Graph Theory

Reference

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".

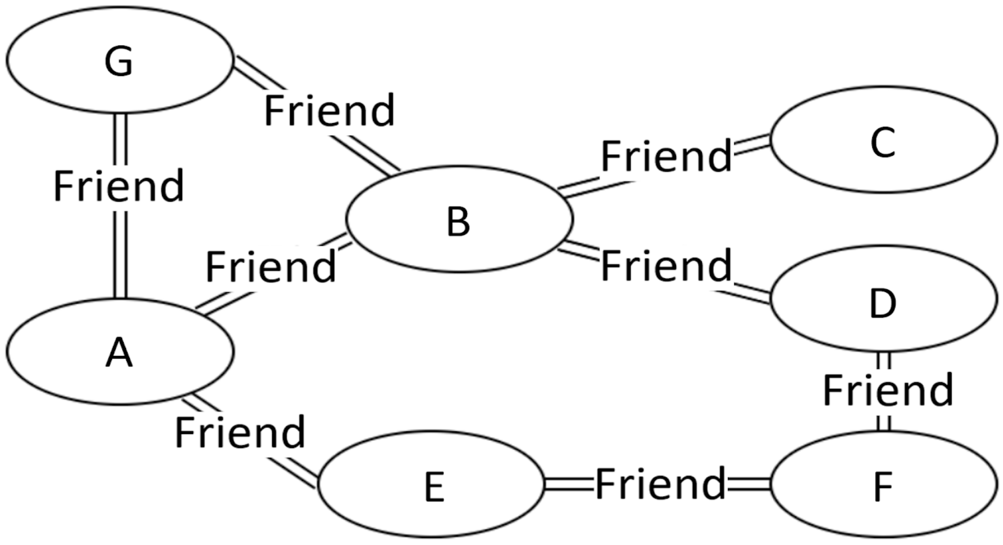

Figure 1. An example of an undirected graph.In Figure 1 above, we can see that person G, B, and E are all direct friends of A, while person C, D, and F are indirect friends of A. This example is a social graph of friendship. So, what is the "graph" data structure?

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.

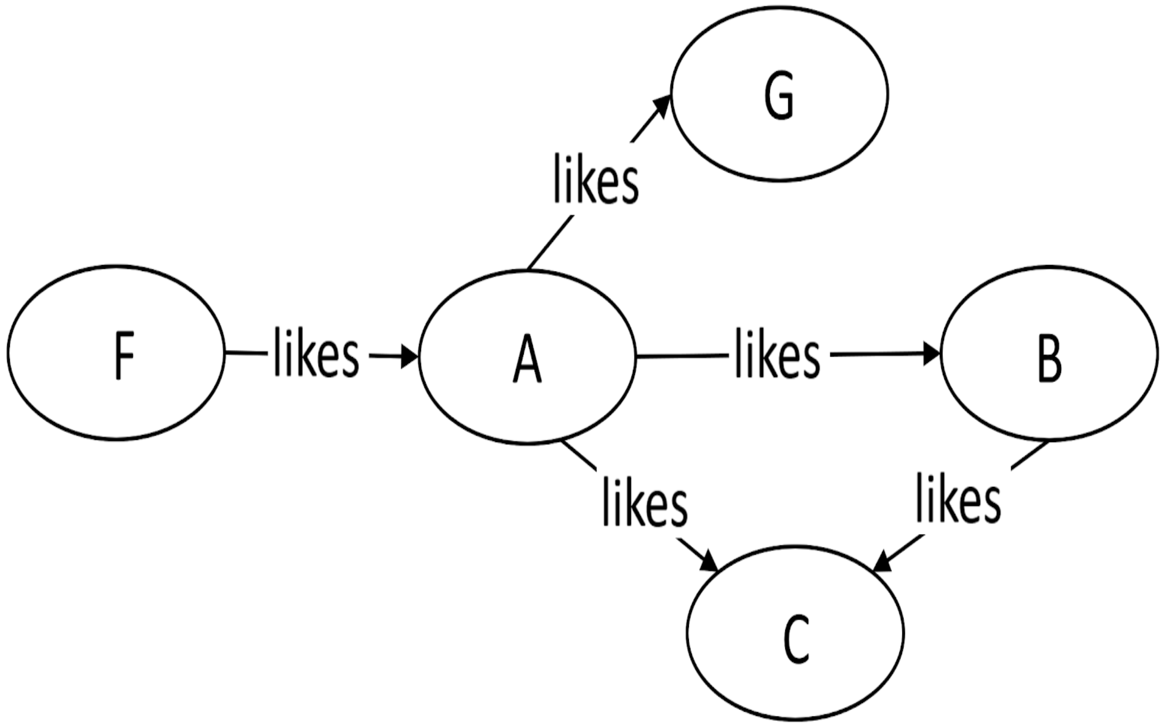

Figure 2. An example of a directed graph.

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.

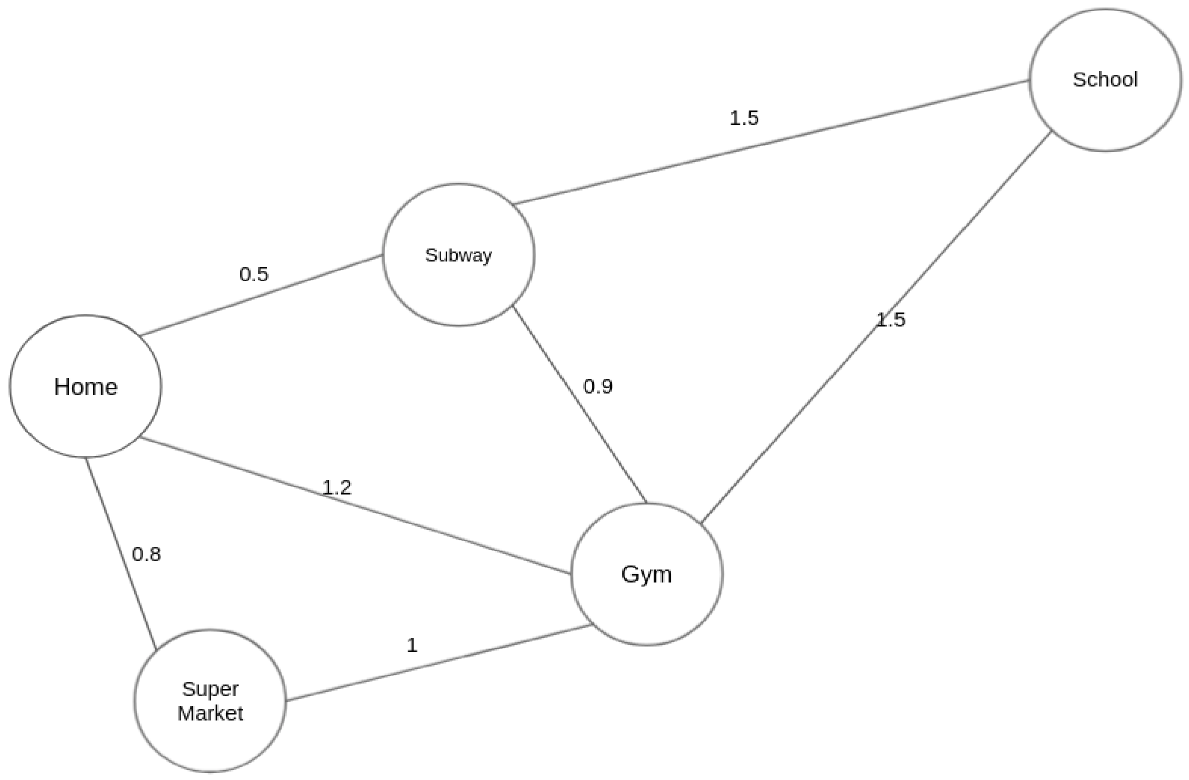

Figure 3. An example of a weighted graph.

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.

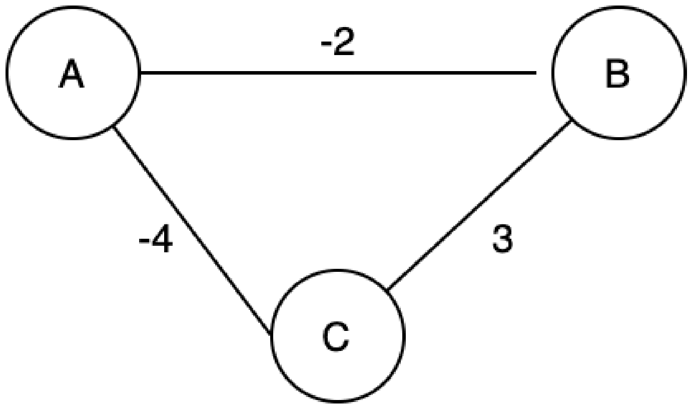

Figure 4. An example of a negative weight cycle.

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.

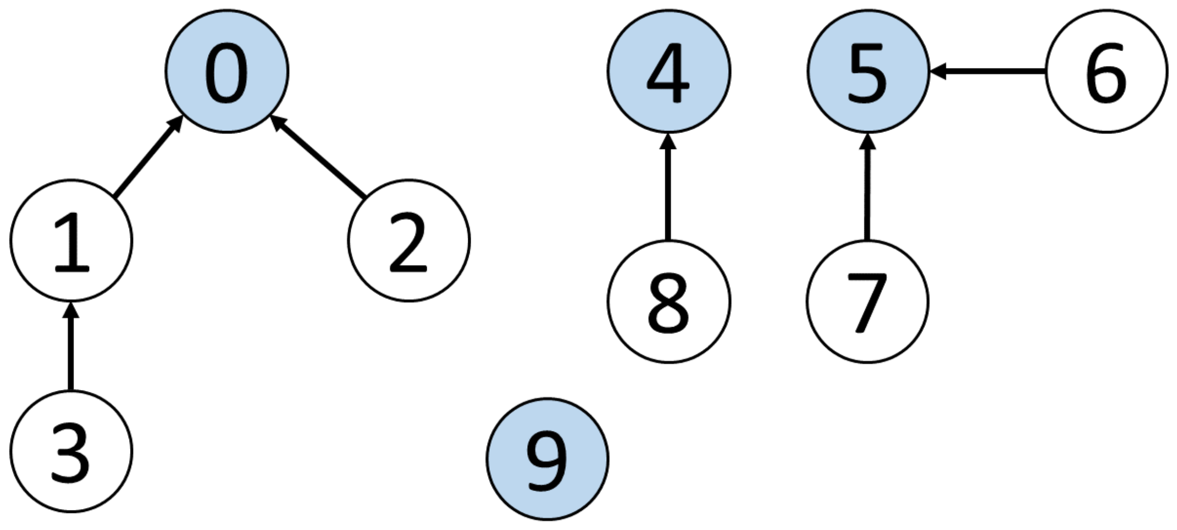

Figure 5. Each graph consists of vertices and edges. The root vertices are in blue.
The primary use of disjoint sets is to address the connectivity between the components of a network. The “network“ here can be a computer network or a social network. For instance, we can use a disjoint set to determine 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.”

In the introduction videos above, we discussed the two important functions in 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.

Explanation of Quick Find

IN THIS METHOD WE STORE ROOT OF EVERY NODE IN THE ARRAY JUST BECAUSE FOR QUICK FIND TIME COMPLEX WILL BE O(1) EVERY TIME.

IN UNION FUNCTION WE HAVE TO TRAVERSE WHOLE ARRAY EVERY TIME.

FOR EXAMPLE : LET SUPPOSE WE WANT TO CONNECT ROOT NODE 1 AND ROOT NODE 2 THEN AS A RESULT WE WILL MAKE ROOT 1 AS THE ROOT OF 2 BUT PREVIOUSLY THE NODE WHICH IS CONNECTED TO NODE 2 WHICH HAS ENTRY 2 AS A ROOT NODE . FOR THIS WE HAVE TO TRAVERSE WHOLE ARRAY AND CHANGE TO ROOT 1 OF EVERY NODE WHICH IS CONNECTED TO ROOT NODE 2.

```
class UnionFind {
public:
    UnionFind(int sz) : root(sz) {
        for (int i = 0; i < sz; i++) {
            root[i] = i;
        }
    }

    int find(int x) {
        return root[x];
    }

    void unionSet(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            for (int i = 0; i < root.size(); i++) {
                if (root[i] == rootY) {
                    root[i] = rootX;
                }
            }
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }

private:
    vector<int> root;
};

// Test Case
int main() {
    // for displaying booleans as literal strings, instead of 0 and 1
    cout << boolalpha;
    UnionFind uf(10);
    // 1-2-5-6-7 3-8-9 4
    uf.unionSet(1, 2);
    uf.unionSet(2, 5);
    uf.unionSet(5, 6);
    uf.unionSet(6, 7);
    uf.unionSet(3, 8);
    uf.unionSet(8, 9);
    cout << uf.connected(1, 5) << endl;  // true
    cout << uf.connected(5, 7) << endl;  // true
    cout << uf.connected(4, 9) << endl;  // false
    // 1-2-5-6-7 3-8-9-4
    uf.unionSet(9, 4);
    cout << uf.connected(4, 9) << endl;  // true

    return 0;
}
```

Time Complexity

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

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

hen initializing a union-find constructor, we need to create an array of size
N 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.

Explanation of Quick Union

Find:
HERE IF ARRAY[INDX]==INDX THEN ITS ROOT NODE.

UNION
IF X AND Y ANR NOT FROM THE SAME SET THEN ROOT[ROOT[Y]]=ROOT[X]

Why is Quick Union More Efficient than Quick Find?

The keen observer may notice that the Quick Union code shown here includes a soon-to-be introduced technique called path compression. While the complexity analysis in the video is sound, the correct Quick Union code (the right side of the video) is the implementation shown below.

```
class UnionFind {
public:
    UnionFind(int sz) : root(sz) {
        for (int i = 0; i < sz; i++) {
            root[i] = i;
        }
    }

    int find(int x) {
        while (x != root[x]) {
            x = root[x];
        }
        return x;
    }

    void unionSet(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            root[rootY] = rootX;
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }

private:
    vector<int> root;
};

// Test Case
int main() {
    // for displaying booleans as literal strings, instead of 0 and 1
    cout << boolalpha;
    UnionFind uf(10);
    // 1-2-5-6-7 3-8-9 4
    uf.unionSet(1, 2);
    uf.unionSet(2, 5);
    uf.unionSet(5, 6);
    uf.unionSet(6, 7);
    uf.unionSet(3, 8);
    uf.unionSet(8, 9);
    cout << uf.connected(1, 5) << endl;  // true
    cout << uf.connected(5, 7) << endl;  // true
    cout << uf.connected(4, 9) << endl;  // false
    // 1-2-5-6-7 3-8-9-4
    uf.unionSet(9, 4);
    cout << uf.connected(4, 9) << endl;  // true

    return 0;
}
```

Time Complexity

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

Note: NN is the number of vertices in the graph. In the worst-case scenario, the number of operations to get the root vertex will be HH where HH 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, HH can be at most NN 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)O(N) time.
The union operation consists of two find operations which (only in the worst-case) will take O(N)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)O(N) in the worst-case.
The connected operation also takes O(N)O(N) time in the worst-case since it involves two find calls.
Space Complexity
We need O(N)O(N) space to store the array of size NN.

Union by Rank - Disjoint Set

IN THIS METHOD WE USED TO STRICT THE HEIGHT OF TREE. IN QUICK UNION WE USED TO MAKE X OR Y AS A ROOT NODE ALWAYS WITHOUT ANY CHECKING OF HEIGHT BUT IN UNION BY RANK WE WILL PICK UP THE NODE WHICH HAS HIGHER RANK OR HIGHER HEIGHT.

```
class UnionFind {
public:
   UnionFind(int sz) : root(sz), rank(sz) {
      for (int i = 0; i < sz; i++) {
         root[i] = i;
         rank[i] = 1;
      }
   }

   int find(int x) {
      while (x != root[x]) {
         x = root[x];
      }
      return x;
   }

   void unionSet(int x, int y) {
      int rootX = find(x);
      int rootY = find(y);
```

```
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                root[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                root[rootX] = rootY;
            } else {
                root[rootY] = rootX;
                rank[rootX] += 1;
            }
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }

private:
    vector<int> root;
    vector<int> rank;
};

// Test Case
int main() {
    // for displaying booleans as literal strings, instead of 0 and 1
    cout << boolalpha;
    UnionFind uf(10);
    // 1-2-5-6-7 3-8-9 4
    uf.unionSet(1, 2);
    uf.unionSet(2, 5);
    uf.unionSet(5, 6);
    uf.unionSet(6, 7);
    uf.unionSet(3, 8);
    uf.unionSet(8, 9);
    cout << uf.connected(1, 5) << endl;  // true
    cout << uf.connected(5, 7) << endl;  // true
    cout << uf.connected(4, 9) << endl;  // false
    // 1-2-5-6-7 3-8-9-4
    uf.unionSet(9, 4);
    cout << uf.connected(4, 9) << endl;  // true

    return 0;
}
```

Time Complexity

	Union-find Constructor	Find	Union	Connected

Time Complexity	O(N)
	O(N)
		O(\log N)
		O(logN)
			O(\log N)
			O(logN)
				O(\log N)
				O(logN)

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

- For the union-find constructor, we need to create two arrays of size N each.
- For the find operation, in the worst-case scenario, when we repeatedly union components of equal rank, the tree height will be at most log(N)+1, so the find operation requires O(logN) time.
- For the union and connected operations, we also need O(logN) time since these operations are dominated by the find operation.

Space Complexity

We need
O(N)
O(N) space to store the array of size
N
 Path Compression Optimization - Disjoint Set
We use path compression so that time complexity will be reduced.
In Path Compression we first recursively find the root node and then simply put the root node instead of the parent node on the array corresponding to every node .

```
class UnionFind {
public:
    UnionFind(int sz) : root(sz) {
        for (int i = 0; i < sz; i++) {
            root[i] = i;
        }
    }

    int find(int x) {
        if (x == root[x]) {
            return x;
        }
        return root[x] = find(root[x]);
    }
    void unionSet(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            root[rootY] = rootX;
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }

private:
    vector<int> root;
};

// Test Case
int main() {
    // for displaying booleans as literal strings, instead of 0 and 1
    cout << boolalpha;
    UnionFind uf(10);
    // 1-2-5-6-7 3-8-9 4
    uf.unionSet(1, 2);
    uf.unionSet(2, 5);
    uf.unionSet(5, 6);
    uf.unionSet(6, 7);
    uf.unionSet(3, 8);
    uf.unionSet(8, 9);
    cout << uf.connected(1, 5) << endl;  // true
    cout << uf.connected(5, 7) << endl;  // true
    cout << uf.connected(4, 9) << endl;  // false
    // 1-2-5-6-7 3-8-9-4
    uf.unionSet(9, 4);
    cout << uf.connected(4, 9) << endl;  // true

    return 0;
}
```

Time Complexity

Time complexities shown below are for the average case, since the worst-case scenario is rare in practice.

	Union-find Constructor	Find	Union	Connected

Time Complexity	O(N)	O(logN)	O(logN)	O(logN)

- As before, we need O(N) time to create and fill the root array.
- For the find, union, and connected operations (the latter two operations both depend on the find operation), we need O(1) time for the best case (when the parent node for some vertex is the root node itself). In the worst case, it would be O(N) time when the tree is skewed. However, on average, the time complexity will be O(logN). Supporting details for the average time complexity can be found in Top-Down Analysis of Path Compression where R. Seidel and M. Sharir discuss the upper bound running time when path compression is used with arbitrary linking.

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

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

```
class UnionFind {
public:
    UnionFind(int sz) : root(sz), rank(sz) {
        for (int i = 0; i < sz; i++) {
            root[i] = i;
            rank[i] = 1;
        }
    }

    int find(int x) {
        if (x == root[x]) {
            return x;
        }
        return root[x] = find(root[x]);
    }

    void unionSet(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                root[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                root[rootX] = rootY;
            } else {
                root[rootY] = rootX;
                rank[rootX] += 1;
            }
        }
    }

    bool connected(int x, int y) {
        return find(x) == find(y);
    }

private:
    vector<int> root;
    vector<int> rank;
};

// Test Case
int main() {
    // for displaying booleans as literal strings, instead of 0 and 1
    cout << boolalpha;
    UnionFind uf(10);
    // 1-2-5-6-7 3-8-9 4
    uf.unionSet(1, 2);
    uf.unionSet(2, 5);
    uf.unionSet(5, 6);
    uf.unionSet(6, 7);
    uf.unionSet(3, 8);
    uf.unionSet(8, 9);
    cout << uf.connected(1, 5) << endl;  // true
    cout << uf.connected(5, 7) << endl;  // true
    cout << uf.connected(4, 9) << endl;  // false
    // 1-2-5-6-7 3-8-9-4
    uf.unionSet(9, 4);
    cout << uf.connected(4, 9) << endl;  // true

    return 0;
}
```

Time Complexity

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

Note:
N

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, $O(\alpha(N))$ is regarded as $O(1)$ on average.

- For the union-find constructor, we need to create two arrays of size N each.
- When using the combination of union by rank and the path compression optimization, the find operation will take $O(\alpha(N))$ 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 $O(\alpha(N))$ time on average.

Space Complexity

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