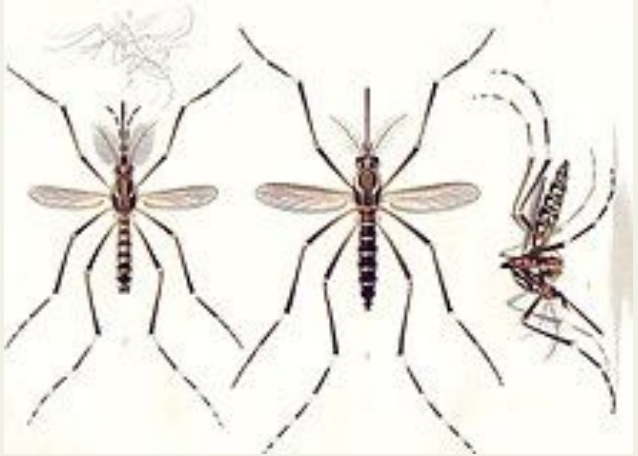
Teoria dos Grafos:

Otimização do combate à dengue em Uberlândia (MG)

Alunos: Cauã Pereira, Marcus Vinícius Almeida e Maria Fernanda Gouveia



Contexto

Segundo o professor João Carlos de Oliveira, especialista em Ciências do Ambiente e Planejamento Urbano e doutor em Geografia pela Universidade Federal de Uberlândia (UFU), o início do ano de 2023 se deu com uma alta alarmante no número de casos de dengue na cidade de Uberlândia (MG). De acordo com dados da Secretaria de Saúde do Estado, os números são 12 vezes maiores em relação ao mesmo período do ano passado, com três óbitos registrados em decorrência da doença até 6 de março. Atualmente, foram constatados mais de 127 mil casos e 70 mortes de dengue no município.

Ao analisar a progressão da dengue na cidade, é possível observar uma constante anual, com o aumento acentuado de casos após o período de chuvas entre janeiro e abril. Pode-se analisar também uma crescente alarmante de um ano para o outro: em 2022, por exemplo, o número de casos foi sete vezes maior que no ano anterior.

Algumas atitudes têm sido tomadas pela Prefeitura Municipal de Uberlândia, como a utilização de carros de fumacê, que objetivam eliminar os mosquitos, em conjunto com a promoção de mutirões de limpeza nas áreas mais afetadas. Apesar disso, a doença retorna anualmente.

O Problema

O caminhão do fumacê é um veículo que aplica inseticidas para combater o mosquito Aedes aegypti. O uso do fumacê é uma medida aplicada para controlar a dengue, mas é importante que o caminhão seja utilizado de forma eficiente para que o combate seja eficaz.

Um dos principais problemas do fumacê é a sua eficiência limitada. Como o fumacê é aplicado na forma de fumaça, ele pode não atingir todos os mosquitos adultos, especialmente aqueles que estão abrigados em locais fechados ou que estão em movimento.

Além disso, o fumacê pode ser ineficiente em áreas com muitos obstáculos, como árvores ou construções. Nesses casos, o inseticida pode ser disperso antes de atingir os mosquitos.

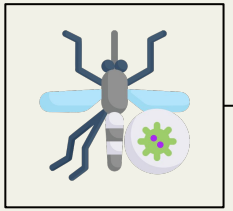
Outro problema do fumacê é o seu impacto ambiental. O inseticida utilizado no fumacê pode contaminar a água e o solo, o que pode prejudicar a saúde humana e o meio ambiente.

Modelagem do Problema

Para minimizar esses problemas, é importante que a aplicação do fumacê seja feita de forma racional e responsável. Esse projeto considera os seguintes fatores:

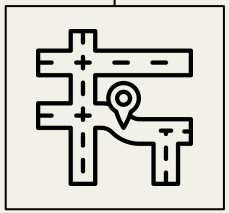
Focos

Localização dos focos de dengue



Distância

Distância entre os focos de dengue

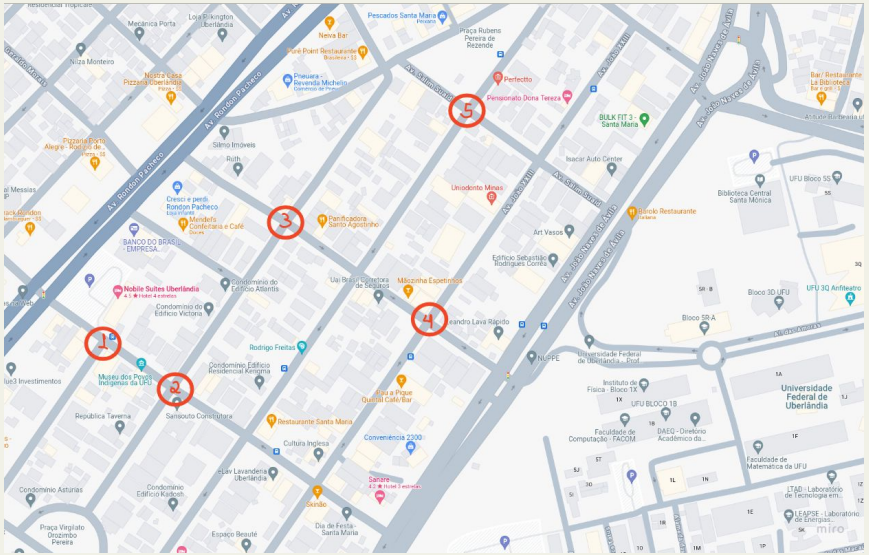


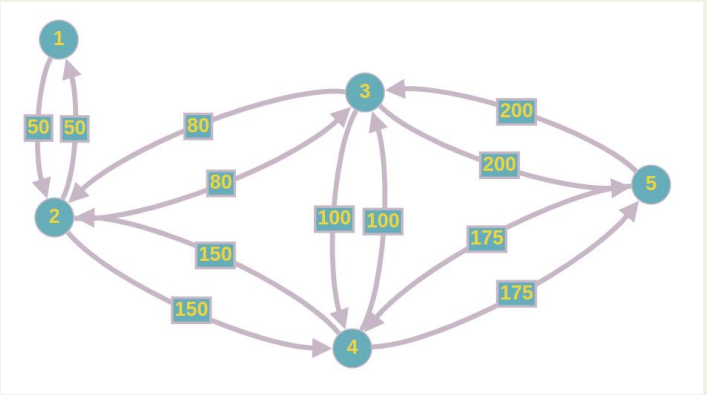
Distância Percorrida

Distância que o mosquito percorre após colocar os ovos (100 metros)

Modelagem do Problema

As ruas são modeladas como arestas do grafo. Os vértices podem ou não serem focos de dengue. O peso de cada aresta representa o comprimento das ruas.





Input

O Input consiste em três parâmetros:

- vértice de origem
- vértice de destino
- distância entre os vértices

```
Digite o numero de vertices do grafo: 5
Digite os dados do grafo (origem, destino, distancia):
Digite 0 0 0, para prosseguir!
0 1 80
0 2 60
1 2 50
1 3 90
2 4 100
3 4 60
0 0 0
```

```
=========================================================
Grafo sem Contaminação:

Lista de adjacência do vértice 0:
  Origem: 0, Destino: 2, Peso: 60, Contaminado: false
  Origem: 0, Destino: 1, Peso: 80, Contaminado: false


Lista de adjacência do vértice 1:
  Origem: 1, Destino: 3, Peso: 90, Contaminado: false
  Origem: 1, Destino: 2, Peso: 50, Contaminado: false
  Origem: 1, Destino: 0, Peso: 80, Contaminado: false


Lista de adjacência do vértice 2:
  Origem: 2, Destino: 4, Peso: 100, Contaminado: false
  Origem: 2, Destino: 1, Peso: 50, Contaminado: false
  Origem: 2, Destino: 0, Peso: 60, Contaminado: false


Lista de adjacência do vértice 3:
  Origem: 3, Destino: 4, Peso: 60, Contaminado: false
  Origem: 3, Destino: 1, Peso: 90, Contaminado: false


Lista de adjacência do vértice 4:
  Origem: 4, Destino: 3, Peso: 60, Contaminado: false
  Origem: 4, Destino: 2, Peso: 100, Contaminado: false

=========================================================
Digite os vértices que você quer contaminar:
Digite -1 para prosseguir!
3
-1
=========================================================
```

```
=========================================================
Grafo Contaminado:

Lista de adjacência do vértice 0:
  Origem: 0, Destino: 2, Peso: 60, Contaminado: false
  Origem: 0, Destino: 1, Peso: 80, Contaminado: true


Lista de adjacência do vértice 1:
  Origem: 1, Destino: 3, Peso: 90, Contaminado: true
  Origem: 1, Destino: 2, Peso: 50, Contaminado: true
  Origem: 1, Destino: 0, Peso: 80, Contaminado: true


Lista de adjacência do vértice 2:
  Origem: 2, Destino: 4, Peso: 100, Contaminado: true
  Origem: 2, Destino: 1, Peso: 50, Contaminado: true
  Origem: 2, Destino: 0, Peso: 60, Contaminado: false


Lista de adjacência do vértice 3:
  Origem: 3, Destino: 4, Peso: 60, Contaminado: true
  Origem: 3, Destino: 1, Peso: 90, Contaminado: true


Lista de adjacência do vértice 4:
  Origem: 4, Destino: 3, Peso: 60, Contaminado: true
  Origem: 4, Destino: 2, Peso: 100, Contaminado: true

=========================================================
Arestas Contaminadas:
Origem: 4, Destino: 2, Peso: 100
Origem: 4, Destino: 3, Peso: 60
Origem: 3, Destino: 1, Peso: 90
Origem: 3, Destino: 4, Peso: 60
Origem: 2, Destino: 1, Peso: 50
Origem: 2, Destino: 4, Peso: 100
Origem: 1, Destino: 0, Peso: 80
Origem: 1, Destino: 2, Peso: 50
Origem: 1, Destino: 3, Peso: 90
Origem: 0, Destino: 1, Peso: 80
=========================================================
De qual vértice o caminhão vai sair? 0
=========================================================
```

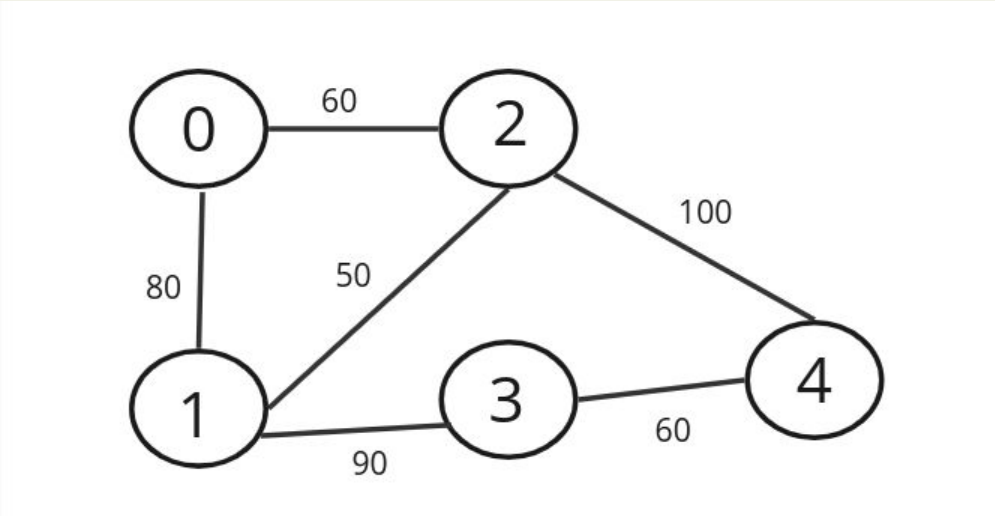
Output

O output gera:

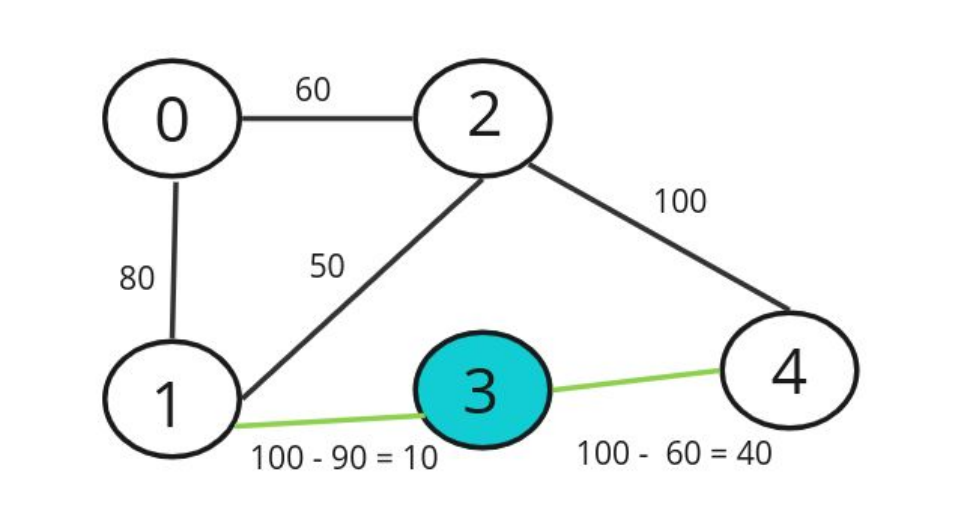
- Menor caminho
- Distância total

```
Menor Caminho:
0 1 2 4 3 1
Distancia Total: 380
```

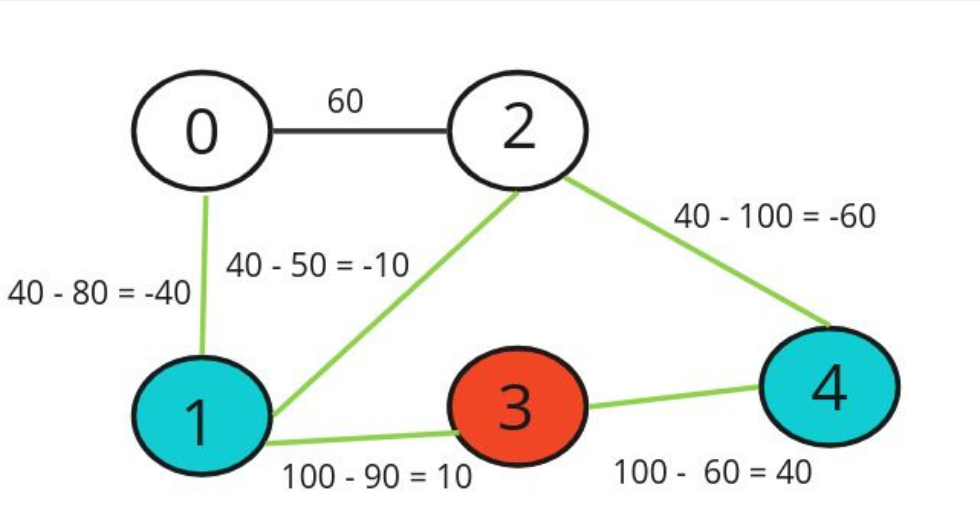
Compreendendo o Problema



Compreendendo o Problema



Compreendendo o Problema



Código Completo da Resolução

```c
// Definição da estrutura para um nó da fila
#define MAX_QUEUE_SIZE 100 // Defina o tamanho máximo da fila conforme necessário

// Definição da estrutura para um nó da fila
struct QueueNode {
    int data1;
    int data2;
};

// Definição da estrutura da fila
struct Queue {
    struct QueueNode array[MAX_QUEUE_SIZE];
    int front, rear, size;
};

// Função para criar uma fila
void createQueue(struct Queue* queue) {
    queue->size = 0;
    queue->front = 0;
    queue->rear = -1;
}
```

struct QueueNode: definição da estrutura para um nó da fila

struct Queue: definição da estrutura da fila

void createQueune: criação de uma fila

Código Completo da Resolução

```c
// Função para verificar se a fila está vazia
bool QueueisEmpty(struct Queue* queue) {
    return (queue->size == 0);
}

// Função para verificar se a fila está cheia
bool isFull(struct Queue* queue) {
    return (queue->size == MAX_QUEUE_SIZE);
}

// Função para enfileirar um elemento
void enqueue(struct Queue* queue, int data1, int data2) {
    if (isFull(queue)) {
        printf("A fila está cheia. Não é possível enfileirar.\n");
        return;
    }
    queue->rear = (queue->rear + 1) % MAX_QUEUE_SIZE;
    queue->array[queue->rear].data1 = data1;
    queue->array[queue->rear].data2 = data2;
    queue->size += 1;
}
```

bool QueueisEmpty: executa a verificação se a fila está vazia.

bool isFull: executa a verificação se a fila está cheia.

void enqueue: adiciona um novo elemento à fila. Antes de fazer isso, ela verifica se a fila está cheia usando a função isFull. Se a fila estiver cheia, imprime uma mensagem de erro e retorna imediatamente. Caso contrário, ela avança o ponteiro traseiro (rear) para a próxima posição circular (usando a operação % para garantir que a fila seja tratada como uma fila circular). Adiciona os dados fornecidos ao nó da fila na posição do ponteiro traseiro e incrementa o tamanho da fila.

Código Completo da Resolução

```c
// Função para desenfileirar um elemento
struct QueueNode dequeue(struct Queue* queue) {
    struct QueueNode emptyNode = { -1, -1 }; // Valor de sentinela para ind
    if (QueueeisEmpty(queue)) {
        printf("A fila está vazia. Não é possível desenfileirar.\n");
        return emptyNode;
    }
    struct QueueNode item = queue->array[queue->front];
    queue->front = (queue->front + 1) % MAX_QUEUE_SIZE;
    queue->size -= 1;
    return item;
}

// Função para imprimir os elementos da fila
void printQueue(struct Queue* queue) {
    if (QueueeisEmpty(queue)) {
        printf("A fila está vazia.\n");
        return;
    }

    printf("Elementos da fila: ");
    int i = queue->front;
    do {
        printf("(%d, %d) ", queue->array[i].data1, queue->array[i].data2);
        i = (i + 1) % MAX_QUEUE_SIZE;
    } while (i != (queue->rear + 1) % MAX_QUEUE_SIZE);
    printf("\n");
}
```

struct QueueNode dequeue: retira o nó na frente da fila e o retorna. Antes de realizar essa operação, ela verifica se a fila está vazia usando a função QueueeisEmpty. Se a fila estiver sem elementos, a função exibe uma mensagem de erro e retorna um nó vazio, representado por valores -1, -1, indicando que a fila está vazia. Caso a fila possuir elementos, a função obtém o nó na frente, atualiza o ponteiro da frente para apontar para o próximo nó na fila, reduz o tamanho da fila e, por fim, retorna o nó que foi desenfileirado.

void printQueue: executa a impressão dos elementos da fila

Código Completo da Resolução

```
// Definindo a estrutura de um nó da lista de arestas
struct EdgeNode {
    int src;
    int dest;
    int weight;
    int circuito_mosquito;
    bool contaminated;
    struct EdgeNode* next;
};
```

struct EdgeNode: definição da estrutura de um nó da lista de arestas

```
// Definindo a estrutura de um vértice com uma lista de arestas
struct Vertex {
    struct EdgeNode* head;
};
```

struct Vertex: definição da estrutura de um vértice com uma lista de arestas

Código Completo da Resolução

```c
void addEdge(struct Vertex graph[], int src, int dest, int weight, bool contaminated) {
    // Adiciona uma aresta da origem para o destino

    struct EdgeNode* newNode = (struct EdgeNode*)malloc(sizeof(struct EdgeNode));
    newNode->src = src;
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->contaminated = contaminated;
    newNode->circuito_mosquito = 0;
    newNode->next = graph[src].head;
    graph[src].head = newNode;

    // Adiciona a aresta do destino para a origem
    newNode = (struct EdgeNode*)malloc(sizeof(struct EdgeNode));
    newNode->src = dest;
    newNode->dest = src;
    newNode->weight = weight;
    newNode->contaminated = contaminated;
    newNode->next = graph[dest].head;
    graph[dest].head = newNode;
}

void printGraph(struct Vertex graph[], int numVertices) {
    for (int i = 0; i < numVertices; i++) {
        struct EdgeNode* current = graph[i].head;
        printf("\nLista de adjacência do vértice %d:\n", i);
        while (current != NULL) {
            printf("  Origem: %d, Destino: %d, Peso: %d, Contaminado: %s\n",
                   current->src, current->dest, current->weight,
                   current->contaminated ? "true" : "false");
            current = current->next;
        }
        printf("\n");
    }
}
```

void addEdge: adiciona uma aresta direcionada ao grafo, criando dois nós de aresta dinamicamente na memória. Cada nó representa a aresta indo da origem para o destino e vice-versa. Os campos desses nós, como vértice de origem, vértice de destino, peso da aresta e estado de contaminação, são preenchidos com as informações fornecidas. O campo circuito_mosquito é inicializado como 0. Ambos os nós são então adicionados às listas de adjacência dos vértices de origem e destino, respectivamente. Esse processo cria uma representação bidirecional da aresta no grafo.

void printGraph: imprime as informações de um grafo ponderado direcionado em forma de lista de adjacência. Para cada vértice no grafo, a função percorre sua lista de adjacência, exibindo as propriedades das arestas, como origem, destino, peso e estado de contaminação. A saída é organizada por vértice, destacando cada aresta e suas características. O estado de contaminação é indicado como "true" ou "false" para cada aresta.

Código Completo da Resolução

```c
// Função para encontrar a aresta reversa
struct EdgeNode* findReverseEdge(struct Vertex graph[], int src, int dest, int numVertices) {
    for (int i = 0; i < numVertices; i++) {
        struct EdgeNode* current = graph[i].head;
        while (current != NULL) {
            if (current->src == src && current->dest == dest) {

                return current;
            }
            current = current->next;
        }
    }
    return NULL; // Se não encontrar a aresta reversa
}

// Função para marcar a aresta reversa como contaminada
void markReverseContaminated(struct Vertex graph[], int src, int dest, int numVertices) {
    // Encontra a aresta reversa na lista (se existir)
    struct EdgeNode* reverseEdge = findReverseEdge(graph, dest, src, numVertices);

    // Marca a aresta reversa como contaminada
    if (reverseEdge != NULL) {
        reverseEdge->contaminated = true;
    }
}
```

struct findReverseEdge: procura pela aresta reversa de uma aresta específica em um grafo direcionado. Percorrendo a lista de arestas conectadas ao nó de destino no grafo, verificando se a origem e o destino de uma aresta correspondem aos valores fornecidos. Se uma correspondência é encontrada, a função retorna um ponteiro para a aresta reversa associada; caso contrário, retorna NULL, indicando que a aresta reversa não foi encontrada no grafo.

void markReverseContaminated: Tem o propósito de marcar a aresta reversa associada a uma aresta direcionada específica como contaminada. Recebe um grafo representado por um array de vértices, os índices dos vértices de origem e destino, e o número total de vértices no grafo . A função inicia procurando a aresta reversa usando findReverseEdge. Se a aresta reversa existe (se reverseEdge não for nulo), então ela marca a aresta reversa como contaminada, definindo o campo contaminated para true. Isso implica que a aresta direcionada do vértice de destino para o vértice de origem está contaminada no grafo.

Código Completo da Resolução

```c
// Função para diminuir a prioridade de um vértice na fila de prioridade
void diminuirPrioridade(struct Queue* fila, int novaPrioridade) {
    fila->array[fila->front].data2 = novaPrioridade;
}

// Função para encontrar a posição de um vértice na fila
int encontrarPosicao(struct Queue* fila, int vertice) {
    int posicao = -1;
    for (int i = fila->front; i != (fila->rear + 1) % MAX_QUEUE_SIZE; i = (i + 1) %
MAX_QUEUE_SIZE) {
        if (fila->array[i].data1 == vertice) {
            posicao = i;
            break;
        }
    }
    return posicao;
}
```

void diminuirPrioridade: diminui a propriedade de um vértice na fila de prioridade.

int encontrarPosicao: encontra a posição de um vértice na fila.

Código Completo da Resolução

```c
// Função para verificar se um vértice está na fila
bool estaNaFila(struct Queue* fila, int vertice) {
    return (encontrarPosicao(fila, vertice) != -1);
}
bool todas_arestas_contaminadas(struct Vertex graph[], int numVertices) {
    for (int i = 0; i < numVertices; i++) {
        struct EdgeNode* current = graph[i].head;
        while (current != NULL) {
            if (!current->contaminated) {
                // Se encontrar uma aresta não contaminada, retorna falso
                return false;
            }
            current = current->next;
        }
    }
    // Se todas as arestas foram contaminadas, retorna verdadeiro
    return true;
}
```

bool estaNafila: verifica se um vértice está na fila.

bool todas_arestas_contaminadas: verifica se todas as arestas de um grafo estão contaminadas. Percorre cada vértice do grafo e, para cada vértice, examina suas arestas para determinar se todas estão marcadas como contaminadas. Se encontrar pelo menos uma aresta não contaminada, a função retorna false, caso contrário, se todas as arestas estiverem contaminadas, retorna true. Em essência, a função avalia a condição de contaminação de todas as arestas no grafo.

Código Completo da Resolução

```c
void marca_arestas_contaminadas(struct Vertex graph[], int ori, int numVertices) {
    struct EdgeNode* current = graph[ori].head;
    int distancia_mosquito_percorre = 60;
    struct Queue myQueue;
    createQueue(&myQueue);

    while (current != NULL) {
        current->contaminated = true;
        //MarkReverseContaminatedEdges(graph, numVertices);
        current->circuito_mosquito = distancia_mosquito_percorre - current->weight;
        enqueue(&myQueue, current->dest, current->circuito_mosquito);
        current = current->next;
    }

    struct QueueNode dequeuedNode;
    while (!QueueisEmpty(&myQueue)) {
        if(todas_arestas_contaminadas(graph, numVertices)){
           break;
        }
        dequeuedNode = dequeue(&myQueue);
        int u = dequeuedNode.data1;
        int v = dequeuedNode.data2;
        struct EdgeNode* temp = graph[u].head;
        while (temp != NULL) {
            if (v <= 0) break;
            temp->contaminated = true;
            temp->circuito_mosquito = v - temp->weight;
            enqueue(&myQueue, temp->dest, temp->circuito_mosquito);
            temp = temp->next;
        }
    }
    markReverseContaminatedEdges(graph, numVertices);
}
```

void marca_arestas_contaminadas: começa marcando todas as arestas que saem do vértice de origem como contaminadas.
Cria uma fila de prioridade com os vértices adjacentes ao vértice de origem.
Enquanto a fila de prioridade não estiver vazia: desenfileira o vértice com a menor distância. Marca todas as arestas que saem do vértice desenfileirado como contaminadas. Adiciona os vértices adjacentes ao vértice desenfileirado à fila de prioridade.

Código Completo da Resolução

```c
void printContaminatedEdges(struct EdgeNode* edges) {
    while (edges != NULL) {
        if (edges->contaminated) {
            printf("Origem: %d, Destino: %d, Peso: %d\n", edges->src, edges->dest, edges->weight);
        }
        edges = edges->next;
    }
}

struct Resultado {
    int* caminhoInvertido;
    int tamanhoCaminho;
    int distanciaTotal;
    int penultimoVertice;
    int ultimoVertice;
};
```

void printContaminatedEdges: imprime as informações das arestas contaminadas de uma lista encadeada de arestas (struct EdgeNode). Recebe como parâmetro um ponteiro para o início da lista de arestas (struct EdgeNode* edges).

struct Resultado: utilizada para armazenar resultados dos caminhos no grafo, incluindo informações sobre o próprio caminho (como a sequência de vértices), o tamanho do caminho, a distância total percorrida, e os vértices inicial e final do caminho.

Código Completo da Resolução

```c
// Função para encontrar o menor caminho usando o algoritmo de Dijkstra
void dijkstra(struct Vertex graph[], int numVertices, int inicio, int destino, int* distancia,
int* caminho) {
    // Inicializa a fila de prioridade como vazia
    struct Queue filaPrioridade;
    createQueue(&filaPrioridade);

    // Inicializa a distância do vértice de início como 0
    distancia[inicio] = 0;
    enqueue(&filaPrioridade, inicio, 0);

    // Loop principal do algoritmo de Dijkstra
    while (!QueueisEmpty(&filaPrioridade)) {
        struct QueueNode noAtual = dequeue(&filaPrioridade);
        int verticeAtual = noAtual.data1;
        int distAtual = noAtual.data2;

        // Percorre as arestas do vértice atual
        struct EdgeNode* current = graph[verticeAtual].head;
        while (current != NULL) {
            int novoPeso = distAtual + current->weight;

            // Se encontrarmos um caminho mais curto para um vizinho, atualizamos a distância
            if (novoPeso < distancia[current->dest]) {
                distancia[current->dest] = novoPeso;
                caminho[current->dest] = verticeAtual;
                enqueue(&filaPrioridade, current->dest, novoPeso);
            }

            current = current->next;
        }
    }
}
```

void disjkstra: implementa o algoritmo de Dijkstra, utilizado para encontrar o caminho mais curto entre dois vértices em um grafo ponderado. Ele usa uma fila de prioridade para processar os vértices de forma ordenada, calculando as distâncias mínimas à medida que avança pelo grafo. Resultado final é um conjunto de distâncias mínimas e um conjunto de vértices que formam o caminho mais curto do vértice de início ao vértice de destino.

Código Completo da Resolução

```c
// Função para imprimir o caminho ao contrário usando uma pilha
int* imprimirCaminhoInvertido(int* caminho, int destino, int* tamanhoCaminho, int numVertices)
{
    // Usar uma pilha para armazenar o caminho invertido
    int* pilha = (int*)malloc(numVertices * sizeof(int));
    int topo = -1;

    // Adicionar os vértices do caminho à pilha
    int verticeAtual = destino;
    while (verticeAtual != -1) {
        pilha[++topo] = verticeAtual;
        verticeAtual = caminho[verticeAtual];
    }

    // Criar um vetor para armazenar o caminho invertido
    int* caminhoInvertido = (int*)malloc((topo + 1) * sizeof(int));
    *tamanhoCaminho = topo + 1;

    // Preencher o vetor com os elementos do caminho invertido
    for (int i = 0; i <= topo; i++) {
        caminhoInvertido[i] = pilha[topo - i];
    }

    // Liberar memória alocada para a pilha
    free(pilha);

    // Retornar o vetor com os elementos do caminho invertido
    return caminhoInvertido;
}
```

int imprimirCaminhoInvertido: recebe informações sobre um caminho em um grafo e retornar esse caminho de forma invertida. O caminho é originalmente representado por um vetor chamado caminho, onde cada elemento indica o vértice anterior no caminho até um determinado vértice de destino. O caminho começa no vértice de destino e percorre seus predecessores até chegar a um ponto inicial.Usando uma pilha para inverter a ordem dos vértices no caminho. Ela percorre o vetor caminho, empilhando cada vértice na pilha até atingir o ponto inicial (onde o valor do vértice é -1). Cria um novo vetor chamado caminhoInvertido e o preenche com os elementos da pilha.Vetor caminhoInvertido é retornado, também fornece o tamanho do caminho por meio do parâmetro tamanhoCaminho.

Código Completo da Resolução

```c
struct Resultado encontrarCaminhoComNovoElemento(struct Vertex graph[], int numVertices, int inicio, int destino, int novoElemento) {
    // Inicializar arrays para armazenar distância e caminho para Dijkstra
    int* distancia = (int*)malloc(numVertices * sizeof(int));
    int* caminho = (int*)malloc(numVertices * sizeof(int));

    // Inicializar distância e caminho para Dijkstra
    for (int i = 0; i < numVertices; i++) {
        distancia[i] = INT_MAX;
        caminho[i] = -1;
    }

    // Executar o algoritmo de Dijkstra
    dijkstra(graph, numVertices, inicio, destino, distancia, caminho);

    // Chamar a função para obter o caminho invertido
    int tamanhoCaminho;
    int* caminhoInvertido = imprimirCaminhoInvertido(caminho, destino, &tamanhoCaminho, numVertices);
```

```c
    caminhoInvertido = (int*)realloc(caminhoInvertido, (tamanhoCaminho + 1) * sizeof(int));
    caminhoInvertido[tamanhoCaminho] = novoElemento;
    tamanhoCaminho++;

    // Calcular a distância total
    int distanciaTotal = distancia[destino];

    // Liberar memória alocada
    free(distancia);
    free(caminho);

    // Criar e retornar a estrutura de resultado
    struct Resultado resultado;
    resultado.caminhoInvertido = caminhoInvertido;
    resultado.tamanhoCaminho = tamanhoCaminho;
    resultado.distanciaTotal = distanciaTotal;

    return resultado;
}
```

struct encontrarCaminhoNovoElemento: utiliza o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois vértices em um grafo. Ela aloca memória para armazenar informações sobre a distância e o caminho mais curto, inicializa esses dados, calcula o caminho e as distâncias usando Dijkstra, obtém o caminho invertido entre os vértices de início e destino, adiciona um novo elemento a esse caminho, calcula a distância total percorrida, libera a memória alocada e retorna uma estrutura contendo o caminho invertido, o tamanho do caminho e a distância total.

Código Completo da Resolução

```
// Função para processar uma aresta e armazenar origem e destino em uma estrutura
struct Resultado processarAresta(struct EdgeNode* aresta, int inicioDijkstra, struct Vertex
graph[], int numVertices) {

    int destinoDijkstra = aresta->dest;

    // Chamar a função para encontrar o caminho invertido com um novo elemento
    struct Resultado resultado = encontrarCaminhoComNovoElemento(graph, numVertices,
inicioDijkstra, destinoDijkstra, aresta->src);

    destinoDijkstra = aresta->src;

    // Chamar a função para encontrar o caminho invertido com um novo elemento
    struct Resultado resultado2 = encontrarCaminhoComNovoElemento(graph, numVertices,
inicioDijkstra, destinoDijkstra, aresta->dest);

    resultado2.ultimoVertice = aresta->dest;
    resultado2.penultimoVertice = aresta->src;
    resultado2.distanciaTotal = resultado2.distanciaTotal + aresta->weight;
    resultado.ultimoVertice = aresta->src;
    resultado.penultimoVertice = aresta->dest;
    resultado.distanciaTotal = resultado.distanciaTotal + aresta->weight;
    if(resultado.distanciaTotal > resultado2.distanciaTotal) return resultado2;
    return resultado;
}
```

struct processarAresta: recebe uma aresta (struct EdgeNode* aresta), o vértice de início para o algoritmo de Dijkstra (int inicioDijkstra), um array de vértices representando o grafo (struct Vertex graph[]), e o número total de vértices no grafo (int numVertices). Processa uma aresta específica, considerando dois cenários: quando o vértice de origem e quando o vértice de destino da aresta são adicionados como novos elementos no caminho, utilizando o algoritmo de Dijkstra modificado(encontrarCaminhoComNovoElemento). Resultado final é o caminho mais curto entre os vértices de início e destino do Dijkstra, considerando a inclusão da aresta como parte desse caminho.

Código Completo da Resolução

```c
// Função para obter as arestas contaminadas de um grafo
struct EdgeNode* getContaminatedEdges(struct Vertex graph[], int numVertices) {
    struct EdgeNode* contaminatedEdges = NULL; // Inicializando como nulo

    for (int i = 0; i < numVertices; i++) {
        struct EdgeNode* current = graph[i].head;
        while (current != NULL) {
            if (current->contaminated) {
                // Adicionando a aresta contaminada ao vetor
                struct EdgeNode* newNode = (struct EdgeNode*)malloc(sizeof(struct EdgeNode));
                newNode->src = current->src;
                newNode->dest = current->dest;
                newNode->weight = current->weight;
                newNode->contaminated = current->contaminated;
                newNode->next = contaminatedEdges;
                contaminatedEdges = newNode;
            }
            current = current->next;
        }
    }

    return contaminatedEdges;
}

// Função para liberar a memória alocada para as arestas
void freeEdges(struct EdgeNode* edges) {
    while (edges != NULL) {
        struct EdgeNode* temp = edges;
        edges = edges->next;
        free(temp);
    }
}
```

struct getContaminatedEdges: Percorre um grafo representado por um array de vértices (graph) e cria uma nova lista encadeada contendo todas as arestas contaminadas do grafo. A função itera sobre todos os vértices do grafo, e para cada vértice, percorre a lista de arestas associada a esse vértice. Se uma aresta estiver marcada como contaminada, a função cria um novo nó de aresta (struct EdgeNode) com as mesmas informações da aresta original e o adiciona no início da lista de arestas contaminadas (contaminatedEdges).

void freeEdges: libera a memória alocada para as arestas.

Código Completo da Resolução

```c
// Função para remover uma aresta do grafo
void removeEdge(struct EdgeNode** headRef, int src, int dest) {
    struct EdgeNode* current = *headRef;
    struct EdgeNode* prev = NULL;

    // Procura a aresta a ser removida na lista
    while (current != NULL && (current->src != src || current->dest != dest)) {
        prev = current;
        current = current->next;
    }

    // Se a aresta foi encontrada, a remove da lista
    if (current != NULL) {
        if (prev != NULL) {
            prev->next = current->next;
        } else {
            *headRef = current->next;
        }

        free(current);
    }
}
```

void removeEdge: remove uma aresta específica de uma lista encadeada de arestas. Ela procura pela aresta com vértices de origem e destino especificados na lista, ajusta os ponteiros para "pular" a aresta e, em seguida, libera a memória alocada para o nó da aresta removida. A remoção é realizada a partir do ponteiro da cabeça da lista (headRef). Se a aresta não está presente na lista, a função não realiza alterações.

Código Completo da Resolução

```c
// Função para encontrar o caminho mais curto que passa por todas as arestas contaminadas
void findShortestPathToContaminatedEdges(struct Vertex graph[], int numVertices, int src,
struct EdgeNode* contaminatedEdges){
    struct Resultado resultado2;
    resultado2.distanciaTotal = INT_MAX;
    struct EdgeNode* contaminatedEdges2 = contaminatedEdges;
    struct Resultado resultado;
    while (contaminatedEdges != NULL) {
      // Função feita para encontrar o menor caminho até a aresta contaminada mais próxima
      struct Resultado resultado = processarAresta(contaminatedEdges, src, graph, numVertices);


        if(resultado.distanciaTotal < resultado2.distanciaTotal){
            resultado2 = resultado;
        }
        // Move para a próxima aresta contaminada
        contaminatedEdges = contaminatedEdges->next;
    }
    // Imprimir o caminho invertido
    printf("\nMenor Caminho:\n");
    for (int i = 0; i < resultado2.tamanhoCaminho; i++) {
        printf("%d ", resultado2.caminhoInvertido[i]);
    }

    int distanciaGlobal = resultado2.distanciaTotal;
```

```c
    removeEdge(&contaminatedEdges2, resultado2.penultimoVertice, resultado2.ultimoVertice);
    removeEdge(&contaminatedEdges2, resultado2.ultimoVertice, resultado2.penultimoVertice);

    while(contaminatedEdges2 != NULL){
      int ultimoVerticeEntrada = resultado2.ultimoVertice;
      contaminatedEdges = contaminatedEdges2;

      resultado2.distanciaTotal = INT_MAX;
      while (contaminatedEdges != NULL) {

          resultado = processarAresta(contaminatedEdges, ultimoVerticeEntrada, graph,
numVertices);

          if(resultado.distanciaTotal < resultado2.distanciaTotal){
              resultado2 = resultado;
          }
          // Move para a próxima aresta contaminada
          contaminatedEdges = contaminatedEdges->next;
      }
      // Imprimir o caminho invertido
      for (int i = 1; i < resultado2.tamanhoCaminho; i++) {
          printf("%d ", resultado2.caminhoInvertido[i]);
      }

      // Imprimir a distância total
      distanciaGlobal += resultado2.distanciaTotal;
      removeEdge(&contaminatedEdges2, resultado2.penultimoVertice, resultado2.ultimoVertice);
      removeEdge(&contaminatedEdges2, resultado2.ultimoVertice, resultado2.penultimoVertice);
    }
    printf("\nDistancia Total: %d\n", distanciaGlobal);
}
```

void findShortestPathToContaminatedEdges: busca para cada aresta contaminada, o caminho mais curto a partir de um ponto de origem (src) usando o algoritmo de Dijkstra. Em seguida, imprime esse caminho e remove as arestas utilizadas desse conjunto de arestas contaminadas. O processo é repetido para todas as arestas contaminadas, utilizando o último vértice do caminho anterior como novo ponto de origem. Ao final, o código imprime a distância total global percorrida nos caminhos mais curtos.

Função Main

```c
int main() {
    int numVertices;
    printf("Digite o numero de vertices do grafo: ");
    scanf( "%d", &numVertices);

    // Criando um array de vértices
    struct Vertex graph[numVertices];

    // Inicializando as listas de arestas
    for (int i = 0; i < numVertices; i++) {
        graph[i].head = NULL;
    }

    // Adicionando arestas ao grafo bidirecionado
    int ori, dest, peso;

    printf("Digite os dados do grafo (origem, destino, distancia):\n");
    printf("Digite 0 0 0, para prosseguir!\n");
    while (1) {
        scanf(" %d %d %d", &ori, &dest, &peso);
        if (ori == 0 && dest == 0 && peso == 0) break;
        addEdge(graph, ori, dest, peso, false);
    }

    printf("============================================================\n");
    printf("Grafo sem Contaminação:\n");
    printGraph(graph, numVertices);

    printf("============================================================\n");
    printf("Digite os vértices que você quer contaminar:\n");
    printf("Digite -1 para prosseguir!\n");
    // Lendo arestas contaminadas
    while (1) {
        scanf("%d", &ori);
        if (ori == -1) break;
        marca_arestas_contaminadas(graph, ori, numVertices);
    }

    printf("============================================================\n");
    printf("Grafo Contaminado:\n");
    printGraph(graph, numVertices);

    struct EdgeNode* contaminatedEdges = getContaminatedEdges(graph, numVertices);

    printf("============================================================\n");
    printf("Arestas Contaminadas:\n");
    printContaminatedEdges(contaminatedEdges);

    int partida;
    printf("============================================================\n");
    printf("De qual vértice o caminhão vai sair? ");
    scanf("%d", &partida);
    printf("============================================================\n");
    findShortestPathToContaminatedEdges(graph, numVertices, partida, contaminatedEdges);


    return 0;
}
```

Obrigado !

Referências

https://comunica.ufu.br/noticias/2023/05/e-possivel-pensar-no-fim-da-dengue

https://jc.ne10.uol.com.br/colunas/saude-e-bem-estar/2023/03/15201478-fumace-falta-de-inseticida-preocupa-ministerio-da-saude-que-pede-intensificacao-no-controle-da-dengue.html

ARAÚJO, V. E. M. D. et al. Aumento da Carga de Dengue no Brasil e UnidadesFederadas, 2000 e 2015: análise do Global Burden of Disease Study 2015. Revista Brasileira de Epidemiologia, [s. l.], v. 20, p. 205-216. 2017. Disponível em: http://www.scielo.br/pdf/rbepid/v20s1/1980-5497-rbepid-20-s1-00205.pdf.