Universidade Estadual de Maringá
Centro de Tecnologia
Departamento de Informática

CIÊNCIA DA COMPUTAÇÃO
ALGORITMOS EM GRAFOS (6898/01)
Relatório do trabalho: Geração de árvores aleatórias

Professor: Marco Aurélio Lopes Barbosa

Discentes

R.A.	Nome
112679	Guilherme Panobianco Ferrari
112683	Stany Helberth de Souza Gomes da Silva

Introdução

O trabalho consiste na geração de árvores aleatórias. Foi desenvolvido e testado três algoritmos para a geração das árvores aleatórias. Todos esses algoritmos recebem como parâmetro um inteiro *n* maior que 0 e retorna uma árvore aleatória, que são:

- Random-Tree-Random-Walk, que retorna uma árvore aleatória.
- Random-Tree-Kruskal, que retorna uma árvore aleatória construída pelas arestas produzidas pela função MST-Kruskal.
- Random-Tree-Prim, que retorna uma árvore aleatória construída pelas arestas produzidas pela função MST-Prim.

Também há a função auxiliar Diameter, que recebe como parâmetro um grafo *G*, e retorna o diâmetro do grafo passado, que será usado para testar a corretude dos algoritmos desenvolvidos para a geração das árvores.

Desenvolvimento

Linguagem utilizada:

Foi utilizada a linguagem Python 3 para o desenvolvimento do trabalho.

Como os grafos e atributos foram representados:

Foram criadas duas classes, uma para o grafo, chamada de *Graph*, que recebe como parâmetro uma lista dos vértices, uma lista de adjacência, o número de vértices, e se o grafo é cíclico, que tem como valor inicial *None*, que posteriormente será utilizado para verificar se o grafo gerado é uma árvore ou não. Além disso, essa classe tem como atributo adicional uma matriz de adjacências, que será manipulada utilizando a biblioteca numpy, e utilizada nos algoritmos de Kruskal e Prim.

A outra classe criada, para os vértices, chamada de *Vertex*, recebe como parâmetro o *índice*, o atributo *d*, o atributo *pai* e o atributo *cor* do vértice que será criado. Além disso, também há dois atributos adicionais, o *rank*, que será utilizado posteriormente no algoritmo de Kruskal, e a *chave*, que será utilizada no algoritmo de Prim.

Informações importantes sobre a implementação

Para conseguir o otimizar ao máximo o tempo de execução do algoritmo de Kruskal, foi utilizado a seguinte ideia:

Como a matriz de adjacências é uma matriz espelhada, ou seja, a parte de cima é igual a parte de baixo, como por exemplo:

```
matrizAdj = [[0.   0.20 0.60 0.40]
             [0.20 0.   0.74 0.46]
             [0.60 0.74 0.   0.30]
             [0.40 0.46 0.30 0.]]
```

Para evitar que o algoritmo de ordenação execute desnecessariamente, iremos preencher a matriz apenas na parte de cima, ficando do seguinte modo:

```
matrizAdj =  [[0.    0.20  0.60  0.40]
              [0.    0.    0.74  0.46]
              [0.    0.    0.    0.30]
              [0.    0.    0.    0.]]
```

Em seguida, é utilizado o método *nonzero* da biblioteca *numpy*, que recebe como parâmetro um arranjo e retorna uma lista com os índices dos elementos que são diferentes de zero, e a partir disso, é criado uma matriz com os índices dos pesos, ficando assim:

```
indicesPesos = [[0  0  0  1  1  2]
                [1  2  3  2  3  3]]
```

Iterando a matriz indicesPesos, criaremos a seguinte lista apenas com os pesos:

listaPesos = [0.2, 0.6, 0.4, 0.74, 0.46, 0.3]

Depois, é utilizado o método *argsort* da biblioteca *numpy*, que recebe como parâmetro um arranjo (neste exemplo, o arranjo listaPesos) e retorna os índices dos elementos na ordem crescente de pesos, o retorno do *argsort* é passado para o método *unravel_index* da biblioteca *numpy*, que irá transformar os índices recebidos pelo *argsort* em uma tupla de coordenadas, e nos retornará a seguinte lista:

indicesSort = [0, 5, 2, 4, 1, 3]

Esta lista contém o índice dos pesos na ordem não decrescente da lista listaPesos.

Se acessarmos as colunas da matriz indicesPesos na ordem dos índices da lista indicesSort, teremos os pesos ordenados, por exemplo:

Coluna	**0**	**5**	**2**	**4**	**1**	**3**
Vértice	0->1	2->3	0->3	1->3	0->2	1->2

Tabela com o tempo médio de execução, em segundos, preenchendo a matriz inteira e preenchendo apenas a metade pra cima, executados no mesmo computador:

N	**Matriz preenchida inteira**	**Matriz preenchida pela metade**	**Diferença**
250	15.1648	10.5193	+4.6455
500	59.6872	42.1020	+17.5852
750	136.7666	91.1699	+45.5967
1000	243.9913	161.5400	+82.4513
1250	385.3217	254.1853	+131.1364
1500	547.4789	358.5312	+188.9477

1750	768.6047	486.3889	+282.2158
2000	986.4254	656.6163	+329.8091
TOTAL:	**3143.4428**	**2061.0531**	**+1082.3897**

Com base na tabela acima, é fácil notar a diferença, principalmente com valores altos de n, no tempo médio de execução, apenas por não ser necessário ordenar a matriz inteira, ordenando somente metade dela.

Resultados

Configuração do computador que os testes foram executados:

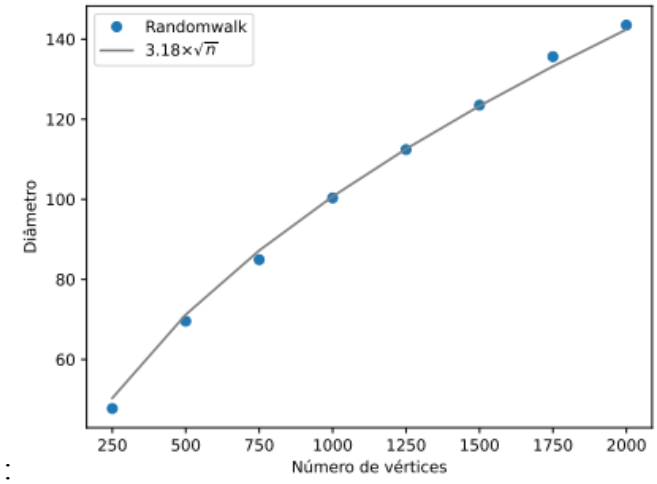
Processador: Intel i5-4460
Memória: 8GB RAM
Placa de vídeo: NVIDIA GeForce GTX 960
Sistema Operacional: Windows 10 - 64 bits

Tabela comparativa do tempo de execução médio dos algoritmos, em segundos:

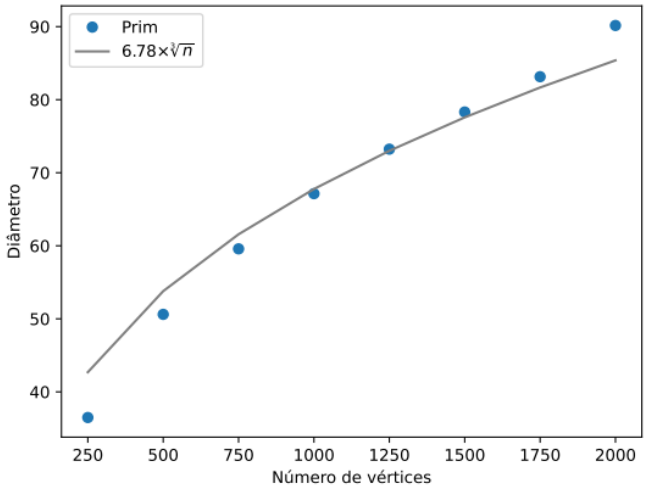
N	Random-Tree-Random-Walk	Random-Tree-Kruskal	Random-Tree-Prim
250	1.3042	10.5193	12.1845
500	2.7159	42.1020	51.3464
750	4.4944	91.1699	117.5681
1000	6.2394	161.5400	215.8055
1250	8.2102	254.1853	344.4293
1500	9.9706	358.5312	491.4315
1750	11.5606	486.3889	692.9757
2000	13.2655	656.6163	888.3649
TOTAL:	**57.7651**	**2061.0531**	**2814.1002**

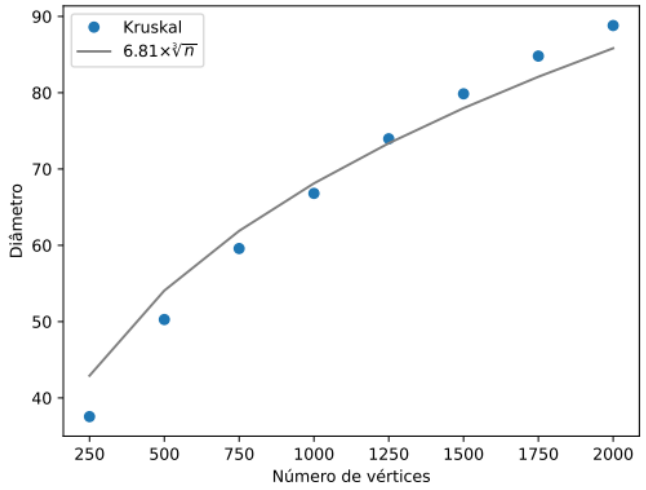
Com base na tabela acima, podemos notar uma diferença grande no tempo médio de execução entre o algoritmo Random-Tree-Random-Walk com o Random-Tree-Kruskal e Random-Tree-Prim, isso se deve pelo fato de que o primeiro algoritmo gera uma árvore aleatória, sem precisar ser mínima, enquanto os outros dois geram uma árvore aleatória mínima, escolhendo os menores pesos disponíveis.

Gráficos dos diâmetros médios:



:





Os gráficos gerados servem para ajudar na corretude dos algoritmos por meio da média do diâmetro das árvores geradas. Com base na documentação do trabalho, os resultados esperados são: O(√n), para o Random-Tree-Random-Walk, e O(∛n) para o

Random-Tree-Kruskal e Random-Tree-Prim. Conforme essas informações e os gráficos, os diâmetros esperados para cada algoritmo foi alcançado.

Descrição da experiência de desenvolvimento do trabalho:

Não houveram dificuldades em relação à implementação do trabalho no contexto geral. As aulas, os slides e o livro usado na disciplina supriram todas as dificuldades em relação a implementação e o funcionamento do código.

A maior dificuldade foi em relação a otimização dos algoritmos, em especial com o algoritmo Random-Tree-Kruskal, que no começo estava demorando muito mais que o esperado, cerca de 7500 segundos, e conseguimos, depois de muitos testes e alterações no código, reduzir para 2060 segundos, aproximadamente.

Todos os testes e alterações feitas para conseguir otimizar ao máximo os algoritmos foi a parte mais interessante e surpreendente para nós, pois aprendemos que até simples alterações que parecem não ter impacto no código, pode reduzir muito o tempo de execução do algoritmo, como foi o caso da alteração na função MST-Kruskal, que usávamos um for para fazer a iteração entre os vértices, e quando trocamos para um while, que visualmente parece ser a mesma coisa, o tempo de execução foi reduzido pela metade, mas depois de recorrer ao professor e pesquisas na internet, entendemos que o while executa menos vezes que o for, pois ele pára quando chega em V-1 vértices, sendo V o número total de vértices, enquanto o for percorria a lista inteira dos vértices.

Este trabalho foi muito importante para o nosso aprendizado, pois é mais fácil de entender como funciona na prática um algoritmo de grafos, principalmente depois de nós mesmos implementarmos, resolver os erros e entender como tudo está funcionando, dando uma visão diferente sobre o conteúdo, que apenas as aulas teóricas não conseguiria nos dar, além de melhorar nossas análises para melhorar o tempo de execução do algoritmo.

O que poderia ter sido diferente é a linguagem utilizada, que poderia ser a linguagem C. A linguagem python foi escolhida pensando na implementação dos algoritmos, pois implementar os pseudo-códigos em python geralmente é mais fácil do que em C, mas como não houveram problemas com a implementação, apenas com a otimização, seria mais interessante ter feito em C para que seja possível notar a diferença entre as duas linguagens lidando com o mesmo problema.