Disjoint Set ADT

Efficient Algorithm to solve Equivalence problem

***Abstract*— One of the most important concepts of mathematics is Sets. Insets we can represent a group of elements and the order is not important for it. The disjoint set ADT is used for it. We can solve equivalence problems using this data structure. They are used to maintain a collection of non-overlapping sets of elements from a limited number of elements. Algorithms developed using this data structure are often called a UNION_FIND algorithm. In this paper, we will have a look a how the routines can be implemented using only a few lines of codes and how it requires only constant average time per operations. We will also have a look at few simple applications of disjoint sets, implementing it with a minimal effort and how can we increase the speed of the disjoint sets using two simple methodologies.**

Keywords—data structure, minimum spanning tree, disjoint set, abstract data type, union-find algorithm, Kruskal's algorithm

I. INTRODUCTION

Ordinarily, the effectiveness of an algorithm relies upon the data structure utilized in the algorithm. An insightful decision in the structure you use in taking care of an issue can lessen the season of execution, an opportunity to actualize the algorithm and the measure of memory utilized. Consider a problem where we want to design an algorithm to schedule a set of jobs on a single server and each job requires one unit of execution time t and has its own deadline d.

One way to construct such a schedule is to assign each job in turn to the latest available time slot prior to its deadline, provided there is such a time slot. But the challenge here is to find an efficient way of locating the latest available time slot prior to the deadline. One way to think about this problem is to partition the time slots into DISJOINT SETS -It's a collection of sets such that no two sets have any element in common.

For the above-given problem suppose, for example, that we have scheduled jobs in time slots 1, 2, 5, 7, and 8. Each set must have a single available time slot, which must be the smallest time slot in that set; Thus, elements 0, 3, 4, 6, and all elements greater than 8 must be in different sets If 10 is the latest deadline, our disjoint sets will, therefore, be {0, 1, 2}, {3}, {4, 5}, {6, 7, 8}, {9}, and {10}. If we then wish to schedule a job with deadline 8, we need to find the latest available time slot prior to 8. This is simply the first time slot in the set containing 8 — namely, 6. When we then schedule the job at time slot 6, the set {6, 7, 8} no longer contains an available time slot.

II. DATA STRUCTURES

A. Definition

Data structures enable optimization of the management of information in memory, and in many cases, data structures build on themselves to create new structures. Programmers can choose and adopt those structures that best fit the given data pattern. Choosing the most efficient data structure for the job significantly improves the performance of the algorithm, which further fastens application processing speeds. This enables computing systems to efficiently manage massive amounts of data within large-scale indexing, massive databases, and structured data in big data platforms.

Programmers and analytics team will not need to program a data structure as many standard libraries are already available to them. But understanding data structures will help them to choose the optimal and the best analytical toolsets which are feasible for the big data environment.

B. Types of data structures

Different types of data structures built on one another which includes primitive, simple and compound structures.

-PRIMITIVE DATA STRUCTURE/TYPES:
They are the basic building blocks of simple and compound data structures: integer, floats and doubles, characters, strings and Boolean.

-SIMPLE DATA STRUCTURE:
It is built on primitive data types to create higher level data structures like arrays and linked lists.

-COMPOUND DATA STRUCTURE:
They are built on primitive and non-simple data structures and can be linear or non-linear in nature-based whether they form a linear sequence. Stacks, Queue, Trees, Hash tables, Heaps, Graphs are its examples. It provides us with a greater level of sophistication.

III. DISJOINT SETS

A. Definition

A disjoint-set abstract data structure also known as a union-find data structure or a merge-find set is a data structure that can track a set of elements that are portioned into n number of disjoint or non-overlapping subsets. We can get near to constant time operations for adding a new set or to merge existing sets and to even determine if the elements are part of the same subset.

They can be implemented very easily, and we can use an array to implement them and coding it also takes a very little amount of effort.

B. Equivalence Relations and Classes

A relation R is defined on a set S if for every pair of elements (a, b), a, b $\in$ S, a R b is either true or false. If a R b is true, we say that a is related to b.

An equivalence relation is a relation R that satisfies three properties.

1. Reflexive: a R a is true for all a $\in$ S.
2. Symmetric: a R b if and only if b R a.
3. Transitive: a R b and b R c implies that a R c.

A union-find data structure, also known as a disjoint-set data structure, is a data structure that can keep track of a collection of pairwise disjoint sets S = {S1, S2, …. Sr} containing a total of n elements. Each set Si has a single, arbitrarily chosen element that can serve as a representative for the entire set, denoted as rep [Si].

Specifically, we wish to support the following operations:

- MAKE-SET(x), which adds a new set {x} to S with rep[{x}] = x.
- FIND-SET(x), which determines which set Sx $\in$ S contains x and returns rep[Sx].
- UNION(x, y), which replaces Sx and Sy with Sx $\cup$ Sy in S for any x, y in distinct sets Sx, Sy.

C. Background of Disjoint Sets

Bernard A. Galler and Michael J. Fischer were the first people who described the disjoint-set forests in 1964. The complexity of their data structure was bounded to an iterated logarithm of n by Hopcroft and Ullman in 1973. O(a(n)) which is an inverse Ackermann function was proved to be the upper bound to the algorithm's time complexity by Robert Tarjan in 1979. In 1989, the optimality of the solution was proved by Fredman and Saks by showing that W(a(n)) i.e. amortized words must be accessed by any disjoint sets.

Ackermann function is one of the simplest and early discovered examp0les of a total computable function. A function i.e. not primitive recursive. A function which is primitive recursive is total and computable, but the Ackermann function illustrates that they don't have to be primitive recursive to be a total computable function. The two-argument Ackermann–Péter function, is defined as follows for nonnegative integers m and n:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Fig 1: Ackermann Function Example

D. The idea behind the disjoint Set

Start with all values in their own sets:

{a} {b} {c} {d} {e} {f}

Label the sets for reference:

{a} {b} {c} {d} {e} {f}

s0 s1 s2 s3 s4 s5

Allowed operations: – union,

e.g. union(s0, s1) or union(a,b)

{a,b} {c} {d} {e} {f}

s0 s2 s3 s4 s5

– find, e.g.

find(b) returns s0

E. Operations to manipulate

Basic operations which we need to manipulate the operations defined on a set are found, union and make set.

- Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- Union: Join two subsets into a single subset.
- Makeset: Makes a set containing only a given element (a singleton).

Union-Find Algorithm can be used to check whether an undirected graph contains a cycle or not. With these three operations, many practical partitioning problems can be solved.

F. Strategies for Solving Union-Find Problem

~Simple solutions:

-Sets of sets.

-Union-Find - Quick Find Algorithm.

~Tree Solution:

-Union-Find - Quick Union Algorithm.

-Rank Union (aka Union by Size).

-Rank Union with Path Compression.

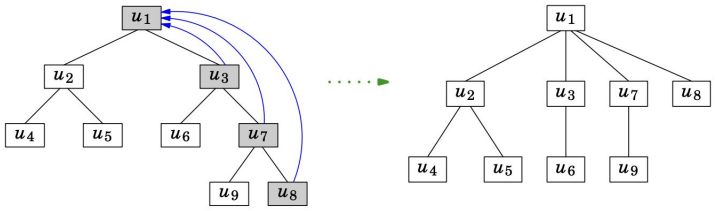

Fig 2: Path Compression

The following table gives us the analysis of each of these approaches when n = number of objects and m = number of operations (union or find) performed on those objects

	For M ops
Sets of Sets	O(mn)
Quick Find	O(mn)
Quick Union + Naive Find	O(mn)
Rank Union + Naive Find	O(n + m log n)
Quick Union + Path Compression	O(n + m log n)
Rank Union + Path Compression	O((m + n) (lg* n))

Fig 3: Analysis

G. Pseudo Code

The disjoint Set class can be implemented as follows:

```
    /**
     * Construct the disjoint sets object.
     * @param numElements the initial number of disjoint sets.
     */
    public DisjointSets( int numElements )
    {
        s = new int[ numElements ];
        for( int i = 0; i < s.length; i++ )
            s[ i ] = -1;
    }
/**
 * Union two disjoint sets using the height heuristic.
 * root1 and root2 are distinct and represent set names.
 * @param root1 the root of set 1.
 * @param root2 the root of set 2.
 * @throws IllegalArgumentException if root1 or root2
 * are not distinct roots.
 */
public void union( int root1, int root2 )
{
    assertIsRoot( root1 );
    assertIsRoot( root2 );
    if( root1 == root2 )
        throw new IllegalArgumentException( );

    if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
        s[ root1 ] = root2;        // Make root2 new root
    else
    {
        if( s[ root1 ] == s[ root2 ] )
            s[ root1 ]--;          // Update height if same
        s[ root2 ] = root1;        // Make root1 new root
    }
}
/**
 * Perform a find with path compression.
 * @param x the element being searched for.
 * @return the set containing x.
 * @throws IllegalArgumentException if x is not valid.
 */
public int find( int x )
{
    assertIsItem( x );
    if( s[ x ] < 0 )
        return x;
    else
        return s[ x ] = find( s[ x ] );
}
```

H. Implementation of a DISJOINT SET

1. USING AN ARRAY

This representation assigns one position for each element. Each position stores the element and an index to the representative. To make the Find-Set operation fast we store the name of each equivalence class in the array. Thus the find takes constant time, O(1).

2. USING A LINKED LIST

All the elements of an equivalence class are maintained in a linked list. The first object in each linked list is it is set's representative. Each object in the linked list contains a set member, a pointer to the object containing the next member of the set, and a pointer back to the representative. Each list maintains head pointer, to the representative, and tail pointer, to the last object in the list.

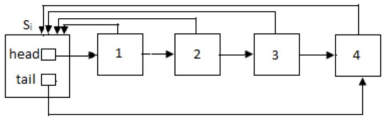

Figure 1.1 Linked list representation of equivalent set S_i

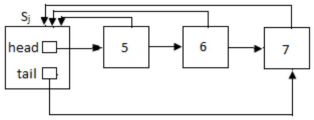

Figure 1.2 Linked list representation of equivalent set S_j

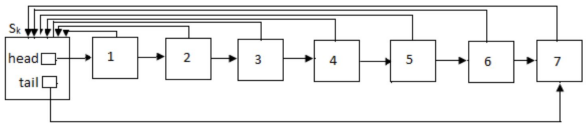

Figure 1.3 Linked list representation after union (S_i, S_j)

Fig 4: Linked List representation

3. Tree

A tree data structure can be used to represent a disjoint set ADT. Each set is represented by a tree. The elements in the tree have the same root and hence the root is used to name the set. Eg: Make-set (DISJ_SET S) After the Make-set operation, each set contains one element. The Make-set operation takes O(1) time Tree representation of disjoint set ADT after Make-set operation

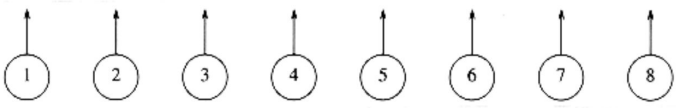

Fig 5: After performing Makeset operation

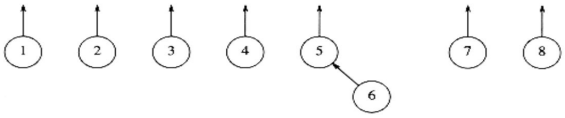

Figure 1.5 Tree representation of disjoint set ADT after union (5,6)

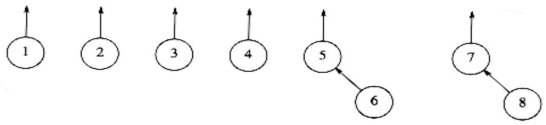

Figure 1.6 Tree representation of disjoint set ADT after union (7,8)

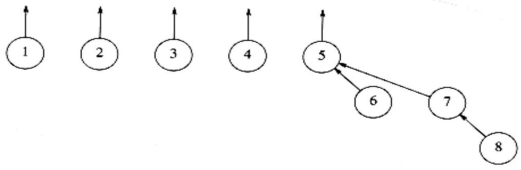

Figure 1.7 Tree representation of disjoint set ADT after union (5,7)

Fig 6: Performing union operations on a Tree

IV. APPLICATIONS

- Percolation.
- Image processing.
- Least common ancestor.
- Equivalence of finite state automata.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.

Kruskal Algorithm Kruskal's algorithm is a minimum-spanning-tree algorithm: finds an edge of the least possible weight that connects any two trees in the forest and performs a union.
Consider the graph which is given to us and the corresponding graph is its minimum spanning tree which we deduce after using Kruskal's algorithm.

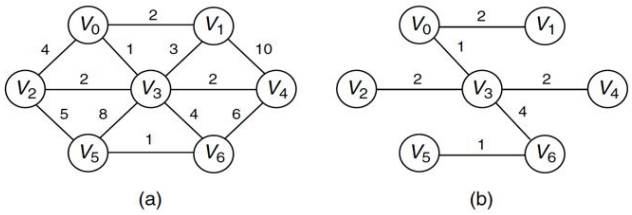

Fig 7: Graph for Kruskal's Example

The following steps are performed in order to get the minimum spanning tree. The first five edges are all accepted because they do not create cycles. The next two edges, ($v1$, $v3$) (of cost 3) and then ($v0$, $v2$) (of cost 4), are rejected because each would create a cycle in the tree. The next edge considered is accepted, and because it is the sixth edge in a seven-vertex graph, we can terminate the algorithm.

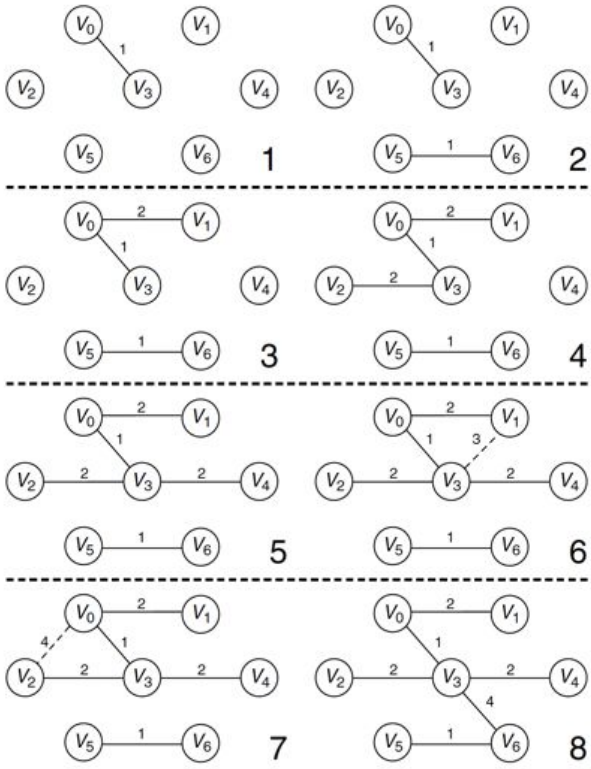

Fig 8: Steps performed to get a minimum spanning tree

VI. CONCLUSION

Here we thus discussed an ADT which can use simple data structures to maintain disjoint sets when the union operation is performed, it does not matter, as far as correctness is concerned, which set retains its name. A valuable lesson that should be learned here is that considering the alternatives when a particular step is not totally specified can be very important. The union step is flexible. By taking advantage of this flexibility, we can get a much more efficient algorithm.

Path compression is one of the earliest forms of self-adjustment, which we have used elsewhere (splay trees and skew heaps). Its use here is extremely interesting from a theoretical point of view because it was one of the first examples of a simple algorithm with a not-so-simple worst-case analysis.

VII. REFERENCES

1. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/recitation-notes/MIT6_046JS15_Recitation3.pdf
2. https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf
3. https://docs.rs/disjoint-sets/0.4.2/disjoint_sets/
4. Book- Data Structures and Problem Solving Using Java
5. https://en.wikipedia.org/wiki/Disjoint-set_data_structure
6. https://en.wikipedia.org/wiki/Ackermann_function