Lista Grafos

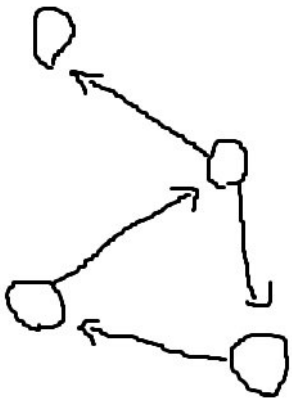
Alunos:

Ian Batista Fornaziero RA: 2677210

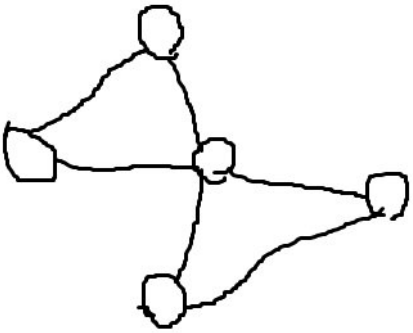
Gabriel Augusto Dupim RA: 2651408

1 – A)

Direcionado

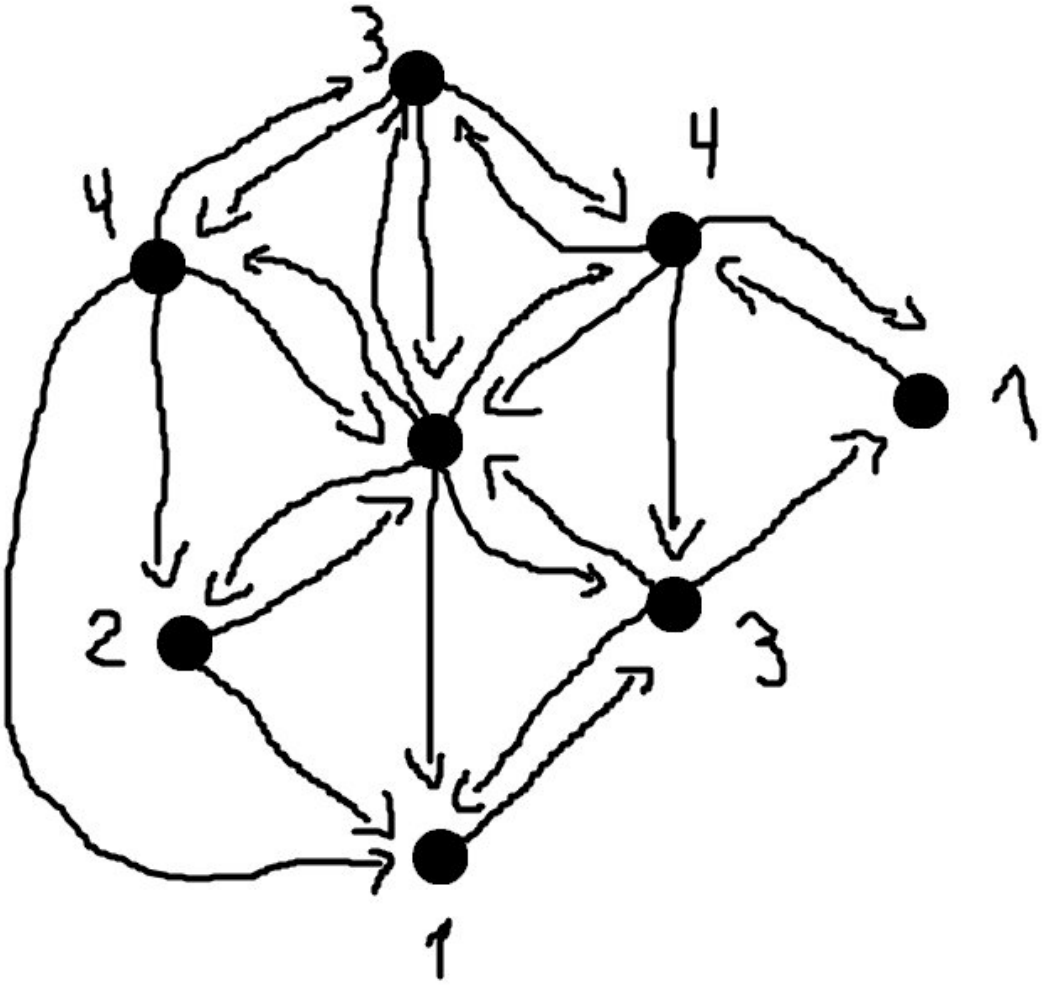


Não Direcionado



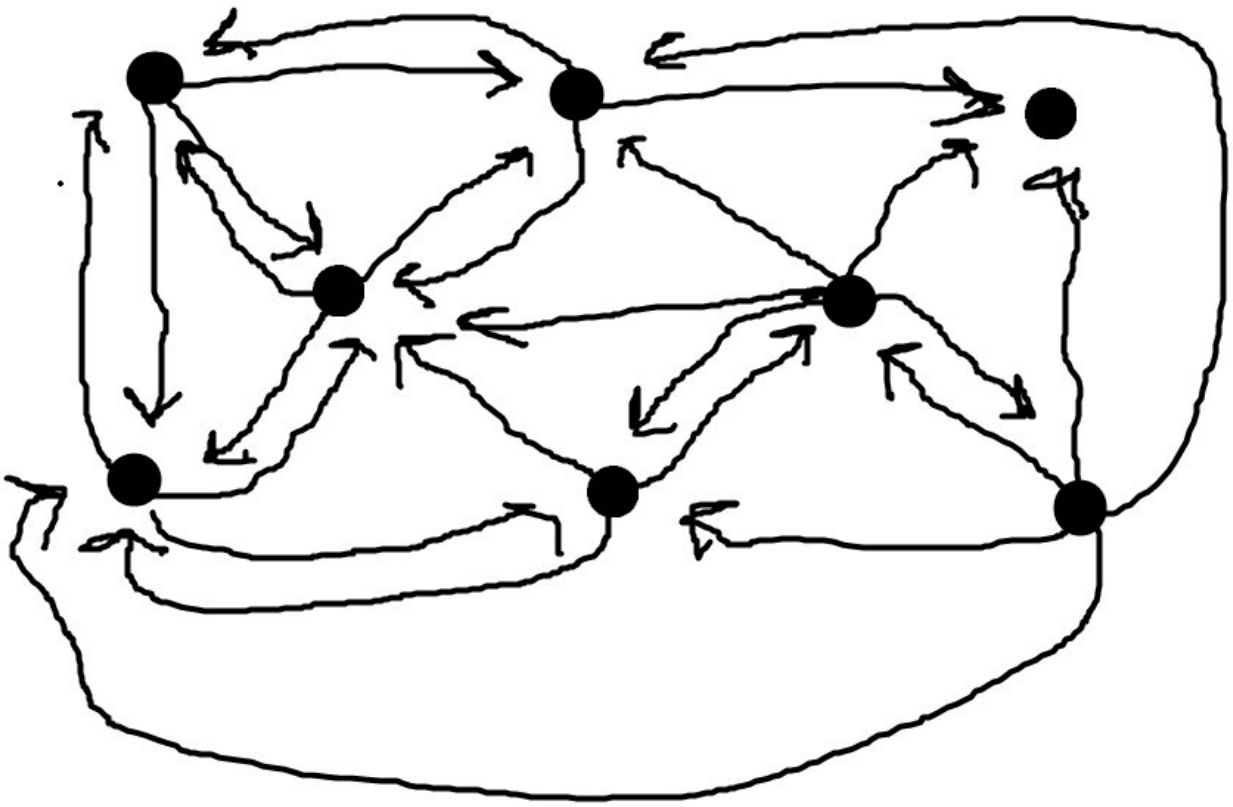
B)

A)



B)

B)



2 -

A) 11 Arestas B) 7 Vértices C) Não, pois, as únicas conexões de DFW acabam nele mesmo, e não passam por JFK

D) O caminho mais curto entre o vértice MIA e LAX, é utilizando a aresta para DFW e em seguida utilizar a aresta para LAX, tendo uma soma de pesos de 523 + 49 que dá 572, no caso, menor, do qual a aresta de MIA e LAX que tem peso de 611.

E)

	SFO	JFK	BOS	MIA	DFW	ORD	LAX
SFO	0	0	0	0	0	0	0
JFK	1	0	0	1	1	0	0
BOS	0	1	0	1	0	0	0
MIA	0	0	0	0	1	0	1
DFW	0	0	0	0	0	1	1
ORD	0	0	0	0	1	0	0
LAX	0	0	0	0	0	1	0

F)

SFO → NULL

JFK → SFO, MIA, DFW

BOS → JFK, MIA

MIA → DFW, LAX

DFW → ORD, LAX

ORD → DFW

LAX → ORD

G) Não é possível utilizar o algoritmo de Prim, pois, o grafo em questão é do tipo direcional, nesse caso, como o algoritmo de Prim foi pensado em grafos não direcionais, se torna, impossível gerar uma árvore geradora mínima de um grafo direcional.

E por mais que fosse feito a tentativa, não seria possível, pois SFO, não tem ligação com ninguém de forma direcional, ou seja, um nó sozinho, que não aceito por uma árvore geradora mínima.

3 –

```c
/*
 * 1. Implementa um construtor de grafos utilizando listas de adjacência.
 * 2. Ela define o número de vértices, o grau máximo de cada vértice e se o grafo é ponderado ou não.
 * 3. Na criação, aloca-se memória para todas as estruturas necessárias, incluindo as arestas e, se for o caso, os pesos das arestas.
 * 4. Se o grafo for ponderado, também aloca-se memória para os pesos das arestas.
 * 5. A função retorna um ponteiro para o grafo criado ou NULL se a alocação falhar.
 */
Grafo *cria_Grafo(int nro_vertices, int grau_max, int eh_ponderado)
{
    Grafo *gr;

    //? 1.
    gr = (Grafo *)malloc(sizeof(struct grafo));
    if (gr != NULL)
    {
        int i;
        //? 2.
        gr->nro_vertices = nro_vertices;
        gr->grau_max = grau_max;
        gr->eh_ponderado = (eh_ponderado != 0) ? 1 : 0;
        gr->grau = (int *)calloc(nro_vertices, sizeof(int));

        //? 3.
        gr->arestas = (int **)malloc(nro_vertices * sizeof(int *));
        for (i = 0; i < nro_vertices; i++)
            gr->arestas[i] = (int *)malloc(grau_max * sizeof(int));

        //? 4.
        if (gr->eh_ponderado)
        {
            gr->pesos = (float **)malloc(nro_vertices * sizeof(float *));
            for (i = 0; i < nro_vertices; i++)
                gr->pesos[i] = (float *)malloc(grau_max * sizeof(float));
        }
    }
    //? 5.
    return gr;
}
```

```c
/*
 * 1. Libera a memória alocada para um grafo.
 * 2. Verifica se o grafo não é NULL.
 * 3. Libera a memória alocada para as arestas de cada vértice.
 * 4. Se o grafo for ponderado, libera a memória alocada para os pesos das arestas.
 * 5. Libera a memória alocada para o vetor de graus e, finalmente,
 * 6. libera a memória alocada para o próprio grafo.
 */
//? 1.
void libera_Grafo(Grafo *gr)
{
    //? 2.
    if (gr != NULL)
    {
        int i;

        //? 3.
        for (i = 0; i < gr->nro_vertices; i++)
            free(gr->arestas[i]);
        free(gr->arestas);

        //? 4.
        if (gr->eh_ponderado)
        {
            for (i = 0; i < gr->nro_vertices; i++)
                free(gr->pesos[i]);
            free(gr->pesos);
        }

        //? 5.
        free(gr->grau);

        //? 6.
        free(gr);
    }
}
```

```c
/*
 * 1. Insere uma aresta entre dois vértices em um grafo.
 * 2. Verifica se o grafo não é NULL e se os índices dos vértices estão dentro dos limites válidos.
 * 3. Adiciona a aresta do vértice de origem para o vértice de destino.
 * 4. Se o grafo for ponderado, também adiciona o peso da aresta.
 * 5. Incrementa o grau do vértice de origem.
 * 6. Se o grafo não for direcionado, insere a aresta de volta do destino para a origem.
 * 7. Retorna 1 se a aresta foi inserida com sucesso, ou 0 se houve algum erro.
 */
//? 1.
int insereAresta(Grafo *gr, int orig, int dest, int eh_digrafo, float peso)
{
    //? 2.
    if (gr == NULL)
        return 0;
    if (orig < 0 || orig >= gr->nro_vertices)
        return 0;
    if (dest < 0 || dest >= gr->nro_vertices)
        return 0;

    //? 3.
    gr->arestas[orig][gr->grau[orig]] = dest;
    //? 4.
    if (gr->eh_ponderado)
        gr->pesos[orig][gr->grau[orig]] = peso;
    //? 5.
    gr->grau[orig]++;

    //? 6.
    if (eh_digrafo == 0)
        insereAresta(gr, dest, orig, 1, peso);
    //? 7.
    return 1;
}
```

```c
/*
 * 1. Remove uma aresta entre dois vértices em um grafo.
 * 2. Verifica se o grafo não é NULL e se os índices dos vértices estão dentro dos limites válidos.
 * 3. Procura a posição da aresta a ser removida no vetor de arestas do vértice de origem.
 * 4. Se a aresta não for encontrada, retorna 0.
 * 5. Diminui o grau do vértice de origem.
 * 6. Substitui a aresta removida pela última aresta do vetor (para manter o vetor compacto).
 * 7. Se o grafo for ponderado, faz o mesmo para o peso da aresta.
 * 8. Se o grafo não for direcionado, remove a aresta de volta do destino para a origem.
 * 9. Retorna 1 se a aresta foi removida com sucesso.
 */
//? 1.
int removeAresta(Grafo *gr, int orig, int dest, int eh_digrafo)
{
    //? 2.
    if (gr == NULL)
        return 0;
    if (orig < 0 || orig >= gr->nro_vertices)
        return 0;
    if (dest < 0 || dest >= gr->nro_vertices)
        return 0;

    //? 3.
    int i = 0;
    while (i < gr->grau[orig] && gr->arestas[orig][i] != dest)
        i++;
    //? 4.
    if (i == gr->grau[orig]) // elemento nao encontrado
        return 0;
    //? 5.
    gr->grau[orig]--;
    //? 6.
    gr->arestas[orig][i] = gr->arestas[orig][gr->grau[orig]];
    //? 7.
    if (gr->eh_ponderado)
        gr->pesos[orig][i] = gr->pesos[orig][gr->grau[orig]];
    //? 8.
    if (eh_digrafo == 0)
        removeAresta(gr, dest, orig, 1);
    //? 9.
    return 1;
}
```

```c
/*
 * 1. Imprime o grafo na tela.
 * 2. Verifica se o grafo não é NULL.
 * 3. Percorre todos os vértices do grafo.
 * 4. Para cada vértice, imprime suas arestas e, se for ponderado, imprime também os pesos.
 */
//? 1.
void imprime_Grafo(Grafo *gr)
{
    //? 2.
    if (gr == NULL)
        return;

    int i, j;
    //? 3.
    for (i = 0; i < gr->nro_vertices; i++)
    {
        printf("%d: ", i);
        //? 4.
        for (j = 0; j < gr->grau[i]; j++)
        {
            if (gr->eh_ponderado)
                printf("%d(%.2f), ", gr->arestas[i][j], gr->pesos[i][j]);
            else
                printf("%d, ", gr->arestas[i][j]);
        }
        printf("\n");
    }
}
```