RAFAELA CARNEIRO
MATRÍCULA 2011483

① Código com a implementação das funções e comentários na próxima página.

(E)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 32 // definindo tamanho maximo de um mapa de bits do tipo int

struct set{
    int size;
    int members;
};
typedef struct set Set;

void setInsert(Set* set, int i){
    if(i >= MAX_SIZE || i < 0){
        printf("%d fora do range do conjunto\n",i);
        return;
    }
    i = 1 << i;     // int com apenas o iesimo bit aceso
    set->members = set->members | i;    // acende o iesimo bit do mapa de bits
}

int setIsMember(Set* set, int i){       // retorna 1 se i é membro e zero caso contrário
    if(i >= MAX_SIZE || i < 0){
        printf("%d fora do range do conjunto\n",i);
        return 0;
    }
    if(!set || set->members == 0){      // se o set for null ou tiver todos os bits apagados, com certeza o elemento i nao faz parte do set
        printf("mapa vazio\n");
        return 0;
    }
    return (1 << i & set->members);
}

Set* setIntersection(Set* set1, Set* set2){
    if(!set1 || !set2){ // se um dos mapas for null retorna null
        return NULL;
    }
    Set* intersec = (Set*) malloc(sizeof(Set));
    intersec->size = set1->size;
    intersec->members = set1->members & set2->members;
    return intersec;
}

Set* setDifference(Set* set1, Set* set2){
    if(!set1){
        return NULL;
    }
    if(!set2 || set2->members == 0){        // se set2 for null ou tiver todos os bits apagados retorna o proprio set1
        return set1;
    }
    Set* res = (Set*) malloc(sizeof(Set));
    res->size = set1->size;
    res->members = set1->members ^ set2->members;       // tira os elementos de set1 que tb tem no set2; mas vai acender os bits que tem no set2 e nao tem no set1
    res->members = res->members & set1->members;        // apaga os que nao tinham antes no set1
    return res;
}

int main(){

    return 0;
}


```

② considerando a estrutura de união-busca da forma.

```
struct ub {
    int n; // qtde de elementos
    int *v; // vetor
};
```

Onde os elementos da estrutura são os índices do vetor e cada um deles aponta para o seu pai e a raiz aponta para -(quantidade de nós da partição).

a) Enquanto ub->v[u] >= 0, o valor de ub->v[u] é o valor do pai de u. Então atribui o valor do pai de u a u e vai verificando o valor de ub->v[u] até esse valor ser negativo (indicando que chegamos na raiz); logo o retorno u é a raiz do conjunto.

b) enquanto x não for o representante do conjunto (raiz), indica o índice x do vetor apontando para o representante do conjunto e atualiza os índices do vetor (elementos do conjunto) apontando para o representante do conjunto (raiz) ao invés de deixá-los apontando para os seus pais

c) código na próxima página.

explicação do código:

a função recebe a estrutura de união-busca ub e dois inteiros, u e v, representando duas partições de ub. primeiro uso a função de busca com u e v para achar as raízes deles;

depois verifico se as raízes são iguais; se sim, eles

pertencem a mesma partição e retorno a raiz de um deles;
depois, como a raiz aponta para -(quantidade de nós que tem nessa partição), verifico se qtde de nós em u é menor que em v; se sim, penduro u em v. Para isso, atualizo a qtde de nós em v com (qtde de nós em v) + (qtde de nós em u);
coloco u apontando para v (que agora é seu pai) e retorno v (que é a raiz da união);
se qtde de nós em u não é menor que qtde de nós em v, penduro v em u (os passos são os mesmos que fiz antes, só que agora é v em u ao invés de u em v).
Código da ub_união:

```
1   int ub_uniao (UniaoBusca* ub, int u, int v){
2       u = ub_busca(ub, u);         // u = raiz da particao que u faz parte
3       v = ub_busca(ub, v);         // v = raiz da particao que v faz parte
4       if(u == v)                   // se as raizes sao iguais entao u e v fazem parte
        da mesma particao
5           return u;
6       if (ub->v[u] > ub->v[v] ) { // v[u] tem valor absoluto menor; vamos pendurar u
        em v
7           ub->v[v] += ub->v[u];    // soma os nós de u em v
8           ub->v[u] = v;            // v é a raiz da uniao e pai de u
9           return v;                // retornando a raiz
10      }
11      else { // v tem menos ós que u
12          ub->v[u] += ub->v[v]; //soma os nós de v em u
13          ub->v[v] = u;            // u é a nova raiz e pai de v
14          return u;                // retorna a raiz da uniao
15      }
16  }
17
18
```

RAFAELA CARNEIRO
MATRÍCULA: 2011483

(3) a)

PASSO 1: todas as distâncias são infinito

PASSO 2: coloca dist 0 no 0, 6 no 1, 9 no 2, 15 no 3, 23 no 5 e marca o vértice 0 como visitado

PASSO 3: visitando o vértice 1 (é o que tem a menor distância entre os não visitados). coloca distância 26 no 4 e marca o 1 como visitado

PASSO 4: visitando o 2: coloca dist 22 no 5, 37 no 8 e marca o 2 como visitado

PASSO 5: visitando 3: coloca dist 19 no 4, 22 no 6 e marca o 3 como visitado;

PASSO 6: visitando o 4: marca o 4 como visitado

PASSO 7: visitando o 5 (poderia ser o 6 porque os dois têm a mesma distância): marca o 5 como visitado

PASSO 8: visitando o 6: coloca distância 37 no 7 e marca o 6 como visitado

PASSO 9: visitando o 7: marca o 7 como visitado

PASSO 10: visita o 8 e marca ele como visitado

3) a)

DISTÂNCIAS

	0	1	2	3	4	5	6	7	8
✓0	∞	0	0	0	0	0	0	0	
✓1	∞	6	6	6	6	6	6	6	
✓2	∞	9	9	9	9	9	9	9	
✓3	∞	15	15	15	15	15	15	15	
✓4	∞	∞	26	26	19	19	19	19	
✓5	∞	23	23	22	22	22	22	22	
✓6	∞	∞	∞	∞	22	22	22	22	
✓7	∞	∞	∞	∞	∞	∞	∞	37	
✓8	∞	∞	∞	37	37	37	37	37	

PASSOS: 1 2 3 4 5 6 7 8

b) código na próxima página

```c
/* considerando que o tipo grafo é uma estrutura como a abaixo */

struct _grafo {
    int nv; /* numero de nos ou vertices */
    int na; /* numero de arestas */
    Viz** viz; /* viz[i] aponta para a lista
                  de arestas incidindo em i */
};
typedef struct _grafo Grafo;

typedef struct _viz Viz;
    struct _viz {
    int noj;
    float peso;
    Viz* prox;
};


void mostraCaminhos (Grafo *g, int* cmc, int no){
    static int custo = 0;
    if(no==0){                                  // chegou na origem
        printf("custo = %d\n", custo);
        return;
    }
    else{
        Viz* list_adj = g->viz[no];             // lista de adjacencias do vertice no
        while(list_adj->no != cmc[no]){         // percorrendo as adjacencias do vertica no ate chegar no ultimo no visitado antes dele
            list_adj = list_adj->prox;
        }
        custo += list_adj->peso;                // soma a distancia entre no e o ultimo no visitado antes dele
        mostraCaminhos(g, cmc, cmc[no]);        // entra recursivamente no ultimo no visitado antes de no
        printf("%d ",no);
    }
}
```

3) c) $\{\{0\}, \ldots, \{8\}\}$ cria partição dinâmica com os vértices;

adiciono as arestas mais leves: primeiro $\{3,4\}$, depois $\{5,6\}$, $\{0,1\}$, $\{0,2\}$, $\{3,6\}$ ficando com:

$\{\{0,1,2\}, \{3,4,6,5\}, \ldots\}$

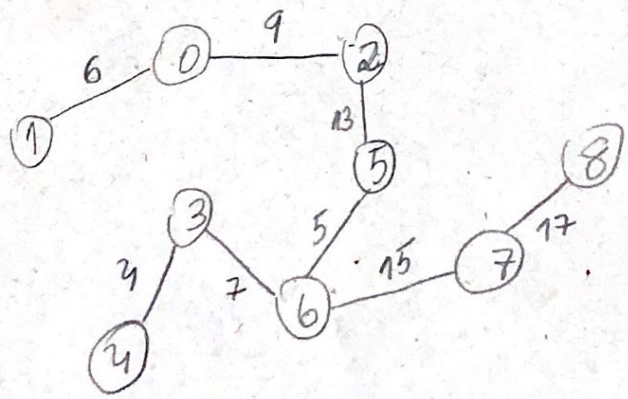
marco as arestas $\{1,6\}$ e $\{3,5\}$ pois elas formariam laços;

adiciono as próximas arestas mais leves e não marcadas: $\{2,5\}$ e $\{6,7\}$ e marco as arestas $\{0,3\}$, $\{0,5\}$ e $\{1,4\}$ pois elas formariam laços; ficando com:

$\{\{0,1,2,3,4,5,6,7\}, \ldots\}$

adiciona a aresta $\{7,8\}$ (mais leve não marcada) e encontro a árvore geradora de custo mínimo:

$\{0,1,2,3,4,5,6,7,8\}$



custo total: 76

(4) código na próxima página:

```c
int auxiliar(Grafo* g, int visited[], int pilha[], int v){          // fazendo dfs retorna 1 se encontrar algum ciclo
    int i;
    visited[v] = 1;                     // estamos visitando o v
    pilha[v] = 1;
    Viz* list_adj = g->viz[v];      // pegando a lista de adjacencias do v
    while(list_adj){                    // enquanto nao chegou no fim da lista de adjacencias
        if(!visited[list_adj->noj]){    // se o vertice da lista de adjacencias ainda nao foi visitado, visita ele;
            auxiliar(g, visited, pilha, list_adj->noj);
        }
        else if(pilha[i]){      // tem ciclo
            return 1;
        }
    } // visitou todos os vertices da lista de adjacencias
    for(i = v; i < g->nv; i++){
        if(!visited[i]){
            auxiliar(g, visited, pilha, i);
        }
        else if(pilha[i]){      // tem ciclo
            return 1;
        }
    }
    return 0;
}

int temCiclos(Grafo *g){
    int visited[g->nv];
    int pilha[g->nv];
    int v = 0;          // comecar a dfs no primeiro indice do grafo
    int i;
    for(i=0; i<g->nv; i++){     // inicializando todos os vertices como nao visitados e a pilha vazia
        visited[i] = 0;
        pilha[i] = 0;
    }
    return auxiliar(g, visited, pilha, v);
}

```