Pakistan Journal of Information and Technology 2 (3): 240-249, 2003
ISSN 1682-6027
© 2003 Asian Network for Scientific Information

An Algorithm to Find Equivalence Classes

Malik Sikander Hayat Khiyal, Muhammad Arshad Zia, Khadija Riaz and Najia Khalid
Department of Computer Science, International Islamic University, Islamabad, Pakistan

Abstract: The purpose of this algorithm is to Find the equivalence class of a particular element in a set and all disjoint classes of a set. The algorithm works on an equivalence relation.

Key words: Discrete Structures, Equivalence Relation, Equivalence Classes, Algorithms.

The problem

The problem is to Find the equivalence classes for a given *equivalence relation* on a set. The concept of equivalence classes is used to partition a set into classes. Two elements are considered to be equivalent if they are in the same class. Equivalent elements form a partition over a set, no element belongs to two equivalence classes and every element belongs to a class. A binary relation that is reflexive, transitive and symmetric is called an equivalence relation. Given a set S of objects and an equivalence relation R over the elements of S, we can partition S into mutually disjoint subsets such that two elements a and b of S fall into the same subset if and only if a R b.
Thus we handle two problems in our algorithm:

Class of an element

Giving an element of a set, return the class of that element. The equivalence classes of any two elements should be same or disjoint.

Disjoint classes

Show all the classes of a set and verify that all classes are disjoint and have distinct elements.

Introduction

There is a family of algorithms for maintaining equivalence classes. Two of the most crucial operations that should be supported by such an algorithm are Find ("Find the equivalence class of an element a") and Union ("unite two equivalence classes"). Therefore, these algorithms are often referred to as Union-Find Algorithms. In this paper we compare our algorithm with two well known Union-Find Algorithms.

We have joined the best qualities from these algorithms into one algorithm. Our algorithm is also based Union-Find operations. We use ADT (abstract data type) the APVECTOR in our algorithm. The use of APVECTOR makes our algorithm much simpler and easier to understand

because it seems like an array in processing, but has no disadvantage of array, like wastage of memory or fix boundary in advance. Suppose there is a set A={1,2,3,4}. The numbers 1,2,3... are representing any kind of object relating with each other through a relation. Initially, all elements are stored in a vector (Fig. 1).

A	i
1	0
2	1
3	2
4	3

Fig. 1: Vector

On the bases of the fact that equivalence relations are reflexive and symmetric, we process only one relation from among the three {(a,a) (a,b) (b,a)} i.e a,b. When pairs of equivalence relations are taken one by one, it is checked whether two elements are same or one is greater than the other (i.e. a=b or a>b). If this condition is true another relation is asked to enter and no processing (Find or Union) is done on above two types of the relations.

Similarly, we presume that if someone enters a relation (a,b), he must enter (b,a). So, the relation (b,a) is neglected and processing is only done for the relation (a,b). It is because, if elements have been connected once there is no need to consult the relations which do the same thing again. The relation (a,a) is also not processed as each element belongs to its own class.

Find ( )

After a relation is found in which two elements are not already connected, Find procedure is called. We implement Find(a) by simply searching that element in the vector and then storing its index. The indexes of a and b in ADT vector are found. Suppose a=2 and b=3, the index of a is "ina=1" and that of b is "inb= 2". It means "a" is located at position "ina" of the vector and "b" is located at position "inb". If there are "n" objects, complexity of the Find() operation is constant i.e O(l). "l" is position of that element in the vector.

Union ( )

In order to unite two classes, four conditions are checked for the elements of relation entered.

- None of the elements of the relation have been entered before.
- Element "a" of the relation is part of any class, while "b" is not.
- Element "b" of the relation is part of any class, while "a" is not.
- Both elements of the relation already occur in different classes and now both are relating with each other in recent relation.

Depending upon the condition met, the two classes are joined to form a single class. In the end all the elements belonging to the one class point towards the same subset. Suppose for a set A={1,2,3,4}, user enters following relations:

R= {(1,1)(1,2)(2,1)(2,2)(2,3)(3,2)(3,3)(4,4)}

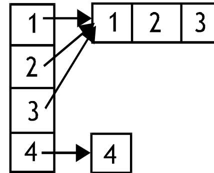


Fig. 2: Output

The distinct classes are {1,2,3} and {4}.

Analysis of algorithms

We compare our algorithm with two Union-Find Algorithms of the equivalence classes.

- List Algorithms
- Tree Algorithms

First is made by using link list while the second uses Tree processing respectively. We developed our algorithm using ADT class APVECTOR. By using ADT class we are able to save the memory. Actually, APVECTOR removes the complexity of using the pointers and makes the algorithm simple. In our algorithm, there is least memory used and minimum time consumption.

List algorithms

The nodes of an equivalence class are maintained in a linked list. To make the implementation of the Union operation faster, the linked list is circular. Each element of the list contains an additional pointer to the representative node of the equivalence class (including the representative itself). The lists are depicted as trees with a depth of at most 1. The root of the tree is the representative element and the leaves of the tree (all direct children of the root) are the other nodes in the equivalence class. The list algorithm is also sometimes known as the "Quick Find" algorithm, since the Find operation on this data structure takes O(1) time.

The Union operation is implemented by making all the "root" fields of the nodes in one class point to the representative of the other class and then splicing the list of nodes of one class into the other. If no Heuristics are used then the time complexity of Union operation is O(n^2). And by using Union By Rank the time complexity of Union operation is O(n log(n))

In Fig. 3 the field "R" of list is showing representative of each element while the field "D" is data field. There are two equivalence classes {{1,3,6}{2,4,5}}.

R	1	2	1	2	2	1
D	1	2	3	4	5	6

Fig. 3: List algorithm

Tree algorithms

The nodes of an equivalence class are maintained in a tree and the root of the tree is the class's representative. The Find operation is implemented by walking up the tree to the root. The Union operation is implemented by Finding the representative elements of the two arguments and then making one representative the child of the other. Notice that with this algorithm, it is the Union operation that takes constant time and the Find operation which can be costly: in the worst case, the Find operation takes time proportional to the height of the deepest tree. To process each equivalence pair, we need to perform two Find and at most one Union. Thus, if we have n objects of a set and m equivalence pairs, we need to process 2m Finds and at most min(n-1,m) Unions. (Note that after n-1 Unions all n objects will be in the same equivalence class and no more Unions can be performed. See Fig. 4).

Comparison

Time complexity and memory

In List Algorithm Find operation take a constant time. An element is directly searched from the list and it is found that where that element in the list exists. Find operation in our algorithm is similar to that of list algorithm in which we store the indices of both elements of a relation. While the Union operation of list algorithm is costly. Here the "R" (representative) field of all the elements of one equivalence class is changed to the representative of the other class. In the Tree algorithm there is reverse situation. The Find operation is costly while the Union operation is efficient than those of the list algorithm. The Find operation may be severe with increase of the tree length. While in the Union operation the root of one tree is made child of the other tree. The Union operation in our algorithm is similar to the Union operation of the Tree algorithm. We just connect one equivalence class at the end of other class and then all the elements of resultant class in the vector point to their equivalence class. Besides these operations there is another advantage in our algorithm, we do not process all the relations entered by the user. Only selective pairs undergo processing. We know that relations entered by the user are reflexive, i.e a=b, symmetric, i.e (a,b) then (b,a) and transitive. For the relations which we do not process we even do not call Find procedure. But it happens in both of other algorithms, though already connected elements are not united by Union operation but the Find operations are called.

Table 1 state that in the List algorithm when two elements of a relation are not already connected both Union and Find operations are performed, if they are already connected only Find operation is performed. In tree algorithm same is happened for non connected elements but for connected elements of a relation Find operation has to perform to check their connectivity. As Find in tree algorithm is costly so in this way it shows bad performance than list

algorithm and our algorithm. In our algorithm we perform these two operations only for non connected pairs in which a<b. Thus our algorithm is taking Find operation from List algorithm and Union operation from tree algorithm so the time complexity of our algorithm is less then both of them. When we consider memory tree algorithm is best of all. But it causes that Find operation becomes inefficient in Tree algorihm. Memory of our algorithm is less then that of List algorithm. Because list algorithm always stores a representative to tell about the class to which that element belongs. But in our algorithm if an element does not belong to any class then no extra node is created. Besides this, memory is almost similar to List algorithm.

Table 1: Relations and their corresponding processing in Algorithms

Relation	Our algorithm	List algorithm	Tree Algorithm
a=b	none	none	none
a,b	none	both Union and Find	both Union and Find
b,a	both Union and Find	only Find	only Find

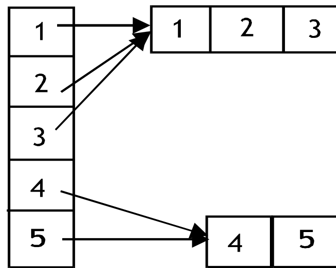


Fig. 3: Our algorithm with equivalence classes, {{1,2,3}{4,5}}

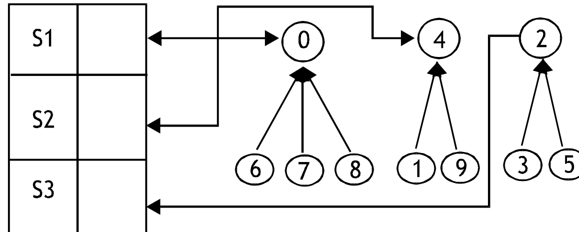


Fig. 4.1: Tree algorithms, Data representation

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Parent	-1	4	-1	2	-1	2	0	0	0	4

Fig. 4.2: Tree Algorithms, array representation of S1,S2 and S3

Algorithm: In this algorithm vect is a vector of link list. Each cell of vector has two parts: one part contains data and the other part contains address of its equivalence class. vect is filled with

all set elements in its data portion while in address part NULL is stored. If set elements are 1,2,3,4 then the vector "vect" is given below.

VECT

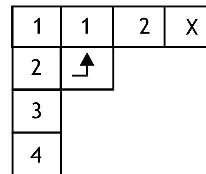
0	1	X
1	2	X
2	3	X
3	4	X

Its length is n. "a" and "b" are elements of the relation entered by the user. Relation entered by user is send to this function as an argument. ina and inb store the values of indexes of elements a and b.

```
Equivalence (a,b)
{
int                 ina, inb;
link                *cur,*newn;

// first of all check whether a = b or a > b because we will process on one of three
//relations(a,a),(a,b),(b,a)

step 1: -           if (a= =b or a>b)
                        return
// then Find indexes of a and b in vector

step 2: -
                    call  Find(a,b)

step 3: -
                    call Union ( )

}
void Find ( a,b)

{
continue  loop for I= 0,1,2,3…..n

                    if  (vect[i]->data = = a)
                        ina = I
                    if  (vect[i]->data = = b)
                        inb = I
```

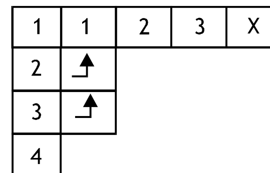
```
end loop

}

void union ( )

{

1st condition:
      //First condition is true  when both elements a and b do not belong to any class.

if ((vect[ina]->next= =null) and (vect[inb]->next= = null))

      newn= new link
      newn->data= b
      newn->next= null

      cur= new link
      cur->data= a
      cur->next=newn
      vect[ina]->next=cur
      vect[inb]->next= vect[ina]->next

end  if
```

suppose a=1 and b=2




```
2nd Condition
// second condition is true if "a" belongs to any class while "b" does not

else if ((vect[ina]->next != null) and (vect[inb]->next = = null))

            cur = vect[ina]->next

            loop  while cur->next!= null

            cur=cur->next

      end loop

      newn =new link
      newn->data=b;
```

Suppose (1,3) is relation after (1,2)



```
    newn->next=null
    cur->next=newn
    vect[inb]->next=vect[ina]->next

end  else if
```

3rd condition:

```
// third condition is true if "b" belongs to any class while "a" does not  belong to any one.
 // suppose user enters first 2-3 and then 1-2
```

(2,3) is first relation entered

1	X		
2	2	3	X
3	↑		
4	X		

```
else if  ((vect[ina]->next= =null) and (vect[inb]->next! =null))
```

Now user enters 1-2
It means "b" is already in a class while
"a" is not in any class

1	↓			
2	2	3	1	X
3	↑			
4				

```
cur = vect[inb]->next

loop while cur->next!= null
        cur=cur->next
end loop
        newn =new link
        newn->data=a;
        newn->next=null
        cur->next=newn
        vect[ina]->next=vect[inb]->next

end  else if
```

4th condition:

```
/* suppose user has   entered 1-2 then he entered 3-4, it means  there are two disjoint classes.
Now he enters 2-3 so the equivalence class will be made by joining of  the both subclasses*/
```

1	1	2	X
2	↑		
3	3	4	X
4	↑		

```
else if ( ((vect[ina]->next! =null) and (vect[inb]->next! =null)) and (vect[ina]->next!= vet[inb]->next))

// store ptr in remaining elements of the class of "b"

    cur=vect[inb]->next

    do

    cur=cur->next
    loop for I=0,1,2....n
        if  vect[i] -> data= cur-> data
                vect[i]->next=vect[ina]->next
                break
        end  if
    end  for loop

while (cur->next!= null)

// now join two equivalent classes

    cur= vect[ina]
    while (cur->next != null)
        cur=cur->next
    end while loop
    cur->next=vect[inb]->next
    vect[inb]->next=vect[ina]->next
end  else if
}
```

the result of this step is

1	1	2	3	X
2	↑			
3	↑			
4	↑			

End of Algorithm

Our algorithm provides a successful and reliable mechanism for finding equivalence class of an element and all disjoint classes of a set. In this algorithm presented in this paper we have used numbers to show set elements. But in general these elements can be any type of object and there can be any relation.

Acknowledgement

The authors wish to thank Almighty Allah who enabled them to do such type of research work. We are grateful to Mr. Nadeem, for his technical help in writing the program in C++. We are also grateful to Dr. Khalid Rashid for his moral support and encouragement.

References

Ellis-Horowitz, Dinesh Mehta and Sartaj Sahni, 1995. Fundamentals of Data Structures in C++.

Thomas A. Standish, 1997. Data Structures, Algorithms and Software Principles.

Narsingh Deo, 1996. Graph Theory with applications to engineering and computer science.

Stephen B.Maurer & Anthony Ralston, 1991. Discrete Algorithmic Mathematics. Corniel's Ph.D. Thesis. [11-9] and [11-12].