UNIVERSIDADE FEDERAL DO CEARÁ

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Antônio Jorge Nobre da Costa

Gustavo Campelo de Sousa

Kauan Santana de Sousa

Weslley Fernando Teixeira Chaves

RELATÓRIO DO TRABALHO DE ALGORITMOS EM GRAFOS

CRATEÚS

2024

RELATÓRIO DO TRABALHO DE ALGORITMOS EM GRAFOS

Relatório do trabalho em grupo realizado na disciplina de algoritmos em grafos, do qual foi implementado o algoritmo para o problema da árvore de caminhos de custo mínimo restrito a grafos acíclicos dirigidos.

Professor: Prof. Rafael Martins Barros

CRATEÚS

2024

RESUMO

Este relatório consiste em expor e argumentar sobre a realização do trabalho da disciplina de Algoritmos em Grafos, do qual foi-se atribuído este presente trabalho com algumas finalidades, com o objetivo de acrescer e fixar o aprendizado do conteúdo aos alunos. A turma foi dividida em grupos arbitrários contendo 4 integrantes, que deveriam se dividir em tarefas distintas que convergirão no decorrer da realização do trabalho atribuído. Cada grupo ficou com uma problemática passada ao professor, e a problemática presente, dissertada e argumentada aqui trata-se da **IMPLEMENTAÇÃO DE UM ALGORITMO PARA O PROBLEMA DA ÁRVORE DE CAMINHOS DE CUSTO MÍNIMO RESTRITO A GRAFOS ACÍCLICOS DIRIGIDOS VISTO. O SEU ALGORITMO DEVE RECEBER UM DIGRAFO (GRAFO DIRIGIDO) PONDERADO $G = (V, E)$ E UM VÉRTICE DE ORIGEM $s \in V$ E RETORNAR UM NOVO GRAFO T QUE É UMA ÁRVORE DE CAMINHOS MÍNIMOS COM RAIZ EM s.**

SUMÁRIO

1 INTRODUÇÃO

Tratando da questão de grafos simples, os dígrafos, ou grafos dirigidos, também podem ser incluídos na implementação do problema da árvore de caminhos mínimos acíclicos dirigidos. Isto porque ao recordar-se no que consiste um problema de caminhos mínimos, pode-se definir da seguinte maneira: Em um problema de caminhos mínimos, temos um grafo dirigido ponderado G = (V, E), com função peso w: E $\rightarrow$ $\mathbb{R}$ que mapeia arestas para pesos de valores reais. O peso do caminho p = ⟨v0, v1, ..., vk⟩ é a soma dos pesos de suas arestas constituintes. Devido a isso, o peso do caminho mínimo pode ser definido como a soma das arestas partindo de um vértice fonte s até um vértice t, em que diferentemente da árvore geradora mínima, não tem a obrigatoriedade de possuir todos os vértices do grafo original em seu resultado final, visto que o caminho mínimo é o caminho de menor valor da soma das arestas partindo de uma fonte s até um vértice t, e não estamos tratando de um subgrafo gerador, e sim de um caminho de custo mínimo dentro do grafo original. E salientando, uma árvore de caminhos mínimos faz com que todo caminho formado pelas arestas do grafo, sejam de custo mínimo, ao que no final, realizando a soma, se mostre como o menor custo de caminho entre todos os pares de vértices que formam essa árvore de caminhos mínimos.

A ideia da implementação desse trabalho se manterá de acordo com as regras dos caminhos mínimos de fonte única, como por exemplo, não permitir a existência de ciclos de peso negativo e positivo, pois de acordo com a explicação do Cormen, isso não seria possível, resultando apenas em que qualquer caminho acíclico em um grafo G = (V, E) contém no máximo |V| vértices distintos, ele também contém no máximo |V| - 1 arestas. Assim, podemos restringir nossa atenção a caminhos mínimos que tenham no máximo |V| - 1 arestas, logo estaremos tratando de grafos com caminhos simples, que não possuem ciclo.

Salientamos nesta introdução a implementação do algoritmo utilizando caminhos mínimos de fonte única, que permite a utilização de custos arbitrários em seus arcos (positivos ou negativos), porém aqui nos limitaremos apenas a arcos de custo não negativo, ou seja, arcos (u,v) >=0, e mesmo que existam arestas de peso negativo, não deve existir nenhum ciclo de peso negativo, visto que o algoritmo sempre deixa bem definida essa questão dos caminhos mínimos acíclicos.

O relatório a seguir trata de: uma explicação sobre a implementação realizada, os componentes implementados, a linguagem utilizada, além da lógica pensada. Há também um capítulo dedicado às dificuldades enfrentadas pelo grupo durante a implementação, a contribuição para o aprendizado e por fim os testes realizados e apresentados em gráficos seguindo o padrão descrito no documento do trabalho.

2 IMPLEMENTAÇÃO

O trabalho foi realizado na linguagem Python, do qual decidiu-se dividir a implementação em três arquivos para que não houvesse confusão no momento da criação do grafo aleatório gerado, do algoritmo do DAG-Shortest Paths e também das estatísticas, estas que inicialmente pensamos em executar de maneira separada, porém ao final do trabalho ficou integrada ao arquivo principal do trabalho.

A parte inicial pensada deu-se de maneira focada e completamente dedicada à geração do grafo aleatório (ou pseudoaleatório), pois foi exatamente esse ponto que nos tirou ponto durante o trabalho anterior, e além de que estava claramente descrito na descrição do documento deste. A sua implementação teria que vir a convergir com o algoritmo a ser implementado, porém inicialmente geramos um grafo completamente aleatório, porém que permitia ciclos, e no fim das contas percebemos que os erros posteriores obtidos eram decorrentes exatamente de que a ordenação topológica necessária não era possível, visto que uma das regras principais do DAG-Shortest Paths é exatamente ser uma maneira de modelar o grafo, onde este não possui ciclo, e de fato era explícito durante a explicação do conteúdo, desde o material de estudo e mesmo nas pesquisas, porém era um detalhe sutil que acabou não sendo observado de início, no entanto, não atrapalhou nas fases posteriores da construção do código Python.

A geração do grafo aleatório se deu a partir da importação da biblioteca Random, que implementa geradores de números pseudoaleatórios de diversos tipos e aplicações diversas. E ela basicamente para todo tipo de aplicação, não sendo diferente neste trabalho, aplica a geração de um ponto flutuante aleatório de maneira uniforme no intervalo semiaberto [0.0, 1.0), e com isso realiza diversas aplicações e operações, e de acordo com a documentação do próprio Python sobre a biblioteca Random: “usa o Mersenne Twister como gerador de núcleo. Produz pontos flutuantes de precisão de 53 bits e possui um período de $2^{19937-1}$. O Mersenne Twister é um dos geradores de números aleatórios mais amplamente testados existentes.” Portanto, ela é aplicada mais à frente, mas não antes de definirmos a classe Grafo por si só, que se inicia passando como parâmetro o próprio grafo que será criado e a quantidade de vértices que ele conterá. Com isso o grafo a ser criado terá um parâmetro com o valor de V atribuído a ele, e com isso será atribuído aos adjacentes do grafo criado, uma lista de adjacências para cada vértice v no alcance da quantidade de vértices de Grafo.

Junto da nossa classe Grafo, é definida uma função de inserção de arestas (arcos) que passam como parâmetro o grafo em si, a conexão de um para outro vértice (v,w) e o peso,

e assim a função irá adicionar na lista de adjacências uma aresta direcionada de v para w adicionando também o peso dessa conexão entre vértices, formando assim uma estrutura de destino e valor, respectivamente. Após isso viu-se necessário mostrarmos as arestas do grafo, assim como seus pesos, assim foi realizada uma varredura por todos os vértices do grafo foi e exibida no terminal o vértice da iteração, seguido da lista de adjacências com o vértice destino e o peso para tal vértice destino, ou seja, representa uma aresta saindo de um vértice v para seu adjacente w com um peso associado (weight). Exemplo: self.adj[0] poderia ser [(1, 8), (2, 5)], o que significa que existe uma aresta do vértice 0 para o vértice 1 com peso 8, e uma outra do vértice 0 para o vértice 2 com peso 5; isso será repetido enquanto durar o for. E visando que o grafo gerado deve ser lido a partir de um arquivo de arestas (contendo a lista de adjacências) de acordo com a descrição do trabalho, foi criada a função de salvar o grafo gerado, onde será salvo ele mesmo junto de seu nome, onde será feito isso no modo de escrita da função e para cada vértice e suas adjacências, e para cada peso contido nessas adjacências, será escrito no arquivo e separado da seguinte maneira (vértice, vértice de destino e peso de aresta vw).

Tendo nosso grafo completamente definido, agora seria preciso gerar o grafo aleatório com base na definição feita do Grafo em si, e para isso passaríamos o conjunto de vértices, de arestas e o peso máximo definido em tempo de execução, e após sua parametrização, é calculada a probabilidade de uma aresta ser inserida entre dois vértices, se baseando no número de arestas pela quantidade total de pares de vértices que um grafo dirigido deve ter, que é V(V-1), e com ela será possível verificar se há ou não uma aresta inserida entre um par de vértices. Após isso atribuímos G a um grafo que inicia vazio e é realizada iterações que percorrem todos os vértices v, e com isso para cada vértice destino w no alcance de v+1 (vértice seguinte, maior que v), garantindo que não haverá vértices de retorno à v (evitando ciclos de w para v) no conjunto V, será gerado um peso real aleatório, e se a probabilidade desse real aleatório (entre 0 e 1, de acordo com a documentação do Python) for menor que a probabilidade calculada acima, ou seja, caso essa condição seja satisfeita, irá gerar um peso aleatório entre 1 e o peso máximo definido na variável max_weight. E com isso tudo satisfeito, é inserida uma aresta no Grafo antes vazio, agora preenchido com vértice, vértice de destino e peso, por meio de um arco.

Com isso já é possível atribuirmos a quantidade de vértices, arestas e o peso máximo da maneira que desejarmos e gerar corretamente o grafo, e após exibir o grafo no terminal e também logo salvar o arquivo de maneira correta em um “.txt”, que será usado depois para aplicar o algoritmo DAG-Shortest Paths.

O arquivo principal(main) é composto pela criação de outra classe Grafo que também é definida iniciando-o junto de seus adjacentes e com isso é atribuído ao número de vértices do grafo o comprimento da própria lista de adjacências. É criada também a função da classificação da ordenação topológica, passando o grafo, o vértice, o vértice visitado e a pilha em que será armazenada, então é necessário verificarmos se o vértice for maior/igual ao número de vértices no grafo, validando a condição, e com isso marcará o nó da vez como visitado. E com isso recorrerá para todos os vértices adjacentes ao vértice da vez, realizando a estrutura de repetição, passando por todos esses vértices e verificando, e caso todas essas forem cumpridas, o grafo em si irá atribuir o nome como visitado e incluí-lo à pilha de visitados, mas o que isso significa? Significa que ao final disso, será adicionado o vértice atual à pilha que armazena a ordenação topológica. Com isso pode-se realizar o algoritmo DAG Shortest-Paths que necessitava dessa ordenação topológica para permitir sua execução, de acordo com as aulas e também com o pseudocódigo do livro do Cormen. Seguindo, para ser feito, precisamos passar o grafo em si e sua fonte, visto que estamos tratando de caminhos mínimos de fonte única, necessitamos de um vértice fonte, e passamos ele, iniciando todos os vértices como não visitados (parecido com o ∞ do pseudocódigo), e iniciando nossa pilha como vazia, e com isso temos uma função recursiva que armazenará a ordenação começando a partir do vértice fonte, verificando se o vértice não for visitado, este é marcado como visitado, a cada iteração. E no fim do algoritmo, imprimimos a ordenação topológica da pilha.

Após, será processado os vértices nessa ordem da iteração, realizando a devolução do valor no topo da pilha e atualizando as distâncias de todos os vértices adjacentes, então iremos realizar a passagem por todo nó e peso na lista de adjacências do vértice i, que contém todos os vértices conectados por arestas que partem de i, com isso será verificado se a distância atualmente conhecida até o vértice nó (dist[node]) pode ser melhorada (relaxada). E caso possa, a nova distância conhecida até o vértice nó é atualizada por um valor melhor, relaxando a aresta, permitindo sua atualização para a distância menor possível. Feito isso a árvore de caminhos mínimos é imprimida no terminal partindo do vértice fonte s e é exibida a distância da fonte até o vértice destino i, e verifica se para retornar a distância “INF” (quando não há conexão de vértices entre s e i), se ocorrer, é imprimida essa distância inexistente, senão, imprime corretamente o caminho mínimo entre o vértice fonte e o de destino.

Feito tudo isso, é possível resgatar o arquivo do grafo contendo a lista de adjacências, e abrindo o arquivo definido no modo de leitura e lendo cada linha contida no arquivo e aplicando uma função map para vértices, vértices destino e o peso, ou seja, será lido

todas essas partes linha por linha. E feito isso o grafo será criado com todos os vértices, retornando o grafo em si. Após isso finalmente será possível carregar o grafo a partir do arquivo de texto nomeado anteriormente. E a partir de G, calcular o algoritmo DAG Shortest Paths partindo da fonte s, que podemos definir arbitrariamente. E a partir da execução do arquivo principal, nos exibirá a ordem topológica e o algoritmo finalizado com a árvore de caminhos mínimos exibida da maneira correta.

3 DIFICULDADES ENFRENTADAS

Relacionado à dificuldades, houveram algumas enfrentadas durante o processo de implementação de todo o sistema, pois foi pensado em reaproveitar algo do primeiro trabalho da disciplina, só que acabou não sendo possível, pois não implementamos um gerador de grafos aleatório, logo foi tido que recorrer a pesquisas de bibliotecas que gerassem números aleatórios e que permitissem ser incluídas à vértices e arestas dentro de um arquivo, então também viu-se a necessidade de inserirmos toda aquela informação em um tipo de arquivo, e o formato mais simples que pensamos foi o de documento de texto padrão “.txt”. Então uma das dificuldades foi ter de implementar essas partes do zero; e quanto à questão do algoritmo DAG Shortes Paths, junto da ordenação topológica e seu relaxamento, houve também uma boa dificuldade, pois são diversos passos a serem seguidos, testados e comprovados, então para superarmos, separamos todos os conceitos dos slides da aula, vimos o que tinha que conter em todas os passos, desde vértices, arestas, pesos nas arestas, ordenação topológica, a ausência de ciclos, como armazenar a ordenação, entre outros. Todos esses pontos considerados acabaram ajudando deveras durante todo o processo do trabalho, e os pseudocódigos do Cormen foram ajudaram na ordem em que teríamos que fazer o código e como ele se comportaria e seria organizado. Uma outra dificuldade que vale a pena relatar foi o tempo que decidimos fazer o trabalho, pois de fato foi-nos dado 1 mês para realizarmos tudo, porém veio toda uma priorização diferente dos membros com outras disciplinas, emprego e outras ocupações, que tudo acabou ficando para perto dos últimos dias de entrega, e ainda assim com tudo isso não nos impediu de concluir e ir relatando cada parte para poder ser incluído nesse relatório. É acreditado por todos que o tema sorteado para o grupo foi um dos mais simples de aplicar, devido à sua facilidade de desenho em um caderno e aplicação dos métodos vistos em sala de aula, mas à medida que o tamanho do grafo tende a crescer, a verificação de corretude fica cada vez mais difícil, e foi aí que entrou a resolução para essa

dificuldade, que também está inserida no capítulo de testes realizados para a validação da nossa implementação. No geral, as dificuldades podem ser resumidas em pontos citados abaixo:

- Tempo restante para começar o trabalho
- Escolha da linguagem de programação
- Relembrar as propriedades que compõem o DAG - Shortest Paths
- Definir as funções que farão parte dessa definição do algoritmo
- Abstração do pseudocódigo para o código em si
- Testes realizados para validação da implementação

Segue abaixo como esses problemas foram resolvidos

- Os quatro membros unidos durante todo o processo para que tudo pudesse ser concluído no tempo correto, dando tempo para estudar para a prova
- Escolha da linguagem Python, que tem uma gama excepcional de recursos e bibliotecas, como a Random, e fóruns online completos para pesquisa de possíveis erros gerados, além da plataforma Minha Biblioteca, que possui o livro Algoritmos, ajudando muito durante todo o processo, e também os slides das aulas
- Utilização do livro Algoritmos e os slides das aulas, sendo anotados para serem aplicados.
- Partimos do pseudocódigo e anotamos o que seria necessário para gerar um grafo aleatório a partir de uma quantidade pré-definida de vértices, arestas e peso máximo de arestas.
- Passo semelhante ao anterior, com um completando o outro
- Utilização de plataformas de grafos online, que tem o algoritmo de maneira visual para realizar os testes com grafos pequenos, e também o teste por meio do caderno, com lápis e borracha

4 TESTES REALIZADOS PARA VALIDAÇÃO DA IMPLEMENTAÇÃO

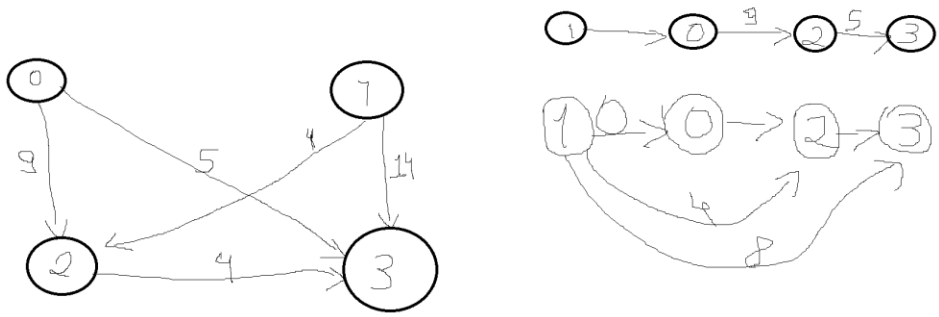
Se tratando dos testes, esses foram divididos em duas etapas maiores, que é a testagem da geração aleatória do grafo, para diversos tamanhos e se este realmente foi salvo corretamente no arquivo de texto, e também sobre a ordenação topológica e aplicação do DAG-Shortest Paths. E dentro desses houveram etapas menores que foram também aplicadas e serão explicitadas a seguir com maiores detalhes.

Ao falar na implementação do grafo gerado aleatoriamente e seu armazenamento, primeiro verificamos se o grafo gerado era realmente aleatório, então foi executado diversas vezes a mesma função de gerar o grafo sem alterar nenhum valor e a cada vez foi retornado um grafo formado de maneira diferente e sem violar a propriedade de não gerar ciclos, pois se houvesse ciclos, não haveria como ter uma ordenação topológica, e isso foi garantido, pois todos os grafos gerados foram permitidos haver uma ordenação topológica posteriormente.

Outro teste que se viu a necessidade de fazer foi o de zerar a quantidade de arestas e vértices, pois pensamos que poderia haver a geração do grafo mesmo assim, só que contendo a lista de adjacências completamente vazia, no entanto, ao realizar isso, o código nem executou, devido à operação da probabilidade definida na função do grafo aleatório, onde essa operação: prob = A / (V * (V - 1)), ao termos o valor 0 ou 1 de vértices, dá erro de divisão por 0, então nosso grafo só é permitido um valor maior ou igual a 2 de vértices, porém de arestas é permitido termos um número maior ou igual a 0.

Por último, mas não menos importante, a questão de salvamento, que não foi muito difícil de testar, pois bastou verificar o grafo gerado junto do arquivo em que as informações foram salvas e vendo que elas eram iguais as exibidas no terminal, o teste foi concluído, mas não antes sem realiza-lo diversas vezes com grafos de diferentes tamanhos.

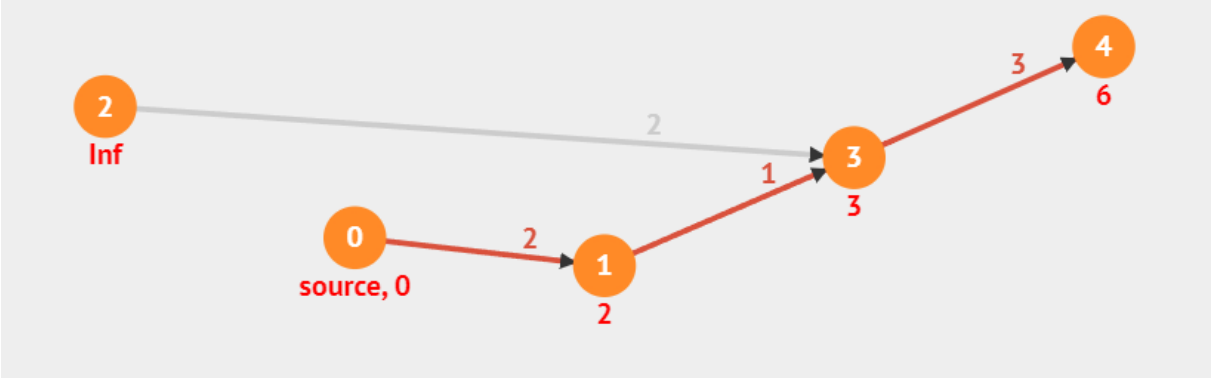
Passando para a main, os testes a serem realizados se resumiam em realmente saber se aquela ordenação topológica gerada para os grafos gerados era verdadeira, e para isso foi realizado para os grafos menores a verificação no papel e também no Paint e para os grafos maiores, foi utilizada uma plataforma que ao inserirmos os vértices e arestas correspondentes, ele retornava uma árvore de caminhos mínimos, essa plataforma se chama Visual Go, e nela há diversos tipos de algoritmos, e ao selecionar o de SSSP (Single-Source-Shortest-Path), ele nos retorna corretamente a ordenação topológica e aplica o algoritmo testado. Abaixo segue imagens da implementação do algoritmo:



Exemplo de verificação da corretude do algoritmo desenhado no Paint (Exemplo diferente da imagem abaixo)

```
Ordem Topológica: [2, 0, 1, 3, 4]
Árvore de Caminhos Mínimos a partir do vértice 0:
Distância do vértice 0 ao vértice 0: 0
Distância do vértice 0 ao vértice 1: 2.0
Distância do vértice 0 ao vértice 2: Inf
Distância do vértice 0 ao vértice 3: 3.0
Distância do vértice 0 ao vértice 4: 6.0
```

Retorno do terminal a partir da lista de adjacências



Retorno do mesmo grafo acíclico dirigido realizado na plataforma online, e vê-se que o resultado é o mesmo. O resultado se manteve alterando o vértice fonte, ou seja, o resultado do site a partir de outras fontes foi igual ao resultado da implementação, começando de outro vértice fonte. Logo, é possível verificar que tanto a ordenação topológica, como o algoritmo é executado corretamente e sem nenhum problema aparente.

5 ANÁLISE DE DESEMPENHO DA IMPLEMENTAÇÃO

A análise de desempenho foi uma parte da implementação que não houve muito segredo nem dificuldades extremas para ser implementada, pois seu funcionamento foi de maneira semelhante com o do primeiro trabalho, só com o acréscimo de mais testes de desempenho a serem realizados. A descrição do trabalho nos pareceu um pouco confusa, e

para evitar possíveis perdas de pontos, foram criados dois arquivos para análise de desempenho, um que gera 10 grafos aleatórios e calcula o tempo médio dessas execuções diferentes, que será exibido abaixo com o resultado desses testes. Já em relação ao algoritmo de Shortest Paths, ele mede o tempo apenas uma vez para cada um dos testes exibidos no gráfico do Matplotlib. Os resultados serão exibidos da maneira que está no padrão do relatório, além de uma pasta onde estarão os arquivos de cada teste salvos no formato de imagem, que a biblioteca utilizada nos permitiu fazer, assim como no primeiro trabalho.

Explicando mais detalhadamente cada um dos 2, começando pelo gerador de grafos aleatórios, que importamos do arquivo a função que gera o grafo em si, e definimos alguns parâmetros vindos do outro arquivo e alguns novos que farão parte da medição das estatísticas. Os tempos de execução serão armazenados em uma lista iniciada vazia, e iremos medir o tempo através do for passando o numero de grafos gerados e pelo número de testes feitos no grafo aleatório gerado, com isso medimos o tempo e geramos a média da soma dos tempos de testes sobre o número de testes e adicionamos o resultado ao tempo médio. Após isso é só mostrar o gráfico com a quantidade de execuções, passando o tempo de execução.

Com a medição do tempo do Shortest Paths, foi conseguido realizar os testes, porém de uma maneira mais grosseira, definindo uma função simples de medição do tempo que chama o arquivo de adjacências e o vértice fonte dela, carregando o grafo e com isso medindo normalmente o tempo após a execução do algoritmo de caminho mais curto, retornando o tempo da execução. E após isso o código é basicamente o mesmo do anterior, com a diferença que não foi possível medir a média da execução para 10 grafos gerados, apenas realizar 10 testes com o mesmo grafo e fazer sua média, não conseguimos adaptar o código para realizar essa tarefa sem que ele quebrasse, porém ele mede o tempo de execução e mostra sua média no próprio gráfico, realizando os testes no mesmo grafo do arquivo de adjacências, mas executando o mesmo teste diversas vezes, podendo ter uma média do tempo de execução, entretanto, foi realizado testes com todos os tipos de tamanhos de grafo, o que permitiu termos uma média correta de tempo para todos os tamanhos de grafo.

A complexidade do algoritmo de caminhos mínimos de fonte única para grafos acíclicos dirigidos tem complexidade O (V+E), onde V é o conjunto de vértices do grafo e E é o número de arestas, e tendo em vista que todos os DAGs gerados e aplicados o algoritmo irão crescer de maneira linear, por exemplo, temos 2 vértices e 1 aresta, caso dobrarmos esse número para 4 vértices e 2 arestas, o tempo de execução também dobrará, pois dobramos a quantidade exata de elementos para processar, ou seja, se V e A dobrarem, o tempo também aumenta, além de que ao relaxar as arestas do grafo acíclico dirigido de acordo com a

ordenação topológica de seus vértices, podemos calcular caminhos mínimos de uma fonte única no tempo O(V + E). E ao realizar diferentes testes com os tamanhos variados de V e E, iniciando com E fixo e V variando, 250 vértices e 500 arestas, o tempo médio foi 17.64ms, e dobrando esse número de vértices e mantendo o de arestas, obtivemos 41.44ms. Realizando de maneira contrária, com V fixo e E variando, obtivemos 41.89ms, e dobrando esse número de vértices, obtivemos 87.81ms. Levando em conta esses testes simplórios, é evidente que ao dobrar o número de vértices, o tempo meio que dobra também. Mas tentando explicar de uma maneira mais coesa, é que é realizada uma ordenação topológica passando por todos os vértices do grafo, e consequentemente por todas as suas arestas adjacentes, logo a quantidade de tempo necessária para que a ordenação topológica percorra tudo é como se fosse a soma do conjunto de vértices e o conjunto de arestas, processando cada aresta exatamente uma vez, e não repetindo verificação de arestas como em outros algoritmos. Resumindo, complexidade total do algoritmo é a soma dos tempos necessários para a ordenação topológica e para o relaxamento das arestas, e sim, ela se mantém durante a execução do nosso algoritmo.

A seguir seguem as tabelas esperadas para a análise de desempenho no padrão solicitado, porém tomamos a liberdade de adicionar uma legenda a mais para que se torne mais simples de visualizar e entender o que tentamos fazer para estar de acordo com as orientações do trabalho:

TABELA DA MÉDIA DO TEMPO DE GERAÇÃO DO GRAFO

Nº de Vértices do Grafo (|A| = 980)	100	200	300	400	500
Tempo médio de execução	0.64ms	1.44ms	2.99ms	5.01ms	8.31ms

Nº de Vértices do Grafo (|V| = 100)	980	1960	2940	3920	4900
Tempo médio de execução	0.66ms	0.88ms	0.92ms	1.06ms	1.09ms

TABELA DO CÁLCULO DO ALGORITMO DAG-SHORTEST PATHS

Nº de Vértices do Grafo (|A| = 980)	100	200	300	400	500
Tempo médio de execução	7.46ms	14.15ms	20.65ms	29.21ms	37.22ms

Nº de Vértices do Grafo (|V| = 100)	980	1960	2940	3920	4900
Tempo médio de execução	7.50ms	7.90ms	8.10ms	8.50ms	9.31ms

Todas as imagens dos testes realizados estão salvas nas pastas com os nomes correspondentes, graças à biblioteca do Matplotlib que nos permite exibir um gráfico mostrando a quantidade de testes feitos e a média desses testes, para cada célula da tabela mostrada acima. Os testes foram realizados em uma máquina com a configuração: Acer Nitro 5, com processador Ryzen 7 4800h com 8 núcleos e 16threads, Memória 24GB DDR4 3200MHz e uma GTX 1650 4GBVRAM. A IDE utilizada foi o Visual Studio Code, onde foi implementado utilizando algumas bibliotecas úteis para o objetivo do trabalho com a linguagem Python.

5 CONCLUSÃO

Esse trabalho mostrou ser de um nível de desafio impressionante e por vezes estressante, tanto que mesmo tendo ficado com o tema, digamos mais simples de implementação, as dificuldades enfrentadas já citadas no outro capítulo se mostraram bem cansativas e diversas vezes tivemos vontade de desistir, mas isso não aconteceu e conseguimos entregar tudo, da maneira pedida na descrição do trabalho; talvez com possíveis deslizes e a falta de verificação de algumas coisas a fim de tratarmos os erros possível, mas não se viu necessário e afim de economizar tempo nos apressamos em entregar as principais funcionalidades da maneira mais caprichosa e rápida que podia ser feito, pois não havia tempo para refinamentos. Ao fim das contas, todos do grupo acreditamos que tudo ficou bem feito, desde o momento de concepção das ideias, da escolha da linguagem de programação, dos testes feitos, da maneira que foi implementado, dos testes de desempenho e principalmente do relatório sempre montado com muito cuidado e dedicação pelos membros do grupo, tentando formalizar da maneira mais didática possível tudo que teríamos de expressar e realmente esperamos que seja entendido. Futuramente se houver a oportunidade de trabalhar com algo do tipo, todos os membros estão aptos a proverem ideias de melhoria e para deixar tudo o mais perfeito possível, com todas as verificações possíveis, detalhes de documentação e até mesmo a elaboração de uma exibição mais gráfica do grafo contendo sua configuração inicial até a ordenação topológica obtida. Pois não há nada que tempo, dedicação e disciplina não consigam fazer quando se está interessado em aprender mais sobre uma área tão interessante que é a de algoritmos de otimização.