Trabalho Prático 1 da disciplina de Algoritmos 1

Alex Eduardo Alves dos Santos
Matrícula: 2021032145

Departamento de Ciência da Computação -
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
alexeduardoaeas@ufmg.br

1. Introdução

Este código é responsável por calcular o caminho mínimo entre o vértice 1 e o vértice n em um grafo, sem passar por arestas de peso ímpar e utilizando um número par de arestas. O grafo é representado por uma lista de adjacências e o algoritmo utilizado para encontrar o caminho mínimo é o algoritmo de Dijkstra.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Classe Graph

A classe Graph possui um construtor que redimensiona o tamanho do grafo e atribui distâncias infinitas a todos os vértices.

A função adicionarAresta é responsável por adicionar as arestas no grafo de forma que a construção do grafo obtenha um caminho mínimo com quantidade par de arestas ao rodar Dijkstra.

A função dijkstra utiliza o algoritmo de Dijkstra para descobrir o menor caminho entre o vértice 1 e o vértice n, passado como argumento da função. É utilizado uma fila de prioridade e um vetor de distâncias mínimas de cada vértice.

A função calculaDistancia é responsável por calcular e imprimir a distância do vértice 1 ao vértice n, recusando arestas com peso ímpar e utilizando o menor caminho com quantidade par de arestas. O algoritmo utilizado é o de Dijkstra com modificações na construção do grafo.

Por fim, a função populaGrafo é utilizada para popular o grafo adicionando a quantidade de arestas que é passada como argumento da função.

2.2. Main

O programa principal lê a quantidade de vértices e arestas do grafo, popula o grafo e calcula o caminho mínimo entre os vértices 1 e n, utilizando a função calculaDistancia. O resultado é impresso na tela.

2.3. Estrutura de Dados/Lista de Adjacência

A lista de adjacência é a estrutura de dados utilizada para representar o grafo. A lista de adjacência consiste em uma lista de listas, onde cada elemento da lista principal representa um vértice do grafo, e a lista associada a esse vértice contém os vértices adjacentes a ele. Cada elemento da lista de adjacência armazena uma tupla, que geralmente contém o número do vértice adjacente e o peso da aresta que liga os dois vértices

2.2. Construção do Grafo

O grafo é reconstruído com o dobro de vértices, particionando cada vértice em 2 um com cada índice, par e ímpar, respectivamente. Essa abordagem é uma técnica utilizada para encontrar o caminho mais curto em um grafo, permitindo apenas a passagem por arestas com peso par. A ideia é transformar o grafo original em um novo grafo G', onde cada nó V é substituído por dois novos nós, V_par e V_ímpar. O nó V_par representa o nó original V com um índice par e o nó V_ímpar representa o mesmo nó com um índice ímpar.

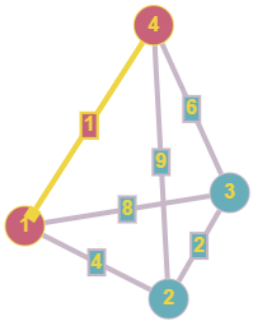
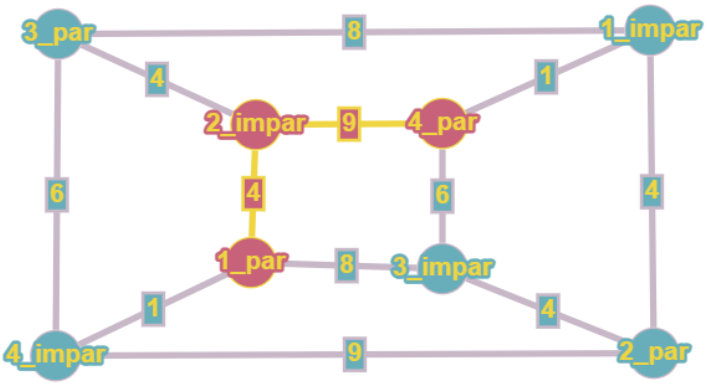


Figura 1: Grafo G

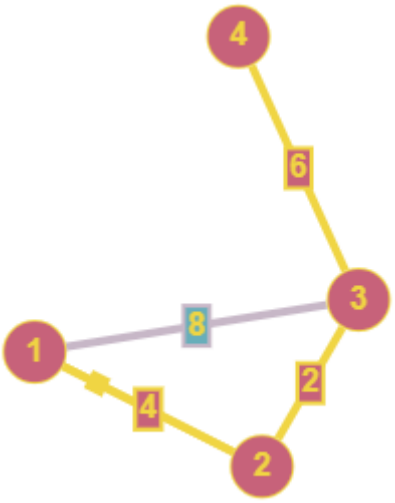
Por exemplo, suponha que o grafo original G tenha os nós 1, 2, 3 e 4 com as arestas (1, 2) com peso 4, (2, 3) com peso 2, (2, 4) com peso 9, (1, 3) com peso 8, (1, 4) com peso 1 e (3, 4) com peso 6.

Figura 2: Grafo G’



Repare que o dijkstra nesse grafo não satisfaz a condição de passar por uma quantidade par de arestas, portanto devemos reconstruir o Grafo com o dobro de vértices com cada vértice com índices V_par e V_ímpar.

O grafo G' será formado pelos nós 1_par, 1_ímpar, 2_par, 2_ímpar, 3_par, 3_ímpar, 4_par, 4_impar e pelas arestas (1_ímpar, 2_par) com peso 4, (1_par, 2_ímpar) com peso 4, (1_ímpar, 3_par) com peso 8, (1_par, 3_ímpar) com peso 8, (1_par, 4_ímpar) com peso 1, (1_ímpar, 4_par) com peso 1, (2_ímpar, 3_par) com peso 2, (2_par, 3_ímpar) com peso 2, (2_ímpar, 4_par) com peso 9, (2_par, 4_ímpar) com peso 9, (3_ímpar, 4_par) com peso 6, e (3_par, 4_ímpar) com peso 6. O grafo G' ficará como na figura 2.



No entanto, percebe que a solução ainda não satisfaz a condição de passar apenas por arestas de peso par, logo o grafo deve ser alterado de forma que não contenha as arestas com peso ímpar, conforme a figura 3.

Figura 3: Grafo G sem arestas ímpares.

Novamente o grafo sem arestas ímpares deve ser alterado como no Grafo G, para conter os vértices com índices par e ímpar, conforme a figura 4.

Agora, para encontrar o caminho mais curto entre o nó origem_par e o nó destino_par, é utilizado o algoritmo de Dijkstra no grafo G' sem arestas impares. O nó origem_par é o nó original de origem com um índice par e o nó destino_par é o nó original de destino com um índice par. O exemplo da figura 4 mostra o resultado do grafo final satisfazendo todas as condições.

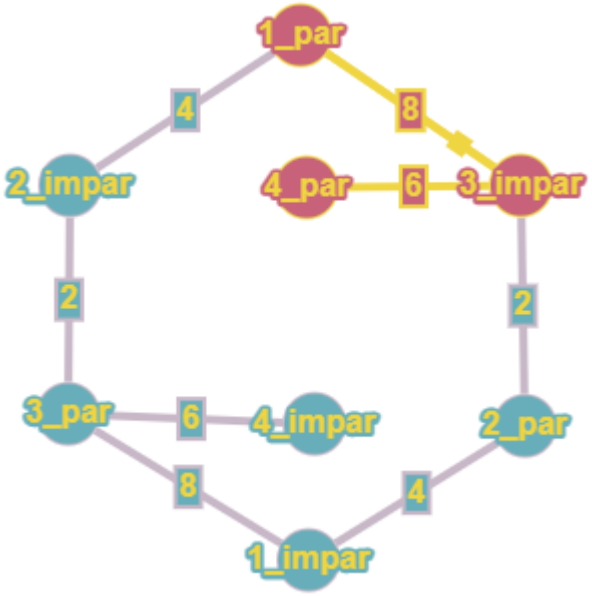


Figura 4: Grafo G' sem arestas ímpares.

3. Análise de Complexidade

A complexidade assintótica do programa é O((m+n)log n), onde n é o número de vértices e m é o número de arestas no grafo.

A função Graph::Graph(int n) tem complexidade O(n) devido ao redimensionamento do grafo e atribuição de distâncias infinitas a todos os vértices.

A função Graph::adicionarAresta(int a, int b, int custo) tem complexidade O(1), pois as operações de inserção em vetor e verificação de paridade do custo são constantes.

A função Graph::dijkstra(int destino) tem complexidade O((m+n)log n) devido ao uso de uma fila de prioridade em conjunto com o algoritmo Dijkstra. A inserção de um elemento na fila de prioridade tem complexidade log n, e a remoção do elemento mínimo tem complexidade log n. Como cada vértice pode ser inserido e removido da fila de prioridade apenas uma vez, a complexidade total dessa função é O((m+n)log n).

A função Graph::calculaDistancia(int destino) chama a função Graph::dijkstra(int destino), portanto, a complexidade assintótica é a mesma: O((m+n)log n).

Por fim, a função Graph::populaGrafo(int quantidadeArestas) tem complexidade O(m), pois ela simplesmente faz um loop de m iterações para adicionar as arestas no grafo.

Assim, a complexidade assintótica total do programa é dada por O(n + m + (m+n)log n) = O((m+n)log n), já que o termo O(m) é dominado pelo termo **O((m+n)log n).**

4. Conclusão

A implementação desse código foi uma experiência enriquecedora, pois permitiu a aplicação prática de conceitos fundamentais de grafos e algoritmos. A construção do grafo foi um passo crucial, pois determinou a representação do problema em termos de nós e arestas. A escolha pelo algoritmo de Dijkstra foi determinante para a solução do problema, uma vez que ele garante a escolha do caminho mais curto em grafos ponderados, como é o caso desse problema.

Durante a implementação, surgiram diversas dificuldades, tais como a manipulação de estruturas de dados e a escolha da melhor forma de representar o grafo em memória. Além disso, foi necessário tomar cuidado com a complexidade espacial e temporal do algoritmo, para garantir que o programa pudesse ser executado em tempo hábil.

Um dos principais aprendizados foi a importância de se planejar bem a implementação do algoritmo, desde a construção do grafo até a escolha da melhor estrutura de dados para armazená-lo. Além disso, a importância de se avaliar a complexidade espacial e temporal do algoritmo foi evidenciada, para garantir que o programa pudesse lidar com grandes quantidades de dados.

Em resumo, a implementação deste código permitiu a aplicação prática de diversos conceitos fundamentais de grafos e algoritmos, além de ter proporcionado valiosos aprendizados sobre planejamento, estruturação e avaliação de algoritmos. O conhecimento adquirido certamente será de grande utilidade em futuros projetos que envolvam a solução de problemas por meio de algoritmos.

Referências Bibliográficas

ZIVIANI, Nivio. Projetos de Algoritmos com Implementações em Java e C + +. Editora Cengage, 2006.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: Teoria e Prática. 6ª edição. Editora Elsevier, 2020.

Instruções para compilação e execução

1. Abra um terminal e navegue até o diretório onde o arquivo Makefile está localizado.

2. Execute o comando "make" na linha de comando para compilar o programa. Certifique-se de que não houve erros durante a compilação.

3. Verifique se o arquivo executável "tp01" foi criado com sucesso.

4. Para executar o programa, utilize o comando "./tp01" na linha de comando. Certifique-se de que o arquivo de entrada é passado como entrada padrão, através da linha de comando (por exemplo, $ ./tp01 < casoTeste01.txt) e a saída deve ser gerada na saída padrão (não gerar saída em arquivo).

5. Confira a solução no terminal.