

Teoria dos Grafos - COS242

Prof. Daniel Ratton e Fábio Botler

Monitora: Maria Luiza Wuillaume

Trabalho - Parte 2

Aluno: Patrick Carneiro Trindade

DRE: 119052764

1 Introdução

1.1 O programa criado

Para este trabalho foi criado um conjunto de funções e estruturas (struct) em Go a fim de explorar grafos com pesos (gerar árvores mínimas, fazer buscas e encontrar distâncias).

O código pode ser encontrado no github, no link abaixo:

<https://github.com/ptk-trindade/graph-project>

1.1.1 Estrutura do código

Abaixo estão listados os principais arquivos do código e o que eles contêm.

Arquivo	Descrição
graphWeighted.go	Funções que leem o grafo com arestas com peso
main.go	A função main deste arquivo permite que o usuário rode e interaja com o programa
linkedList.go	Métodos da lista encadeada
heapTree.go	Métodos da árvore de Heap
tests.go	Uma função criada durante a para o tópico 3 deste relatório (Estudos de caso)

1.2 A máquina utilizada

Para rodar este programa foi utilizada uma máquina Windows, com um processador Intel i7 6ª geração e 24GB de memória RAM.

Especificações do dispositivo

XPS 8900

Nome do dispositivo	CR2
Processador	Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz 4.01 GHz
RAM instalada	24,0 GB
ID do dispositivo	78246873-6873-478B-A9C5-AA91FB63F255
ID do Produto	00342-41340-30253-AAOEM
Tipo de sistema	Sistema operacional de 64 bits, processador baseado em x64

imagem 1: especificações da máquina utilizada

1.3 Os grafos analisados

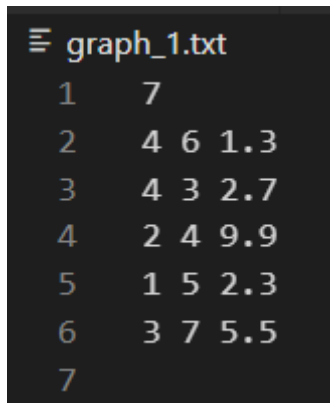
Para este relatório, foram analisados 5 grafos, que possuem as características descritas abaixo:

nome	quantidade de vértices	quantidade de arestas	peso das arestas			
			máximo	mínimo	médio	mediana
grafo 1	1000	25975	10.00	0.00	4.90	4.84
grafo 2	10000	259980	10.00	0.00	4.90	4.81
grafo 3	100000	2599967	10.00	0.00	4.91	4.82
grafo 4	1000000	25999970	10.00	0.00	4.90	4.82
grafo 5	5000000	24999996	10.00	0.00	4.50	4.17

Essas e algumas buscas pelo grafo podem ser encontradas na pasta “output_log” do repositório. Mais informações sobre os arquivos de saída na seção a seguir.

2 Input e Output (I/O)

Para input do grafo, o usuário deve criar um arquivo txt. O nome do arquivo será perguntado assim que o programa começa sua execução, e o arquivo deverá ter o seguinte formato:



```
graph_1.txt
1 7
2 4 6 1.3
3 4 3 2.7
4 2 4 9.9
5 1 5 2.3
6 3 7 5.5
7
```

imagem 2: arquivo de input “grafo.txt”

Neste, a primeira linha representa o número de vértices do grafo, e cada linha seguinte representa uma aresta com os dois vértices (inteiros) e o peso da aresta (float).

O programa manipula então os seguintes arquivos:

I/O	nome	descrição
input	grafo.txt*	Número de vértices e arestas do grafo
output	output.txt	Informações básicas sobre o grafo
output	dijkstraHeap.txt	A árvore gerada pelo Dijkstra usando árvore de Heap
output	dijkstraList.txt	A árvore gerada pelo Dijkstra usando lista encadeada
output	path.txt	O menor caminho entre 2 vértices e o custo dele
output	mst.txt	A árvore geradora mínima do grafo

*Pode ser qualquer nome, porém não é recomendado utilizar o mesmo nome de um dos arquivos de output.

3 Estudos de caso

Foram feitos vários testes de performance com o programa rodando para cada um dos 5 grafos, eles podem ser vistos abaixo:

3.1 e 3.3 Distâncias e Árvore geradora mínima (MST)

Para cada grafo foi calculada a distância entre o vértice 10 e os vértices 20, 30, 40, 50 e 60. Além da árvore geradora mínima daquele grafo.

Grafo	Tempo médio das distâncias (seg)	Distância (10, 20)	Distância (10, 30)	Distância (10, 40)	Distância (10, 50)	Distância (10, 60)	Peso da MST
grafo 1	$7.95 \cdot 10^{-4}$	1.52	1.48	1.52	1.39	1.38	220.11
grafo 2	$8.91 \cdot 10^{-3}$	2.08	1.92	1.61	1.34	1.69	2247.07
grafo 3	0.173	1.98	2.08	2.14	1.82	2.23	22218.71
grafo 4	3.23	2.46	2.43	2.33	2.36	2.61	221937.54
grafo 5	4.40	14.03	12.01	13.66	9.22	14.47	4785814.85

Esses resultados e outros resultados podem ser encontrados na pasta output_log do github. (https://github.com/ptk-trindade/graph-project/tree/main/output_log)

3.1.1 Calculando distâncias

Para encontrar a distância, iniciou-se o algoritmo de Dijkstra (usando Heap, falaremos mais sobre ele na seção 3.2) em um dos vértices e rodou-se até que o outro vértice fosse explorado. Com a árvore gerada basta ver o custo no vértice final para saber o custo do caminho, e para saber os vértices vamos subindo para os pais de cada vértice, até chegar na raiz.

3.3.1 Árvore geradora mínima

Para encontrar a MST foi utilizado o algoritmo de Prim, com a mesma árvore de Heap que usamos para o Dijkstra. A única diferença é que o custo de cada vértice não explorado não é mais a distância da origem até ele, mas a menor aresta conhecida (incide em um vértice já explorado) que incide nele.

3.2 Tempo de execução Dijkstra

A função de Dijkstra foi implementada de duas formas, uma utilizando uma lista encadeada e outra usando uma árvore de heap (binária). Para cada uma delas rodou-se o algoritmo começando em k vértices aleatórios e contabilizou-se o tempo total. Nos grafos 4 e 5, o valor entre parênteses é uma estimativa do tempo que levaria para rodar 100 vezes.

3.2.1 Lista encadeada

Neste modelo, foi mantida uma lista encadeada ordenada com base nos custos conhecidos para cada um dos vértices. Dessa forma, o próximo vértice a ser explorado é sempre o primeiro da lista, e tirá-lo tem um custo $O(1)$.

Manter essa lista ordenada, no entanto, é um processo custoso, sempre que um vértice for adicionado na lista ou tiver seu valor atualizado, reordenar a lista pode ter custo de 1 a n . O que não a torna uma boa opção para grafos densos e com pesos de arestas muito variados (o que implica em mais atualizações de valores).

3.2.1 Árvore de heap (binário)

A árvore de heap foi armazenada em um vetor na memória e são usadas as funções abaixo para encontrar os pais e filhos de cada nó da árvore.

pai	$\text{floor}((\text{índice} - 1) / 2)$
filho_esquerdo	$(\text{índice} * 2) + 1$
filho_direito	$(\text{índice} * 2) + 2$

Nela o custo de inserir ou atualizar um nó varia de 1 a $\log(n)$ (note que o limite $\log(n)$ é bem menor que o limite n da lista), porém o custo de tirar um nó é de $\log(n)$. Dessa forma o processo mais rápido dependerá do grafo que usamos. (Geralmente o heap tem um desempenho melhor, como veremos abaixo).

	Lista encadeada	Árvore de heap	k
grafo 1	169.50ms	61.48ms	100
grafo 2	33.38s	1.44s	100
grafo 3	2h 25m 43s	31.00s	100
grafo 4	14h 45m 27s (~61d)	5.45s (9m 5s)	1
grafo 5	+48h	13.95s (23m 15s)	1

4 Rede de Colaboração Pesquisadores da Computação

Caminhos encontrados entre os pesquisadores no grafo da rede de colaboração.

Pesquisador	Distância (até Dijkstra)	Caminho mínimo (com os nomes)
Alan M. Turing	-	(sem caminho)
J. B. Kruskal	3.48 (8 arestas)	Edsger W. Dijkstra > John R. Rice > Dan C. Marinescu > Howard Jay Siegel > Edwin K. P. Chong > Ness B. Shroff > R. Srikant > Albert G. Greenberg > J. B. Kruskal
Jon M. Kleinberg	2.71 (9 arestas)	Edsger W. Dijkstra > A. J. M. van Gasteren > Gerard Tel > Hans L. Bodlaender > Dimitrios M. Thilikos > Prabhakar Ragde > Avi Wigderson > Eli Upfal > Prabhakar Raghavan > Jon M. Kleinberg
Eva Tardos	2.75 (11 arestas)	Edsger W. Dijkstra > A. J. M. van Gasteren > Gerard Tel > Hans L. Bodlaender > Jan van Leeuwen > Mark H. Overmars > Micha Sharir > Haim Kaplan > Robert Endre Tarjan > Andrew V. Goldberg > Serge A. Plotkin > Eva Tardos
Daniel R. Figueiredo	2.94 (8 arestas)	Edsger W. Dijkstra > John R. Rice > Dan C. Marinescu > Chuang Lin > Bo Li > Y. Thomas Hou > Zhi-Li Zhang > Donald F. Towsley > Daniel R. Figueiredo