Alunos: Matheus Augusto Schiavon Parise - RA:107115

Gabriel de melo Osório - RA:107862

Relatório do trabalho de Grafos

Introdução

Este relatório busca esclarecer o desenvolvimento da implementação prática, em python, dos algoritmos estudados pela disciplina de Algoritmo em Grafos(6898) lecionada por Marco Aurélio Lopes Barbosa, sendo eles algoritmo de Prim, Kruskal, Diameter e Random tree random walk. Cada algoritmo gerador de árvores aleatórias, prim e kruskal, foi testado com entradas padrões, 250, 500, 750, 1000, 1250, 1500, 1750 e 2000 vezes, e seus tempos de execução estão registrados no presente documento, além das dificuldades enfrentadas durante todo o seu desenvolvimento.

Desenvolvimento

A linguagem de programação utilizada para fazer esse trabalho foi o Python 3.8, O Grafo foi representado através de uma classe a qual possui uma lista de Vértices e uma lista de Arestas, a Aresta foi representada através de uma classe que tem 3 atributos v1 um vértice, v2 outro vértice diferente de v1 e w o peso da aresta entre v1 e v2, o Vértice foi representado através de uma classe, ele possui Número um inteiro identificador, Adj uma lista de números inteiros que representam os vértices adjacentes de desse vértice, d o valor desse vértice, Pai que é o atributo que descobriu o vértice, e Cor que representa o estado do vértice, por último AdjPeso, uma lista de números float com o valor do peso da aresta entre o vértice e a lista de vértices do grafo, como os vértices são criados e inseridos na lista de vértices a partir do index, então AdjPeso[x], irá retornar o peso da aresta do vértice atual com o que possui número x, a forma de adicionar as arestas foi encapsulada no classe do Grafo.

Este algoritmo faz uso do ploy.py que foi disponibilizado para esse trabalho com o intuito de gerar o gráfico dos diâmetros automaticamente ele escreve um arquivo com os resultados dos diâmetros das função com o “nome da função”.txt e executa um comando no terminal ativa ploy.py, se ele não tiver ploy.py ele irá retornar um erro dizendo que não conseguiu abrir o arquivo, mas ainda irá gerar o txt.

Linguagem Utilizada: Python 3.8

Configuração das Máquinas utilizadas:

Máquina Gabriel:

Processador: Ryzen 7 3700x

Memória: 16GB kingston hyperx

Placa de Video: Evga gtx 1650 super 4GB

Armazenamento: Hd 1tb

Máquina Matheus:

Processador: i7-6700HQ CPU 2.60GHz

Memória: 16GB kingston hyperx

Placa de Video: Evga gtx 1060 6GB

Armazenamento: SSD 256Gb + Hd 1Tb

Entrega Diameter(25/03)

Primeiramente o grupo precisou revisar todos os conceitos da matéria pois as outras matérias do curso estavam tomando muito tempo com seus trabalhos e provas, também revisamos a linguagem python pois já fazia um tempo que os integrantes não a utilizavam. Começamos tentando criar o algoritmo requisitado para o dia 25/03 o Diameter, contudo ele precisa do BFS e de um grafo, então definimos o que é um grafo e um vértice. Após definir o grafo e a estrutura do vértice, definimos e fizemos os exemplos dos vértices e do BFS, em seguida criamos exemplos e definimos a função Diameter.

Foi descoberto um problema ao utilizar as funções da própria biblioteca do python para as listas, houve um descuido na forma de implementar o método FIFO(First in, First Out), o custo da função pop() que removia o último valor da direita era O(1) porém o custa de inserir na posição zero era N tornando meu algoritmo ineficiente perto dos padrões vistos em sala de aula. Para corrigir isso utilizamos o “deque” uma generalização de pilhas e filas (o nome é pronunciado "deck" e é uma abreviação de "fila dupla" em inglês), cujo tempo das suas funções utilizadas para o Enqueue e Dequeue, respectivamente “append” e “popleft” são constantes. Após isso, melhoramos as descrições das funções.

Entrega Random-Tree-Random-Walk(09/04)

Para a segunda entrega, é necessário enviar a função Random Tree Random Walk. Nesta função há a necessidade de criar um grafo aleatório, portanto, criamos uma função para resolver isto. Nossa maior dificuldade foi calcular os custos dentro desta função, pois a bibliografia não deixa explícito os custos de random.choice, após perguntar ao professor mais uma vez sobre os custos, foi sugerido usar randint ao invés de choise. Você pode assumir que randint é O(1). len (para list) e range também são O(1). list é O(n), dessa forma garantimos um bom tempo. No momento dos testes, o computador estava testando o Random Tree Random Walk, contudo, o tempo consumido para fazer o exemplo das árvores aleatórias de 250 até 2000 (500 vezes cada) incrementando de 250 em 250 consumiu mais de 10 minutos. Ao conferir com o professor o mesmo levou 30 segundos, ao conferir o custo de tempo de cada função com o comando “python3 -m cProfile -s tottime seuarquivo.py” percebeu-se que o custo da verificação era quadrático, no algoritmo de verificação de árvore, para verificação da quantidade de arestas cada aresta era comparada para ver se já estava na lista de arestas contudo não era necessário fazer dessa forma, se adicionarmos cada aresta de cada lista de adjacência no total teríamos 2 vezes a quantidade total de arestas, pois para um vértice x e outro vértice diferente de x chamado y seja r um aresta entre eles, r vai estar na lista de adjacências de x e y. Após corrigir a função o tempo melhorou para 1:30. No dia seguinte, percebemos que nosso algoritmo não verificava

se o grafo é conexo,gerando falsos positivos no assert. Para contornar isso, implementamos a função DFS e conferimos se existem mais de um vértice com pai nulo, pois, se existir, significa que o dfs foi executado novamente para um vértice b qualquer, que não era acessível a partir do primeiro vértice explorado,portanto, o grafo era desconexo, a representação do Infinito mudou de math.inf para int = -1.

Entrega Random-Tree-Kruskal(23/04)

Após verificar que o problema mudou e precisava de mais informações mudamos a estrutura e definição da aresta, agora existem 3 classes, Grafo que tem V uma lista de vértices e E uma lista de arestas, Vértice que está da mesma forma contudo retiramos o atributo f e mudamos o valor o pai para Union(None, Vértice) e aresta que possui v1 e v2, vértices que são adj entre si e w o peso da aresta entre eles. Para fazer Random Tree Kruskal foi fazer uma função de criação de grafo completo, um grafo completo é uma grafo onde cada vértice tem todos os outros vértices na sua lista de adjacência a criação de um grafo com tamanho n tem custo O(n) e a inserção de n-1 arestas em cada um dos n vértice também tem custo $O(n^2)$, conteúdo com um erro na forma de implementação fez que com que a função tivesse tempo $O(n^4)$, basicamente a ideia inicial foi inserir todas as combinações de arestas e retirar as repetidas,o erro foi tentar usar remove() que tem custo n para cada aresta na lista só isso daria $O(n^2)$ mas a lista tinha tamanho n^2, além dessa função na era necessário implementar 4 funções auxiliares, Make_Set, Find_Set, Link e Union, todas elas eram simples e fáceis de implementar exceto Find_Set, o pseudocódigo do livro não estava claro posteriormente o professor informou os alunos que havia um erro no pseudocódigo e o problema foi resolvido assim como todas as dúvidas teóricas sobre o algoritmo foram resolvidas por email, chegou a hora de fazer o Random Tree Krukal, Esse algoritmo foi a maior dificuldade do trabalho até então, foi necessário refazer todo o progresso em última hora por problemas de implementação, mesmo tendo enviar diversos emails a maior parte das alterações do trabalho feitas para o dia 23/04 foram descartadas, dentre elas podemos citar o construtor da classe, encapsulamento da adição de arestas, utilização da função de ordenação do próprio python, entre outras coisas, só foi mantido o essencial para o funcionamento do algoritmo a teoria em si foi compreendida sem problemas, podemos citar os como os problemas principais:

1- O problema das classes do tipo Pai e o problema do Pai Infinito
2- O suposto “ponteiro” ligando os objetos
3- Diferença enorme de tempo

Vamos começar do início a equipe havia mudado o tipo do pai de inteiro para vértice pois ele seria mais adequado para o kruskal e não afetaria drasticamente os outros algoritmos, e isso deu certo no começo, quando eu adicionei o construtor para a classe vértice, deixando todos os atributos exceto número como opcionais comecei a ter problemas com as outra funções não identificando o tipo dos atributos, erros como: “Esse elemento não tem esse atributo”, começaram a aparecer muito frequentemente, posso ter começado a mexer em python desde PAA mas ainda sou leigo na linguagem não achamos uma forma em que poderia afirmar com certeza absoluta, que a implantação de atributos opções estava boa então ela foi descartada, ainda mantive o pai como Vértice, quando eu criava um vértice por padrão o pai dele nulo quando eu usei o make_set o vértice recebia si mesmo como pai, isso era feito para Find_Set achar a raiz em teoria, infelizmente quando eu colocava ele sendo pai dele mesmo existiam 2 instâncias uma que ele era o pai dele mesmo e outra que ele tinha None como pai, essa última era sempre a raiz, isso parou de ocorrer após eu remover o construtor de classe.

No algoritmo Kruskal havia G o grafo completo e A que seria uma árvore aleatória gerada a

partir de G, quando eu fazia o make_set para os vértices de G eu adicionava o resultado do make_set em A.V, eu não criava uma nova instância do vértice, eu usava a mesma do de G, percebemos isso tarde demais pois quando eu mudava em G eu mudava em A e vice-versa.

A diferença enorme de tempo, enquanto no computador do matheus o algoritmo demora 8hrs o no do gabriel demorava 5h, como isso é possível ? só sei que daqui em diante só compro AMD, adeus intel foi bom enquanto durou.

Matheus:

```
        -991138292 function calls (-3061489041 primitive calls) in 29016.846 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
-140045258/-2210379811 8693.498   -0.000 16142.818   -0.000 Trabalho 1 Grafos RA107115_RA107862.py:345(Find_Set)
```

Gabriel

```
Traceback (most recent call last):
  File "E:\Facul\Grafos\plot.py", line 2, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
         -991160675 function calls (-3061512978 primitive calls) in 18304.409 seconds
```

Entrega Random-Tree-Prim(05/05)

A primeira parte do tempo da entrega de prim foi dedicado a melhorar o código e corrigir o algoritmo de kruskal, as correções e dicas do recebidas por e mail foram essenciais para compreender e corrigir o algoritmo, em seguida após a correção do algoritmo de kruskal, o mesmo demorou 2,78 horas no computador de gabriel e 3,7 horas no computador do matheus construiu com grafico muito melhor.

Gabriel:

```
 -2171012960 function calls (-2203535233 primitive calls) in 10383.551 seconds
```

Matheus:

```
-2180509385 function calls (-2212867214 primitive calls) in 13336.742 seconds
```

Após isso partimos para o desenvolvimento de random tree prim esse algoritmo faz a mesma coisa de kruskal porém de uma forma diferente, a estrutura dos Grafos e Arestas estavam muito boas mas nós precisávamos de acessar o peso das arestas adjacentes em tempo constante e da forma que estava isso não era possível então foi criado um atributo chamado AdjPeso para o Vertice, uma lista de números float que representa o peso da aresta desse vértice em relação aos outros vértices da lista, para um vértice v se o vértice for adjacente de v ele vai ter peso “w” na lista, caso contrário o peso será math.inf, e por fim o AdjPeso do próprio vértice é 0, dessa forma consigo obter o peso em tempo constante. O MST_Prim utiliza uma função chamada Extract_min, ela recebe uma lista de vértices e extrai da lista o item com menor valor, na nossa implementação do Extract_min decidimos não retirar o item da lista, ao invés disso utilizamos o atributo Cor do vértice para dizer se ele estava na lista ou não, se a cor for a string “false” significa que ele ainda está na lista, se for “True” significa que ele não está mais na lista e já faz parte da árvore, então no Extract_min ele só verifica os vértices cuja a cor é “False”. Durante a implementação do MST_Prim o grupo ficou com dificuldades na hora da implementação novamente, e após a aula do dia 28/04 finalmente foi descoberto o motivo por trás da dificuldade em implementar kruskal e prim, a falta de exemplos para o MST_Prim e MST_Kruskal, o grupo pensava que para os algoritmo geradores de árvores só era necessário 2 verificações, primeira

cada resultado individual deve ser uma árvore e a segunda, um conjunto de várias árvores geradas por cada ca um dos algoritmos deve se aproximar o suficiente “da curva gerado por ploy.py”, contudo essa verificação era necessária para os casos "aleatórios", faltavam a implementação de exemplos para casos concretos o qual era possível predizer a resposta, atrás do aprendizado, o algoritmo de prim foi finalizado.

Testes Realizados

Resultados para os testes realizados no computador do Gabriel, abaixo a tabela da média dos diâmetros para cada valor de n paras os 3 algoritmos:

N	250	500	750	1000	1250	1500	1750	2000
Random	46,58	69,76	85,52	99,83	110,98	123,48	133,59	142,81
Kruskal	7,38	8,12	8,47	8,75	9,05	9,24	9,47	9,60
Prim	36,87	49,64	59,21	66,65	73,31	79,15	84,66	88,83

Em seguida a tabela da média dos tempos em segundos para cada valor de n para os 3 algoritmos

N	250	500	750	1000	1250	1500	1750	2000	Total
Random	0,91	2,09	3,35	4,49	6,03	7,25	8,40	9,71	42,28
Kruskal	24,85	136,01	395,70	780,02	1296,66	1870,40	2547,97	3437,85	10489,48
Prim	26,26	122,21	331,78	652,58	1013,38	1479,39	2086,53	2964,70	8676,87

Resultados para os testes realizados no computador do Matheus, abaixo tabela da média dos diâmetros para cada valor de n paras os 3 algoritmos:

N	250	500	750	1000	1250	1500	1750	2000
Random	47,37	69,55	86,29	99,98	113,43	123,10	133,23	141,65
Kruskal	7,36	8,14	8,43	8,76	9,01	9,24	9,48	9,67
Prim	36,74	49,73	59,51	68,34	73,37	79,16	84,62	88,18

E por fim a Tabela da média dos tempos em segundos para cada valor de n para os 3 algoritmos

N	250	500	750	1000	1250	1500	1750	2000	Total
Random	1,50	4,37	5,62	7,42	8,56	10,56	12,25	13,16	63,47
Kruskal	35,69	172,78	456,55	870,59	1415,31	2100,28	2867,93	3855,68	11774,84
Prim	36,53	163,81	415,94	785,62	1274,57	2371,04	3290,83	3874,43	12212,81