P3 INF1010 - 2021.2

DANIEL STULBERG HUF - 1920468 - TURMA 3WA

10/10

QUESTÃO 1

```
#include <stdio.h>
#include <stdlib.h>
#define MEMBROS_MAX X   /* sendo X um número arbitrário máximo
                           de elementos a depender do tipo
                           escolhido de armazenamento
                           dos elementos */

struct set {
    int n;
    unsigned int membros;   -> ou algum outro tipo...
};

/* insere o elemento i no conjunto */
void setInsert (set *set, int i) {
    if ((set == NULL) || (i<0) || (i>= MEMBROS_MAX)) return;
    set->membros |= (1<<i);   /* conjunto || Mascara(i) */
    return;
}

/* testa se o elemento i pertence ao conjunto */
int setIsMember (set *set, int i) {
    if ((set == NULL) || (i<0) || (i>= MEMBROS_MAX)) return 0;
    return ((1<<i) & (set->membros))   /* mascara(i) & conjunto */
}.
```

```
/* calcula a intersecção de dois conjuntos */
Set *setIntersection (Set *set1, Set *set2) {
    Set *intersec = (Set *) malloc (sizeof (Set));
    if (intersec == NULL) exit(1);
    else {
        intersec->n = MEMBROS_MAX;
        intersec->membros = 0;
    }
    if ((set1 == NULL) || (set2 == NULL)) return intersec;
    intersec->membros = (set1->membros) & (set2->membros);
    return intersec;
}

/* calcula a diferença de dois conjuntos set1 - set2 (elementos de set1 que
   não pertencem a set2) */
Set *setDifference (Set *set1, Set *set2) {
    Set *diff = (Set *) malloc (sizeof (Set));
    if (diff == NULL) exit(1);
    else {
        diff->n = MEMBROS_MAX;
        diff->membros = 0;
    }
    if ((set1 == NULL) & (set2 == NULL)) return diff;
    if (set1 == NULL) return diff;
    if (set2 == NULL) {
        diff->membros = set1->membros;
        return diff;
    }
    diff->membros = (set1->membros) & ~(set2->membros);
    return diff;
}
```

A = 1 1 0 0
B = 1 0 1 0
A - B = 0 1 0 0 ≡
A & ~(B) = 0 1 0 0

QUESTÃO 2

(a) A função de busca é responsável por identificar a qual conjunto o elemento pertence. Para isso, a função irá retornar o representante do conjunto, isto é, a raiz da árvore que contem o elemento passado como parâmetro.

(b) → Após ter encontrado qual é o representante de u, as linhas 6-8 serão responsáveis por pendurar todos os nós intermediários entre u e a raiz (incluindo u) diretamente na raiz (que é o representante do conjunto).

→ Enquanto o nó corrente não for a raiz, passamos o nó corrente para aux, o nó corrente recebe o próximo da chain e fazemos com que aux aponte diretamente para o representante.

→ Essa lógica é extremamente importante, pois otimiza o algoritmo para as próximas buscas nesse elemento ou de seus intermediários, já que agora o elemento e todos os seus nós intermediários apontam diretamente para o representante. Isso significa que todas as próximas buscas envolvendo qualquer um desses nós irão requerer apenas 1 passo.

(c)

→ Criei uma função que retorna o representante do resultado da união de duas partições de maneira otimizada, isto é, "pendurando" a árvore referente à partição com o menor número de nós.

↓ Função na próxima página

```c
#include <stdio.h>

typedef struct uniaoBusca{
    int n;      /* tamanho do vetor */
    int *v;     /* vetor de elementos */
} UniaoBusca

/* retorna o representante do resultado */
int ub_uniao (UniaoBusca * ub, int u, int v) {
    u = ub_busca(ub, u);  /* busca representante da partição de u */
    v = ub_busca(ub, v);  /* busca representante da partição de v */

    if (u == v) return u;  /* partição de u e v já é a mesma */

    if (ub->v[v] > ub->v[u]) {  /* árvore de u tem mais nós em modulo */
        ub->v[u] += ub->v[v];  /* número de nós de u é aumentado */
        ub->v[v] = u;  /* pendura árvore de v ao representante de u */
        return u;
    }
    else {  /* árvore de v tem mais nós em modulo */
        ub->v[v] += ub->v[u];  /* numero de nós de v é aumentado */
        ub->v[u] = v;  /* pendura árvore de u ao representante de v */
        return v;
    }
}
```

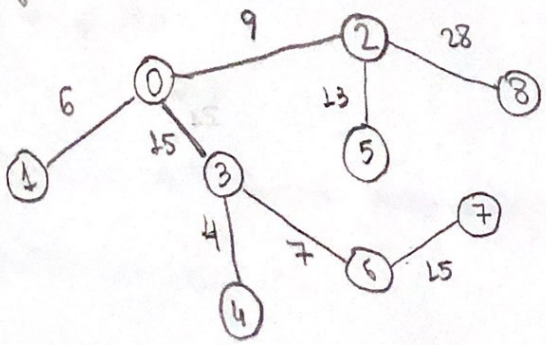
QUESTÃO 3

OBS: NOTAÇÃO DA TABELA → X, Y (X: custo; Y: vindo do vértice Y)

(a) Utilizando o algoritmo de Dijkstra para encontrar o caminho mais curto de G começando no vértice 0:

Vértices de G:	0	1	2	3	4	5	6	7	8
Vértices percorridos: na ordem de seleção do algoritmo 0	0,0	6,0	9,0	15,0	∞	23,0	∞	∞	∞
1			9,0	15,0	26,1	23,0	∞	∞	∞
2				15,0	26,1	22,2	∞	∞	37,2
3					19,3	22,2	22,3	∞	37,2
4						22,2	22,3	∞	37,2
5							22,3	∞	37,2
6								37,6	37,2
7									37,2

Grafo com os caminhos mais curtos partindo de 0 a todos os outros vértices:



Caminhos mais curtos:

0: 0

1: 0 —6— 1

2: 0 —9— 2

3: 0 —15— 3

4: 0 —15— 3 —4— 4 = 19

5: 0 —9— 2 —13— 5 = 22

6: 0 —15— 3 —7— 6 = 22

7: 0 —15— 3 —7— 6 —15— 7 = 37

8: 0 —9— 2 —28— 8 = 37

(b)

```
#include <stdio.h>

/* Fazemos uma iteração reversa no array de nós vizinhos para computar
o caminho mínimo até o nó escolhido, além de que a cada iteração,
somaremos ao custo total entre o nó corrente e seu predecessor. */

/* Levando em consideração que o grafo já foi populado na forma de uma
   matriz de adjacências. */

Void mostraCaminhos (grafos *g, int * cmc, int no) {
    int u = no, custoTot = 0;
    Pilha *p = cria_pilha(p);
    while (cmc[u] != NULL) {
        pilha_push (p, u);
        custoTot += g [cmc[u]] [u]
        u = cmc [u];
    }

    while (!pilha_vazia(p)) {
        u = pilha_pop(p);
        printF ("%d ", u);
    }

    printF ("\n Custo mínimo da origem ao nó %d = %d",
                                        no, custoTot);
    return;
}
```

Também levando em consideração que o TAD básico de pilhas já foi implementado

/* enquanto nó de origem não é atingido */

/* acessa seu predecessor */

/* insere nó na pilha */

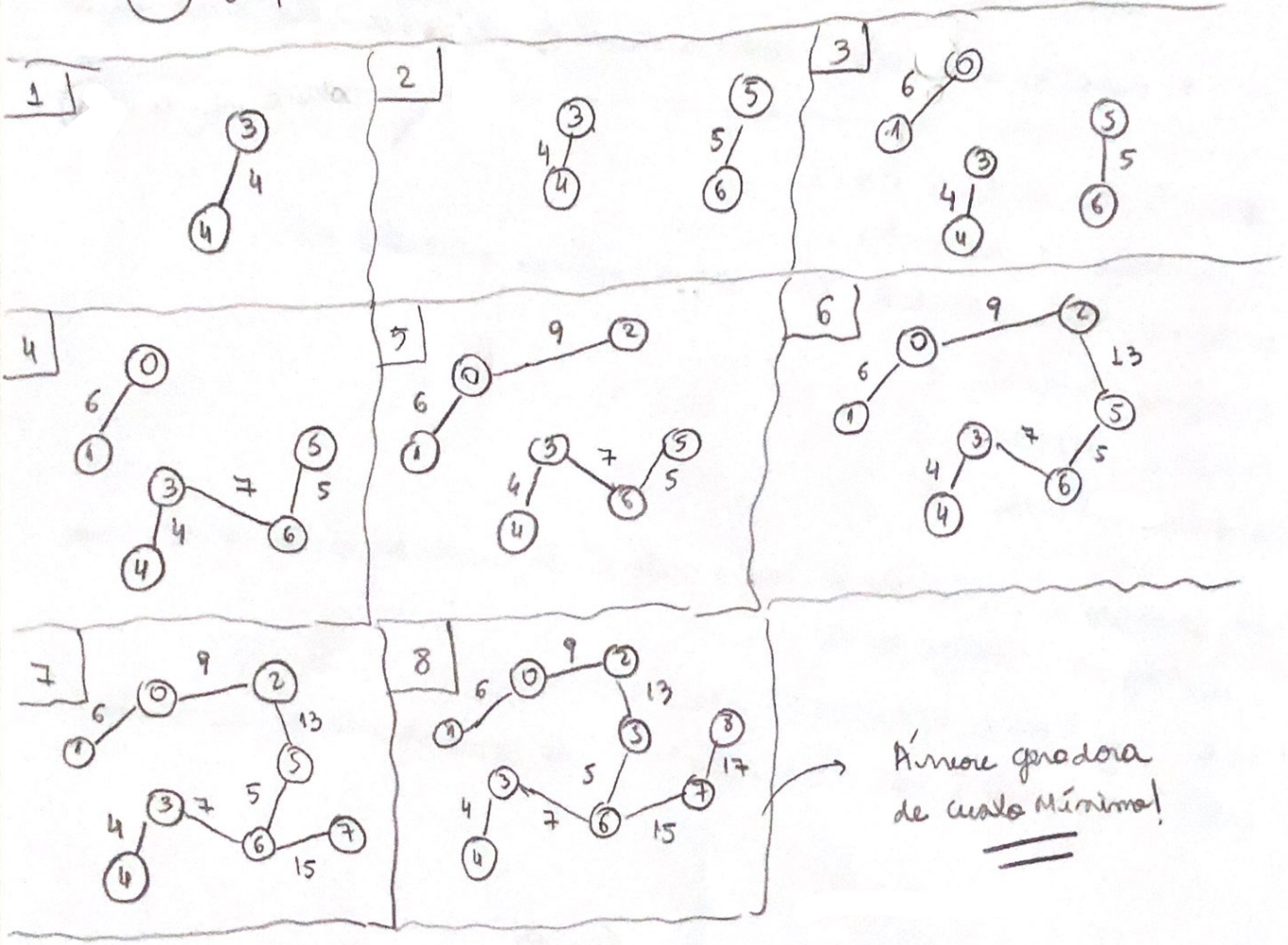
/* somando ao custo total o custo do predecessor até o nó atual */

(c) Utilizando o algoritmo de Kruskal para criar a árvore geradora de custo mínimo do grafo.

Passo 1 → Make-set de todos os nós do grafo

Passo 2 → Procurando as arestas de menor custo e unindo árvores que possuem representantes distintos (sem formar ciclos).

↻ loop até conectar todos os nós



QUESTÃO 4

→ Encontrar ciclos em um grafo direcionado:

- Regra 1 → Visita os nós adjacentes que ainda não foram visitados, marcá-los como visitados e colocá-los na pilha.
- Regra 2 → Se não foi encontrado um nó adjacente, tirar o nó da pilha (iremos tirar todos os nós da pilha que não contem nós adjacentes).
- Regra 3 → Repetir regra 1 e regra 2 até que a pilha esteja vazia.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES X    /* um número arbitrário */
#define INICIAL 1
#define VISITADO 2
#define FINAL 3

/* Levando em consideração que o grafo já foi populado na forma de uma
   matriz de adjacências */
int g[MAX_VERTICES][MAX_VERTICES];
int estado[MAX_VERTICES]    /* estados de cada vertice do grafo */
```

↓ Funções na próxima página

```
int dfs (int v, grafo * g) {
    int i, respTemp = 0;
    estado [v] = VISITADO;
    for (i = 0; i < MAX_VERTICES; i++) {
        if (g[v][i] != 0) {  /* nós não adjacentes */
            if (estado[i] == INICIAL)
                respTemp += dfs(i);
            else if (estado[i] == VISITADO) {
                printf ("O grafo possui ciclos!\n");
                return 1;
            }
        }
    }
    estado [v] = FINAL;
    return respTemp;
}

int temCiclo (grafo * g) {
    int v, resp = 0;
    for (v=0; v < MAX_VERTICES; v++)
        estado [v] = INICIAL;
    for (v=0; v < MAX_VERTICES; v++)
        if estado [v] == INICIAL
            resp += dfs (v, g);
    return (resp != 0);   /* se há pelo menos um ciclo, resp
                             será maior do que 0 */
}
```