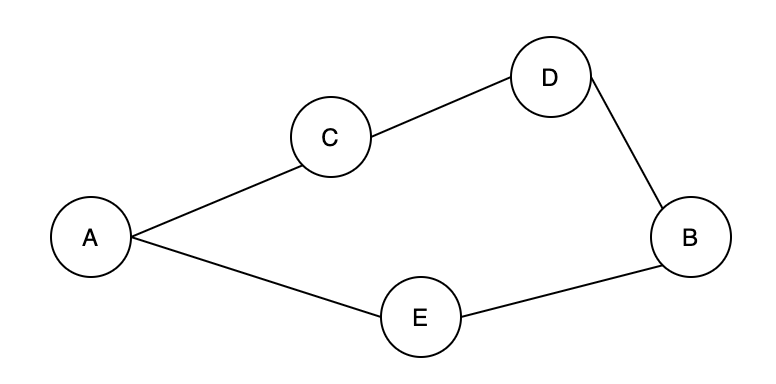
The two most important functions for the “disjoint set” data structure are the find function and the union function. The find function locates the root node of a given vertex. The union function connects two previously unconnected vertices by giving them the same root node. There is another important function named connected, which checks the “connectivity” of two vertices. The find and union functions are essential for any question that uses the “disjoint set” data structure.

# **The Depth-First Search Algorithm**

## **Overview of Depth-First Search Algorithm**

Previously, we learned how to check the connectivity between two vertices with the “disjoint set” data structure. Now, let's switch gears and consider: Given a graph, how can we find all of its vertices, and how can we find all paths between two vertices?

The depth-first search algorithm is ideal in solving these kinds of problems because it can explore all paths from the start vertex to all other vertices. Let's start by considering an example. In Figure 7, there are five vertices [A, C, D, B, E]. Given two vertices A and B, there are two paths between them. One path is [A, C, D, B], and the other is [A, E, B].



**Figure 7. An undirected graph**

In Graph theory, the depth-first search algorithm (abbreviated as DFS) is mainly used to:

1. Traverse all vertices in a “graph”;
2. Traverse all paths between any two vertices in a “graph”.

## **Traversing all Vertices – Depth-First Search Algorithm**

### **Complexity Analysis**

* Time Complexity: *O*(*V*+*E*). Here, *V* represents the number of vertices, and *E* represents the number of edges. We need to check every vertex and traverse through every edge in the graph.
* Space Complexity: *O*(*V*). Either the manually created stack or the recursive call stack can store up to *V* vertices.

### **Traversing all paths between two vertices – Depth-First Search Algorithm**

### **Complexity Analysis**

* **Time Complexity:** O((V−1)!) The above example is for an undirected graph. The worst-case scenario, when trying to find all paths, is a complete graph. A complete graph is a graph where every vertex is connected to every other vertex.

In a complete graph, there will be V−1 unique paths of length one that start at the source vertex; one of these paths will go to the target and end. Each of the remaining paths will have V−2 unique paths that extend from it (since none of them will go back to the source vertex which was already visited). This process will continue and lead to approximately (V−1)! total paths. Remember, once a path reaches the target vertex, it ends, so the total number of paths will be less than (V−1)!.

The precise total number of paths in the worst-case scenario is equivalent to the **Number of Arrangements** of the subset of vertices excluding the source and target node, which equals e⋅(V−2)!.

While finding all paths, at each iteration, we add all valid paths from the current vertex to the stack, as shown in the video. Each time we add a path to the stack requires O(V) time to create a copy of the current path, append a vertex to it, and push it onto the stack. Since the path grows by one vertex each time, a path of length V must have been copied and pushed onto the stack V times before reaching its current length. Therefore, it is intuitive to think that each path should require O(V^2) time in total. However, there is a flaw in our logic. Consider the example above; at 2:50 we add ADE to the stack. Then at 3:20, we add ADEC, ADEB, and ADEF to the stack. ADE is a subpath of ADEC, ADEB, and ADEF, but ADE was only created once. So the time required for each path to create ADE can be thought of as O(V) divided by the number of paths that stem from ADE. With this in mind, the time spent to create a path is V plus V−1 divided by the number of paths that stem from this subpath plus V−2 times... For a complete graph with many nodes, this averages out to O(2⋅V)=O(V) time per path.

Thus, the time complexity to find all paths in an undirected graph in the worst-case scenario is equal to the number of paths found O((V−2)!) times the average time to find a path O(V) which simplifies to O((V−1)!).

**Space Complexity:** O(V^3)

The space used is by the stack which will contain:

* (V−1) paths after adding first V −1 paths to the stack.
* (V - 1) - 1 + (V - 2) paths after popping one path and adding second set of paths.
* (V - 1) - 1 + (V - 2) - 1 + (V - 3) - 1 + ...
* ≈V⋅(V−1)/2+1 paths will be at most in the stack, and each path added to the stack will take O(V) space.

Therefore, in total, this solution will require = O(V⋅(V−1)/2+1)⋅V = O(V ^3) space. Note that the space used to store the result does not count towards the space complexity.

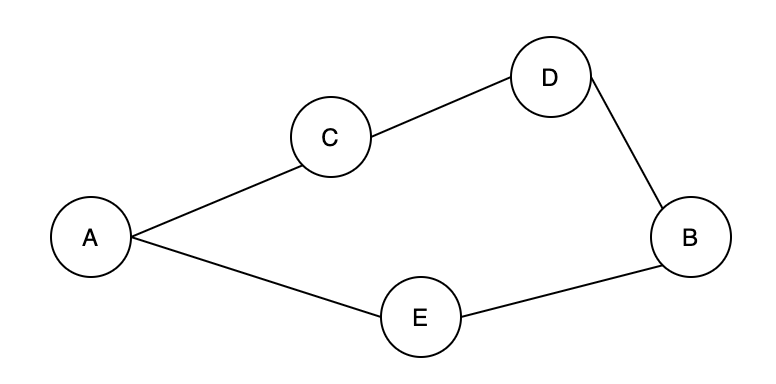
# **The Breadth-First Search Algorithm**

## **Overview of Breadth-First Search Algorithm**

Previously, we discussed the “depth-first search” algorithm. This section will talk about a closely related and equally popular algorithm called “breadth-first search”. Similarly, the “breadth-first search” algorithm can traverse all vertices of a “graph” and traverse all paths between two vertices. However, the most advantageous use case of “breadth-first search” is to efficiently find the shortest path between two vertices in a “graph” where **all edges have equal and positive weights**.

Although the “depth-first search” algorithm can find the shortest path between two vertices in a “graph” with equal and positive weights, it must traverse all paths between two vertices before finding the shortest one. The “breadth-first search” algorithm, in most cases, can find the shortest path without traversing all paths. This is because when using "breadth-first search", as soon as a path between the source vertex and target vertex is found, it is guaranteed to be the shortest path between the two nodes.

In Figure 8, the vertices are [A, C, D, B, E]. Given vertices A and B, there are two paths between them. One path is [A, C, D, B], and the other is [A, E, B]. Obviously, [A, E, B] is the shortest path between A and B.



**Figure 8. An undirected graph**

In Graph theory, the primary use cases of the “breadth-first search” (“BFS”) algorithm are:

1. Traversing all vertices in the “graph”;
2. Finding the shortest path between two vertices in a graph where **all edges have equal and positive weights**.

## **Traversing all Vertices - Breadth-First Search**

### **Complexity Analysis**

**Time Complexity:** O(V+E). Here, VV represents the number of vertices, and E represents the number of edges. We need to check every vertex and traverse through every edge in the graph. The time complexity is the same as it was for the DFS approach.

**Space Complexity:** O(V). Generally, we will check if a vertex has been visited before adding it to the queue, so the queue will use at most O(V) space. Keeping track of which vertices have been visited will also require O(V) space.

## **Shortest Path Between Two Vertices - Breadth-First Search**

### **Complexity Analysis**

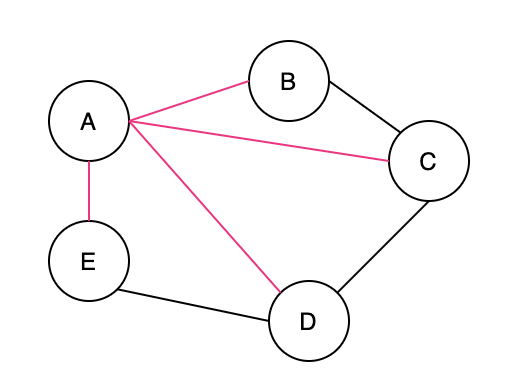
**Time Complexity:** O(V+E). Here, VV represents the number of vertices, and E represents the number of edges. In the worst case, when the distance between the starting vertex and the target vertex is the maximum possible, we need to check every vertex and traverse through every edge in the graph.

**Space Complexity:** O(V). The queue will take up to O(V) space to store all the graph vertices in the worst-case scenario. We must also use O(V) space to keep track of which vertices have been visited.

# **Algorithms to Construct Minimum Spanning Tree**

## **Overview of Minimum Spanning Tree**

You might wonder: what is a spanning tree? A **spanning tree** is a connected subgraph in an undirected graph where **all vertices** are connected with the **minimum number** of edges. In Figure 9, all pink edges [(A, B), (A, C), (A, D), (A, E)] form a tree, which is a spanning tree of this undirected graph. Note that [(A, E), (A, B), (B, C), (C, D)] is also a spanning tree of the undirected graph. Thus, an “undirected graph” can have multiple spanning trees.



**Figure 9. Spanning tree**

After learning what a spanning tree is, you might have another question: what is a **minimum spanning tree**? A minimum spanning tree is a spanning tree with the minimum possible total edge weight in a “weighted undirected graph”. In Figure 10, a spanning tree formed by green edges [(A, E), (A, B), (B, C), (C, D)] is one of the minimum spanning trees in this weighted undirected graph. Actually, [(A, E), (E, D), (A, B), (B, C)] forms another minimum spanning tree of the weighted undirected graph. Thus, a “weighted undirected graph” can have multiple minimum spanning trees.

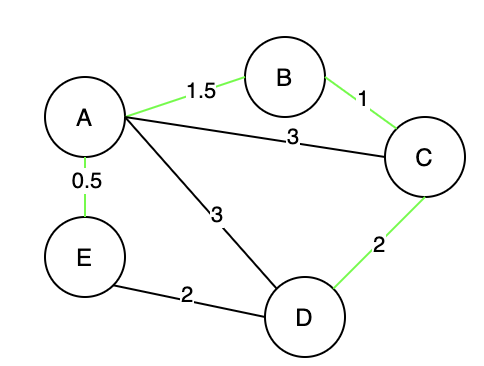


Figure 10. Minimum spanning tree

In this chapter, we will learn about the “cut property and two algorithms for constructing a “minimum spanning tree”:

* **Kruskal’s Algorithm**
* **Prim’s algorithm**

## **Cut Property**

What is a “cut”? Although many theorems are named after people’s names, “cut” is not one of them. To understand the “cut property”, we need to understand two basic concepts.

* First, in Graph theory, a “cut” is a partition of vertices in a “graph” into two disjoint subsets. Figure 11 illustrates a “cut”, where (B, A, E) forms one subset, and (C, D) forms the other subset.
* Second, a crossing edge is an edge that connects a vertex in one set with a vertex in the other set. In Figure 11, (B, C), (A, C), (A, D), (E, D) are all “crossing edges”.

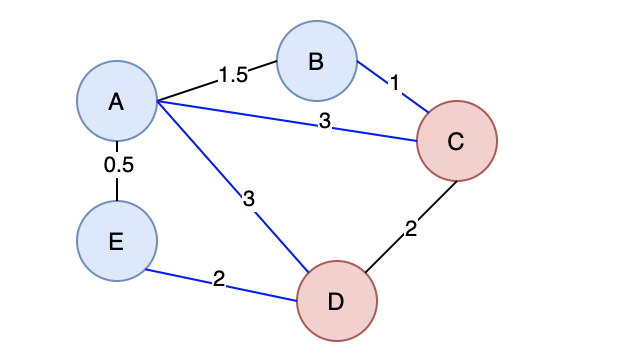


Figure 11. Graph with a cut

After knowing the basics of a graph cut, let’s delve into the “cut property”. The cut property provides theoretical support for Kruskal’s algorithm and Prim’s algorithm. So, what is the “cut property”? According to [**Wikipedia**](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cut_property), the “cut property” refers to:

For any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph.

## **Kruskal’s Algorithm**

“Kruskal’s algorithm” is an algorithm to construct a “minimum spanning tree” of a “weighted undirected graph”. We need to choose exactly **N−1** edges of the graph with N vertices in total to construct a “minimum spanning tree” of that graph. The greedy approach does work in accomplishing our task.

### **Complexity Analysis**

* **Time Complexity:** O(*E*⋅log*E*). Here, *E* represents the number of edges.
  + At first, we need to sort all the edges of the graph in ascending order. Sorting will take *O*(*E*log*E*) time.
  + Next, we can start building our minimum spanning tree by selecting which edges should be included. For each edge, we will look at whether both of the vertices of the edge belong to the same connected component; which is an *O*(*α*(*V*)) operation, where \alpha*α* refers to the Inverse Ackermann function. In the worst case, the tree will not be complete until we reach the very last edge (the edge with the largest weight), so this process will take *O*(*Eα*(*V*)) time.
  + Therefore, in total, the time complexity is O(*E*log*E*+*Eα*(*V*)) = *O*(*E*log*E*).
* **Space Complexity:** *O*(*V*). *V* represents the number of vertices. Keeping track of the root of every vertex in the union-find data structure requires O(V)*O*(*V*) space. However, depending on the sorting algorithm used, different amounts of auxiliary space will be required to sort the list of edges in place. For instance, **Timsort** (used by default in python) requires *O*(*E*) space in the worst-case scenario, while Java uses a variant of quicksort whose space complexity is *O*(log*E*).

## **Prim’s Algorithm**

"Prim's algorithm", that can be used to construct a “minimum spanning tree” of a “weighted undirected graph”.

### **The difference between the “Kruskal’s algorithm” and the “Prim’s algorithm”**

“Kruskal’s algorithm” expands the “minimum spanning tree” by adding edges. Whereas “Prim’s algorithm” expands the “minimum spanning tree” by adding vertices.

### **Complexity Analysis**

V represents the number of vertices, and *E* represents the number of edges.

* **Time Complexity:** *O*(*E*⋅log*V*) for Binary heap, and *O*(*E*+*V*⋅log*V*) for Fibonacci heap.
  + For a Binary heap:
    - We need *O*(*V*+*E*) time to traverse all the vertices of the graph, and we store in the heap all the vertices that are not yet included in our minimum spanning tree.
    - Extracting minimum element and key decreasing operations cost *O*(log*V*) time.
    - Therefore, the overall time complexity is *O*(*V*+*E*)⋅*O*(log*V*) = *O*(*E*⋅log*V*).
  + For a Fibonacci heap:
    - Extracting minimum element will take *O*(log*V*) time while key decreasing operation will take amortized *O*(1) time, therefore, the total time complexity would be (*E*+*V*⋅log*V*).
* **Space Complexity:** *O*(*V*). We need to store V*V* vertices in our data structure.

# **Single Source Shortest Path Algorithm**

## **Overview of Single Source Shortest Path**

Previously, we used the “breadth-first search” algorithm to find the “shortest path” between two vertices. However, the “breadth-first search” algorithm can only solve the “shortest path” problem in “unweighted graphs”. But in real life, we often need to find the “shortest path” in a “weighted graph”.

For example, there may be many routes from your home to a target location, such as a bus station, and the time needed for each route may be different. The route with the shortest distance may not be the one that requires the least amount of time because of the speed limit and traffic jams. So, if we want to find the route that takes the least time from home to a certain bus station, then the weights should be time instead of distance. With that in mind, how can we solve the “shortest path” problem given two vertices in a “weighted graph”?

The main focus of this chapter is to solve such “single source shortest path” problems. Given the starting vertex, find the “shortest path” to any of the vertices in a weighted graph. Once we solve this, we can easily acquire the shortest paths between the starting vertex and a given target vertex.

### **Edge Relaxation**

An alternative way to understand why this process is called ‘relaxation’ is to imagine that each path is a rubber band of length 1. The original path from A to D is of length 3, so the rubber band was stretched to 3 times its original length. When we relax the path to length 2, by visiting C first, the rubber band is now only stretched to twice its length, so you can imagine the rubber band being relaxed, hence the term edge relaxation.

In this chapter, we will learn two “single source shortest path” algorithms:

1. Dijkstra’s algorithm
2. Bellman-Ford algorithm

“Dijkstra's algorithm” can only be used to solve the “single source shortest path” problem in a graph with non-negative weights.

“Bellman-Ford algorithm”, on the other hand, can solve the “single-source shortest path” in a weighted directed graph with any weights, including, of course, negative weights.

## **Dijkstra's Algorithm**

“Dijkstra’s algorithm” solves the “single-source shortest path” problem in a weighted directed graph with non-negative weights.

### **The Main Idea**

We take the starting point u as the center and gradually expand outward while updating the “shortest path” to reach other vertices.

“Dijkstra's Algorithm” uses a “greedy approach”. Each step selects the “minimum weight” from the currently reached vertices to find the “shortest path” to other vertices.

### **Proof of the Algorithm**

The “greedy approach” only guarantees that, at each step, it takes the optimal choice in the current state. It does not guarantee that the final result is optimal.

### **Limitation of the Algorithm**

“Dijkstra’s Algorithm” can only be used on graphs that satisfy the following condition:

* Weights of all edges are non-negative.

### **Complexity Analysis**

V*V* represents the number of vertices, and E*E* represents the number of edges.

* Time Complexity: *O*(*E*+*V*log*V*) when a Fibonacci heap is used, *O*(*V*+*E*log*V*) for a Binary heap.
  + If you use a [Fibonacci heap](https://en.wikipedia.org/wiki/Fibonacci_heap) to implement the “min-heap”, extracting minimum element will take *O*(log*V*) time while key decreasing operation will take amortized *O*(1) time, therefore, the total time complexity would be *O*(*E*+*V*log*V*).
  + If you use a [Binary heap](https://en.wikipedia.org/wiki/Binary_heap), then the time complexity would be *O*(*V*+*E*log*V*).
* Space Complexity: *O*(*V*). We need to store V*V* vertices in our data structure.

**Kahn's Algorithm for Topological Sorting**

## **Overview of Kahn's Algorithm**

When selecting courses for the next semester in college, you might have noticed that some advanced courses have prerequisites that require you to take some introductory courses first. In Figure 12, for example, to take Course C, you need to complete Course B first, and to take Course B, you need to complete Course A first. There are many courses that you must complete for an academic degree. You do not want to find out in the last semester that you have not completed some prerequisite courses for an advanced course. So, how can we arrange the order of the courses adequately while considering these prerequisite relationships between them?

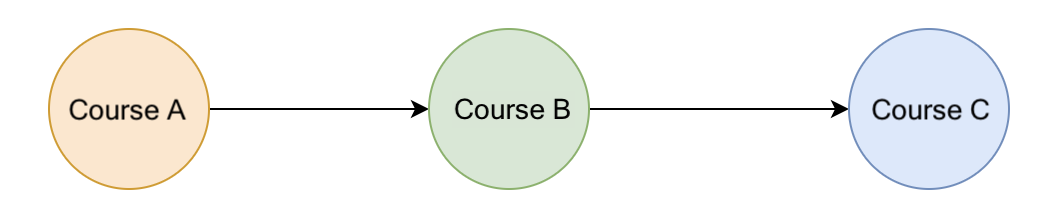


Figure 12. Prerequisite relationships between courses

“Topological sorting” helps solve the problem. It provides a linear sorting based on the required ordering between vertices in directed acyclic graphs. To be specific, given vertices u and v, to reach vertex v, we must have reached vertex u first. In “topological sorting”, u has to appear before v in the ordering. The most popular algorithm for “topological sorting” is Kahn’s algorithm.

Note, for simplicity while introducing Kahn's algorithm, we iterated over all of the courses and reduced the in-degree of those for which the current course is a prerequisite. This requires us to iterate over all E*E* prerequisites for all V*V* courses resulting in *O*(*V*⋅*E*) time complexity at the cost of O(V) space to store the in degree for each vertex.

However, this step can be performed more efficiently by creating an adjacency list where adjacencyList[course] contains a list of courses that depend on course. Then when each course is taken, we will only iterate over the courses that have the current course as a prerequisite. This will reduce the total time complexity to *O*(*V*+*E*) at the cost of an additional O(E) space to store the adjacency list.

### **Limitation of the Algorithm**

* “Topological sorting” only works with graphs that are directed and acyclic.
* There must be at least one vertex in the “graph” with an “in-degree” of 0. If all vertices in the “graph” have a non-zero “in-degree”, then all vertices need at least one vertex as a predecessor. In this case, no vertex can serve as the starting vertex.

### **Complexity Analysis**

*V* represents the number of vertices, and E represents the number of edges.

* Time Complexity: O(V + E).
  + First, we will build an adjacency list. This allows us to efficiently check which courses depend on each prerequisite course. Building the adjacency list will take *O*(*E*) time, as we must iterate over all edges.
  + Next, we will repeatedly visit each course (vertex) with an in-degree of zero and decrement the in-degree of all courses that have this course as a prerequisite (outgoing edges). In the worst-case scenario, we will visit every vertex and decrement every outgoing edge once. Thus, this part will take *O*(*V*+*E*) time.
  + Therefore, the total time complexity is O(E) + O(V + E) = O(V + E).
* Space Complexity: O(V+E).
  + The adjacency list uses O(E) space.
  + Storing the in-degree for each vertex requires O(V) space.
  + The queue can contain at most V nodes, so the queue also requires O(V) space.