Trabalho Prático 1

Gabriela Tavares Barreto

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

gbarreto@ufmg.br

1 Introdução

O objetivo desse trabalho prático consistia em resolver o seguinte problema: dado um grafo de entrada G, e sendo este grafo não direcionado e ponderado, obter o menor caminho possível do vértice de origem 1 ao vértice de maior índice do grafo(vértice de destino N), sob duas restrições. Sendo elas:

- é vetado o uso de arestas de peso ímpar;
- o número total de arestas percorridas no caminho entre os vértices deve ser par.

2 Método

O programa foi desenvolvido na linguagem C++, e compilado pelo G++ da GNU Compiler Collection, usando o WSL. O computador utilizado tem como sistema operacional o Windows 11, com 12 GB de RAM.

2.1 Solução

Para poder resolver este problema, foi feito uso do algoritmo de Dijkstra. Ao receber um grafo G de entrada, é construído um grafo modificado G baseado no grafo original. A construção do grafo G dá-se da seguinte forma:

- para cada vértice v presente no grafo original, são criados dois vértices no grafo modificado, chamados v_{par} e v_{impar};
- para cada aresta ligando os vértice u e v do grafo original, são criadas duas arestas no grafo modificado. Uma ligando v_{par} à u_{impar} e outra ligando $v_{ímpar}$ à v_{par}.

Sabe-se que a partir do vértice de origem 1, para atingir o vértice de destino N, o menor caminho pode ser par ou ímpar(o caminho "ser par"ou "ser ímpar"diz respeito ao número de arestas usadas). E para chegar ao vértice de destino N usando um número par de arestas, é necessário chegar a algum dos vértices adjancentes a N usando um número ímpar de arestas. Em todo o caminho a partir de 1, para cada vértice passado, alterna-se entre um número par de arestas usadas e um número ímpar. Na prática, rodar o algoritmo de Djkstra nessa representação permite calcular ambas maneiras de se chegar aos vértices a partir da origem, seja de forma par ou ímpar. A representação alternada dos vértices(um vértice com caminho par só é adjacente a um vértice com caminho ímpar) garante que a restrição de paridade seja obedecida no caminho entre 1_{par} e N_{par}.

2.2 Implementação

Nessa seção será apresentada a parte técnica da solução.

2.2.1 Estruturas

Foram implementadas duas estruturas auxiliares, a **struct Vertice** e a **struct ComparadorDistancia.**

A struct Vertice serve como auxiliar a implementação algortimo de Dijkstra, e é usada no heap mínimo necessário para esse algoritmo. A estrutura armazena dois inteiros, **vertice** e **distancia**, como nomenclauras autoexplicativas.

Já a struct ComparadorDistancia sobrecarrega o operador bool(), tal que ela compara dois objetos do tipo Vertice e retorna true se o atributo distância do primeiro objeto é considerado maior do que o do segundo objeto. Essa struct é usada para implementação de um heap mínimo usando a classe priority queue da biblioteca padrão de c++.

2.3 Classes

Foram implementadas duas classes para a representação do grafo, sendo elas as classes **VerticeAdjacente** e **Grafo**.

2.3.1 VerticeAdjacente

Essa classe serve como auxiliar a classe Grafo, e armazena dois inteiros, **vertice** e **peso_aresta**. Ela tem dois construtores, um construtor padrão default, que não recebe parâmetros, e um construtor que recebe dois inteiros para inicializar os atributos vertice e peso_aresta.

2.3.2 Classe Grafo

Essa classe armazena um vector de vectors de VerticeAdjacentes **g**, de forma a representar o grafo como uma lista de adjacências. Além disso, ela contém também os inteiros **n**, que guarda o valor de vértices usados no grafo modificado G, e **N**, que guarda o valor dos vértices do grafo original. Seu uso ficára explícito mais adiante. As funções dessa classe estão descritas abaixo:

- **int par(int x):** retorna x transformado para um valor par, no caso $(x * 6) + 2$;
- **int impar(int x):** retorna x transformado para um valor ímpar, no caso $(x * 6) + 1$;
- **Grafo(int a):** construtor que inicializa o grafo com $a * 10 + 3$ linhas, e guarda a em N;
- **add_vertice(int v, int u, int w):** inicializa as arestas entre os vértices (v_{par}, $u_{ímpar}$), e ($v_{ímpar}$,u_{par}), com o peso w passado como parâmetro, e as adiciona à lista de adjacências;
- **int peso_aresta(int u, int v):** retorna o peso da aresta entre u e v.
- **dijkstra():** implementa o algoritmo de Dijkstra. Ao final da função, é impresso o valor da distância até o vértice N_{par}.

2.4 Programa principal

O main.cpp tem a função **void inicializar_grafo(Grafo& g)**, que faz a leitura do terminal para adicionar os vértices e arestas corretamente no grafo g.

A função main é bem concisa, e faz o seguinte: primeiro, ela lê o valor N do terminal para poder inicalizar o grafo g com o tamanho correto. Depois ela chama a função inicializar_grafo(), e é feita a adição de vértices adjacentes na lista de adjacências. Por fim, é chamada a função dijkstra, de forma a calcular as distâncias e o programa é encerrado.

3 Análise de Complexidade

Nesta seção, será apresentada a análise de complexidade das funções apresentadas na seção 2. As funções add_vertice(), par(), impar() e peso_aresta() da classe Grafo são todas $\Theta(1)$, pois sempre executam a mesma quantidade de operações. Já a função construtora Grafo(int a) tem complexidade de tempo $\Theta(n)$, pois o tempo que ela gasta para ser executada é o tempo para modificar o número de linhas da lista de adjacências para 6n+3 linhas.

No programa principal, a função inicializar_grafo() tem complexidade $\Theta(V + E)$, já que para cada um dos vértices do grafo original são adicionados dois vértices no grafo modificado, e para cada aresta são adicionadas duas arestas ao grafo.

Agora, será apresentada a complexidade da função dijkstra(). A complexidade do algoritmo de Dijkstra é $O((E+V)logV)$. Essa complexidade se deve à utilização do heap mínimo, que é utilizado para armazenar os vértices que ainda não foram visitados. A inserção e remoção de elementos em um heap mínimo tem complexidade $O(logV)$, e isso é feito V vezes. Além disso, todas as arestas do grafo são percorridas uma vez, o que contribui com um fator de E.

Por fim, dado um grafo de entrada g com V vértices e E arestas, a complexidade de tempo total do programa é $O((E + V)logV)$ e de espaço $\Theta(E + V)$.

4 Conclusão

Neste trabalho, enfrentei dificuldades para conseguir resolver o problema. Primeiro tentei, de diferentes maneiras, alterar o algoritmo de dijkstra, para que ele pudesse calcular a distância par de 1 até N. Porém, não tive sucesso. Somente após essas tentativas, mudei a estratégia de alterar o dijkstra para mantê-lo em sua forma original e alterar a entrada.

Apesar de ter sido um processo com diversos tropeços, pude aprender bastante com esse trabalho, pelos meus acertos e pelos meus erros também. Consegui reforçar o que havia sido aprendido em sala de aula de forma teórica sobre grafos e o algoritmo, e ampliei meus conhecimentos ainda mais ao implementar a essa solução.

5 Bibliografia

- Introduction to Algorithms, 3ª edição
- http://wiki.icmc.usp.br

6 Instruções de compilação e execução

Para compilar o programa, deve-se seguir os seguintes passos:

- Acessar o diretório TP01-Template-CPP;
- Utilizando um terminal, digitar make, de forma a gerar o arquivo executável **tp0**;
- Para a execução, chamar o TP0 seguido de $<$ e passando um arquivo de entrada do tipo .txt com os dados do grafo como parâmetro.