Relatório

O código apresentado é uma implementação do Algoritmo de Dijkstra para encontrar os caminhos mais curtos a partir de um nó inicial em um grafo ponderado. Vou descrever o funcionamento do código e fornecer um cenário de prova de conceito (POC) que ilustra a adição e remoção de nós durante o processo de descoberta e roteamento.

Funcionamento do Código

1. Inicialização:

- O algoritmo começa criando uma lista de todos os nós do grafo.

- Inicializa um objeto distances para armazenar a distância mínima conhecida de cada nó ao nó inicial.

- Cria um conjunto visited para acompanhar os nós que já foram processados.

- Utiliza uma fila de prioridade (priorityQueue) para armazenar os nós a serem explorados, ordenada pela distância mínima.

2. Configuração Inicial:

- Define a distância do nó inicial como 0 e as distâncias de todos os outros nós como infinito.

- Insere todos os nós na fila de prioridade com suas distâncias iniciais.

3. Processamento dos Nós:

- O nó com a menor distância é retirado da fila de prioridade.

- Se o nó já foi visitado, ele é ignorado.

- so contrário, o nó é marcado como visitado e suas distâncias para os vizinhos são atualizadas.

- Se uma nova distância menor for encontrada para um vizinho, essa nova distância é registrada e o vizinho é adicionado na fila de prioridade com a nova distância.

4. Finalização:

- O algoritmo continua até que todos os nós tenham sido processados e retorna as distâncias mínimas conhecidas de todos os nós ao nó inicial.

Cenário de Prova de Conceito (POC)

Configuração Inicial

Vamos considerar um grafo com os seguintes nós e arestas:

```
const graph = {
    A: { B: 1, C: 4 },
    B: { A: 1, C: 2, D: 5 },
    C: { A: 4, B: 2, D: 1 },
    D: { B: 5, C: 1 }
};
```

Teste 1: Adição de Nós

1. Início da Simulação:

- Começamos com o grafo fornecido e aplicamos o Algoritmo de Dijkstra a partir do nó 'A'.

2. Adição de um Novo Nó (Nó E):

- Adicionamos o nó 'E' ao grafo, conectando-o a 'C' e 'D' com pesos específicos:

```
graph.E = { C: 3, D: 6 };
graph.C.E = 3;
graph.D.E = 6;
```

3. Atualização e Re-execução:

- Reexecutamos o Algoritmo de Dijkstra a partir do nó 'A'.

- Observe como as distâncias para o novo nó 'E' e os caminhos que passam por ele são recalculados.

4. Observação:

- Verifique como as distâncias mínimas para todos os nós foram ajustadas, especialmente para o nó 'E' e quaisquer outros nós cujas distâncias possam ter mudado devido à adição do nó 'E'.

Teste 2: Remoção de Nós

1. Remoção de um Nó (Nó B):

- Removemos o nó 'B' do grafo e suas conexões:

```
delete graph.B;

for (const node in graph) {

    delete graph[node][B];

}
```

2. Atualização e Re-execução:

- Reexecutamos o Algoritmo de Dijkstra a partir do nó 'A'.

- Observe como as distâncias para os nós restantes são recalculadas considerando a ausência do nó 'B'.

3. Observação:

- Verifique como a remoção do nó 'B' afetou as distâncias e as rotas entre os nós restantes.

Conclusão

Este código é uma implementação eficiente do Algoritmo de Dijkstra para encontrar os caminhos mais curtos em um grafo ponderado. A prova de conceito demonstra a capacidade do algoritmo de lidar com mudanças na topologia do grafo, como a adição e remoção de nós, e como as distâncias mínimas são recalculadas para refletir essas mudanças.