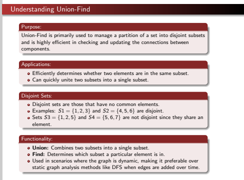


Union-Find is a valuable data structure for managing a collection of disjoint sets. It provides an efficient method for determining the connected components in a graph and is particularly useful for cycle detection. While Depth First Search (DFS) can also be used for detecting cycles by maintaining a set of visited nodes, it is more suited to static graphs where the structure does not change over time. Conversely, Union-Find excels in dynamic scenarios where the graph may be updated with new edges or vertices.

Disjoint Sets:

At the heart of Union-Find is the concept of *disjoint sets*. These are sets that do not share any elements; formally, they are sets whose intersection is the empty set. For example, consider two sets:

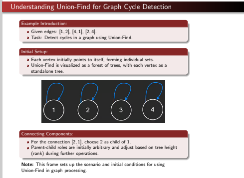
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{4, 5, 6\}$

S_1 and S_2 are disjoint because there is no element common to both. On the other hand, sets $S_3 = \{1, 2, 5\}$ and $S_4 = \{5, 6, 7\}$ are not disjoint as they both contain the element 5.

Operations: Union-Find supports two primary operations:

- **Find:** Determines the set to which a particular element belongs. This can also be used to check if two elements are in the same set.
- **Union:** Merges two disjoint sets into a single set. This operation is based on the principle that the two sets being merged should initially be disjoint.

These operations are typically implemented with optimizations such as *path compression*



Concept

Consider a scenario where we are given an array of edges, such as $\{[1,2],[4,1],[2,4]\}$. Each pair in this array represents an undirected edge connecting two vertices. For instance, vertex 1 is connected to vertex 2, and so forth.

Union-Find as a Forest of Trees: Initially, every vertex operates independently, forming its own set. In Union-Find terminology, each vertex initially points to itself, indicating that it is its own parent.

Connecting Components: To illustrate, when connecting vertex 2 to vertex 1, we may decide that vertex 2 will become the child of vertex 1. The selection of the parent in such cases is arbitrary unless the sets being united differ significantly in size or "rank" (a concept used to optimize Union-Find operations).

Union and Find Operations: Union-Find is primarily defined by two operations:

1. **Find:** This operation determines the "root" or the ultimate parent of a vertex. If two vertices have the same root, they belong to the same set.
2. **Union:** This operation merges two disjoint sets into a single set. It utilizes the Find operation to determine the roots of the vertices and then merges the sets by linking one root to the other.

These operations are critical in preventing cycles when building a graph. For instance, before connecting two vertices, the Find operation can check if they already share a root, which would indicate that connecting them would create a cycle.

Implementation Notes

When implementing Union-Find, especially for graph-related operations like cycle detec-

Implementation of Union-Find

UnionFind Class:

- **Data Structures:**
  - **Parent Map:** Tracks the parent of each node.
  - **Rank Map:** Tracks the "rank" of each tree in the forest, used to optimize union operations.
- **Initialization:**
  - Each node is its own parent initially.
  - Each node has an initial rank of 0.
- **Choice of Data Structure:**
  - Using hashmaps for flexibility and ease of use.
  - Arrays can also be used, especially when vertex identifiers are compact and sequential.

```
class UnionFind:
def __init__(self, n):
    self.par = {}
    self.rank = {}

    for i in range(1, n + 1):
        self.par[i] = i
        self.rank[i] = 0
```

Methods:

- **Find:** Path compression to keep trees flat.
- **Union:** Union by rank to ensure that the smaller tree is always attached under the larger tree, preventing deep trees.

To implement Union-Find, we can have a UnionFind class, using which we can instantiate our parent hashmap and our rank hashmap. Some people like to use arrays instead of hashmaps, and that is also a valid option.

Implementation of ‘Find’ in Union-Find

Purpose

Locate the root parent of a given vertex, employing path compression to optimize future lookups.

Implementation:

- Input: A vertex n.
- Output: The root parent of vertex n.

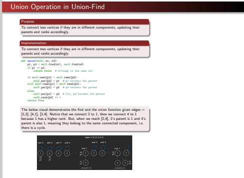
```python
def find(self, n):
    p = self.parent[n]
    while p != self.parent[p]:
        self.parent[p] = self.parent[self.parent[p]]  # Path Compression
        p = self.parent[p]
    return p
```

Path Compression:

- Reduces the height of the tree by making nodes directly point to their grandparent.
- Improves the efficiency of future ‘find’ operations.

Our find function will take a vertex, n, as an argument and return its parent. We can do this by using our parent hashmap where the key is the vertex and the value is the parent. If a vertex is a parent to itself, we can just return the vertex itself.

Path Compression As we are performing union on a large number of vertices, it can end up creating a pretty long chain, similar to a long linked list. Then, to determine if two nodes are part of the same component, it will take us a lot of steps. However, we can reduce the amount of these steps by traversing up two vertices at a time instead of one. This would mean that when we are going up the tree, the parent is actually the grandparent. This won’t have any improvements in performance the first time but if we ever encounter the same vertex again, we can retrieve the parent immediately. This is referred to as path compression.



Our find function will take a vertex, n, as an argument and return its parent. We can do this by using our parent hashmap where the key is the vertex and the value is the parent. If a vertex is a parent to itself, we can just return the vertex itself.

Path Compression As we are performing union on a large number of vertices, it can end up creating a pretty long chain, similar to a long linked list. Then, to determine if two nodes are part of the same component, it will take us a lot of steps. However, we can reduce the amount of these steps by traversing up two vertices at a time instead of one. This would mean that when we are going up the tree, the parent is actually the grandparent. This won't have any improvements in performance the first time but if we ever encounter the same vertex again, we can retrieve the parent immediately. This is referred to as path compression.

Kruskal's Algorithm Implementation

Implementation Strategy

- Utilize a minHeap to sort all edges by their weight to ensure we're always considering the edge with the minimum weight next.
- Directly work with edges, bypassing the need for an adjacency list, focusing on global minimum weights.

Union-Find Data Structure

- Initialize the UnionFind structure with each vertex pointing to itself.
- For each edge popped from the minHeap:
  - Attempt to union the vertices of the edge.
  - If union is successful (no cycle created), include the edge in the MST.
  - If a cycle would be formed, discard the edge.
- Continue until the MST contains $n - 1$ edges.

```python
    # Given an list of edges of a connected undirected graph,
# with nodes numbered from 1 to n,
# return a list edges making up the minimum spanning tree.
def minimumSpanningTree(edges, n):
    minHeap = []
    for n1, n2, weight in edges:
        heapq.heappush(minHeap, [weight, n1, n2])

    unionFind = UnionFind(n)
    mst = []
    while len(mst) < n - 1:
        weight, n1, n2 = heapq.heappop(minHeap)
        if not unionFind.union(n1, n2):
            continue
        mst.append([n1, n2])
    return mst
```

Key Implementation Note

The UnionFind class, used here, facilitates efficient cycle detection and set management.

Topological Sort: The Idea

The Idea:

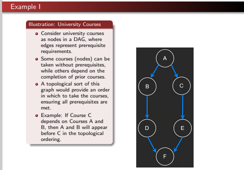
- Topological sort is a linear ordering of vertices in a directed acyclic graph (DAG), where for each directed edge uv, vertex u comes before v in the ordering.
- This sorting is particularly useful in scenarios where you need to understand the sequence of tasks or events, like prerequisites in university courses.
- In a DAG, if there's a path from vertex u to vertex v, u must appear before v in the order.
- It provides a way to execute tasks while respecting all precedence relationships.

Illustration: University Courses

- Consider university courses as nodes in a DAG, where edges represent prerequisite requirements.
- Some courses (nodes) can be taken without prerequisites, while others depend on the completion of prior courses.
- A topological sort of this graph would provide an order in which to take the courses, ensuring all prerequisites are met.
- Example: If Course C depends on Courses A and B, then A and B will appear before C in the topological ordering.

Key Points in Topological Sorting

- It's applicable only to DAGs. A cyclic graph does not have a valid topological ordering.
- The order is not necessarily unique. There can be multiple valid topological sorts for a given graph.
- Topological sorts are used in scheduling tasks, ordering compiler tasks, resolving symbol dependencies in linkers, etc.



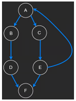
Topological Sort

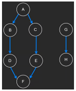
The idea Topological sort is a way of sorting a directed acyclic graph (DAG) such that each node comes before its dependent nodes. A simple example of this is university courses. There are some courses that can be taken without any pre-requisites and then there are those that have pre-requisites, i.e. you cannot take them unless you have taken other courses first.

In other words, some courses can be taken independent of other courses and others have to be taken in a specific order. We can represent this scenario using a DAG, where the edges represent the dependencies between the courses.

So, if we have node C and it has node A and B as its dependents, A and B will appear before C in the topological ordering. What order they appear in is not important unless A and B also have a dependency on each other.

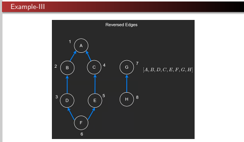
Example-II





Suppose we are given the following directed acyclic graph (DAG). The topological ordering for this graph would be A,B,C,D,E,F. Notice that each node appears before its dependent node.

This is a rather simple example. We mentioned previously that topological sort works on acyclic graphs. What if we had a cycle in our graph? Let's take a slight modification of the graph above and apply the same concept to it. In this case, we have an edge coming out of E, going into A. The order would be: E,A,B,C,D,E,F. This actually contradicts the idea of topological sort since it is not possible to have E before A, and also after A. This would be like saying to take course A, you must take course E first, but to take course E, you must take course A first - it is a cycle.



Even if there are no cycles allowed, topological sort will still work on disconnected graphs. If we have two connected components in a graph, the ordering in which we place the vertices of the individual disconnected components does not matter as they are independent of each other. The graph has two connected components and one possible valid ordering could be A,B,C,D,E,F,G,H.