Projeto e Análise de Algoritmos - Lista 02

Maria Eduarda Mesquita Magalhães

FGV EMAp

1 Dado um grafo $G = (V, E)$, uma árvore enraizada T de G produzida por uma varredura, e uma lista L_1 contendo vértices dessa árvore, escreva três algoritmos que retornem uma lista L_2 contendo vértices de forma que um vértice está nessa lista se e somente se ele está em L_1 ou:

(a) possui um número ímpar de descendentes em L_2.

```
// Função auxiliar para contar os descendentes em L2
int countDescendants(TreeNode* node, const unordered_set<vertex>& L2) {
    int count = 0;
    for (TreeNode* filho : node->filhos) {
        if (L2.count(filho->v)) {
            count++;
        }
        count += countDescendants(filho, L2); // Contagem recursiva
    }
    return count;
}

// Função para encontrar vértices com número ímpar de descendentes em L2
unordered_set<vertex> findDescendantCondition(const unordered_set<vertex>& L1,
GraphAdjList& graph) {
    unordered_set<vertex> L2 = L1;
    vector<TreeNode*> nodes(graph.numVertices(), nullptr);

    // Construa a árvore a partir do grafo
    buildTree(graph, 0, nodes, nullptr);  // Supondo que o vértice 0 seja a raiz da árvore

    for (vertex v = 0; v < graph.numVertices(); v++) {
        int descendantCount = countDescendants(nodes[v], L2);
        if (descendantCount % 2 == 1) {
            L2.insert(v);  // Se o número de descendentes é ímpar, adicione v a L2
        }
    }

    return L2;
}
```

O algoritmo para contar os descendentes em um grafo tem como objetivo identificar os vértices que possuem um número ímpar de descendentes em uma árvore construída a partir do grafo. Para isso,

começamos com uma árvore de expansão gerada por uma busca em profundidade a partir de um vértice raiz. A partir dessa árvore, o algoritmo recursivamente conta os descendentes de cada nó, verificando se o número de descendentes é ímpar. Se for, o vértice é adicionado a um conjunto $L2$. A contagem de descendentes é feita percorrendo todos os filhos de um nó e seus filhos, até que todos os descendentes sejam contados.

A principal característica desse algoritmo é a recursividade na contagem de descendentes. Ao percorrer cada nó, o algoritmo segue a estrutura da árvore para calcular o número de filhos e netos de cada nó, e assim sucessivamente. A complexidade do algoritmo é $O(V + E)$, onde V é o número de vértices e E é o número de arestas do grafo, uma vez que realizamos uma busca em profundidade (DFS) para construir a árvore e, em seguida, contamos os descendentes para cada nó. Esse tempo é eficiente para grafos esparsos e densos.

(b) possui um número ímpar de ancestrais em L_2.

```
// Função auxiliar para contar os ancestrais em L2
int countAncestors(TreeNode* node, const unordered_set<vertex>& L2) {
    int count = 0;
    TreeNode* current = node->pai;
    while (current) {
        if (L2.count(current->v)) {
            count++;
        }
        current = current->pai;
    }
    return count;
}

// Função para encontrar vértices com número ímpar de ancestrais em L2
unordered_set<vertex> findAncestorCondition(const unordered_set<vertex>& L1,
GraphAdjList& graph) {
    unordered_set<vertex> L2 = L1;
    vector<TreeNode*> nodes(graph.numVertices(), nullptr);

    // Construa a árvore a partir do grafo
    buildTree(graph, 0, nodes, nullptr);  // Supondo que o vértice 0 seja a raiz da árvore

    for (vertex v = 0; v < graph.numVertices(); v++) {
        int ancestorCount = countAncestors(nodes[v], L2);
        if (ancestorCount % 2 == 1) {
            L2.insert(v);  // Se o número de ancestrais é ímpar, adicione v a L2
        }
    }

    return L2;
}
```

O algoritmo para contar os ancestrais em um grafo segue uma lógica similar ao algoritmo de contagem de descendentes, mas ao invés de explorar os filhos de cada nó, ele percorre os pais de cada vértice até a raiz da árvore. O objetivo é verificar se o número de ancestrais de um vértice é ímpar. Se for, o vértice é incluído no conjunto $L2$. Assim como no algoritmo de descendentes, esse algoritmo também utiliza a árvore de expansão gerada pela busca em profundidade.

O principal desafio neste algoritmo é percorrer corretamente a cadeia de pais de cada vértice. Isso é feito de maneira iterativa, indo de um nó até a raiz, contando quantos ancestrais estão presentes no

conjunto $L2$. A complexidade desse algoritmo também é $O(V + E)$, pois a busca em profundidade precisa ser realizada uma vez para construir a árvore e, em seguida, cada nó é visitado para contar seus ancestrais. Assim, o algoritmo é eficiente para grafos com grande número de vértices e arestas.

(c) possui um número ímpar de primos mais velhos em L_2.

```
// Função auxiliar para contar primos mais velhos em L2
int countOlderCousins(TreeNode* node, const unordered_set<vertex>& L2) {
    int count = 0;
    TreeNode* parent = node->pai;

    if (parent) {
        // Para cada irmão do pai de node
        for (TreeNode* sibling : parent->filhos) {
            if (sibling != node) {
                // Contar primos mais velhos (vértices no mesmo nível, mas não irmãos)
                for (TreeNode* cousin : sibling->filhos) {
                    if (L2.count(cousin->v)) {
                        count++;
                    }
                }
            }
        }
    }

    return count;
}

// Função para encontrar vértices com número ímpar de primos mais velhos em L2
unordered_set<vertex> findCousinCondition(const unordered_set<vertex>& L1, GraphAdjList& graph) {
    unordered_set<vertex> L2 = L1;
    vector<TreeNode*> nodes(graph.numVertices(), nullptr);

    // Construa a árvore a partir do grafo
    buildTree(graph, 0, nodes, nullptr);  // Supondo que o vértice 0 seja a raiz da árvore

    for (vertex v = 0; v < graph.numVertices(); v++) {
        int cousinCount = countOlderCousins(nodes[v], L2);
        if (cousinCount % 2 == 1) {
            L2.insert(v);  // Se o número de primos mais velhos é ímpar, adicione v a L2
        }
    }

    return L2;
}
```

O algoritmo de contagem de primos mais velhos lida com uma situação mais complexa, onde é necessário verificar os irmãos e primos de um nó. Nesse caso, para cada nó, o algoritmo verifica seus irmãos e, para cada irmão, seus filhos (ou seja, os primos do nó). O objetivo é contar quantos primos mais velhos estão presentes no conjunto $L2$ e, se o número for ímpar, o nó é adicionado a esse conjunto. O algoritmo segue uma abordagem semelhante à do algoritmo de ancestrais, mas com uma maior profundidade de análise, incluindo irmãos e primos.

A complexidade deste algoritmo também é $O(V + E)$, uma vez que, como nos algoritmos anteriores, a busca em profundidade é utilizada para construir a árvore de expansão. O problema dos primos mais velhos

implica em um número maior de verificações, pois para cada nó, precisamos percorrer todos os irmãos e seus filhos. Contudo, como estamos lidando com uma estrutura de árvore, o número de operações adicionais é linear em relação ao número de vértices e arestas. Portanto, a complexidade permanece linear no tamanho do grafo.

2 Dado um grafo direcionado fortemente conectado sem pesos e uma lista L de vértices no grafo, retorne o menor caminho que passe por todos vértices de L em ordem.

```
// Função de BFS para encontrar o caminho mais curto entre dois vértices
vector<vertex> bfs(const GraphAdjList& graph, vertex start, vertex end) {
    vector<int> parent(graph.numVertices(), -1);  // Para reconstruir o caminho
    vector<bool> visited(graph.numVertices(), false);
    queue<vertex> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        vertex u = q.front();
        q.pop();

        if (u == end) {
            break;  // Encontramos o destino, podemos parar a busca
        }

        for (EdgeNode* edge = graph.getAdj(u); edge != nullptr; edge = edge->next()) {
            vertex v = edge->otherVertex();
            if (!visited[v]) {
                visited[v] = true;
                parent[v] = u;
                q.push(v);
            }
        }
    }

    // Reconstruir o caminho de start a end
    vector<vertex> path;
    for (vertex v = end; v != -1; v = parent[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());  // Reverter o caminho para a ordem correta
    return path;
}

// Função principal que encontra o menor caminho passando por todos os vértices de L em ordem
vector<vertex> findMinPathThroughList(GraphAdjList& graph, const vector<vertex>& L) {
    vector<vertex> fullPath;

    for (size_t i = 0; i < L.size() - 1; ++i) {
        vector<vertex> path = bfs(graph, L[i], L[i+1]);
        if (path.empty()) {
```

```
            return {};  // Se não houver caminho entre L[i] e L[i+1], retornar vetor vazio
        }

        // Adiciona o caminho entre L[i] e L[i+1] ao caminho total (sem repetir o vértice inicial)
        if (!fullPath.empty()) {
            path.erase(path.begin());  // Remove o primeiro vértice, pois já foi incluído
        }
        fullPath.insert(fullPath.end(), path.begin(), path.end());
    }

    return fullPath;
}
```

A função fornecida tem como objetivo resolver um problema clássico de grafos: encontrar o caminho mais curto que passe por um conjunto específico de vértices em um grafo direcionado fortemente conectado, sem pesos, e dado um conjunto de vértices L. O grafo é representado por listas de adjacência, e a lista L contém os vértices pelos quais o caminho deve passar, seguindo uma ordem específica. O algoritmo utiliza uma abordagem de busca em largura (BFS) para encontrar o caminho mais curto entre dois vértices consecutivos da lista L e, assim, construir um caminho que passe por todos os vértices de L, na ordem dada.

A função principal do algoritmo é `findMinPathThroughList`, que percorre a lista L de vértices, buscando o menor caminho entre cada par consecutivo de vértices. Para isso, a função utiliza a busca em largura (BFS), que é ideal para encontrar o caminho mais curto em grafos não ponderados. A função `bfs` recebe dois vértices (início e fim) e retorna o caminho mais curto entre eles, utilizando uma fila para explorar os vértices de forma iterativa, garantindo que o caminho seja encontrado na ordem mais eficiente. O vetor `parent` é usado para reconstruir o caminho após a busca, já que a BFS, por si só, apenas fornece os vértices visitados.

O algoritmo começa percorrendo a lista L, que contém os vértices pelos quais o caminho precisa passar. Para cada par consecutivo de vértices na lista L, a função `bfs` é chamada para calcular o caminho mais curto entre eles. Uma vez encontrado o caminho, ele é adicionado ao caminho total, mas o primeiro vértice de cada novo caminho é removido, para evitar que o vértice seja repetido. Este procedimento é repetido para todos os pares de vértices consecutivos na lista L, garantindo que o caminho seja formado de maneira eficiente, sem retrabalho e sem incluir vértices desnecessários.

Em termos de complexidade, a função `bfs` tem um custo de $O(V+E)$, onde V é o número de vértices e E é o número de arestas no grafo. Isso ocorre porque a BFS visita cada vértice uma vez e processa cada aresta uma vez. A função `findMinPathThroughList`, por outro lado, chama a função `bfs` para cada par consecutivo de vértices na lista L, o que implica que a função realiza $L-1$ chamadas para `bfs`, onde L é o número de vértices na lista. Portanto, a complexidade total do algoritmo é $O((L-1)\times(V+E)) = O(L\times(V+E))$, onde L é o número de vértices na lista L, e V e E são o número de vértices e arestas no grafo, respectivamente.

Essa complexidade é eficiente para o tipo de problema abordado, já que a BFS é uma abordagem ótima para encontrar o caminho mais curto em grafos não ponderados. Além disso, a multiplicação pela quantidade de vértices em L (representando o número de segmentos do caminho a serem encontrados) não representa uma complexidade excessiva, dado que L é tipicamente menor do que V e E, e, portanto, o algoritmo pode ser executado de maneira eficaz mesmo em grafos relativamente grandes. O uso de BFS, uma técnica bastante eficiente, permite que o algoritmo lide com grafos direcionados de forma bastante eficaz, mantendo a eficiência computacional necessária para grafos de grande escala.

Ao analisar a complexidade, podemos observar que a solução é bastante escalável, especialmente quando comparada a outras abordagens mais ingênuas, como algoritmos de busca exaustiva ou até mesmo algoritmos de programação dinâmica que poderiam ser usados para resolver um problema de caminho mínimo em um conjunto de vértices. A escolha do BFS como a técnica central da solução permite que o algoritmo se concentre nas características do grafo direcionado sem pesos, aproveitando sua estrutura para encontrar caminhos eficientes de maneira incremental. Isso torna o algoritmo uma solução bastante robusta e eficiente para o problema proposto, além de ser fácil de entender e implementar.

3 Dado um grafo direcionado fortemente conectado com pesos e uma lista L de vértices no grafo, retorne o vértice cuja maior distância de um vértice da lista até ele seja mínima, ou seja, retorne o vértice V de forma que:

$$\min\left(\max\{d(L[0], V), d(L[1], V), \ldots, d(L[N-1], V)\}\right)$$

seja mínimo.

```
// Função auxiliar para calcular a menor distância de uma fonte usando Dijkstra
vector<int> dijkstra(GraphAdjList &graph, vertex source) {
    int numVertices = graph.numVertices(); // Número de vértices
    vector<int> dist(numVertices, INF);    // Inicializa as distâncias como infinito
    vector<bool> visited(numVertices, false); // Marca os vértices visitados
    dist[source] = 0;

    // Fila de prioridade (min-heap)
    priority_queue<pair<int, vertex>, vector<pair<int, vertex>>, greater<pair<int, vertex>>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int d = pq.top().first;
        vertex u = pq.top().second;
        pq.pop();

        if (visited[u]) continue; // Ignorar se já foi processado
        visited[u] = true;

        for (const auto &[v, weight] : graph.getAdj(u)) {
            if (!visited[v] && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist; // Retorna as menores distâncias do vértice de origem
}

// Função principal para encontrar o vértice desejado
vertex findOptimalVertex(GraphAdjList &graph, const vector<vertex> &L) {
    int numVertices = graph.numVertices();
    vector<vector<int>> allDistances(L.size(), vector<int>(numVertices, INF));

    // Calcula as distâncias de cada vértice na lista L
    for (size_t i = 0; i < L.size(); i++) {
        allDistances[i] = dijkstra(graph, L[i]);
    }

    vertex optimalVertex = -1;
    int minMaxDistance = INF;

    // Para cada vértice do grafo, calcula o máximo das distâncias
```

```
    for (vertex v = 0; v < numVertices; v++) {
        int maxDistance = 0;
        for (size_t i = 0; i < L.size(); i++) {
            maxDistance = max(maxDistance, allDistances[i][v]);
        }
        if (maxDistance < minMaxDistance) {
            minMaxDistance = maxDistance;
            optimalVertex = v;
        }
    }

    return optimalVertex;
}
```

Este algoritmo tem como objetivo encontrar o vértice mais "central" em um grafo, considerando a menor distância entre um conjunto de vértices e os demais do grafo. A função `findOptimalVertex` recebe uma lista de vértices de interesse e retorna o vértice que minimiza o maior valor das distâncias de todos os vértices da lista para os demais vértices do grafo. O algoritmo faz uso do algoritmo de *Dijkstra* para calcular as menores distâncias a partir de cada vértice da lista, e posteriormente, determina o vértice "ótimo" com base nas distâncias calculadas.

A função auxiliar `dijkstra` é responsável por calcular a menor distância entre o vértice de origem e todos os outros vértices do grafo. Este é um problema clássico de grafos, que é resolvido de maneira eficiente utilizando uma fila de prioridade (min-heap), onde, a cada iteração, o vértice com a menor distância conhecida é processado. A complexidade temporal dessa função é dominada pela operação de extração e inserção na fila de prioridade. Para um grafo com V vértices e E arestas, a complexidade do algoritmo de *Dijkstra* com min-heap é $O((V + E) \log V)$.

No contexto do problema maior, a função `findOptimalVertex` invoca a função `dijkstra` para calcular as distâncias mínimas de cada vértice na lista L para todos os outros vértices do grafo. O resultado de cada execução de `dijkstra` é armazenado em uma matriz `allDistances`, onde cada linha contém as distâncias de um vértice de origem específico para todos os vértices do grafo. Isso faz com que o tempo de execução da função `findOptimalVertex` seja proporcional ao número de vértices na lista L e ao tempo necessário para calcular as distâncias de Dijkstra, resultando em uma complexidade de $O(L \cdot (V + E) \log V)$.

Após calcular as distâncias, a função `findOptimalVertex` percorre todos os vértices do grafo e calcula a maior distância (máximo) de cada vértice de L para o vértice atual. Em seguida, ela escolhe o vértice cuja maior distância mínima é a menor entre todos os vértices do grafo. Essa parte da função possui complexidade $O(V \cdot L)$, já que, para cada vértice do grafo, é necessário calcular o máximo das distâncias para os L vértices.

Finalmente, a complexidade total do algoritmo é uma combinação da complexidade das distâncias de *Dijkstra* e do cálculo da "distância máxima mínima". Isso resulta em uma complexidade de $O(L \cdot (V + E) \log V)$, que é eficiente quando comparada a outras abordagens possíveis. O uso de *Dijkstra* com min-heap permite que o algoritmo lide com grafos grandes de forma eficaz, aproveitando o fato de que as distâncias mínimas podem ser calculadas de forma incremental, reduzindo o custo computacional em relação a abordagens mais ingênuas como a busca em largura ou busca exaustiva. Assim, o algoritmo é capaz de fornecer uma solução ótima dentro de uma complexidade aceitável para grafos de tamanho significativo.

4 Dado um grafo direcionado com pesos, um caminho C e um inteiro positivo X, projete um algoritmo que retorne o caminho mais barato C' com o mesmo começo e fim de C de forma que a distância de qualquer vértice em C' ao seu vértice mais próximo em C não ultrapasse X.

```
const int INF = numeric_limits<int>::max(); // Valor infinito para inicialização
```

```
// Função para calcular a menor distância de qualquer vértice do caminho C
// até os demais vértices, com restrição de distância máxima X
void calculateDistanceFromPath(GraphAdjList& graph,
const vector<int>& path, int X, vector<int>& minDistance) {
    int n = graph.numVertices();
    minDistance.assign(n, INF);
    set<vertex> pathSet(path.begin(), path.end()); // Facilita checar se um vértice está em C

    // Fila de prioridade para BFS com pesos
    queue<pair<vertex, int>> q; // {vértice, distância atual}

    // Inicializa as distâncias para os vértices em C
    for (vertex v : path) {
        minDistance[v] = 0;
        q.push({v, 0});
    }

    // BFS para calcular as distâncias mínimas de cada vértice de C
    while (!q.empty()) {
        auto [u, dist] = q.front();
        q.pop();

        // Percorre as arestas do vértice u
        for (auto& edge : graph.edges()) {
            vertex v1, v2;
            int weight;
            tie(v1, v2, weight) = edge; // Desempacota a tupla (v1, v2, weight)

            // Verifica se u é um dos vértices da aresta
            if (u == v1 || u == v2) {
                vertex v = (u == v1) ? v2 : v1; // Determina o outro vértice da aresta

                // Verifica a restrição de distância e atualiza se necessário
                if (dist + 1 <= X && minDistance[v] > dist + 1) {
                    minDistance[v] = dist + 1;
                    q.push({v, dist + 1});
                }
            }
        }
    }
}

// Algoritmo principal para encontrar o caminho mais barato C' respeitando as restrições
vector<int> findCheapestRestrictedPath(GraphAdjList& graph, const vector<int>& path, int X) {
    int n = graph.numVertices();
    vector<int> minDistance; // Distância mínima de um vértice de C
    calculateDistanceFromPath(graph, path, X, minDistance);

    // Algoritmo de Dijkstra modificado para encontrar C'
    priority_queue<tuple<int, vertex, vector<vertex>>,
    vector<tuple<int, vertex, vector<vertex>>>, greater<>> pq;
    vector<int> distance(n, INF);
    vector<bool> visited(n, false);
```

```
    vertex start = path.front(); // Início do caminho
    vertex end = path.back();    // Fim do caminho

    // Inicializa a fila de prioridade com o vértice inicial
    pq.push({0, start, {start}});
    distance[start] = 0;

    while (!pq.empty()) {
        auto [cost, u, currentPath] = pq.top();
        pq.pop();

        if (visited[u]) continue;
        visited[u] = true;

        // Se alcançarmos o destino, retornamos o caminho
        if (u == end) return currentPath;

        // Explora as vizinhanças de u
        for (auto& edge : graph.edges()) {
            vertex v1, v2;
            int weight;
            tie(v1, v2, weight) = edge; // Desempacota a tupla

            // Verifica se u é um dos vértices da aresta
            if (u == v1 || u == v2) {
                vertex v = (u == v1) ? v2 : v1;

                if (!visited[v] && distance[v] > cost + weight && minDistance[v] <= X) {
                    distance[v] = cost + weight;
                    vector<vertex> newPath = currentPath;
                    newPath.push_back(v);
                    pq.push({distance[v], v, newPath});
                }
            }
        }
    }
    return {}; // Retorna vazio se não encontrar caminho
}
```

O problema a ser resolvido consiste em encontrar um caminho C' em um grafo direcionado ponderado, que conecta os mesmos vértices inicial e final de um caminho C, minimiza o custo total do caminho, e respeita a restrição de que a distância de qualquer vértice em C' ao vértice mais próximo em C não ultrapasse um valor X. Para resolver este problema, propomos um algoritmo eficiente que combina dois passos principais: (1) a busca em largura (BFS), para calcular as distâncias mínimas de C para os demais vértices, e (2) uma versão adaptada do algoritmo de Dijkstra, que incorpora as restrições de distância para encontrar o caminho C' de menor custo.

O grafo é representado utilizando listas de adjacência, implementadas na classe `GraphAdjList`. Essa estrutura é eficiente para armazenar e consultar vizinhanças de vértices, o que é essencial para algoritmos de busca e cálculo de caminhos mínimos. Além disso, todas as arestas são armazenadas em uma lista separada, facilitando o acesso às informações de peso. Essa escolha de estrutura de dados permite otimizações específicas durante a execução do algoritmo, mantendo a eficiência mesmo para grafos grandes.

A função `calculateDistanceFromPath` utiliza uma busca em largura (BFS) para calcular a menor distância entre cada vértice do grafo e o conjunto de vértices C. Inicialmente, todos os vértices de C

têm distância zero (0), e os demais vértices têm distância infinita (∞). Durante a execução da BFS, cada vértice visitado tem sua distância atualizada apenas se a nova distância calculada for menor que a anterior e não ultrapassar X. Este passo é fundamental, pois cria uma restrição inicial que filtra os vértices elegíveis para C', garantindo que a condição de distância seja respeitada de maneira eficiente durante a próxima etapa.

Após calcular as distâncias mínimas de C, a função `findCheapestRestrictedPath` utiliza uma versão modificada do algoritmo de Dijkstra para encontrar o caminho C'. Este algoritmo considera não apenas o custo acumulado ao visitar os vértices, mas também verifica se a distância do vértice em questão até C respeita a restrição de X. Para cada vértice u, seus vizinhos são explorados e, se o custo acumulado ao visitá-los for menor do que o custo registrado e a restrição de X for satisfeita, o vértice é atualizado na fila de prioridade (min-heap). Este processo continua até que o vértice final do caminho C seja alcançado ou todos os caminhos válidos tenham sido explorados.

O algoritmo possui duas etapas principais. A primeira etapa, a BFS, tem complexidade $O(V + E)$, onde V é o número de vértices e E é o número de arestas. A segunda etapa, baseada no algoritmo de Dijkstra, tem complexidade $O((V + E) \log V)$, devido ao uso de uma fila de prioridade para gerenciar os vértices a serem explorados. Assim, a complexidade total do algoritmo é $O((V + E) \log V)$, o que o torna eficiente, especialmente para grafos esparsos (aqueles em que $E \approx V$).

Essa abordagem é particularmente elegante porque combina técnicas clássicas de grafos (BFS e Dijkstra) com condições adicionais de restrição, mantendo a eficiência computacional. Além disso, a separação das etapas garante modularidade e clareza no desenvolvimento, o que facilita sua extensão para problemas relacionados, como restrições mais complexas ou otimizações adicionais.

5 Dado um grafo não direcionado conexo com pesos, retorne a aresta cuja remoção causaria o maior aumento no custo da árvore geradora mínima.

```
// Classe auxiliar para a Union-Find (Disjoint-Set)
class UnionFind {
private:
    vector<int> parent, rank;

public:
    UnionFind(int size) {
        parent.resize(size);
        rank.resize(size, 0);
        for (int i = 0; i < size; ++i) {
            parent[i] = i;
        }
    }

    int find(int u) {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }

    void unionSets(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
```

```
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                rank[rootU]++;
            }
        }
    }
};

// Algoritmo de Kruskal para encontrar a MST
int kruskal(GraphAdjList& graph, vector<tuple<vertex, vertex, int>>& mstEdges) {
    vector<tuple<vertex, vertex, int>> edges = graph.edges();
    sort(edges.begin(), edges.end(), [](auto& e1, auto& e2) {
        return get<2>(e1) < get<2>(e2);
    });

    UnionFind uf(graph.numVertices());
    int mstCost = 0;

    for (auto& [v1, v2, weight] : edges) {
        if (uf.find(v1) != uf.find(v2)) {
            uf.unionSets(v1, v2);
            mstCost += weight;
            mstEdges.push_back({v1, v2, weight});
        }
    }

    return mstCost;
}

// Calcula o aumento crítico no custo ao remover uma aresta da MST
int criticalEdgeMST(GraphAdjList& graph) {
    vector<tuple<vertex, vertex, int>> mstEdges;
    int mstCost = kruskal(graph, mstEdges);

    int maxIncrease = 0;

    for (auto& edge : mstEdges) {
        GraphAdjList tempGraph(graph.numVertices());

        // Reconstroi o grafo sem a aresta removida
        for (auto& e : graph.edges()) {
            if (e != edge) {
                tempGraph.addEdge(get<0>(e), get<1>(e), get<2>(e));
            }
        }

        vector<tuple<vertex, vertex, int>> newMSTEdges;
        int newMSTCost = kruskal(tempGraph, newMSTEdges);

        maxIncrease = max(maxIncrease, newMSTCost - mstCost);
```

```
    }

    return maxIncrease;
}
```

O objetivo principal do código é resolver o problema de encontrar a aresta crítica em um grafo não direcionado e ponderado. Essa aresta é aquela cuja remoção resulta no maior aumento no custo da árvore geradora mínima (MST) do grafo. Para isso, o código utiliza técnicas clássicas de teoria dos grafos, como o algoritmo de Kruskal, em conjunto com estruturas auxiliares como listas de adjacência e o sistema de união e busca (Union-Find). A seguir, exploramos detalhadamente o funcionamento do código e sua eficiência.

Primeiramente, o código constrói uma representação do grafo utilizando uma classe chamada `GraphAdjList`, que armazena as arestas em listas de adjacência. Essa abordagem é eficiente para manipular grafos esparsos, onde o número de arestas é muito menor do que o quadrado do número de vértices. Uma árvore geradora mínima é uma subestrutura que conecta todos os vértices do grafo com o menor custo possível, e o código a encontra utilizando o algoritmo de Kruskal. Esse algoritmo seleciona as arestas em ordem crescente de peso, garantindo que nenhuma delas forme ciclos.

A lógica central do código está na função que calcula o impacto de remover cada aresta da árvore geradora mínima. Para cada aresta na MST, o grafo é temporariamente reconstruído sem essa aresta, e o custo de uma nova MST é calculado. A diferença entre o custo da MST original e a nova MST determina o impacto da remoção dessa aresta. Esse processo é repetido para todas as arestas da MST, e o maior impacto encontrado é o resultado final. Assim, o código identifica a aresta cuja remoção mais prejudica a conectividade mínima do grafo.

Em termos de complexidade, o algoritmo de Kruskal tem custo $O(E \log E)$, onde E é o número de arestas, devido à ordenação inicial das arestas e às operações do Union-Find, que possuem custo amortizado quase constante. Para cada aresta na MST, o código reconstrói o grafo temporariamente e calcula uma nova MST, o que é realizado $O(E \log E)$ vezes. Portanto, a complexidade final do código é $O(E^2 \log E)$, o que é aceitável para grafos moderadamente grandes, mas pode se tornar um gargalo para grafos densos.

Esse código faz o que faz porque a remoção de uma aresta da MST pode aumentar o custo de manter todos os vértices conectados, obrigando o grafo a adotar uma rota mais cara para compensar. O algoritmo considera todas as possibilidades para garantir que a resposta seja correta e completa. Embora não seja o método mais eficiente possível para o problema, o código é direto, utiliza conceitos bem estabelecidos e é relativamente fácil de entender e modificar, o que o torna uma escolha prática em muitos cenários acadêmicos e profissionais.

LINK DO REPOSITÓRIO COM AS IMPLEMENTAÇÕES