**SOLUÇÃO E ANÁLISE DE PROBLEMAS COMPUTACIONAIS EM GRAFOS**

*Bruno Duarte de Paula Assis*

*Lincoln Antunes Nogueira Coutinho*

*Lucas Guimarães Campregher*

*Ciência da Computação / PUC-MG*

*Projeto e Análise de Algoritmos*

1. **Introdução**

Neste trabalho iremos abordar soluções para três problemas em grafos. O primeiro deles, consiste na implementação de um algoritmo guloso para resolver o problema do caminho mais curto em um grafo com peso não negativo, utilizando o algoritmo de Dijkstra. O segundo problema consiste em, determinar se existe ou não um ciclo negativo em um grafo. O terceiro problema consiste em, desenvolver um algoritmo de programação dinâmica para resolver o problema do caminho mais curto em um grafo com aresta negativa, mas sem ciclo negativo.

Para todas as resoluções, foram implementados códigos na linguagem C++. Iremos abordá-los um por um, e discutir detalhadamente seu funcionamento e as soluções propostas, com o objetivo de analisar os códigos e estabelecer uma visão clara sobre os conceitos de análise de complexidade e otimização.

1. **Implementação**

Os códigos foram implementados no sistema operacional Windows 10, utilizando o Visual Studio Code, e o compilador GNU Compiler Collection (g++). Os códigos estão devidamente comentados para facilitar o entendimento.

* 1. **Execução**

Primeiramente, para executar os programas, deve-se baixar o projeto, ou clonar o repositório git (utilizando git clone).

A estrutura inicial do projeto consiste em uma pasta **/doc** que contém toda a documentação do projeto. (Trabalho Prático com a descrição dos problemas e a Documentação do Trabalho). Na pasta **/io** estão os arquivos de entrada padrão, utilizados para testes dos algoritmos e um arquivo geral de saída. Na pasta raiz do projeto estão os códigos **.cpp** e seus executáveis, além de um README do projeto.

Uma vez com os arquivos, para executar quaisquer um dos programas, basta estar na pasta raiz do projeto e rodar uma linha de comando com o nome do executável, como no exemplo:



Ao executar um dos programas o usuário deve informar parâmetros de forma a construir seu grafo. Se necessário, deverá informar ainda os grafos de origem e destino para execução do código. Esse processo está detalhado na seção “Testes” neste documento.

Para executar os programas com arquivos de entrada pré-definidos, basta rodar uma linha de comando como a seguinte na pasta raiz do projeto:

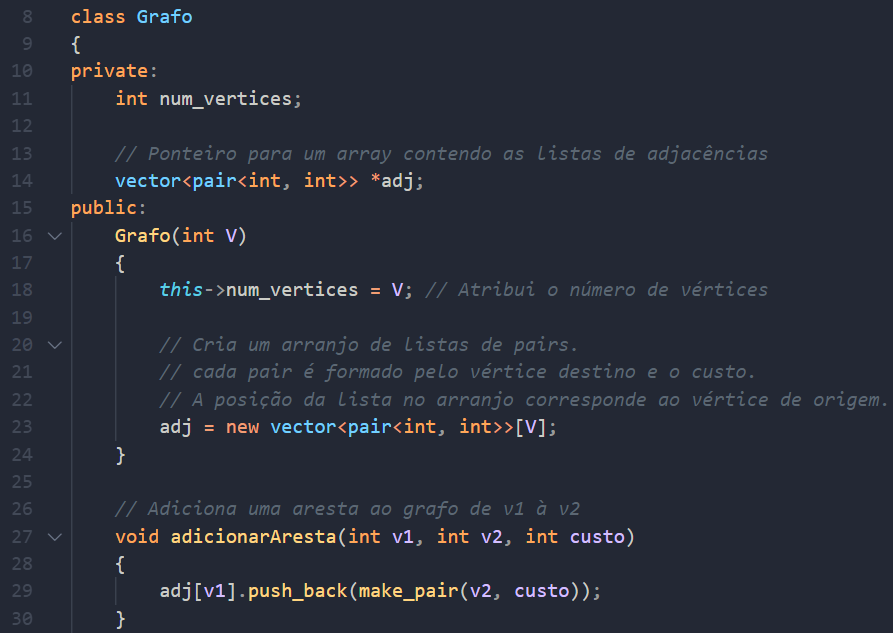
****

Para execução de algum dos outros dois programas, deve-se substituir o nome do executável (“dijkstra.exe” no exemplo). Para substituir um arquivo de entrada basta substituir o nome do arquivo (“grafoPositivo.in” no exemplo). O arquivo de saída para esses casos é sempre o mesmo, e contém a resposta na última linha.

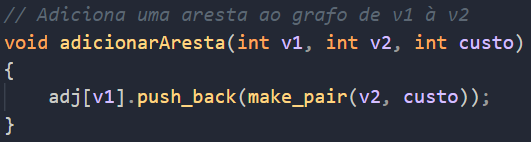
* 1. **Problema 1**

O problema 1 consiste em propor um algoritmo guloso para resolver o problema do caminho mais curto em um grafo com peso não negativo (Algoritmo de Dijkstra).

Para este caso, foi utilizada a seguinte estrutura de dados:



Temos a classe Grafo, que possui seu número de vértices, e um ponteiro com sua lista de adjacência, facilitando assim sua manipulação na execução.



A função *adicionarAresta* recebe o vértice de origem, o vértice de destino, e o peso do caminho. Adiciona um par na posição do ponteiro correspondente ao vértice de origem. Este par consiste no vértice de destino e o peso do caminho.



A função *dijkstra* é responsável por encontrar o menor caminho entre dois vértices. Primeiramente instanciamos um vetor de distâncias com cada posição como infinito, e um vetor que indica se os vértices foram visitados ou não.

Instanciamos uma fila de prioridades que é responsável por controlar quais vértices serão expandidos.

Iniciamos a distância da origem para ela mesma como 0 e inserimos o vértice de origem para ser expandido na fila de prioridades.



Em seguida, um loop que só é interrompido quando não houverem mais vértices para serem expandidos (quando a fila de prioridades estiver vazia) é executado. A variável *u* corresponde ao vértice de destino mais prioritário. Então, é realizada a expansão do vértice, caso este ainda não tenha sido expandido. Nesse processo, percorrem-se todos os vértices adjacentes de *u*, atualizando a distância de cada somente se, a distância da origem até *u* somada ao peso da aresta adjacente, for menor do que a distância atual.

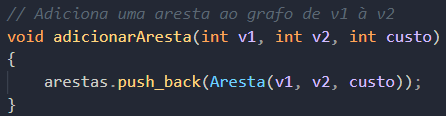
Assim, o algoritmo consegue encontrar sempre o caminho menos custoso, seguindo vértice a vértice, analisando a melhor possibilidade. Após o relaxamento de todos os vértices, retorna-se o valor da distância da origem até o destino informado.

* 1. **Problema 2**

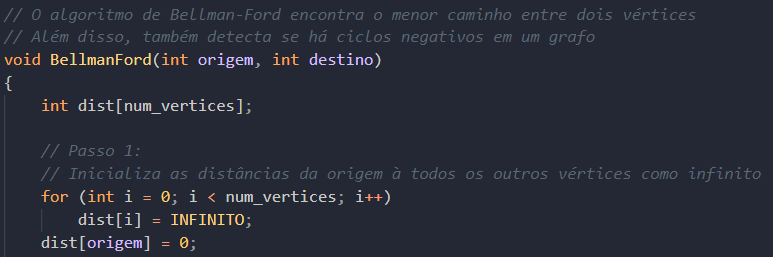
No segundo problema proposto, pede-se para que implemente um algoritmo capaz de identificar em um grafo qualquer com arestas negativas, a existência de um ciclo negativo. Para solucioná-lo, foi usado o Algoritmo de Bellman-Ford, que serve para identificar o menor caminho em um grafo com arestas negativos, sendo capaz de identificar ciclos negativos.



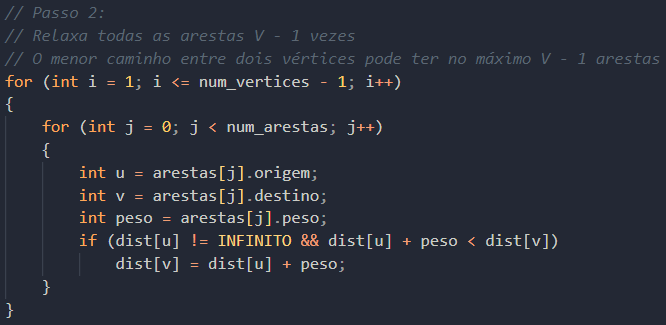
A estrutura de dados acima se diferencia da anterior pois, não possui uma lista de adjacência, mas sim um vetor de objetos da classe *Aresta*, bem como o número de arestas presentes no grafo.



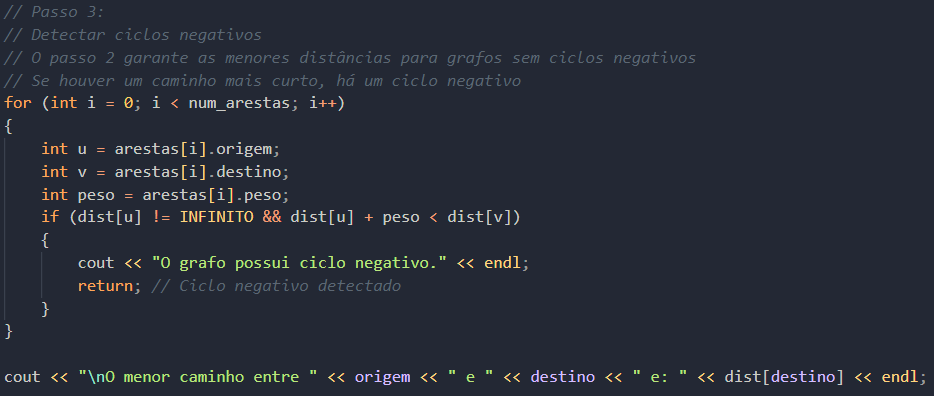
A função responsável por adicionar arestas é bem semelhante. A cada inserção, é colocado na lista um novo objeto do tipo *Aresta* a com seus respectivos vértices de origem, destino, e seu peso.



O algoritmo de BellmanFord consiste em três passos. O primeiro deles é a inicialização de uma lista de distâncias ao vértice origem, em que todas as posições se iniciam em infinito, e a distância da origem para ela mesma é 0.



O passo 2 é relaxar todas as arestas (número de vértices - 1 vezes). Para isso, percorre-se todos os vértices e arestas, atribuindo ao vetor de distâncias à origem os valores somados dos melhores caminhos.

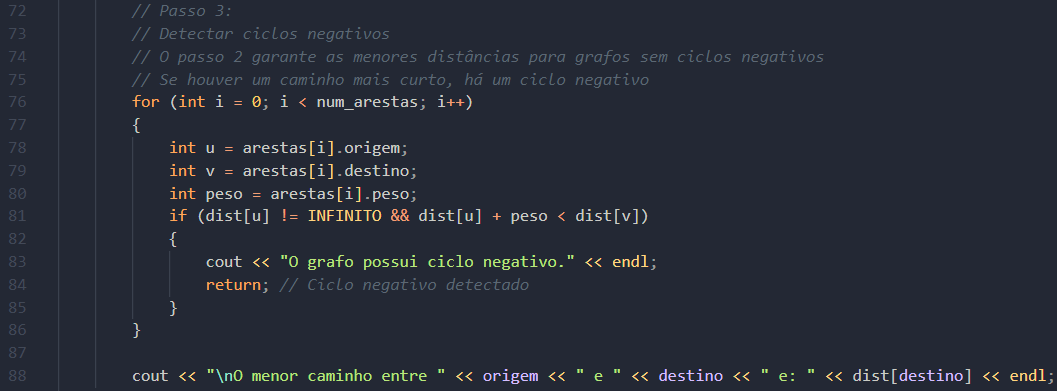


O passo 3 é a detecção de ciclos negativos. Para isso ele percorre todas as arestas verificando se a origem tem distância (soma dos pesos) diferente de infinito e se a distância da origem somada ao peso é menor que a distância do destino. Caso as duas verificações sejam verdadeiras a função retorna que encontrou um ciclo negativo.

Como iniciamos todas as distâncias em infinito, ao percorrer todas as arestas, sempre atribuímos um novo valor à distância, porque os pesos serão menores do que infinito. Quando esses valores das distâncias já estão definidos, e passamos pelas arestas outra vez, podemos dizer que, ao encontrar uma distância que somada à próxima aresta é menor do que a distância ao vértice de destino, temos um ciclo negativo, já que isso não poderia acontecer.

* 1. **Problema 3**

O problema 3 utiliza da mesma abordagem do problema 2. Utilizamos o algoritmo de Bellman-Ford para, verificar a não ocorrência de um ciclo negativo, e encontrar o menor caminho entre dois vértices.



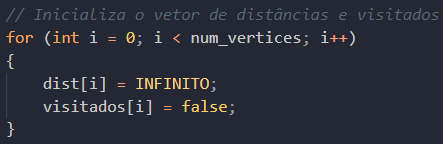
No segundo problema implementamos uma função booleana que retornava a existência de um ciclo negativo. Para o terceiro problema, caso exista um ciclo negativo, a solução é interrompida, e uma mensagem de aviso é exibida ao usuário. Caso não exista um ciclo negativo, o programa apenas retorna o valor constatado no vetor distância, na posição do vértice de destino.

1. **Análise de Complexidade**

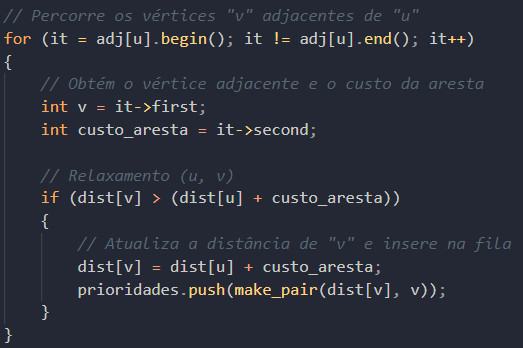
Para a análise de complexidade, iremos avaliar as funções de Dijkstra e Bellman-Ford, considerando quais são suas funções mais relevantes, e os melhores e piores casos encontrados.

* 1. **Dijkstra**

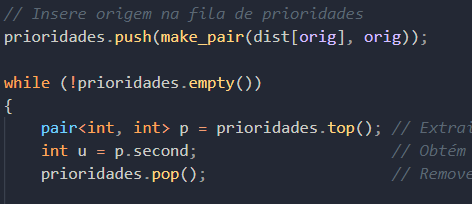
No algoritmo de Dijkstra que utilizamos, o tempo para melhor ou pior caso será o mesmo, já que ele sempre encontra a distância de todos os vértices até a origem definida, e retorna somente a distância relativa ao destino escolhido.



No primeiro momento, os vetores são inicializados, passando por todos os vértices; ou seja, em tempo **O(|V|)**, em que V corresponde ao número de vértices.



Dentro do loop onde ocorre o relaxamento dos vértices, é necessário utilizar um loop for, que percorre todos os vértices adjacentes ao vértice em análise. Nesse caso, pode-se dizer que, ao executar o loop para cada um dos vértices adjacentes, estamos concomitantemente percorrendo todas as arestas. Logo, em tempo **O(|A|)**, em que A corresponde ao número de arestas.



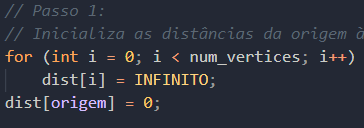
Como a lista de prioridades funciona como um heap, cada vez que precisamos inserir ou retirar um elemento, gastamos tempo correspondente a **O(log|A|)**.

Assim, temos tempo total igual a: **O(|V|) + O(|A|) \* O (log|A|)**; já que o for se encontra dentro do loop while.

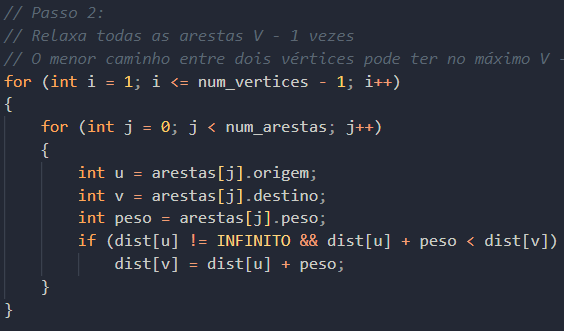
Portanto, o tempo para todos os casos da execução do algoritmo de Dijkstra corresponde a **O(|V| + |A| log|A|)**, ou **O(N log|N|)**.

* 1. **Bellman-Ford**

Os problemas 2 e 3 correspondem a execução do mesmo algoritmo (Bellman-Ford).

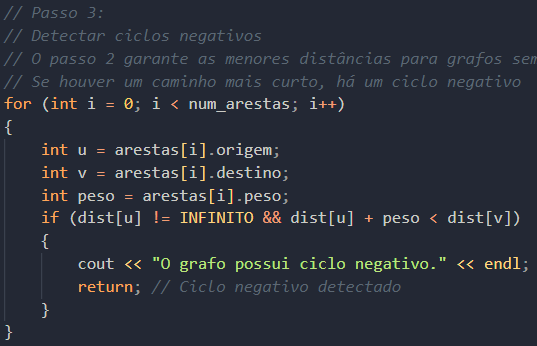


Para inicializar as distâncias realizamos um loop **V** vezes, gastando tempo **O(|V|)**, em que V é o número de vértices do grafo.



No passo 2 do algoritmo de Bellman-Ford, temos a nossa operação relevante (condição de atribuição de distâncias). O loop exterior é executado V - 1 vezes. O loop mais interno é executado A vezes, em que V é o número de vértices e A é o número de arestas. Assim, o tempo gasto nesse trecho do código é sempre equivalente a:

**O(|V-1| \* |A|)** = **O(|V| \* |A|)**



No terceiro passo, temos apenas um loop, que percorre todas as arestas. Para isso gastamos tempo igual a **O(|A|)** no pior caso. No melhor caso, temos apenas 1 checagem, portanto sendo **O(1)**.

Considerando todos os trechos do algoritmo, o nosso tempo total gasto para o melhor caso é **O(|V|) + O(|V| \* |A|) + O(1)**, que pode ser escrito como:

**O(|V| + |V|\*|A|)**

Considerando todos os trechos do algoritmo, o nosso tempo total gasto para o pior caso é **O(|V|) + O(|V| \* |A|) + O(|A|)**, que pode ser escrito como:

**O(|V| + |V|\*|A| + |A|)**

Portanto, analisando a complexidade do algoritmo temos que o tempo gasto é sempre de:

**O(N²)**

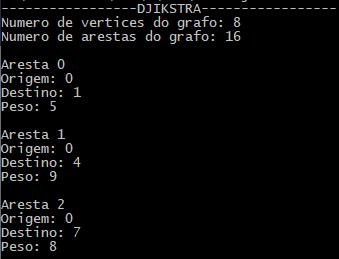
1. **Testes**

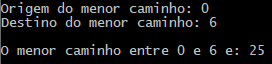
Para esse trabalho foram realizados diversos testes genéricos para cada um dos 3 problemas apresentados. Para exemplificar, foram criados arquivos de entrada para cada um desses problemas, a fim de possibilitar ao usuário entrar com seu próprio teste futuramente. O usuário também tem a opção de entrar com os dados manualmente.

O algoritmo não funcionará caso o usuário insira valores inválidos nos seus casos de teste.

* 1. **Funcionamento** 
     1. **Teste manual**

Para realizar um teste manual, basta executar um dos programas como visto anteriormente, e inserir os dados conforme é solicitado. Podemos ver um exemplo a seguir:

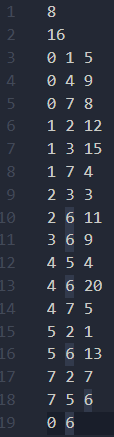




A resposta é dada logo em seguida.

* + 1. **Teste com arquivo de entrada**

Os testes pré-definidos ficam na pasta /**io**, que possui 3 arquivos. Caso o usuário queira incluir seu próprio arquivo de teste, basta inserir um novo arquivo com o seguinte formato:



Na primeira linha se encontra o número de vértices(V) do grafo. Na segunda linha se encontra o número de arestas(A) do grafo. Depois existem mais um número A de linhas . Existem 3 números em cada uma dessas A linhas. O primeiro desses números representa a origem da aresta, enquanto o segundo número representa o destino dessa aresta e o terceiro o peso da aresta.

Para o programa de detecção de ciclos negativos não é necessário informar os vértices de origem e destino.

* 1. **Problemas**

Para testar os problemas os quais nos foram propostos, utilizamos de arquivos pré-definidos que contemplam todas as possibilidades de retorno na execução de cada um dos programas. Também fizer testes personalizados, inserindo entradas alternativas para garantir que o algoritmo estava funcionando corretamente.

Os testes realizados permitiram que nos fosse garantido o bom funcionamento dos programas conforme foi proposto no enunciado.

1. **Conclusão**

Com a elaboração deste trabalho, pudemos entender mais profundamente o funcionamento de algoritmos gulosos, como o de Dijkstra, bem como a execução de algoritmos dinâmicos, como o de Bellman-Ford. Percebeu-se que o algoritmo de Bellman-Ford se assemelha bastante ao de Dijkstra em seu conceito (expandir todos os vértices), porém sua maior diferença está em como isto acontece, visto que Dijkstra usa de uma estrutura auxiliadora (como uma fila de prioridades), enquanto Bellman-Ford apenas passa por todas as arestas e vértices fora de uma ordem específica.

Ademais, foi possível rever conceitos aprendidos na disciplina de PAA (Projeto e Análise de Algoritmos), como análise de complexidade. O entendimento de todas as partes de cada um dos algoritmos foi a maior dificuldade encontrada para fazer uma análise que fosse coerente.

Devido às diferenças de como os dois algoritmos expandem os vértices, notamos por meio da análise de complexidade que, o tempo de execução de Dijkstra é menor. Apesar de uma execução mais eficiente em relação ao tempo gasto, o algoritmo de Dijkstra é incapaz de reconhecer ciclos negativos, uma limitação que Bellman-Ford soluciona perfeitamente.

1. **Bibliografia**

***Advanced Data Structures: Dijkstra's Algorithm Time Complexity*** *-* [*https://www.youtube.com/watch?v=YMyO-yZMQ6g&ab\_channel=NiemaMoshiri*](https://www.youtube.com/watch?v=YMyO-yZMQ6g&ab_channel=NiemaMoshiri)

***Bellman-Ford Algorithm | DP-23***

[*https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/*](https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/)

***Dijkstra's algorithm: Finding shortest path between all nodes -***

[*https://iq.opengenus.org/dijkstras-algorithm-finding-shortest-path-between-all-nodes/*](https://iq.opengenus.org/dijkstras-algorithm-finding-shortest-path-between-all-nodes/)

***Dijkstra’s Algorithm -***

[*https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm*](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

***Dijkstra on sparse graphs***

[*https://cp-algorithms.com/graph/dijkstra\_sparse.html*](https://cp-algorithms.com/graph/dijkstra_sparse.html)

***Algoritmo de Bellman-Ford***

[*https://www.ime.usp.br/~pf/algoritmos\_para\_grafos/aulas/bellman-ford.html*](https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bellman-ford.html)

***Detect a negative cycle in a Graph | (Bellman Ford)***

[*https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/*](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/)

1. **Anexos**

***Repositório do projeto -***

<https://github.com/Guimma/PAA>