MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE -5162 Program Lab (ADSAL)

Mini Project Report On:

DISJOINT SET

Submitted by:

Bijay Regmi

210913032

M-Tech(CSE)

Under the guidance of:

Mr. Gururaj

Assistant Professor -Selection Grade

Department of Computer Science & Engineering

DESCRIPTION

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. We identify each set by a representative, which is some member of the set. As in implementation of dynamic-set we always represent each element of a set by an object. Letting x denote an object, it supports the following operations:

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x , y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of *UNION* specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y , removing them from the collection S. In practice, we often absorb the elements of one of the sets into the other set.

FINDSET(x) returns a pointer to the representative of the (unique) set containing x.

Let n be the number of *MAKE-SET* operations, and m, the total number of *MAKE-SET*, *UNION*, and *FIND-SET* operations. Since the sets are disjoint, each *UNION* operation reduces the number of sets by one. After $n - 1$ *UNION* operations, therefore, only one set remains. The number of UNION operations is thus at most $n - 1$. Note also that since the *MAKE-SET* operations are included in the total number of operations m, we have $m \geq n$. We assume that the n *MAKE-SET* operations are the first n operations performed.

LINKED LIST REPRESENTATION OF DISJOINT SET

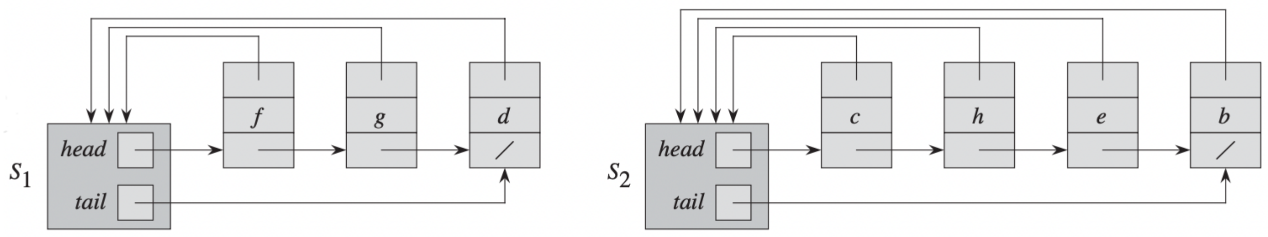


Fig 1. Linked-list representations of two sets S_1 and S_2.

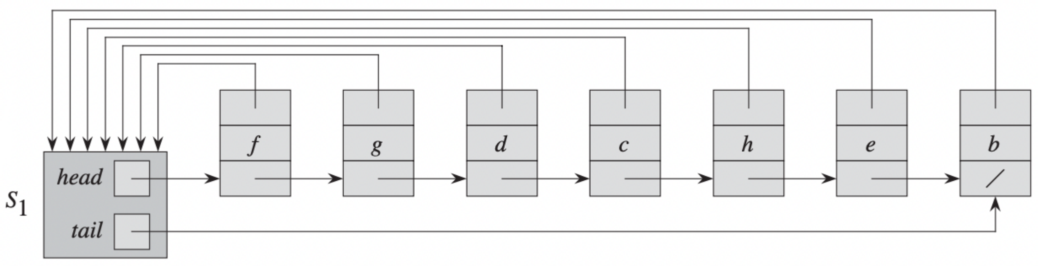


Fig. 2 *UNION* of set S_1 and S_2

Figure 1 represents set S_1 contains members *d* , *f* , and *g*, with representative f , and set S_2 contains members *b, c, e*, and *h*, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers head and tail to the first and last objects, respectively.

Figure 2 The result of *UNION(g , e)*, which appends the linked list containing e to the linked list containing g. The representative of the resulting set is *f* . The set object for *e*'s list, S_2, is destroyed.

ALGORITHM

```
CONNECTED-COMPONENTS(G)
1    for each vertex $v \in G.V$
2        MAKE-SET(V)
3    for each vertex $(u , v) \in G.E$
4        if FIND-SET(u) $\neq$ FIND-SET(v)
5            UNION(u , v)
```

```
SAME-COMPONENT(u , v)
1    if FIND-SET(u) == FIND-SET(v)
2        return TRUE
3    else return FALSE
```

```
MAKE-SET(x)
1    Create empty set $S$
2    Insert $x$ to $S$
3    return $S$
```

```
UNION($S_x$ , $S_y$)
1    for each y $\in S_y$
2        if $y \notin S_x$
3            Insert $y$ to $S_x$
4    Make $S_y$ empty set
5    return $S_x$
```

```
FIND(x)
1    for each $S$ in $G.S$
2        if $x \in S$
3            return $S$
```

ANALYSIS

The simplest implementation of the *UNION* operation using the linked-list set representation takes significantly more time than *MAKE-SET* or *FIND-SET*. We can easily construct a sequence of m operations on *n* objects that requires Θ*(n*^{2}*)* time. Suppose that we have objects $x_1, x_2, \ldots, x_n$. We execute the sequence of *n MAKE-SET* operations followed by *n - 1 UNION* operations shown in Figure 3, so that *m=2n-1*. We spend Θ*(n)* time performing the *n MAKE-SET* operations. Because the *i*th *UNION* operation updates *i* objects, the total number of objects updated by all *n -1 UNION* operations is

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

Fig 3. A sequence of *2n - 1* operations on *n* objects that takes Θ(*n*^{2}) time, or Θ(*n*) time per operation on average, using the linked-list set representation and the simple implementation of *UNION*.

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

The total number of operations is *2n - 1*, and so each operation on average requires Θ(*n*) time. That is, the amortized time of an operation is Θ(*n*).

A WEIGHTED-UNION HEURISTIC

In the worst case, the above implementation of the *UNION* procedure requires an average of Θ(*n*) time per call because we may be appending a longer list onto a shorter list; we must update the pointer to the set object for each member of the longer list. Suppose instead that each list also includes the length of the list (which we can easily maintain) and that we always append the shorter list onto the longer, breaking ties arbitrarily. With this simple ***weighted-union heuristic***, a single *UNION* operation can still take Ω*(n)* time if both sets have Ω*(n)* members. A sequence of *m MAKE-SET*, *UNION*, and *FIND-SET* operations, *n* of which are *MAKE-SET* operations, takes **O*(m + n* lg *n)*** time.

Each *UNION* operation unites two disjoint sets, we perform at most *n - 1 UNION* operations over all. We now bound the total time taken by these *UNION* operations. Consider a particular object *x*. We know that each time *x*'s pointer was updated, *x* must have started in the smaller set. The first time *x*'s pointer was updated, therefore, the resulting set must have had at least 2 members. Similarly, the next time *x*'s pointer was updated, the resulting set must have had at least 4 members. Continuing on, we observe that for any $k \leq n$, after x's pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least *k* members. Since the largest set has at most *n* members, each object's pointer is updated at most $\lceil \lg n \rceil$ times over all the *UNION* operations. Thus the total time spent updating object pointers over all *UNION* operations is O*(n* lg *n)* .We must Θ(1) also account for updating the tail pointers and the list lengths, which take only time per *UNION* operation. The total time spent in all *UNION* operations is thus O*(n* lg *n).*

The time for the entire sequence of *m* operations follows easily. Each MAKE- SET and FIND-SET operation takes O(1) time, and there are O(*m*) of them. The total time for the entire sequence is thus **O*(m + n* lg *n).***

EXAMPLE

Undirected graph given in the Figure 4 has vertices *V* = *{a, b, c, d, e, f, g, h, i, j, k}* and edges *E* = *{(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e)}*.

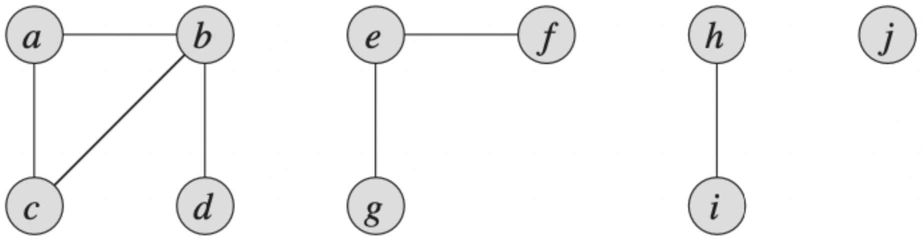


Fig. 4 Undirected graph with four connected components *{a, b, c, d}*, *{e, f, g}*, *{h,i}*, *{j}*

Edge processed	Collection of disjoint sets									
initial sets	{*a*}	{*b*}	{*c*}	{*d*}	{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*b*,*d*)	{*a*}	{*b*,*d*}	{*c*}		{*e*}	{*f*}	{*g*}	{*h*}	{*i*}	{*j*}
(*e*,*g*)	{*a*}	{*b*,*d*}	{*c*}		{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*a*,*c*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*}	{*i*}	{*j*}
(*h*,*i*)	{*a*,*c*}	{*b*,*d*}			{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*a*,*b*)	{*a*,*b*,*c*,*d*}				{*e*,*g*}	{*f*}		{*h*,*i*}		{*j*}
(*e*, *f*)	{*a*,*b*,*c*,*d*}				{*e*, *f*,*g*}			{*h*,*i*}		{*j*}
(*b*,*c*)	{*a*,*b*,*c*,*d*}				{*e*, *f*,*g*}			{*h*,*i*}		{*j*}

Fig. 5 The collection of disjoint sets after processing each edge.

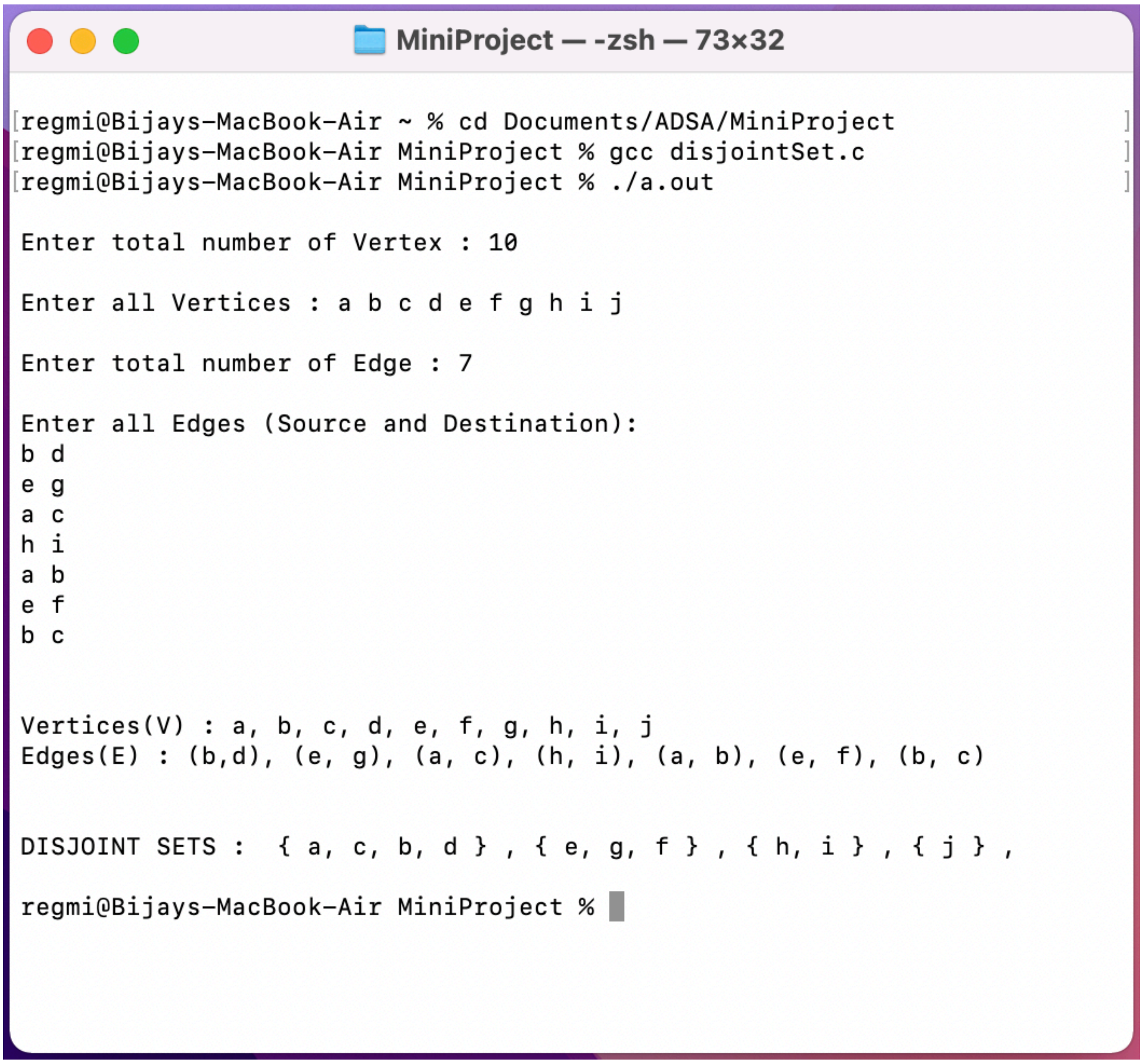
SOURCE CODE

```c
#include<stdio.h>
#include<stdlib.h>

#define MAX 20

// Structure of Edge
typedef struct Edge{
    char source, destination;
}edge;

// Structure of Node
typedef struct Node{
    char data;
    struct Node *next;
}node;

// Structure of Set
typedef struct Set{
    node *head;
    int cardinal;
}set;

// Finds whether key(int) is present in s(set) or not
node* findData(set *s, char key){
    if(s->cardinal == 0)
        return NULL;
    node *temp = s->head;
    while(temp){
        if(temp->data == key)
            return temp;
        temp=temp->next;
    }
    return NULL;
}

// Adds Item n to the Set s
void addItem(set *s, int data){
    node *n = (node *)malloc(sizeof(node));
    n->next = NULL;
    n->data = data;
    if(s->head == NULL){
        s->head = n;
        s->cardinal += 1;
        return;
    }
    node *temp = s->head;
    while(temp->next){
        temp = temp->next;
    }
    temp->next = n;
    s->cardinal += 1;
}

// Union of Set s1 and s2
set * unionOperation(set *s1, set *s2){
    if(s2->cardinal == 0)
        return s1;
    node *temp2 = s2->head;
    // Traversing through Set 1
    while(temp2){
        if(findData(s1,temp2->data) == NULL)
            addItem(s1,temp2->data);
        temp2 = temp2->next;
    }
    return s1;
}

// Make a new set for given Vertex
set * makeSet(char data){
    set *s = (set*)malloc(sizeof(set));
    node *newNode = (node*)malloc(sizeof(node));
    newNode->data = data;
    newNode->next = NULL;
    s->head = newNode;
    s->cardinal = 1;
    return s;
}

// Returns sets from allSets[MAX]
int findSet(set *allSets[MAX], int noOfVertices, char key){
    for (int i=0; i<noOfVertices; i++){
        if(findData(allSets[i],key))
            return i;
    }
    return -1;
}

// Prints set
void printSet(set *s){
    if(s->cardinal == 0){
        printf(" {} ");
        return;
    }
    node *temp = s->head->next;
    printf(" { %c",s->head->data);
    while(temp){
```

```c
            printf(", %c", temp->data);
            temp=temp->next;
        }
        printf(" } ");
}

// Makes s(set) Empty
void makeEmpty(set *s){
        node *temp = s->head, *temp1;
        while(temp){
            temp1=temp;
            temp = temp->next;
            free(temp1);
        }
        s->cardinal = 0;
        s->head = NULL;
}

// Finds index of the vertes form all vertices
int findIndex(char *vertices, int n, char key){
        for(int i=0; i<n; i++){
            if(key == vertices[i]){
                return i;
            }
        }
        return -1;
}

//Main Function
int main(){
        // Taking input for vertices and edges
        int noOfVertices = 10, noOfEdges = 6, i, j;
        printf("\nEnter total number of Vertex : ");
        scanf("%d",&noOfVertices);
        char *vertices = (char*)malloc(noOfVertices * sizeof(char));
        set *allSets[MAX];
        printf("\nEnter all Vertices : ");
        for(i=0; i<noOfVertices; i++){
            scanf("%s",&vertices[i]);
            allSets[i] = makeSet(vertices[i]);
        }
        printf("\nEnter total number of Edge : ");
        scanf("%d",&noOfEdges);
        edge *edges = (edge*)malloc(noOfEdges * sizeof(edge));
        printf("\nEnter all Edges (Source and Destination):\n");
        for(i=0; i<noOfEdges; i++){
            scanf("%s",&edges[i].source);
            scanf("%s",&edges[i].destination);
        }
```

```
    // Printing vertices and edges
    printf("\n\nVertices(V) : %c",vertices[0]);
    for(i=1; i<noOfVertices; i++)
        printf(", %c",vertices[i]);
    printf("\nEdges(E) : (%c,%c)", edges[0].source,
    edges[0].destination);
    for(i=1; i<noOfEdges; i++){
        printf(", (%c, %c)", edges[i].source, edges[i].destination);
    }

    int sIndex, dIndex;
    // Looping through all the edges
    for(i=0; i<noOfEdges; i++){
        sIndex = findIndex(vertices, noOfVertices, edges[i].source);
        dIndex = findIndex(vertices, noOfVertices,
         edges[i].destination);
        if(allSets[sIndex]->cardinal == 0){
            sIndex = findSet(allSets, noOfVertices, edges[i].source);
        }
        if(allSets[dIndex]->cardinal ==0){
            dIndex = findSet(allSets, noOfVertices,
             edges[i].destination);
        }
        unionOperation(allSets[sIndex], allSets[dIndex]);
        if(sIndex != dIndex)
            makeEmpty(allSets[dIndex]);
    }

    // Printing all non empty sets
    printf("\n\n\nDISJOINT SETS : ");
    for(i=0; i<noOfVertices; i++){
        if(allSets[i]->cardinal != 0){
            printSet(allSets[i]);
            printf(",");
        }
    }
    printf("\n\n");
    return 0;
}
```

INPUT/OUTPUT



```
regmi@Bijays-MacBook-Air ~ % cd Documents/ADSA/MiniProject
regmi@Bijays-MacBook-Air MiniProject % gcc disjointSet.c
regmi@Bijays-MacBook-Air MiniProject % ./a.out

Enter total number of Vertex : 10

Enter all Vertices : a b c d e f g h i j

Enter total number of Edge : 7

Enter all Edges (Source and Destination):
b d
e g
a c
h i
a b
e f
b c


Vertices(V) : a, b, c, d, e, f, g, h, i, j
Edges(E) : (b,d), (e, g), (a, c), (h, i), (a, b), (e, f), (b, c)


DISJOINT SETS :  { a, c, b, d } , { e, g, f } , { h, i } , { j } ,

regmi@Bijays-MacBook-Air MiniProject %
```