Trabalho Prático 1
Aritmofobia

Vinicius Trindade Dias Abel
Matrícula: 2020007112

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

viniciustda@ufmg.br

1. Introdução

O problema proposto foi desenvolver um algoritmo que determine o menor caminho entre duas cidades, dado um grafo que representa a rede de cidades e estradas que interconectam a região. No entanto, o algoritmo possui duas restrições:

1. A estrada que conecta duas cidades deve ter comprimento par para ser utilizada;
2. O caminho traçado pelo algoritmo deve passar por um número par de estradas.

2. Método

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de um grafo, implementado por um vetor de adjacência (escolhido pela facilidade de trabalhar com vetor), onde os vértices representam cidades e as arestas representam estradas.

Para a resolução do problema, foi utilizado o algoritmo Dijkstra (escolhido por ser um algoritmo que supre exatamente a necessidade de encontrar um caminho mínimo). Por sua vez, o algoritmo que busca o caminho mínimo utiliza um heap, implementado por uma fila de prioridade (escolhido por apresentar uma boa complexidade), e um vetor que armazena as distâncias mínimas entre cada vértice e o vértice de origem.

2.2. Solução para as restrições

a) A estrada que conecta duas cidades deve ter comprimento par para ser utilizada

Para garantir que o caminho traçado não passe por nenhuma estrada de comprimento ímpar, no momento da criação do grafo, antes de adicionar uma aresta, é verificado o comprimento dela. Caso o comprimento seja par, a aresta é adicionada ao grafo, caso contrário, ela não é adicionada.

b) O caminho traçado pelo algoritmo deve passar por um número par de estradas

O algoritmo constrói um grafo adaptado com base no grafo original. Para isso, são criados dois vértices (par e ímpar) no grafo adaptado para cada vértice do grafo original. As arestas são criadas de forma que intercale entre os tipos de vértice, isto é, o vertice_a par é ligado ao vertice_b ímpar e o vertice_a ímpar é ligado ao vertice_b par, sendo que ambas as arestas possuem mesmo peso.

Como os caminhos de tamanho par ficam separados por essa implementação, ao executar o algoritmo Dijkstra encontramos o menor caminho que atende às restrições.

2.3. Funções

AdicionaAresta

A função recebe como parâmetro as duas cidades que serão interligadas e o tamanho da estrada. Se a estrada tiver tamanho par, é criada a conexão entre as cidades nos dois sentidos, de cidade A para cidade B e cidade B para cidade A.

Entretanto, para atender a segunda restrição, criei duas versões de cada vértice (cidade), sendo um vértice par e um ímpar. O vetor de vértices foi dividido pela metade, de forma que na primeira metade fica uma versão e na segunda a outra versão de cada vértice.

`Ex.: vertices[3] = [0][vertice_a par][vertice_b par][vertice_c par][vertice_a impar][vertice_b impar][vertice_c impar]`

Para a criação de arestas (estradas), a ligação foi feita de forma intercalada, sendo vértice par ligado com vértice ímpar. Os pesos para as ligações semelhantes são iguais.

Desta forma, são criadas quatro arestas no grafo adaptado, para cada aresta do grafo original. De forma que indique que o grafo é não direcional e intercale de par para ímpar e ímpar para par:

vertice_a par – vertice_b ímpar
vertice_b ímpar – vertice_a par
vertice_b par – vertice_a ímpar
vertice_a ímpar – vertice_b par

CalculaMenorDistancia

A função é uma implementação do algoritmo Dijkstra, ela utiliza uma fila de prioridade que implementa um heap e um vetor para armazenar as distancias mínimas entre as cidades e a cidade de origem.

A função funciona da seguinte forma:

```
Enquanto a fila de prioridade/heap não estiver vazio
        vértice v = vértice do topo do heap
        exclui vértice do topo do heap
        para cada aresta que liga o vértice v a um vértice w
            vértice w = vértice ligado ao vértice v
            tamanho = tamanho/peso da aresta que liga vértices v e w
            se a distância do vértice w ao vértice inicial for maior que
(a distância do vértice v ao vértice inicial + o tamanho da aresta v-w)
                atualiza a distância do vértice w
                insere a distância do vértice w ao vértice inicial na
fila de prioridade
```

Por fim, caso exista um caminho mínimo, a distância é retornada, caso não exista, é retornado -1.

Main

A função main recebe o número de cidades e o número de estradas que compõem a rede de cidades. Em seguida, recebe as estradas da rede de cidades por meio de um loop que chama a função AdicionaAresta, criando um grafo já adaptado para a resolução do problema. Após a criação do grafo, é chamada a função CalculaMenorDistancia para calcular a menor distância entre a última cidade e a cidade de origem.

2.4. Configuração para teste

– Sistema Operacional do computador: Linux Ubuntu;
– Linguagem de programação implementada: C++;
– Compilador utilizado: G++ da GNU Compiler Collection;
– Dados do seu processador: i7;
– Quantidade de memória RAM: 16GB.

3. Análise de Complexidade

AdicionaAresta

A função tem complexidade O(1). Ou ela não faz nada (tamanho da estrada é ímpar), ou adiciona quatro arestas no grafo (tamanho da estrada é par).

CalculaMenorDistancia

A função tem complexidade $O(n^2)$. A complexidade do laço é a que prevalece, e o laço de repetição “while” possui outro laço de repetição “for” interno, levando a complexidade $O(n^2)$.

Geral

O TP tem complexidade $O(n^2)$. A função main chama a função AdicionaAresta n vezes, então o laço de repetição da função main tem complexidade O(n), a inicialização do grafo também tem complexidade O(n), uma vez que o vetor distancias é percorrido uma vez. Como a função CalculaMenorDistancia é chamada uma vez, sua complexidade de $O(n^2)$ sobressai, logo a complexidade do TP é $O(n^2)$.

4. Conclusões

Este trabalho lidou com cálculo de caminho mínimo em um grafo com peso e não direcionado, na qual a abordagem utilizada para sua resolução foi o algoritmo Dijkstra.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a grafos e algoritmos de grafos.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a melhor forma para implementar o grafo e como atender a segunda restrição do problema.

Inicialmente, tentei modificar o algoritmo Dijkstra para atender a segunda restrição. Para isso, o algoritmo calculava a distância do último vértice, verificava o número de arestas que foram percorridas até o momento, e só realizava a atualização da distância se o número de arestas fosse par. No entanto, ao realizar os testes, percebi que não funcionava. Somente após diversas tentativas de alterar o Dijkstra pensei em alterar o grafo antes de buscar o caminho mínimo.

Primeiro criei o grafo, já atendendo a primeira restrição, e tentei altera-lo para cumprir com a segunda regra após toda a construção. A complexidade ficou alta, então tentei criar o grafo já atendendo a segunda restrição, como foi feito para a primeira restrição.

5. Bibliografia

- Standard C++ Library reference. Disponível em: https://cplusplus.com/reference/

- Algoritmo de caminho de custo mínimo de Dijkstra - uma introdução detalhada e visual.
Disponível em: https://www.freecodecamp.org/portuguese/news/algoritmo-de-caminho-de-custo-minimo-de-dijkstra-uma-introducao-detalhada-e-visual/

- Grafos e algoritmos. Disponível em: https://medium.com/programadores-ajudando-programadores/os-grafos-e-os-algoritmos-697c1fd4a416

6. Instruções para compilação e execução

- Acesse o diretório tp1
- Utilizando o terminal, compile o TP 01 utilizando o comando: make
- Com esse comando, será gerado o arquivo tp01.exe e os arquivos *.o
- Utilizando o terminal, teste o tp01.exe utilizando o comando: make eval

Obs.: o comando “make eval” apresentou falhas ao ser utilizado em ambiente Windows, foi necessário realizar os testes separadamente. Entretanto, no ambiente Linux não houve problema.