Projeto Prático 3: Nano-robôs cerebrais

1. Estruturas a serem usadas:

-TAD grafo ponderado com lista de adjacência;
-TAD fila de prioridade mínima com um heap binário mínimo;
-Algoritmo de Dijkstra para caminhos mínimos com um TAD fila de prioridade mínima;
-Algoritmo de Kruskal para obter uma árvore geradora mínima utilizando um TAD Union-Find.

2. Descrição do problema:

Foi criado um modelo de cérebro representado por um grafo, onde os blocos de neurônios são representados por vértices dos grafos, e sinapses ligando neurônios (e ligando blocos de neurônios) são arestas do grafo, como mostrado na imagem a seguir(o grafo da imagem não é equivalente ao exemplo fornecido de entradas e saídas mais abaixo, porém a ideia é a mesma para o exemplo fornecido logo mais abaixo):

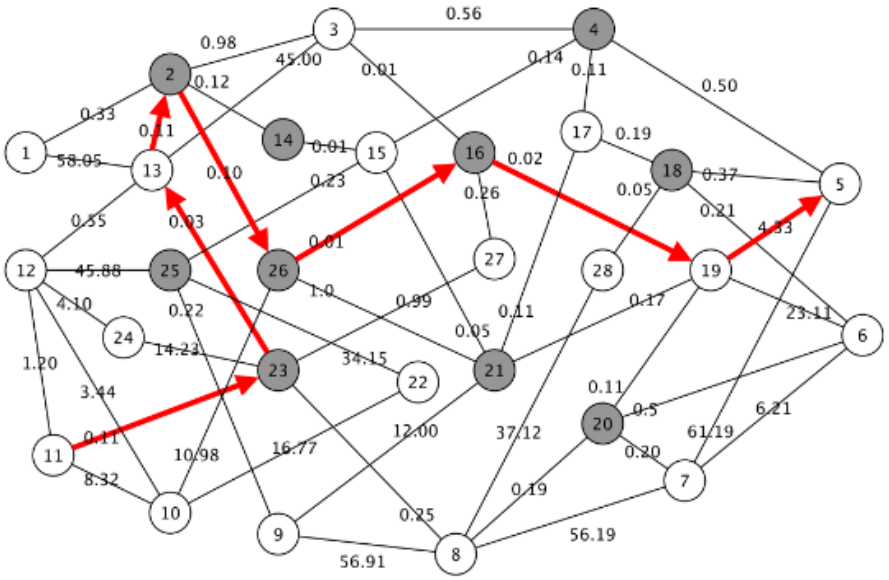

Figura 1: Grafo representando blocos de neurônios do cérebro e o percurso de um nano-robô.

Nesta figura, os vértices 11 e 5 representam, respectivamente, os blocos de entrada e saída do robô. As setas em vermelho indicam o caminho a ser percorrido pelo robô. Vértices em cinza representam blocos de neurônios doentes. Um bloco é doente se contém pelo menos um neurônio doente; ou é sadio, em caso contrário. Em um bloco doente, neurônios doentes estão marcados com a cor preta, conforme mostra a figura a seguir (conforme a figura 1 de cima):

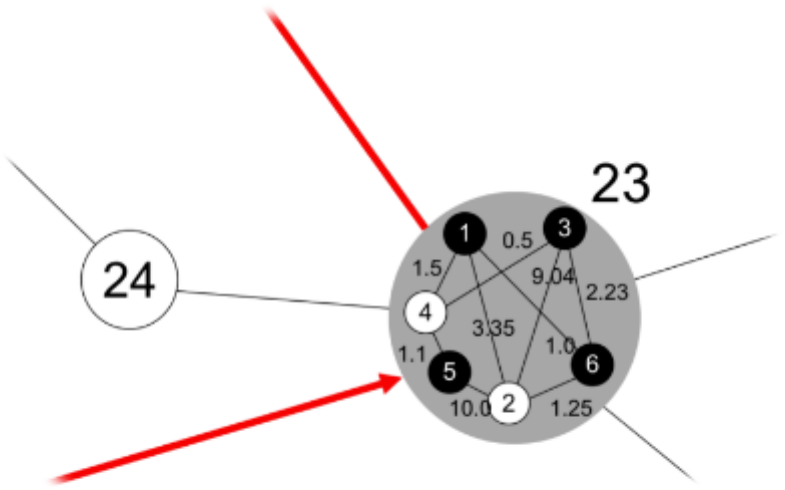

Figura 2: Bloco com neurônios doentes (em preto).

O processo executado por um nano-robô é descrito à seguir:

1. O robô inicia em um bloco de neurônios (ponto de entrada no cérebro);
2. O robô calcula um percurso pela rede de neurônios até alcançar o bloco de saída do cérebro. Este percurso deve ser mínimo, pois o robô dispõe de pouca energia. Assim, utilize o algoritmo de Dijkstra com uma fila de prioridade mínima;
3. O robô caminha pelo percurso calculado visitando os blocos de neurônios do caminho;
4. Ao visitar um bloco de neurônios o robô identifica se está diante de um bloco doente ou de um bloco sadio;
5. Se o bloco for doente, o nano-robô (para economizar tempo e energia), determina a MST (minimum spanning tree) dentro do bloco de neurônios doentes, e caminha na MST injetando uma enzima sintética no núcleo de cada neurônio do bloco (doente ou não), que corrige o código genético do neurônio, tornando-o sadio (a enzima é inócua às células sadias). Se o bloco não for doente o robô não faz nada e segue para o próximo passo (observe que somente serão calculadas as MSTs dos blocos doentes do percurso do robô). Para determinar cada MST, implemente o algoritmo de Kruskal com um Union-Find;
6. O robô abandona o bloco visitado e parte em direção do próximo bloco do percurso.
7. Os passos de 4-6 se repetem até o robô concluir o percurso chegando ao bloco que representa o ponto de saída do cérebro.

Durante o percurso, nem todos os blocos identificados pelo robô são doentes. Para tentar "forçar" o robô a passar por alguns blocos doentes para curá-los, os cientistas injetam uma solução no cérebro do paciente que faz o robô interpretar as sinapses que se conectam a um neurônio doente como tendo um valor de comprimento muito pequeno. Assim, ao processar o cálculo do percurso, haverá maior chance do nano-robô passar por esse

neurônio doente (pois a “distância” da sinapse até ele terá um valor menor do que as sinapses que se ligam a neurônios sadios).

3. Entradas e Saídas do Problema

1º Parte da entrada: O grafo do cérebro, seguido dos blocos de entrada e saída do cérebro. Exemplo:

18 43 (ordem e tamanho do grafo, respectivamente)
1 3 1.0 (1º aresta ligando os blocos 1 e 3, com valor 1.0)
1 9 1.0 (2º aresta ligando os blocos 1 e 9, com valor 1.0)
1 15 1.0 (3º aresta ligando os blocos 1 e 15, com valor 1.0)
1 16 1.0 (4º aresta ligando os blocos 1 e 16, com valor 1.0)
1 17 1.0 (continua…)
2 4 99.0
2 5 99.0
2 11 99.0
2 12 99.0
2 18 99.0
3 8 1.0
3 9 1.0
3 14 1.0
4 5 99.0
4 6 99.0
4 7 99.0
4 9 99.0
4 15 99.0
5 7 1.0
5 12 1.0
5 14 1.0
5 15 1.0
6 7 1.0
6 9 1.0
7 8 99.0
7 12 99.0
7 13 99.0
7 15 99.0
7 17 99.0
8 11 1.0
8 12 1.0
8 14 1.0
9 10 99.0
9 11 99.0
9 12 99.0
9 13 99.0
9 14 99.0
10 14 1.0
10 15 1.0

11 12 99.0
11 13 99.0
11 16 99.0
11 18 99.0 (43º aresta ligando os blocos 11 e 18, com valor 99.0)
1 18 (bloco de entrada e bloco de saída respectivamente)

2º Parte da entrada: Os grafos de cada bloco de neurônios, na ordem de cada bloco dada no grafo do cérebro. Exemplo:

6 10 (ordem e tamanho do grafo do bloco 1)
2 (número de neurônios doentes do bloco 1)
1 3 (neurônios doentes mencionados acima)
1 4 4.0 (aresta ligando o neurônio 1 ao neurônio 4, e sua distância)
1 6 8.0 (aresta ligando o neurônio 1 ao neurônio 6, e sua distância)
2 3 51.0 (aresta ligando o neurônio 2 ao neurônio 3, e sua distância)
2 4 59.0 (aresta ligando o neurônio 2 ao neurônio 4, e sua distância)
2 5 5.0 (continua…)
2 6 67.0
3 5 50.0
3 6 91.0
4 6 70.0
5 6 1.0 (aresta ligando o neurônio 5 ao neurônio 6, e sua distância; termina bloco 1)
7 15 (ordem e tamanho do grafo do bloco 2)
0 (número de neurônios doentes do bloco 2)
1 2 91.0 (aresta ligando o neurônio 1 ao neurônio 2, e sua distância)
1 3 5.0 (aresta ligando o neurônio 1 ao neurônio 3, e sua distância)
1 4 80.0 (aresta ligando o neurônio 1 ao neurônio 4, e sua distância)
1 5 26.0 (continua…)
2 3 20.0
2 4 77.0
2 5 21.0
2 7 44.0
3 4 22.0
3 5 73.0
3 6 5.0
3 7 11.0
4 7 76.0
5 7 5.0
6 7 27.0 (aresta ligando o neurônio 6 ao neurônio 7, e sua distância; termina bloco 2)
6 9 (ordem e tamanho do bloco 3)
3 (número de neurônios doentes do bloco 3)
4 5 1 (neurônios doentes do bloco 3)
1 2 35.0 (e assim continua a mesma coisa de antes para os demais blocos)
1 4 16.0
1 5 82.0
2 3 15.0
2 4 31.0
2 5 38.0

3 4 2.0
4 5 92.0
5 6 41.0
10 30 (agora o bloco 4)
0
1 3 93.0
1 5 83.0
1 6 97.0
1 7 50.0
2 3 10.0
2 4 74.0
2 6 45.0
2 7 2.0
2 8 23.0
2 9 99.0
3 4 64.0
3 5 92.0
3 8 98.0
3 9 6.0
3 10 0.0
4 5 52.0
4 8 85.0
4 9 59.0
5 6 34.0
5 7 60.0
5 9 22.0
6 7 4.0
6 8 88.0
6 9 16.0
6 10 77.0
7 9 61.0
7 10 34.0
8 9 7.0
8 10 56.0
9 10 93.0
9 24 (bloco 5)
5
8 6 2 3 9
1 2 58.0
1 3 10.0
1 5 48.0
1 9 17.0
2 3 18.0
2 5 89.0
2 6 57.0
2 8 76.0
2 9 81.0
3 5 29.0

3 6 26.0
3 8 28.0
4 7 71.0
4 8 65.0
4 9 30.0
5 6 64.0
5 7 82.0
5 8 39.0
5 9 77.0
6 7 36.0
6 8 59.0
7 8 42.0
7 9 60.0
8 9 61.0
10 24 (bloco 6)
2
10 5
1 4 8.0
1 5 82.0
1 6 77.0
1 7 67.0
1 9 95.0
2 3 17.0
2 4 65.0
2 6 3.0
2 7 85.0
2 8 76.0
3 4 68.0
3 5 25.0
3 7 75.0
3 10 33.0
4 6 13.0
4 8 66.0
4 9 22.0
4 10 21.0
5 6 7.0
5 8 69.0
6 9 31.0
6 10 41.0
7 9 0.0
8 10 94.0
9 24 (bloco 7)
0
1 2 39.0
1 5 20.0
1 6 52.0
1 8 11.0
2 4 10.0

2 5 2.0
2 6 56.0
2 7 33.0
2 9 55.0
3 4 69.0
3 8 36.0
3 9 66.0
4 5 33.0
4 6 67.0
4 9 92.0
5 6 57.0
5 8 6.0
5 9 34.0
6 7 82.0
6 8 45.0
6 9 35.0
7 8 73.0
7 9 72.0
8 9 47.0
6 13 (bloco 8)
4
4 5 2 3
1 2 48.0
1 3 86.0
1 4 34.0
1 5 66.0
2 3 82.0
2 4 53.0
2 5 67.0
2 6 9.0
3 4 79.0
3 6 38.0
4 5 14.0
4 6 47.0
5 6 83.0
10 29 (bloco 9)
0
1 3 51.0
1 4 16.0
1 5 84.0
1 7 61.0
1 9 38.0
1 10 52.0
2 3 55.0
2 7 86.0
2 8 24.0
2 10 31.0
3 4 13.0

3 5 47.0
3 6 29.0
3 7 52.0
3 8 67.0
3 9 94.0
3 10 3.0
4 8 89.0
4 9 79.0
4 10 8.0
5 6 92.0
5 8 19.0
5 9 54.0
6 7 86.0
6 8 80.0
6 9 46.0
7 9 64.0
8 10 77.0
9 10 48.0
8 22 (bloco 10)
6
8 1 3 5 7 6
1 2 87.0
1 5 30.0
1 7 56.0
1 8 72.0
2 3 59.0
2 4 40.0
2 6 85.0
2 7 53.0
2 8 86.0
3 4 33.0
3 6 4.0
3 7 21.0
3 8 23.0
4 6 41.0
4 7 70.0
4 8 54.0
5 6 95.0
5 7 10.0
5 8 71.0
6 7 93.0
6 8 73.0
7 8 13.0
10 22 (bloco 11)
0
1 2 30.0
1 3 70.0
1 4 58.0

1 5 34.0
1 7 45.0
2 3 34.0
2 6 85.0
2 8 29.0
2 10 89.0
3 6 47.0
3 7 99.0
3 8 95.0
4 5 26.0
4 8 23.0
4 10 84.0
5 6 90.0
5 8 75.0
5 10 52.0
6 7 45.0
7 8 80.0
7 9 32.0
7 10 90.0
7 13 (bloco 12)
0
1 2 71.0
1 7 89.0
2 3 44.0
2 4 63.0
2 7 22.0
3 5 94.0
3 6 89.0
3 7 96.0
4 5 85.0
4 6 79.0
4 7 15.0
5 7 70.0
6 7 80.0
5 6 (bloco 13)
0
1 2 23.0
2 4 2.0
2 5 33.0
3 4 74.0
3 5 40.0
4 5 13.0
5 7 (bloco 14)
0
1 2 50.0
1 3 73.0
1 5 95.0
2 3 70.0

2 4 61.0
3 4 57.0
4 5 46.0
10 34 (bloco 15)
0
1 2 79.0
1 4 27.0
1 5 51.0
1 7 88.0
1 8 80.0
1 9 82.0
1 10 46.0
2 3 98.0
2 4 51.0
2 5 99.0
2 6 57.0
2 7 1.0
2 9 75.0
2 10 30.0
3 4 49.0
3 5 95.0
3 7 48.0
3 9 92.0
3 10 91.0
4 5 24.0
4 8 68.0
4 9 43.0
4 10 79.0
5 6 44.0
5 8 24.0
5 9 91.0
5 10 29.0
6 7 94.0
6 8 33.0
7 8 78.0
7 10 51.0
8 9 64.0
8 10 75.0
9 10 52.0
6 8 (bloco 16)
0
1 3 52.0
1 6 28.0
2 3 66.0
2 5 47.0
3 4 0.0
3 6 31.0
4 5 33.0

5 6 13.0
6 11 (bloco 17)
0
1 2 0.0
1 3 61.0
1 5 95.0
2 3 63.0
2 4 16.0
2 5 75.0
2 6 88.0
3 4 88.0
3 6 39.0
4 5 56.0
5 6 29.0
9 25 (bloco 18)
0
1 2 77.0
1 3 49.0
1 4 10.0
1 5 26.0
1 6 16.0
1 8 81.0
1 9 2.0
2 3 9.0
2 4 67.0
2 6 60.0
2 7 43.0
2 8 65.0
3 5 10.0
3 7 80.0
3 8 37.0
3 9 4.0
4 7 38.0
4 8 68.0
4 9 54.0
5 7 72.0
5 8 21.0
6 8 80.0
7 8 72.0
7 9 96.0
8 9 7.0

Assim, a entrada é única e consiste na 1º parte seguida da 2º parte logo abaixo.

Saída: Deve ser apresentado a soma dos pesos de cada MST visitada no caminho percorrido pelo nano-robô (na verdade, o nano-robô não irá percorrer cada MST, apenas calcular seu valor).
Assim a saída da entrada de exemplo é: 322

4. Requisitos:
    - Deve ser codificado em C++ 20
    - Os contêineres da STL permitidos std:: list e std:: pair, o uso de qualquer biblioteca não é permitido com exceção de iostream, list, cstdlib, limits e iomanip;
    - As TADs das estruturas devem ser codificadas com orientação a objetos;

TAD fila de prioridade máxima com um heap binário máximo
(Basta usar a mesma ideia para desenvolver a fila de prioridade mínima com um heap binário mínimo)

Fila de prioridade (FP) (priority queue):
-Estrutura de dados que mantém um conjunto S = {e1,...,en}, onde cada elemento tem uma chave k ∈ N0, ou seja, ei.k, que indica a sua prioridade de desenfileiramento.

Estrutura da FP:
-INSERT(S, x). Insere o elemento x na fila de prioridade S.
-MAXIMUM(S). Retorna o elemento de S com chave máxima
-EXTRACT-MAX(S). Remove e retorna o elemento de S com chave máxima
-INCREASE-MAX(S, x, k). Atualiza o valor do elemento x com o novo valor de chave k (supõe-se que a chave antiga seja maior que a nova chave).

Heap binário:
É um vetor usado para representar uma árvore binária quase completa (da esquerda para a direita).

Heap binário máximo:
A chave de cada nó é maior ou igual às chaves de seus filhos.

Seja A[1..n] um heap binário máximo.
▸ A.tam = n é o tamanho do vetor A.
▸ A.tam-heap é o tamanho do heap (número de itens no heap, contidos em A).
▸ 0 ≤ A.tam-heap ≤ A.tam.
▸ A[1] é a raiz da árvore.
▸ Dado um índice i, obtém-se o pai de i e os filhos de i por meio dos procedimentos PARENT(i), LEFT(i) e RIGHT(i).

Estrutura do heap binário:

```
PARENT(i)
1       retorne [i/2]

LEFT(i)
1       retorne 2i

RIGHT(i)
1       retorne 2i + 1
```

Heap:
Seja A[1..n] um heap binário máximo.
▸ A propriedade do heap máximo deve ser satisfeita para todo nó i, não-raiz:
A[PARENT(i)] ≥ A[i].

▸ Os procedimentos MAX-HEAPFY(A, i) e BUILD-MAX-HEAP(A) mantém a propriedade do heap e constrói o heap, respectivamente.

```
MAX-HEAPFY(A, i)
1       l = LEFT(i), r = RIGHT(i)
2       se l ≤ A.tam-heap e A[l] > A[i]
3               largest = l
4       senão largest = i
5       se r ≤ A.tam-heap e A[r] > A[largest]
6               largest = r
7       se largest ≠ i
8               troque A[i] com A[largest]
9       MAX-HEAPFY(A,largest)

BUILD-MAX-HEAP(A)
1       A.tam-heap = A.tam
2       para i = ⌊A.tam/2⌋ até 1
3               MAX-HEAPFY(A, i)
```

Algoritmo de Dijkstra para caminhos mínimos com um TAD fila de prioridade mínima

Algoritmo de Dijkstra:

O algoritmo constrói uma árvore de caminhos dada por $G\pi = (V\pi, E\pi)$, onde:
- s é o vértice origem.
- v.π: armazena o vértice predecessor de v.
- v.d: acumula a "distância" (soma dos pesos) da origem s até v.
- $V\pi = \{v \in V : v.\pi \neq NULO\} \cup \{s\}$, ondes.π = NULO.
- $E\pi = \{(v.\pi, v) \in E : v \in V\pi - \{s\}\}$.
- Uma fila de prioridades mínimas Q mantém os vértices ainda não processados.
- Um conjunto S de vértices cujos pesos finais do caminho mínimo com origem s já tenham sido descobertos.

Estrutura:

```
INICIALIZA(G,s)
1       para cada v ∈ G.V
2               v.d = ∞
3               v.π = NULO
4       s.d = 0


RELAXA(u,v,w)
1       se v.d > u.d+w(u,v)
2                v.d = u.d+w(u,v)
3               v.π = u


DIJKSTRA(G,w,s)
1       INICIALIZA(G,s)
2       S = ∅
3       Q = G.V
4       enquanto Q ≠ ∅
5               u = EXTRACT-MIN(Q)
6               S = S ∪ {u}
7               para cada v ∈ G.Adj[u]
8                       RELAXA(u,v,w)
```

Algoritmo de Kruskal para obter uma árvore geradora mínima utilizando um TAD Union-Find

Elementos do algoritmo de Kruskal:
▸ Algoritmo constrói a MST no conjunto de arestas A.
▸ Utiliza uma estrutura de dados Union-Find para representar a floresta de MSTs.

Elementos do Union-Find:
▸ Estrutura de dados que trata da partição de uma coleção de conjuntos disjuntos S = {S1,S2,...,Sn}.
▸ S inicia com n subconjuntos com um único elemento, e depois sofre modificações por meio da união de alguns destes subconjuntos.

Union-Find - operações:
▸ MAKE-SET(x). Cria um novo conjunto com x como membro único (representante do conjunto).

▸ UNION(x,y). Une os dois conjuntos disjuntos Sx e Sy cujos membros são x e y, respectivamente, em um novo conjunto disjunto, digamos, Sxy.

▸ FIND-SET(x). Retorna um ponteiro para o elemento representante do único conjunto contendo x.

```
MST-KRUSKAL(G,w)
1       A = ∅
2       para cada vértice v ∈ G.V
3               MAKE-SET(v)
4       ordene as arestas de G.E em ordem
        não-decrescente de peso
5       para cada aresta {u,v} ∈ G.E // ordem acima
6               if FIND-SET(u) ≠ FIND-SET(v)
7                       A = A ∪ {u,v}
8                       UNION(u,v)
9        retorne A
```

Árvore geradora mínima

Uma Árvore Geradora (Spanning Tree) T de um grafo G é um subgrafo de G tal que:

1. T é uma árvore (acíclico e conexo).
2. T conecta todos os vértices de G.

Teorema (Cayley, 1889). Há n^(n−2) árvores geradoras sobre um grafo completo com n vértices.

Seja um grafo ponderado G = (V,E) com uma função w : E → R de pesos sobre seus vértices.

Uma Árvore Geradora Mínima (Minimum Spanning Tree - MST) T de G é uma árvore geradora cujo peso w(T) (a soma dos pesos de suas arestas) é mínimo.

~\OneDrive\Área de Trabalho\UEA\4-Período\AED2\PP1\Q3 - Copia.cpp

```cpp
// TAD grafo ponderado com lista de adjacência para referência

#include <iostream>
#include <list>

typedef unsigned int uint;
typedef unsigned int Vertex;
typedef float Weight;

class VertexWeightPair
{
    public:
        Vertex vertex;
        Weight weight;

        VertexWeightPair(Vertex v, Weight w) : vertex(v), weight(w) {} //uso de lista de inicializacao
};

class WeightedGraphAL
{
    private:
        uint num_vertices;
        uint num_edges;
        std::list<VertexWeightPair> *adj;

    public:
        WeightedGraphAL(uint num_vertices);
        ~WeightedGraphAL();
        void add_edge(Vertex u, Vertex v, Weight w);
        void remove_edge(Vertex u, Vertex v);
        uint get_num_vertices() { return num_vertices; };
        uint get_num_edges() { return num_edges; };
        std::list<VertexWeightPair> get_adj(Vertex v) { return adj[v]; };
        void print_graph();
};

WeightedGraphAL::WeightedGraphAL(uint _num_vertices): num_vertices(_num_vertices)
{
    adj = new std::list<VertexWeightPair>[num_vertices];
    num_edges = 0;
}

WeightedGraphAL::~WeightedGraphAL()
{
    for (uint i = 0; i < num_vertices; i++)
        adj[i].clear();

    delete[] adj;
    adj = nullptr;
    num_vertices = num_edges = 0;
}
```

```cpp

void WeightedGraphAL::add_edge(Vertex u, Vertex v, Weight w)
{
    adj[u].emplace_back(v, w);
    adj[v].emplace_back(u, w);
    num_edges++;
}

void WeightedGraphAL::remove_edge(Vertex u, Vertex v)
{
    // Funcoes lambda para remocao de arestas
    adj[u].remove_if( [v](const VertexWeightPair& pair) { return pair.vertex == v; } );
    adj[v].remove_if( [u](const VertexWeightPair& pair) { return pair.vertex == u; } );
    num_edges--;
}

void WeightedGraphAL::print_graph()
{
    std::cout << "num_vertices: " << get_num_vertices() << std::endl;
    std::cout << "num_edges: " << get_num_edges() << std::endl;

    for (Vertex v = 0; v < get_num_vertices(); v++) {
        std::cout << v << ": ";
        for (const auto& pair : get_adj(v)) {
            std::cout << "(" << pair.vertex << ", " << pair.weight << "), ";
        }
        std::cout << std::endl;
    }
}

int main()
{
    uint num_vertices, num_edges;
    std::cin >> num_vertices >> num_edges;

    WeightedGraphAL graph(num_vertices);
    for (uint i = 0; i < num_edges; ++i) {
        Vertex u, v;
        Weight w;
        std::cin >> u >> v >> w;
        graph.add_edge(u, v, w);
    }

    graph.print_graph();

    return 0;
}

```