TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Unit 11 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem-solving journey! Use this as a reference as you solve the breakout problems for Unit 11. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 11. In addition to the material below, you will be expected to know any required concepts from previous units.

Standard Concepts

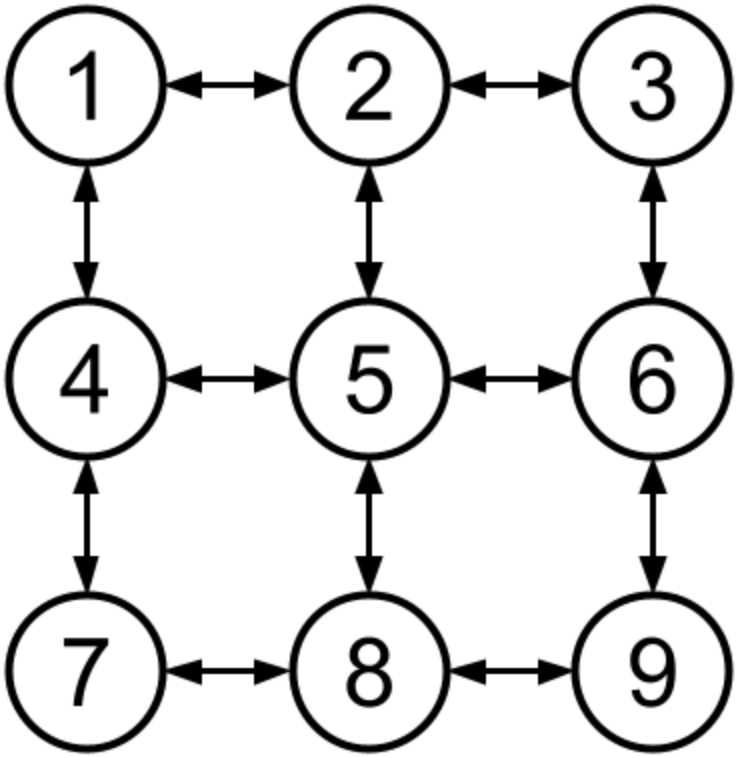
Matrices

Many matrix problems can be solved using graph algorithms by imagining the cells in the matrix as a grid of connected nodes. Each cell in the matrix represents a node. Edges exist between horizontally or vertically adjacent cells/nodes in the matrix/graph.

For example, we can take the following matrix, `grid`:

```
grid = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
    ]
```

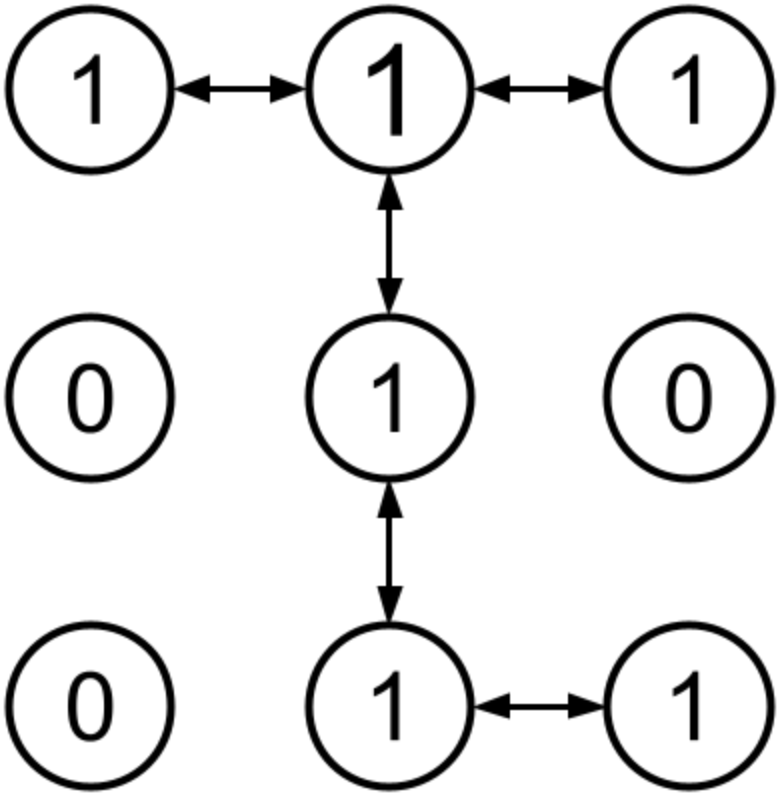
And represent it as the following graph:



Certain matrix problems may place additional constraints on whether or not an edge exists between two cells in a matrix. For example, imagine a binary matrix where each cell has a value of either `0` or `1`. You would like to find if there exists a path of `1` values from some position `start = (start_row, start_col)` in the matrix to another position `destination = (dest_row, dest_col)` in the matrix.

```
grid = [
    [1, 1, 1],
    [0, 1, 0],
    [0, 1, 1]
]
```

Since we may only travel between cells with value `1`, in our graph representation of this matrix there should only exist edges between horizontally and vertically adjacent cells with value `1`.



When we reimagine matrices as graphs, we can then apply graph algorithms such as Breadth First and Depth First Search to the matrix. We can perform these algorithms directly on the matrix without needing to convert the matrix to another graph representation like an adjacency list or adjacency matrix. To do so, we choose a starting cell at some `(row, column)` position in the matrix as our starting "node" and consider cells up, down, left, and right of the current cell potential neighbors. Depending on the problem constraints, we may need to make additional considerations to determine whether each horizontally and vertically adjacent cell is an actual neighbor (ex. whether each potential neighbor has value `1`).

Pseudocode for BFS on a Matrix:

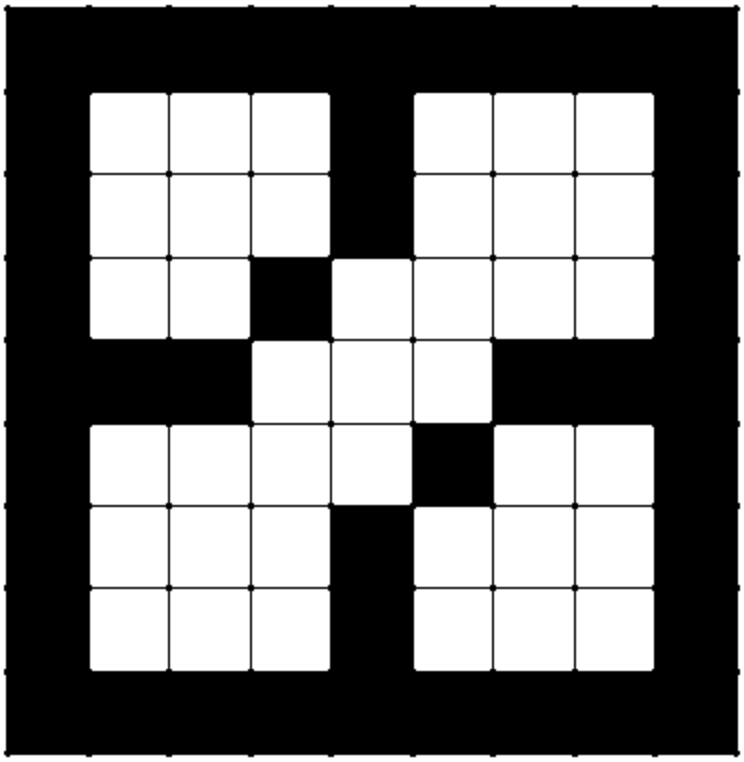
```
- Start from a cell, add it to a queue, and mark it as visited.
- While the queue isn't empty, take the cell at the front of the queue, and check its ne
  (up, down, left, right).
- For each neighbor, if it hasn't been visited, add it to the queue and mark it as visit
- Repeat this process until all reachable cells are visited.
```

Pseudocode for DFS on a Matrix:

```
- Start from a cell, mark it as visited.
- Recursively move to one of its neighboring cells (up, down, left, right) if it hasn't
- If no unvisited neighbors remain, backtrack to the previous cell and explore the next
- Repeat this process until all reachable cells are visited.
```

Flood Fill

Flood fill is an algorithm commonly used to fill a connected area in a matrix, starting from a specific point. It spreads out to neighboring cells based on certain conditions. The idea is to "flood" an unfilled area with a new color or value by filling all connected cells that meet a given condition, typically horizontally and vertically adjacent cells that have the same initial color.



Flood fill is commonly used in applications like:

- **Graphics**: For filling a region in an image (think of the "bucket" tool in image editing software).
- **Game development**: To detect connected areas or territories.
- **Puzzle-solving**: In grid-based problems where you need to explore or modify a region (like Minesweeper).

The flood fill algorithm can be implemented using either BFS or DFS. In both cases, we start from the initial cell and "flood" outward to neighboring cells that meet a specific condition.

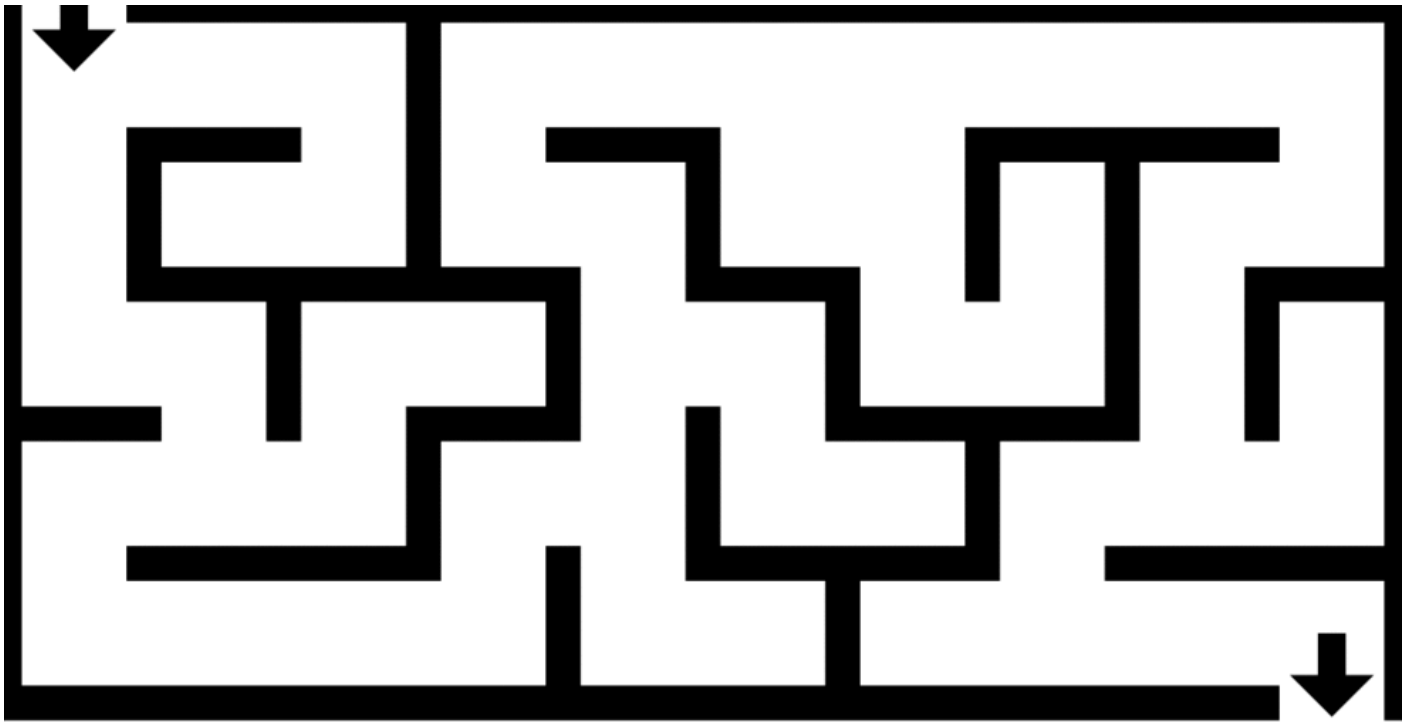
Pseudocode for flood fill:

```
1. Start from a given cell: Begin the flood fill from a starting cell (row, column).
2. Check if the cell meets the condition: Ensure the cell has the same color/value as th
   or satisfies the conditions for spreading.
3. Fill the cell: Change the cell's value to the new fill value (e.g., a new color or a
   as visited).
4. Move to neighboring cells: Recursively or iteratively spread the fill to neighboring
   down, left) that meet the condition.
5. Repeat until the entire region is filled: The algorithm continues to spread until the
   cells that satisfy the conditions.
```

Backtracking

Backtracking is an algorithmic technique that involves building a solution incrementally. Every time there is a choice to be made, a backtracking algorithm will choose one possible choice and continue to try to build a solution. If the algorithm finds that the solution cannot be completed with the chosen sequence of choices, it will "backtrack" by undoing the last choice it made and trying a different option.

A classic example of a backtracking problem is finding a path through a maze. At each fork in the maze, we choose one option. At each successive fork, we continue to choose an option until we hit a dead end. When that happens, we turn around to the last fork in the maze and choose another route. We continue doing this until we have exhausted all our options or found a viable path out of the maze!



Source: via Brilliant.org

Backtracking is not a graph-specific algorithm. Rather, it describes a general technique that is commonly applied to problems that involve:

- Making choices at each step
- Exploring multiple paths
- Meeting constraints

That being said, one of the most common backtracking algorithms is Depth First Search! DFS can be thought of as a form of backtracking, particularly when solving problems where:

- We are looking for all paths or combinations.
- We need to undo choices when a certain path or configuration does not lead to a valid solution.

DFS works by exploring one path in a graph or tree as deeply as possible before it gets to a point where no further progress can be made either because all its neighbors are already visited or the path doesn't meet the problems constraints (ex. in the matrix problem above trying to path of `1`s, but all unvisited

neighboring cells are `0`s). When this happens, DFS "backs up" to the previous step and tries a different path.

Advanced Concepts

Python Syntax

Priority Queues

Priority queues are a data structure we will see used when implementing Dijkstra's Algorithm for the shortest path in a weighted graph.

A priority queue is a variation of a traditional queue data structure. A traditional queue removes elements in first-in-first-out (FIFO) order. In contrast, a priority queue assigns a priority or cost to each item added to the queue. It then removes items by order of their priority.

There are two types of priority queues:

- **minimum priority queue**: removes items with the smallest priority/cost first
- **maximum priority queue**: removes items with the largest priority/cost first

To implement a priority queue in Python, we can import the `heapq` module. The `heapq` module primarily supports mininmum priority queues, meaning the smallest element is always at the root. However, you can simulate a maximum priority queue by using negative values. The module is called `heapq` because behind the scenes, priority queues are implemented with another data structure called a heap.

In Python, the `heapq` module provides an implementation of the **heap** queue algorithm, also known as a **priority queue**. Heaps are binary trees where the parent node is always smaller (for a min-heap) or larger (for a max-heap) than its children, making it efficient for operations like finding the smallest or largest element.

The `heapq` module primarily supports **min-heaps**, meaning the smallest element is always at the root. However, you can simulate a max-heap by using negative values.

Example Usage:

```
# Adding an item to a heap: heapq.heappush(heap, item)
import heapq
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
print(heap)  # Output: [1, 3, 4]

# Removing an item from the heap: heapq.heappop(heap)
smallest = heapq.heappop(heap)  # Removes 1, the smallest element in the heap
print(smallest)  # Output: 1

# Adding an item to a heap where value is different than priority
# heapq.heappush(heap, (priority, item))
heap_2 = []
heapq.heappush(heap_2, (3, 'a'))
heapq.heappush(heap_2, (1, 'b'))
heapq.heappush(heap_2, (4, 'c'))
print(heap_2) # Output: [(1, 'b'), (3, 'a'), (4, 'c')]

# Removing an item from the heap where value is different than priority
smallest = heapq.heappop(heap_2)
print(smallest) # Output: (1, 'b')

# Simulating a max heap with negative values
max_heap = []
heapq.heappush(max_heap, -3)
heapq.heappush(max_heap, -1)
heapq.heappush(max_heap, -4)
max_value = -heapq.heappop(max_heap)  # Output: 4
```

Topological Sort

Often, we encounter situations where tasks or events have dependencies. For an example, before you can take WEB102 you need to take WEB101. We can represent such scenarios using a graph, where each task or event is a node in the graph and each dependency is a directed edge between two nodes. If task `A` depends on task `B`, then there is a directed edge from `B` to `A`.

Topological sort is an algorithm that allows us to sort tasks and events based on their dependencies. It is an algorithm that only works on directed acyclic graphs (DAGs). As a reminder, a DAG is a sepcial type of graph that maintains the following two properties:

1. **Directed Edges**: All edges have a specific direction.
2. **Acyclic**: There are no cycles in the graph, meaning it is impossible to visit the same node along the same path, once you've followed an outgoing edge.

Formally, topological sort will find an ordering of nodes in a DAG such that for every directed edge `u -> v`, `u` comes before `v` in the final ordering.

There are two common approaches to implementing topological sort: BFS (also referred to as Kahn's Algorithm) and DFS.

Breadth First Search Approach (Kahn's Algorithm)

Kahn's algorithm uses the **in-degree** of a node as part of its implementation. The in-degree of a node in a directed graph, is the number of incoming edges to that node.

Pseudocode for Kahn's Algorithm:

```
1. Calculate the in-degree of each node (how many edges are coming into the node).
2. Start with nodes that have an in-degree of 0. These nodes have no dependencies and ca
3. Remove the processed node from the graph
4. For each of the processed node's neighbors, reduce their in-degree by 1. Since one of
   been processed, you are effectively "removing" an incoming edge.
5. Add new nodes with in-degree 0 to the queue and repeat until all nodes are processed.
```

Depth First Search Approach

Topological sort using DFS works by performing a depth-first traversal on the graph and processing each node only after all its descendants have been visited. This results in a valid topological ordering where all nodes appear after all their dependencies.

```
1. Perform DFS on the graph. Visit all nodes in the graph one by one. For each unvisited
   perform a DFS that explores all its neighbors (dependencies). This ensures that all n
   are processed.
2. When visiting a node, recursively visit all its neighbors. Once you finish visiting a
   the node to a stack or result list. This way, nodes are processed after their depende
   have been processed.
3. Continue until all nodes are visited.
4. Once all nodes have been visited and processed, the stack will contain the nodes in r
   order. To get the correct order, reverse the stack and return it.
```

Comparing Kahn's Algorithm and DFS

There can be multiple valid topological sorts of the same graph. Different implementations of the algorithm, or even different executions of the same implementation, may produce different valid orderings. Depending on the problem statement, we may prefer one implementation strategy over the other.

Criterion	BFS (Kahn's Algorithm)	DFS-based Topological Sort
Processing "layers" of nodes	Useful when tasks can be processed in levels	DFS explores paths deeply first, not level by level

Criterion	BFS (Kahn's Algorithm)	DFS-based Topological Sort
Cycle detection	Easier and more intuitive to detect cycles	Cycle detection is possible but more complex
Algorithm complexity	Easier for beginners to understand and implement	Requires comfort with recursion or stack-based logic
Memory efficiency	Uses more memory due to maintaining an explicit queue	More memory-efficient since it uses recursion or a stack
When to explore dependencies deeply	Not as suitable for deep exploration of dependencies	Good for exploring and resolving deep dependencies first

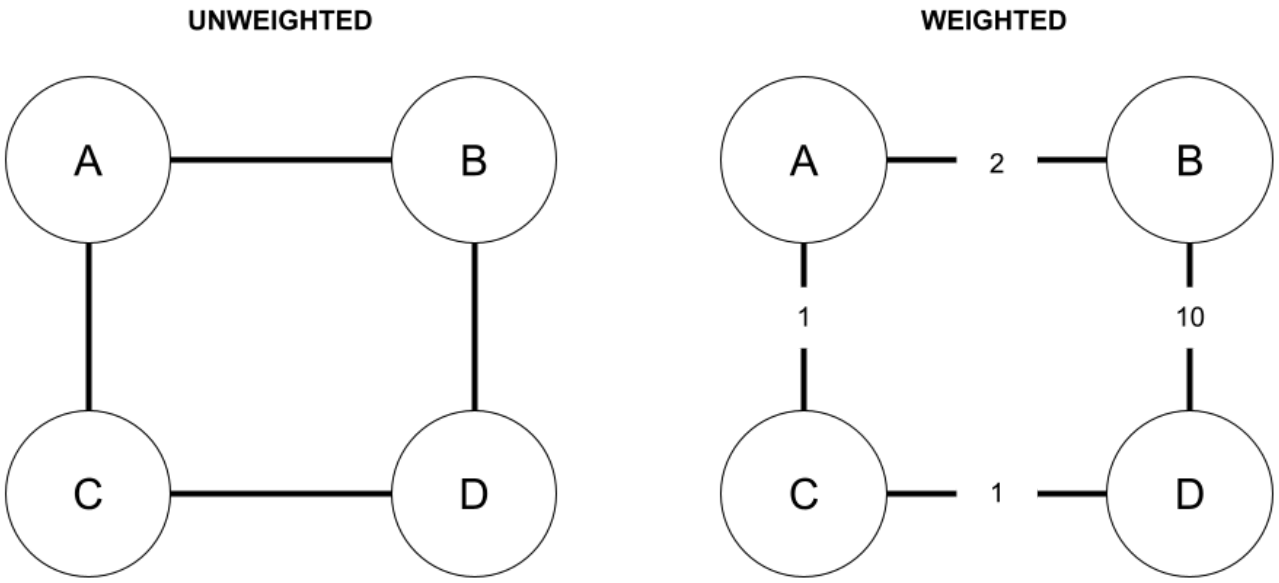
Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between two nodes in a weighted graph.

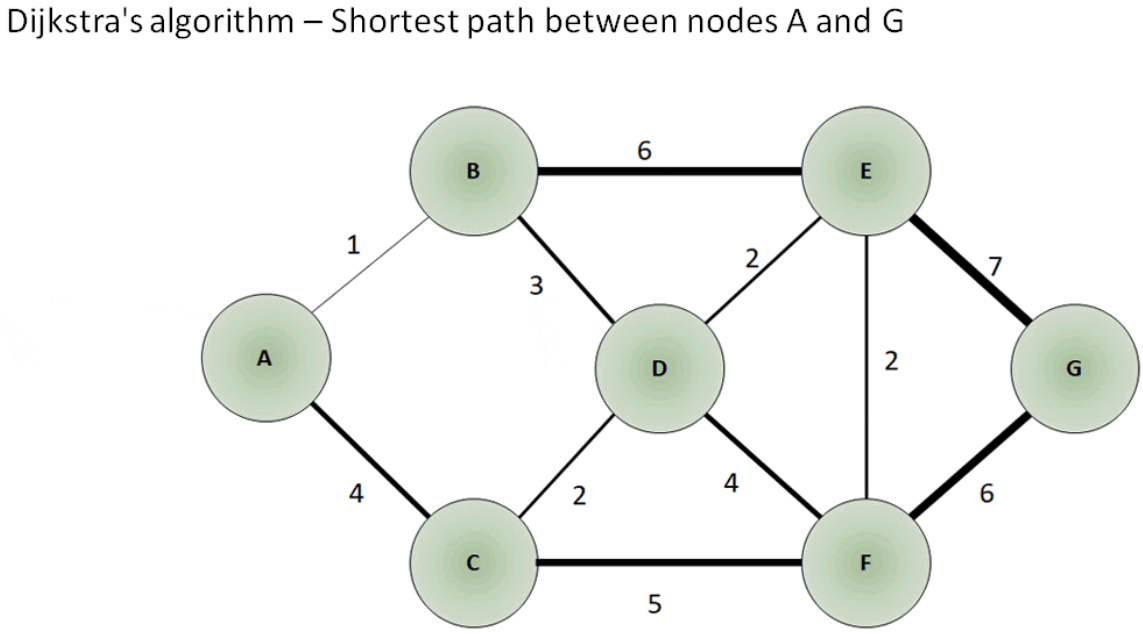
Recall that a normal BFS traversal allows us to find the shortest path between any two nodes in an unweighted graph. This is because BFS explores nodes in order of proximity to the starting node, and in an unweighted graph, the length of a path is equal to the number of the edges in the path.

With a weighted graph, finding the shortest path between two nodes is different. The weight of an edge is equivalent to the 'length' or 'distance' of the edge. The total length of a path is calculated by summing the weights of all the edges that make up at the path. The shortest path is also sometimes referred to as the minimum cost path where an edge's weight is equivalent to its cost.

For example, below are two versions of the same graph. One is weighted while the other is weighted. If we wanted to find the shortest path from B to D, in the unweighted version we would say the shortest path is the direct edge from `B -> D` for a path length of `1`. In the weighted version however, we would say the shortest path is `B -> A -> C -> D` for a total path length of `4` equal to the sum of the edge weights. This is because the direct edge `B -> D` has a greater weight of `10`.



Dijkstra's algorithm works by initially overestimating the cost to travel from the start node to every other node in the graph. Then, it follows a pattern similar to BFS. Beginning at the start node, the algorithm visits each neighbor of the current node. As it visits new nodes, the algorithm uses the weights of the corresponding edges to adjust the original estimated cost to each neighbor. If it finds that the total cost of a path from the start node to the neighbor is less than the currently estimated cost, it will update the minimum cost and path. Once Dijkstra's has explored the entire graph, we know we have found the shortest path to each node from the start node.



Source: via graphable.ai

Pseudcode for Dijkstra's Algorithm:

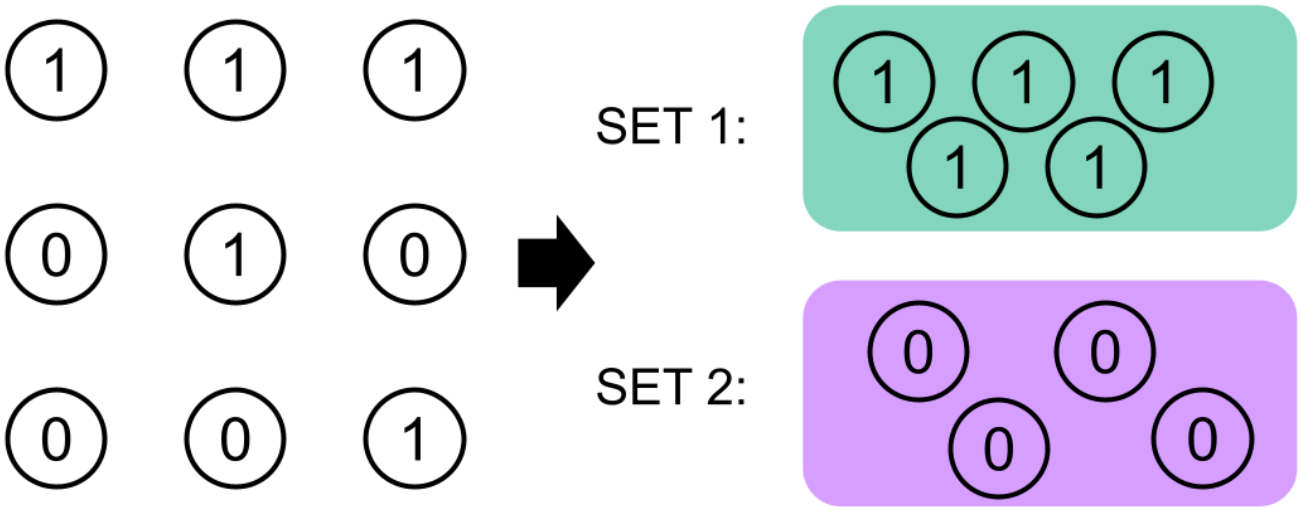
```
1. Initialize a distances list equal in length to the number of nodes in the graph
    a. Tracks the minimum cost/distance from start node to each node in graph
    b. Initialize each value to infinity
    c. Initialize distances[start_node] to zero
2. Initialize a previous list equal in length to the number of nodes in the graph
    a. Tracks the previous node in each node's shortest/minimum cost path
    b. Initialize each value to None since we have not yet traversed any paths
3. Initialize a visited list to track which nodes have already been visited
4. Initialize a priority queue and add the start node to it
5. While the queue is not empty:
    a. Set current as node in the queue with minimum cost
    b. Add current to the list of visited nodes
    c. loop through all the current node's neighbors
        i. if the neighbor has not yet been visited:
            1. calculate distance from start node to neighbor via current node
            2. If calculated distance < distances[neighbor]
                a. distances[neighbor] = calculated distance
                b. previous[neighbor] = current_node
            3. queue.append(neighbor)
6. Return the previous and distances list
```

Union Find (Disjoint Set Union)

Union Find, also called Disjoint Set Union (DSU), is a data structure that provides an easy way to group objects into different non-overlapping (disjoint) sets.

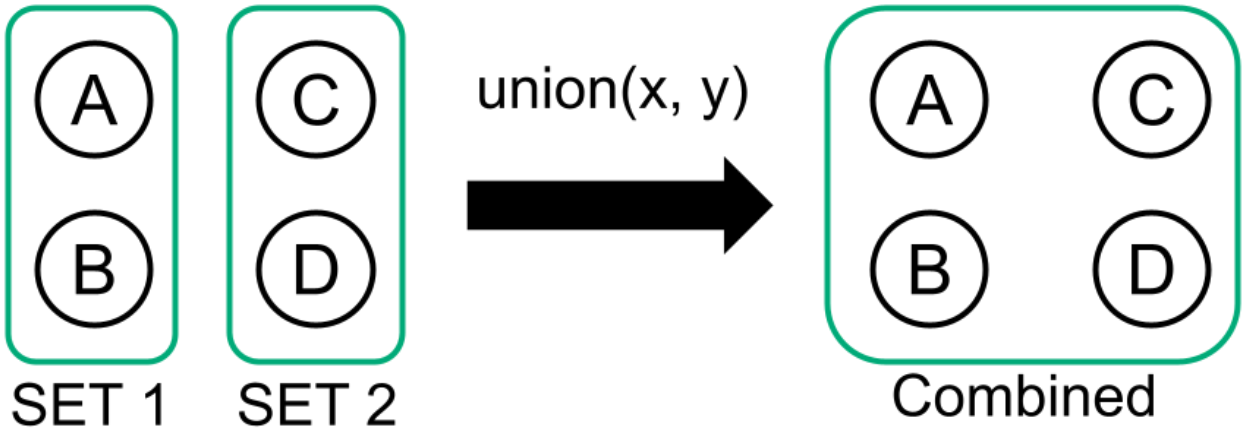
For example, in many binary matrix/graph problems, we may want to group cells/nodes into two sets:

1. Nodes with value `1`
2. Nodes with value `0`



The Union Find data structure also provides two key methods:

1. **Union**: Combines two sets into a single set.



2. **Find**: Determines which set a particular element belongs to.

Union-Find Applications

This structure is commonly used to efficiently solve problems related to connectivity, where you need to keep track of which elements are in the same set or connected component, such as:

- Finding connected components in graphs.
- Detecting cycles in undirected graphs.
- MST (Minimum Spanning Tree) algorithms like Kruskal's Algorithm.

Union-Find is particularly useful in the following scenarios:

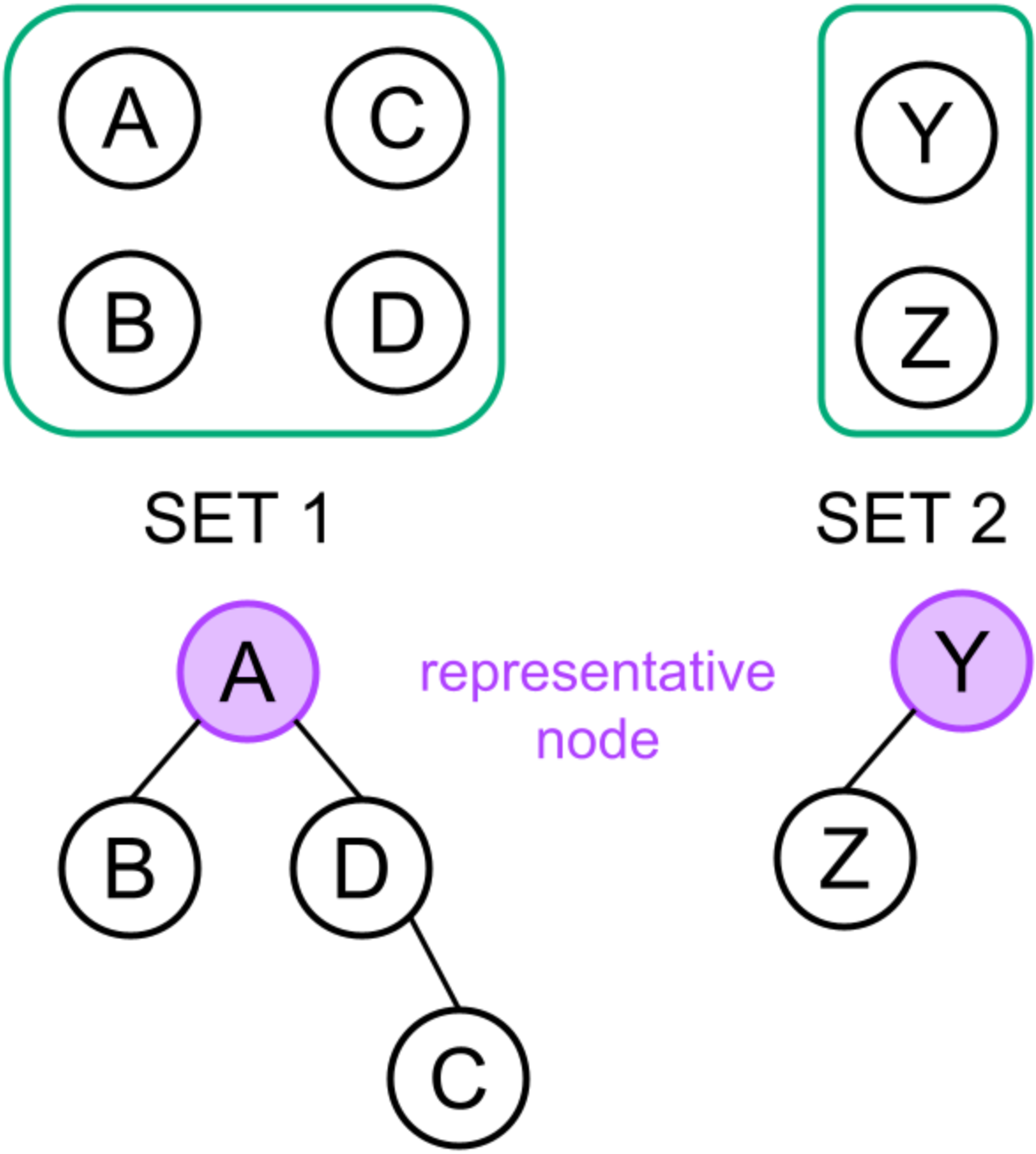
- When there are multiple groups of elements, and you want to efficiently combine them or check if two elements belong to the same group.
- When the operations need to be performed frequently, and you want a fast way to handle merging groups and checking connections.

Note that the *efficiency* is one of the key advantages to using Union Find. It is often not the *only* way to solve a problem like finding the number of connected components in a graph, but rather an optimized alternative.

Union-Find Data Structure

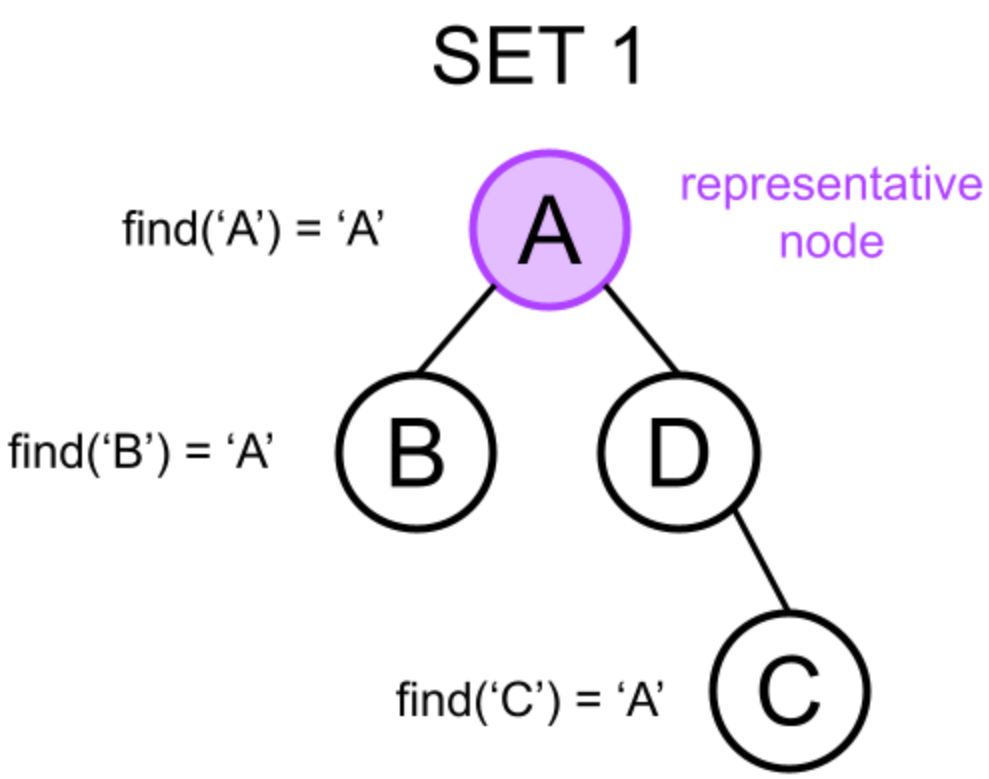
When implementing the Union Find data structure, each distinct set is thought of as a tree like structure, with edges connecting nodes who are a part of the same set.

The root node of each structure is also called the **representative node** and is used to represent each set.



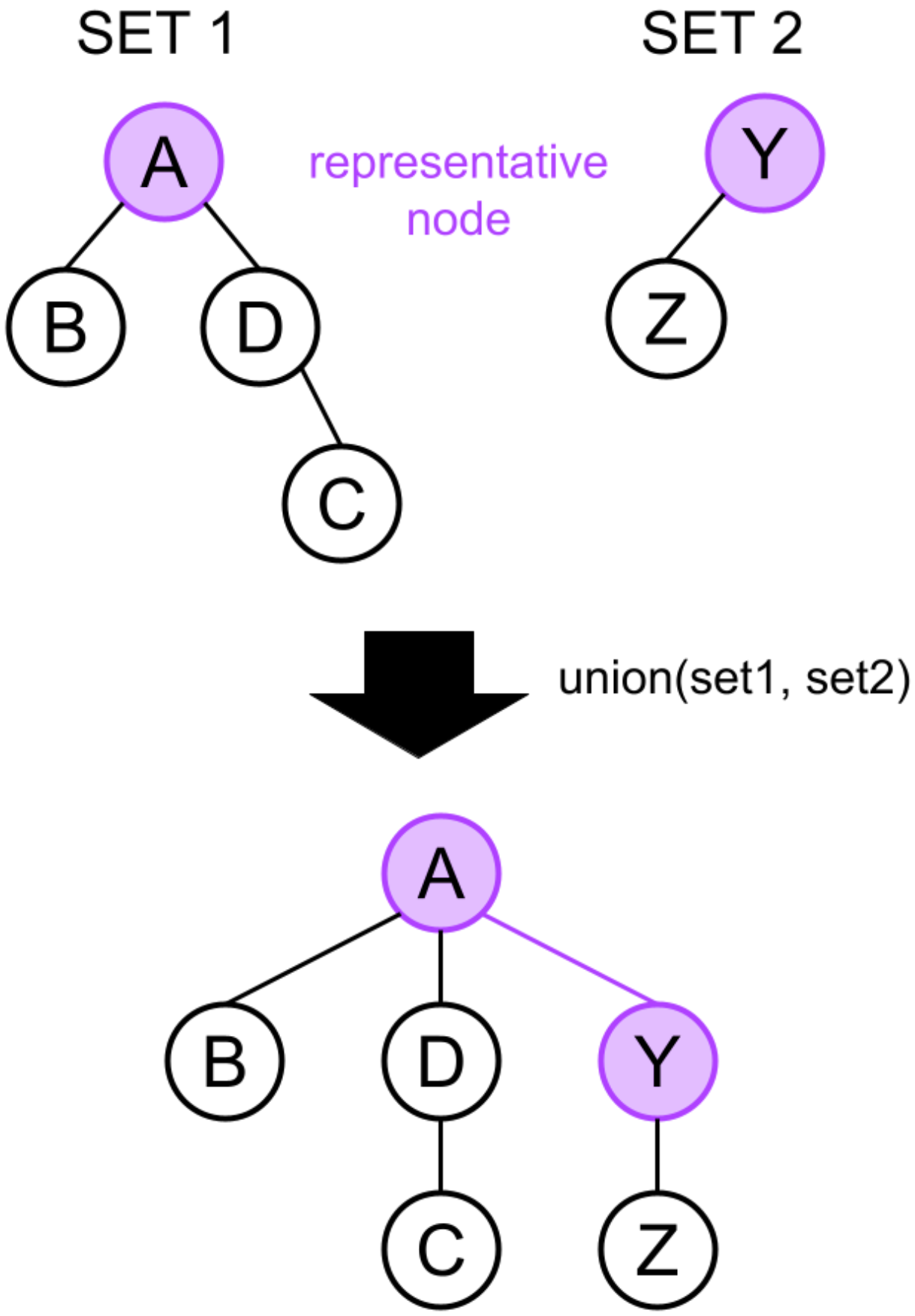
When we use the `find()` method, the operation will determine which set an element `x` belongs to by returning the representative of the set. All elements in the same set share the same representative. The representative of the representative node is itself.

Example Usage:




```
find('A') # Output: 'A'
find('B') # Output: 'A'
find('C') # Output: 'A'
find('D') # Output: 'A'
```

The **rank** of a set is the depth or size of its tree. When we `union()` or merge two sets together, we take the representative/root node of the tree with the smaller rank and set it as the child of the representative/root node of the tree with the larger rank. The representative node of the larger set becomes the representative of the merged set


SET 1
SET 2
A
representative
node
Y
B
D
Z
C
union(set1, set2)
A
B
D
Y
C
Z

Union Find Constructor

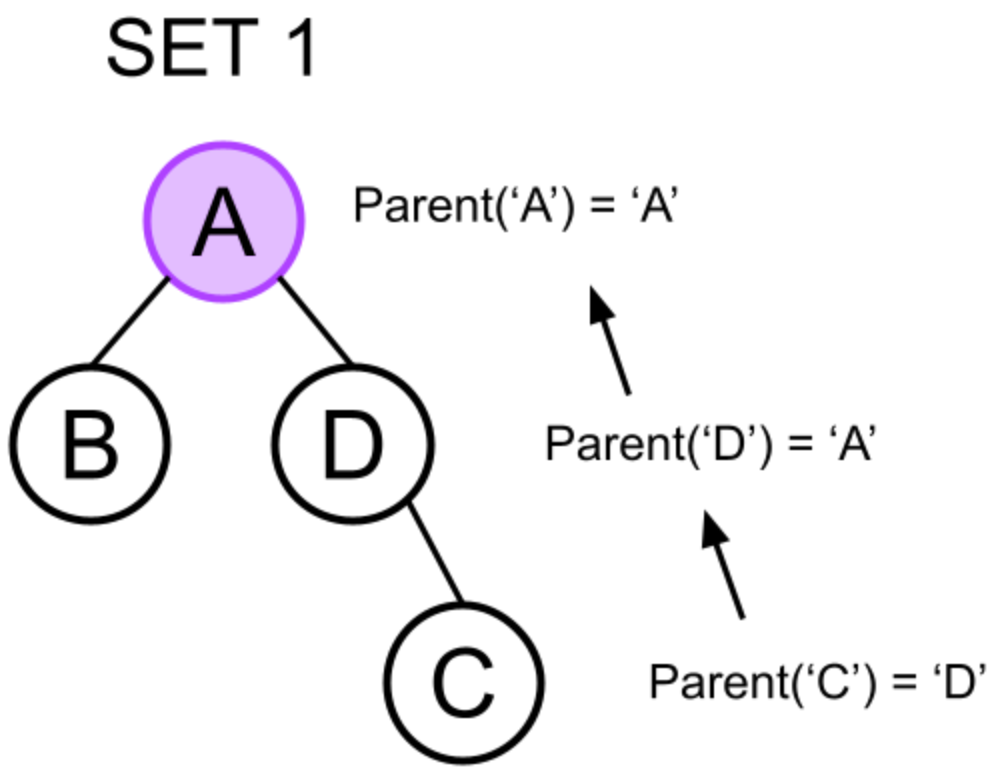
The constructor of the Union Find data structure usually includes the following two attributes:

1. **Parent Array**: Each element points to its parent node, and the "representative" or "root" of a set is the element that points to itself. We keep track of the parents to help us with the `find()` operation. Initially, each element is its own parent.
2. **Rank/Size Array**: Keeps track of the depth (or size) of the tree for each set. This helps ensure that the tree stays balanced during the union operation, which improves performance. This attribute is optional but found in most implementations.

```
class DSU:
    def __init__(self, n):
        # Initialize parent and rank arrays
        self.parent = [i for i in range(n)]  # Initially, each element is its own parent
        self.rank = [0] * n                  # Rank (or size) of the trees initialized t
```

Find Method Implementation

The find method follows the parent pointers up the tree until it finds the representative/root of the set (an element that is its own parent).



```
class DSU:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, x):
        # Find the root of the set containing x
        if self.parent[x] != x:
            # Recursively find the root
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

Union Method Implementation

The union method combines two sets by connecting the roots of their trees. To keep the tree balanced, we attach the smaller tree to the root of the larger tree. This helps ensure that the trees don't become too deep, which keeps operations efficient.

```
class DSU:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Union by rank (merge two sets)
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            # Attach the smaller tree under the larger tree
            if self.rank[rootX] > self.rank[rootY]:
                self.parent[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.parent[rootX] = rootY
            else:
                self.parent[rootY] = rootX
                self.rank[rootX] += 1
```

DSU Class Example Usage

Let's say we have 5 elements (0 to 4) and we want to perform some union and find operations.

```
# Create a DSU for 5 elements
dsu = DSU(5)

# Perform some union operations
dsu.union(0, 1)
dsu.union(1, 2)

# Check if elements are in the same set (connected)
print(dsu.connected(0, 2))  # Output: True (0, 1, 2 are in the same set)
print(dsu.connected(0, 3))  # Output: False (0 and 3 are in different sets)

# Perform more unions
dsu.union(3, 4)

# Check again after more unions
print(dsu.connected(3, 4))  # Output: True (3 and 4 are in the same set)
print(dsu.connected(2, 4))  # Output: False (2 and 4 are in different sets)
```

Bonus Syntax & Concepts

The following concepts are nice to know and may improve your graphs knowledge and help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit and we recommend mastering BFS and DFS first. There are endless graph algorithms that you can learn and these are only a few. These concepts are **not in scope for either the Standard or Advanced Unit 11 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

- Minimum Spanning Trees is a subset of edges in an undirected, connected, weighted graph that includes all nodes in the graph while minimizing the total edge weight.
- Prim's Algorithm finds the minimum spanning tree of a graph.
- Kruskal's Algorithm also finds the minimum spanning tree of a graph.
- Heaps a tree data structure used to implement priority queues where any given node is always greater than its child nodes (for a max heap) or smaller than its child nodes (for a min heap).