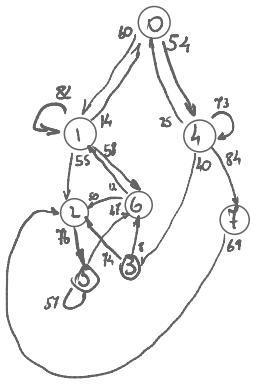
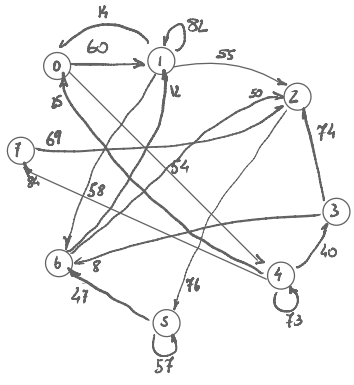
Exemplo de Prova 1 – ACH2024 – Algoritmos e Estruturas de Dados II

Questão 1. Desenhe o **grafo direcionado e ponderado** correspondente à matriz de adjacências com pesos abaixo. Nesta matriz, os hifens indicam arestas inexistentes.

	v0	v1	v2	v3	v4	v5	v6	v7
v0	-	60.0	-	-	54.0	-	-	-
v1	14.0	82.0	55.0	-	-	-	58.0	-
v2	-	-	-	-	-	76.0	-	-
v3	-	-	74.0	-	-	-	8.0	-
v4	25.0	-	-	40.0	73.0	-	-	84.0
v5	-	-	-	-	-	57.0	47.0	-
v6	-	12.0	50.0	-	-	-	-	-
v7	-	-	69.0	-	-	-	-	-

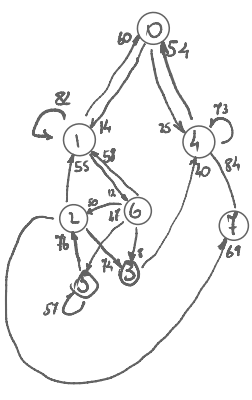
Desenhe o grafo aqui:





Questão 2. Desenhe o grafo transposto do grafo da Questão 1.

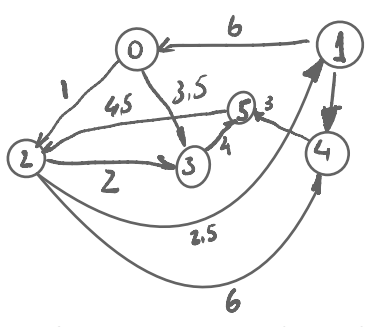
mudar a direção das setas



Questão 3 A função de impressão de grafos vista em aula foi chamada para imprimir um grafo direcionado e ponderado representado por listas de adjacências. O resultado do que foi impresso pode ser visto a seguir. Desenhe o **grafo direcionado e ponderado** correspondente.

```
Imprimindo grafo (vertices: 6; arestas: 10).
[ 0] -> 2 (1.00) -> 3 (3.50)
[ 1] -> 0 (6.00) -> 4 (5.00)
[ 2] -> 1 (2.50) -> 3 (2.00) -> 4 (6.00)
[ 3] -> 5 (4.00)
[ 4] -> 5 (3.00)
[ 5] -> 2 (4.50)
```

Desenhe o grafo aqui:

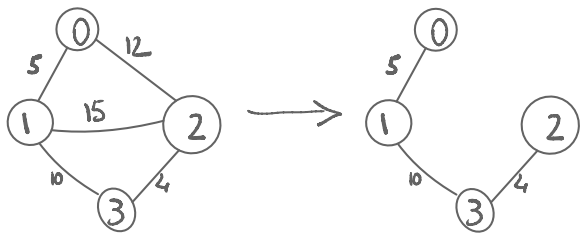


Questão 4. Dado o seguinte grafo **ponderado e não direcionado**, representado pela matriz de adjacências com pesos apresentada a seguir, desenhe a **árvore geradora de custo mínimo** para esse grafo.

árvore geradora mínima não tem ciclo

	v0	v1	v2	v3
v0	-	5.00	12.00	-
v1	5.00	-	15.00	10.00
v2	12.00	15.00	-	4.00
v3	-	10.00	4.00	-

Desenhe a árvore geradora de custo mínimo aqui:



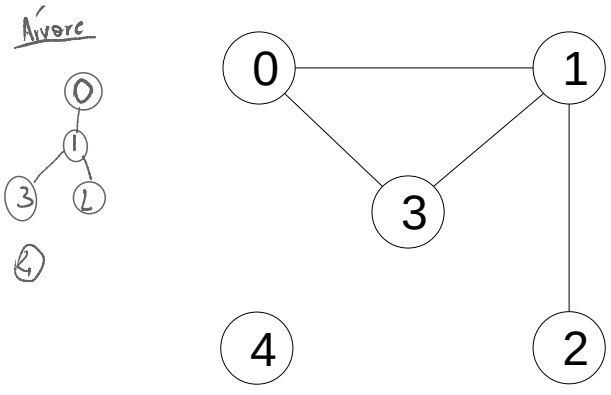
Questão 5. Dadas as seguintes funções relacionadas à busca em profundidade e considerando a representação de **listas de adjacências para grafos não ponderados e não dirigidos** vista em aula **(lembre-se que ordenamos as listas de adjacência de acordo com o número dos vértices)**, escreva o que será impresso na tela após a execução da função *buscaEmProfundidadeCores*, considerando as duas situações descritas a seguir.

```
void visitaProfundidadeCores(Grafo* g, int atual, int* tempo, int* cor, int*
tDescoberta, int* tTermino, int* ant){
  (*tempo)++;
  cor[atual] = 1; //cinza
  tDescoberta[atual] = *tempo;
  int x;
  ElemLista* end = g->A[atual];
  while (end){
    x=end->vertice;
    if(cor[x]==0){ // branco
      ant[x] = atual;
      visitaProfundidadeCores(g, x, tempo, cor, tDescoberta, tTermino, ant);
    } else if(ant[atual] != x) printf("Este grafo nao e aciclico.\n");
    end = end->prox;
  }
  cor[atual] = 2; // preto
  (*tempo)++;
  tTermino[atual] = *tempo;
}

void buscaEmProfundidadeCores(Grafo* g){
  if (!g || g->numVertices<1) return;
  int x;
  int* cor = malloc(sizeof(int)*g->numVertices);
  int* tDescoberta = malloc(sizeof(int)*g->numVertices);
  int* tTermino = malloc(sizeof(int)*g->numVertices);
  int* ant = malloc(sizeof(int)*g->numVertices);
  int tempo = 0;
  for (x=0;x<g->numVertices;x++){
    cor[x] = 0;
    tDescoberta[x] = -1;
    tTermino[x] = -1;
    ant[x] = -1;
  }
  for (x=0;x<g->numVertices;x++)
       if (cor[x]==0) visitaProfundidadeCores(g, x, &tempo, cor, tDescoberta,
tTermino, ant);

  printf("\n");
  printf("Resumo da Busca em Profundidade:\n");
  printf("No\tant\tDescoberta\tTermino\tCor:\n");
  for (x=0;x<g->numVertices;x++)
       printf("%2i\t%8i\t%10i\t%7i\t%3i\n",
                              x,ant[x],tDescoberta[x],tTermino[x],cor[x]);
  printf("\n");
  free(cor);
  free(tDescoberta);
  free(tTermino);
  free(ant);
}
```

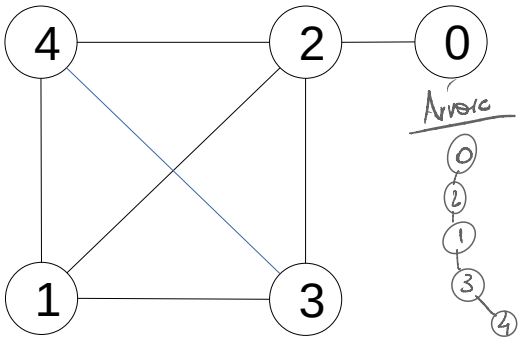
Item a: A função `buscaEmProfundidadeCores` foi chamada recebendo como parâmetro o endereço do grafo representado abaixo.



Escreva o que será impresso:

no	Anterior	tDescoberta	tTermino	cor
0	-1	1	8	2
1	0	2	7	2
2	1	3	4	2
3	1	5	6	2
4	-1	9	10	2

Item b: A função `buscaEmProfundidadeCores` foi chamada recebendo como parâmetro o endereço do grafo representado abaixo.



Escreva o que será impresso:

no	Anterior	tDescoberta	tTermino	cor
0	-1	1	10	2
1	2	3	8	2
2	0	2	9	2
3	1	4	7	2
4	3	5	6	2

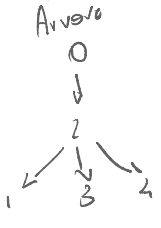
Questão 6. Considere os grafos da questão anterior, representados por listas de adjacências para grafos não ponderados e não dirigidos **(lembre-se que ordenamos as listas de adjacência de acordo com o número dos vértices)**.

a) Escreva a ordem em que os vértices do grafo da questão anterior **item (a)**, serão visitados de acordo com a Busca em Largura, conforme visto em aula (iniciando do vértice 0 [zero]).

Resposta: $V_0 \rightarrow V_1 \rightarrow V_3 \rightarrow V_2$

b) Escreva a ordem em que os vértices do grafo da questão anterior - **item (b)**, serão visitados de acordo com a Busca em Largura, conforme visto em aula (iniciando do vértice 0 [zero]).

Resposta: $V_0 \rightarrow V_2 \rightarrow V_1 \rightarrow V_3 \rightarrow V_4$



Questão 7. Considerando a representação de grafos **não ponderados e não direcionados** usando **matrizes de adjacências booleanas**, com base na estrutura apresentada a seguir (que é a mesma que foi vista em aula), implemente as duas funções solicitadas.

```
typedef struct {
  int numVertices;
  int numArestas;
  bool** matriz;
} Grafo;
```

a) verticeMaiorGrau: função que recebe o endereço de um grafo (g) e retorno o número/identificador do vértice que tem o maior grau no grafo (se mais de um vértice tiver o maior grau, sua função deve retornar o menor deles [aquele com o menor identificador]). Você pode considerar que g contém um endereço válido para um grafo.

```
int verticeMaiorGrau(Grafo* g){
```

```
int verticeMaiorGrau(Grafo* g){
    if (!g) return -1;
    int x, y, grau = 0;
    int vertice = -1;
    for (y=0; y<g->numVertices;y++){
        for (x=0;x<g->numVertices;x++){
            if (g->matriz[y][x]) grau++;
        }
        if (grau>vertice) vertice = y;
    }
    return vertice;
}
```

```
}
```

b) vizinhosEmComum: função que recebe o endereço de um grafo (g) e o identificador de dois vértices (u e v) e retorno o número de vizinhos em comum dos vértices u e v, isto é, a quantidade de vértices que são vizinhos de u e também vizinhos de v. Você pode considerar que g contém um endereço válido para um grafo.

```
int vizinhosEmComum(Grafo* g, int u, int v){
```

```
int vizinhosEmComum(Grafo* g, int u, int v){
    if (!g || v < 0 || v >= g->numVertices || u < 0 || u >= g->numVertices) return -1;
    int vizinhos = 0;
    int x;
    for (x = 0; x<g->numVertices;x++){
        if (g->matriz[u][x]==g->matriz[v][x]) vizinhos++;
    }
    return vizinhos;
}
```

```
}
```

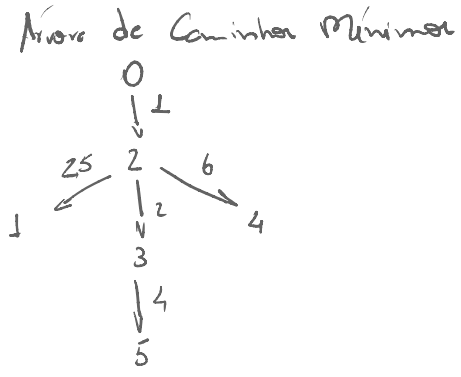
Questão 8. O algoritmo de Dijkstra, que calcula o arranjo de distância e de predecessores de um vértice para os demais vértices de um grafo, foi executado para um dado grafo, a partir do vértice 0 (zero) e produziu os seguintes arranjos de distância e de predecessores:

```
Exibindo arranjo de distâncias.
```

v0	v1	v2	v3	v4	v5
0.00	3.50	1.00	3.00	7.00	7.00

```
Exibindo arranjo de predecessores.
```

v0	v1	v2	v3	v4	v5
0	2	0	2	2	3



Com base nesses arranjos, escreva qual será o caminho de menor custo (escreva a sequência de vértices na ordem em que serão percorridos) e o custo (ou distância) total para:

a) Sair do vértice **v0** e chegar no vértice **v5**:

Caminho: $V_0 \rightarrow V_2 \rightarrow V_3 \rightarrow V_5$ Distância: $1 + 2 + 4 = 7$

b) Sair do vértice **v0** e chegar no vértice **v1**:

Caminho: $V_0 \rightarrow V_2 \rightarrow V_1$ Distância: $1 + 2{,}5 = 3{,}5$

c) Sair do vértice **v0** e chegar no vértice **v3**:

Caminho: $V_0 \rightarrow V_2 \rightarrow V_3$ Distância: $1 + 2 = 3$

Questão 9. O algoritmo de Floyd-Warshall, que calcula as distâncias (e os predecessores) entre todos os pares de vértices, foi executado para um dado grafo (ilustrado a seguir pela impressão do grafo no formato de listas de adjacências) e produziu as seguintes matrizes de distâncias e de predecessores:

```
Imprimindo grafo (vértices: 6; arestas: 13).
[ 0] ->  3 (10.00) ->  5 (76.00)
[ 1] ->  2 (21.00) ->  4 (14.00)
[ 2] ->  2 (73.00) ->  5 (44.00)
[ 3] ->  1 (57.00) ->  2 (70.00) ->  5 (78.00)
[ 4] ->  1 (16.00) ->  3 (55.00)
[ 5] ->  0 (55.00) ->  4 (47.00)
```

Exibindo matriz de distâncias.

	v0	v1	v2	v3	v4	v5
v0	0.00	67.00	80.00	10.00	81.00	76.00
v1	120.00	0.00	21.00	69.00	14.00	65.00
v2	99.00	107.00	0.00	109.00	91.00	44.00
v3	133.00	57.00	70.00	0.00	71.00	78.00
v4	136.00	16.00	37.00	55.00	0.00	81.00
v5	55.00	63.00	84.00	65.00	47.00	0.00

Exibindo matriz de predecessores.

	v0	v1	v2	v3	v4	v5
v0	0	3	3	0	1	0
v1	5	1	1	4	1	2
v2	5	4	2	0	5	2
v3	5	3	3	3	1	3
v4	5	4	1	4	4	2
v5	5	4	1	0	5	5

a) Qual é a menor distância entre o vértice v0 e o vértice v4? 81

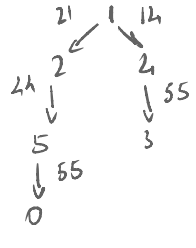
b) Qual vértice está mais distante de v2? V3

c) Qual é o par de vértices mais distante (isto é, se você precisar sair de um vértice x e chegar a um vértice y, qual é o par x e y cujo caminho tem a maior distância)? V4 → V0 (109)

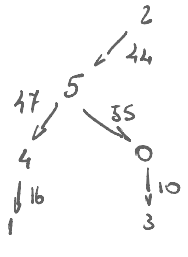
d) O grafo desta questão é cíclico ou acíclico? cíclico

e) O grafo desta questão é fortemente conexo? sim

f) Desenhe a Árvore de Caminhos de Menor Custo (ou de Custo Mínimo) do vértice v1 (isto é, aquela em que v1 é a raiz):



f) Desenhe a Árvore de Caminhos de Menor Custo (ou de Custo Mínimo) do vértice v2 (isto é, aquela em que v2 é a raiz):



Questão 10. Cite vantagens e desvantagens (ou pontos forte e fracos) em relação à escolha por se usar a representação de grafos por matrizes de adjacência ou contraste com o uso de listas de adjacências.

Matriz de Adjacência
Vantagens
Acesso Rápido: Acesso a uma célula da matriz é feito em tempo constante O(1).
Memória: Em grafos densos (com muitas arestas), a matriz de adjacência pode ocupar menos memória do que uma lista de adjacências.
Desvantagens
Memória: Em grafos grandes com poucas arestas, a matriz de adjacência pode consumir uma quantidade significativa de memória, pois precisa armazenar informações para todas as possíveis arestas, mesmo aquelas que não existem.
Lista de Adjacências
Vantagens
Economia de Espaço: Em grafos grandes com poucas arestas, a lista de adjacências pode economizar memória.
Desvantagens
Acesso mais Lento: Para buscar por uma aresta específica é necessário percorrer a lista de adjacências de um vértice.
Gasto de Memória: Em grafos densos, uma lista de adjacências pode consumir mais memória devido à necessidade de armazenar ponteiros para vértices adjacentes em vez de apenas armazenar uma matriz de tamanho fixo.

Questão 11. Considerando grafos ponderados e direcionados, faça duas implementações da função *arranjoDeArestas*, uma considerando matrizes de adjacência e outra considerando listas de adjacências.

A função *arranjoDeArestas* recebe o endereço de um grafo como parâmetro e retorna o endereço de um arranjo contendo todas as arestas do grafo.

Assinatura da função:

```
Aresta* arranjoDeArestas(Grafo* g)
```

```
typedef float Peso;
```

Estrutura Aresta:

```
typedef struct {
  int origem;
  int destino;
  Peso peso;
} Aresta;
```

Estruturas usadas na representação de grafos usando listas de adjacência:

```
typedef struct aux{
  int vertice;
  Peso peso;
  struct aux* prox;
} ElemLista, *PONT;

typedef struct {
  int numVertices;
  int numArestas;
  ElemLista** A;
} Grafo;
```

Estrutura usada na representação de grafos usando matrizes de adjacência (com pesos):

```
typedef struct {
  int numVertices;
  int numArestas;
  Peso** matriz;
} Grafo;
```

```
Aresta * arranjoDeArestas(Grafo * g){ //MATRIZ
    if (!g ) return NULL;
    int a = g->numArestas;
    int i = 0;
    Aresta * arr = (Aresta*)malloc(sizeof(Aresta)*a);
    for (int x=0; x<g->numVertices; x++){
        for (int y=0; y<g->numVertices; y++){
            if(g->matriz[x][y] != 0){
                arr[i].origem = x;
                arr[i].destino = y;
                arr[i].peso = g->matriz[x][y];
                i++;
            }
        }
    }
    return arr;
}
```

```
Aresta * arranjoDeArestas(Grafo * g){ //LISTA
    if (!g ) return NULL;
    int a = g->numArestas;
    Aresta * arr = (Aresta*)malloc(sizeof(Aresta) * a);
    int i = 0;
    Elemlista * atual;
    for (int x=0; x<g->numVertices; x++){
        atual = g->A[x];
        while (atual){
            arr[i].origem = x;
            arr[i].destino = atual->vertice;
            arr[i].peso = atual->peso;
            i++;
            atual = atual->prox;
        }
    }
    return arr;
}
```