Trabalho Prático 1

Igor Eduardo Martins Braga - 2022425671

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

igoreduardobraga@ufmg.br

1 - Introdução

O problema proposto foi realizar um programa que encontrasse o menor caminho entre a cidade de origem para a cidade de destino, em que esse caminho tivesse uma quantidade de estradas par e essas estradas tivessem comprimento par.

2. Método

2.1 Implementação

O programa foi desenvolvido a partir da linguagem C++. Para a compilação utilizou-se o compilador G++ da GNU Compiler Collection.

2.2 Modelagem

Para representar o grafo no programa, foi utilizado uma lista de adjacência, feita a partir da biblioteca de vector da seguinte forma: <vector<vector<par_vertice_distancia>>, no qual, par_vertice_distancia é uma estrutura independente que representa um par de dois inteiros (dist e vertice). “vertice” representa o primeiro elemento do par, que é o vértice adjacente e “dist” representa o segundo elemento do par que é o peso da aresta (distância). Foi optado por utilizar a lista de adjacência para representar o grafo devido à sua eficiência em termos de memória. Isso ocorre porque a lista de adjacência é uma estrutura mais compacta em comparação com a matriz de adjacência. Em vez de armazenar todas as informações em uma matriz, cada vértice é representado por uma lista ligada que contém apenas os vértices adjacentes a ele. Para grafos esparsos, essa abordagem ocupa menos espaço do que a matriz de adjacência, tornando a lista de adjacência uma escolha mais eficiente em termos de memória. Além disso, a lista de adjacência é muito flexível para representar grafos ponderados (que possuem pesos), já que, com poucas alterações podemos representar os vértices adjacentes juntamente com os pesos. Isso foi feito por meio da struct par_vertice_distancia, comentada anteriormente.

O algoritmo utilizado para encontrar o menor caminho do grafo foi o Dijkstra. A preferência por utilizá-lo foi porque o grafo de entrada não contém arestas negativas. Além disso, o algoritmo de Dijkstra é eficiente para grafos densos ou esparsos, sendo totalmente aplicável a problemas para encontrar menor caminho em um grafo ponderado não negativo, ou seja, encontrar a menor rota entre um conjunto de locais diferentes. Para construir esse algoritmo, criou-se uma função chamada dijkstra, onde possui duas variáveis importantes: vetor “distancia”, que armazena as distâncias mínimas até cada vértice e a fila de prioridade “fila_prioridade”, que armazena os vértices em ordem crescente de distância mínima. Em seguida, começa o loop principal do algoritmo, que roda enquanto a fila de prioridade não estiver vazia. Dentro desse loop, é selecionado o vértice com a menor distância até agora, usando a função “top” da fila de prioridade. Depois disso, são percorridos os vizinhos desse vértice, atualizando as distâncias mínimas se encontrarmos um caminho mais curto. Por fim, essa função verifica se há um caminho mínimo ou não, imprimindo a distância mínima caso tenha e “-1” caso não tenha. Para melhorar a eficiência do código, foi criado uma condição dentro do loop while de que se a distância atual já foi atualizada, então não precisamos processar esse vizinho novamente, ou seja, essa parte do código ajuda a evitar que o algoritmo processe vizinhos desnecessários.

Para impor a restrição de que as estradas percorridas tenham somente comprimento par, armazenou-se apenas as arestas de comprimento par ao ler o grafo, ignorando as ímpares. Se houver alguma aresta de tamanho ímpar no grafo, ela será eliminada, para evitar que o algoritmo de Dijkstra a percorra. Outra forma de fazer essa restrição era modificar diretamente o algoritmo de Dijkstra, de modo que ele só considerasse as arestas de comprimento par ao percorrer o grafo. No entanto, essa abordagem é menos eficiente, pois o algoritmo modificado de Dijkstra teria que verificar cada vez que percorresse uma aresta, se ela é par ou não, o que aumentaria consideravelmente a complexidade do código. Assim, verifica-se um ganho de performance do código com essa pequena mudança de implementação, já que **a primeira maneira foi utilizada para construir essa restrição**.

Para impor a restrição de que o caminho mínimo da cidade de origem até a cidade de destino tivesse uma quantidade par de estradas, foi necessário alterar a ideia do grafo original G. Para isso, o grafo original G foi modificado e recebeu o

nome de G1, visando facilitar a compreensão do algoritmo. Para cada vértice X em G, foram adicionados dois novos vértices em G1, sendo um com um índice par (2X) e outro com um índice ímpar (2X+1). Por exemplo, se G tiver um vértice com índice 3, G1 terá dois vértices adicionais: um representando o vértice par com índice 6 e outro representando o vértice ímpar com índice 7. Assim, percebemos que também vamos ter que adicionar duas arestas em G1 para cada aresta de G. Essas duas arestas devem ligar vértices de paridade oposta, ou seja, uma aresta deve ligar um vértice par a um vértice ímpar, e a outra deve ligar um vértice ímpar a um vértice par. Essas duas arestas são adicionadas em ambos os sentidos para garantir que a aresta tenha ida e volta e garantir que o grafo seja não direcionado. Para distinguir os vértices adicionados, é colocado um sufixo "_par" ou "_impar" ao nome do vértice original, dependendo se ele foi transformado em um vértice par ou ímpar, respectivamente. Outra maneira de realizar essa restrição era armazenar todos os vértices normalmente no grafo G e, após isso, percorrê-lo, modificando o índice dos seus vértices para par e ímpar e adicionando-os no grafo modificado G1. Isso aumentaria a complexidade de tempo do código, pois seria necessário percorrer todo o grafo original G, converter os vértices para ímpar e par e armazená-los no grafo G1, o que gastaria mais memória também, pois seria necessário armazenar os grafos G e G1 separadamente. **Como utilizou-se a primeira opção de implementação**, onde convertemos o grafo G em G1 diretamente na função adiconaAresta(), teve-se um ganho considerável de desempenho.

A partir disso, basta percorrer o grafo G1 com o dijkstra do vértice de origem transformado em índice par para o vértice de destino também transformado em índice par. Assim, fica garantido que o caminho percorrido pelo dijkstra tenha, obrigatoriamente, uma quantidade par de estradas (arestas).

3. Análise de complexidade

Considere **E** como o número de arestas e **V** como o número de vértices

3.1 Tempo

A complexidade de tempo total foi construída a partir das funções do código e é:

$O(E \log V) + O(1) + O(E) = O(E \log V)$

3.1.1 Função dijkstra()

A função dijkstra implementa o algoritmo de Dijkstra para encontrar o caminho mais curto entre o vértice de origem e o vértice de destino. Apesar de o grafo modificado a ser percorrido (G1) não tenha a mesma estrutura do grafo original, a função dijkstra ainda tem a mesma complexidade de tempo que se fosse utilizado o grafo original já que em ambos precisam percorrer todas as arestas e executar operações de atualização de distância em cada iteração. Logo, a complexidade dessa função é O(E log V).

3.1.2 Função adicionaAresta()

Essa função tem como papel adicionar as arestas no grafo nos dois sentidos (ida e volta) e tem complexidade O(1), já que utiliza somente a função push_back da biblioteca vector para adicionar as arestas e pesos no grafo.

3.1.3 Função main()

Na função main há um loop que percorre de 0 à quantidade de estradas do grafo. Logo, sua complexidade é O(E).

3.2 Espaço

A análise de complexidade de espaço para esse código envolve a alocação de memória para as variáveis utilizadas. O espaço necessário para alocar o grafo, representado pela lista de adjacência é: O(E+V)

Além disso, na função dijkstra é criado um vetor com 2*V de tamanho, tendo portanto, uma complexidade de O(V). A fila de prioridade também criada nessa função, armazena no máximo V elementos, então sua complexidade também é de O(V).

Logo, a complexidade de espaço total é:

O(E+V) + 2*O(V) = O(E+V)

4. Conclusão

Logo, o trabalho permitiu a recordação de conceitos fundamentais no desenvolvimento de programas na linguagem C/C++, como o uso de estruturas de dados, modularização do código a partir de classes e métodos, entre outros. Além disso, desenvolveu-se o conhecimento acerca do algoritmo de dijkstra e de como implementar e fazer alterações em um grafo de acordo com determinadas restrições.

Além disso, percebe-se a melhora na eficiência do código por meio de pequenas modificações utilizadas durante o trabalho, como mostrado na seção de modelagem.

5. Bibliografia

LAPAUGH, A. S.; PAPADIMITRIOU, C. H. The even-path problem for graphs and digraphs. Computer Science, 1984.

PAULO FEOFILOFF. Algoritmos em grafos: algoritmo de Dijkstra. São Paulo: IME-USP, 2009. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dijkstra.html. Acesso em: 4 maio 2023.