08/04/2024

Relatório INF05016:

Augusto Falcão Flach

a) Possíveis detalhes relevantes da implementação:

Não sou um bom programador C++, mas estou tentando aprender, então resolvi tentar usar a linguagem. Isso talvez fique evidente se olhar o código. Então apesar de a complexidade assintótica continuar correta, acredito que o programa ficou drasticamente mais lento devido a alguma falta de habilidade pontual (e à minha necessidade de ter ponteiros para todos os lados).

Eu rodei o algoritmo nos Estados Unidos partindo do vértice 1 ao 2 com os 2,3,4,5,10-Heap para testar qual seria mais rápido. O que venceu foi o 4-Heap. A partir disso, usei ele para os outros testes.

O 10-Heap foi o mais lento, apesar de ter sido o que necessitou de menos iterações de SiftUp e SiftDown. Acredito que isso se deve a “precisar” fazer a comparação entre o valor de todos os filhos de um nó para escolher o menor. Dessa forma, por exemplo, K comparações para cada Sift Down => 10*82330664 > 2 * 224355914. Então apesar de ter menos Sift Downs, possui mais comparações (para esse grafo e vértices escolhidos).

b) O ambiente de teste:

Rodei os testes em um notebook Ninkear com 16GB DDR4 (2666GHz) - i7-1165G7 @ 2.80GHz - SSD NVMe 512GB (3000Mb/s) rodando Kubuntu 23.10. O compilador pode até importar, mas entendo pouco disso. Rodei os testes pela IDE CLion (JetBrains), diretamente pelo console da IDE (em modo Debug, o que provavelmente é mais lento).

c-d) O resultado dos experimentos e análise:

NOTA: Recomendo ver os resultados pelo PDF por mais fácil visualização. Fiz no Google Sheets e o .csv ficou sem a formatação.

Grafo nos Estados Unidos:

Nodes 1 -> 2	Inserts	DeleteMins	Updates	SiftUp (iterations)	SiftDown (iterations)	Duration
10-Heap	23947347	23947347	1712835	1203067	82330664	80880ms
5-Heap	23947347	23947347	1712871	2150936	108893981	72396ms
4-Heap	23947347	23947347	1712828	2667524	124658356	67212ms
3-Heap	23947347	23947347	1712869	3679047	150927286	70406ms
2-Heap	23947347	23947347	1712809	6788424	224355914	74184ms
30-Iterations average	**Inserts**	**DeleteMins**	**Updates**	**SiftUp (iterations)**	**SiftDown (iterations)**	**Duration**
4-Heap	23947347	23947347	1717347	2694603	123436326	57383ms

Grafos gerados Aleatoriamente:

20-Iteractions Average (Random Graph) - 4-Heap	Inserts	DeleteMins	Updates	SiftUp (iterations)	SiftDown (iterations)	Duration
5000 Nodes - 0.4 Edge Chance	5001	5001	30891	11984	25473	101ms
5000 Nodes - 0.3 Edge Chance	5001	5001	29508	11560	25554	77ms
5000 Nodes - 0.2 Edge Chance	5001	5001	27368	10662	25634	57ms
5000 Nodes - 0.1 Edge Chance	5001	5001	23959	9520	25684	39ms
5000 Nodes - 0.05 Edge Chance	5001	5001	20549	8437	25679	22ms
5000 Nodes - 0.01 Edge Chance	5001	5001	12496	6020	25669	14ms
20-Iteractions Average (Random Graph) - 4-Heap	**Inserts**	**DeleteMins**	**Updates**	**SiftUp (iterations)**	**SiftDown (iterations)**	**Duration**
40000 Nodes - 0.01 Edge Chance	40001	40001	182824	73359	266133	346ms
20000 Nodes - 0.01 Edge	20001	20001	77569	32382	122767	106ms

Chance						
10000 Nodes - 0.01 Edge Chance	10001	10001	31882	14242	56471	43ms
5000 Nodes - 0.01 Edge Chance	5001	5001	12496	6020	25669	14ms
50-Iteractions Average (Random Graph) - 4-Heap	**Inserts**	**DeleteMins**	**Updates**	**SiftUp (iterations)**	**SiftDown (iterations)**	**Duration**
2000 Nodes - 0.8 Edge Chance	2001	2001	11888	4710	8911	28ms
1000 Nodes - 0.8 Edge Chance	1000	1000	5270	2096	3999	7ms
2000 Nodes - 0.6 Edge Chance	2001	2001	11358	4433	8938	20ms
1000 Nodes - 0.6 Edge Chance	1001	1001	5009	2018	4011	6ms
2000 Nodes - 0.4 Edge Chance	2001	2001	11358	4433	8938	20ms
1000 Nodes - 0.4 Edge Chance	1001	1001	4526	1836	4014	7ms

Também fiz outros testes para olhar como os tempos variam entre cada mudança de variável. Por exemplo, tentei projetar o comportamento do algoritmo para um Edge Chance “semelhante” ao dos estados unidos (#arcs / #nodes) / #nodes = 1.01718995e-7. Como é de se imaginar, isso gerou um grafo quase totalmente desconexo, e todos testes geram “inf”. Comecei a conseguir uma taxa < 50% de “inf” para EdgeChance > 0.0002.

Abaixo seguem os resultados calculados e a confirmação de que estão dentro dos limites teóricos.

Nodes 1 -> 2 (US Graph)	***n***	***m***	***r***	***i***	***d***	***u***	***K***
10-Heap	23.947.347	58.333.344	0,228	1	1	0,029	10
5-Heap	23.947.347	58.333.344	0,212	1	1	0,029	5
4-Heap	23.947.347	58.333.344	0,209	1	1	0,029	4
3-Heap	23.947.347	58.333.344	0,202	1	1	0,029	3
2-Heap	23.947.347	58.333.344	0,190	1	1	0,029	2
30-Iterations Average (US Graph)	***n***	***m***	***r***	***i***	***d***	***u***	***K***
4-Heap	23.947.347	58.333.344	0,207	1	1	0,029	4

20-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	***r***	***i***	***d***	***u***	***K***
5000 Nodes - 0.4 Edge Chance	5.000	10.000.000	0,149	1	1	0,003	4
5000 Nodes - 0.3 Edge Chance	5.000	7.500.000	0,153	1	1	0,004	4
5000 Nodes - 0.2 Edge Chance	5.000	5.000.000	0,158	1	1	0,005	4
5000 Nodes - 0.1 Edge Chance	5.000	2.500.000	0,169	1	1	0,010	4
5000 Nodes - 0.05 Edge Chance	5.000	1.250.000	0,182	1	1	0,016	4
5000 Nodes - 0.01 Edge Chance	5.000	250.000	0,229	1	1	0,050	4
20-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	***r***	***i***	***d***	***u***	***K***
40000 Nodes - 0.01 Edge Chance	40.000	16.000.000	0,169	1	1	0,011	4
20000 Nodes - 0.01 Edge Chance	20.000	4.000.000	0,185	1	1	0,019	4
10000 Nodes - 0.01 Edge Chance	10.000	1.000.000	0,205	1	1	0,032	4
5000 Nodes - 0.01 Edge Chance	5.000	250.000	0,229	1	1	0,050	4
50-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	***r***	***i***	***d***	***u***	***K***
2000 Nodes - 0.8 Edge Chance	2.000	3.200.000	0,156	1	1	0,004	4
1000 Nodes - 0.8 Edge Chance	1.000	800.000	0,168	1	1	0,007	4
2000 Nodes - 0.6 Edge Chance	2.000	2.400.000	0,159	1	1	0,005	4
1000 Nodes - 0.6 Edge Chance	1.000	600.000	0,173	1	1	0,008	4
2000 Nodes - 0.4 Edge Chance	2.000	1.600.000	0,159	1	1	0,007	4
1000 Nodes - 0.4 Edge	1.000	400.000	0,180	1	1	0,011	4

Chance							

Também fiz medições de tempo. Passei quase meio dia tentando fazer essas execuções, mas bem dificilmente elas tinham resultados consistentes. Boa parte do tempo empregado nas execuções não era rodando o algoritmo, mas sim gerando os grafos. Esse processo levava centenas de vezes mais tempo do que a execução do Dijkstra.

Mesmo rodando Dijkstra centenas de vezes, havia muita variação entre os resultados. Esses foram os conjuntos de dados mais consistentes que consegui:

Nodes 1 -> 2 (US Graph)	***n***	***m***	**Duration (ms)**	**T/((n+m)log(n))**
10-Heap	23.947.347	58.333.344	80.880	0,0001332
5-Heap	23.947.347	58.333.344	72.396	0,0001192
4-Heap	23.947.347	58.333.344	67.212	0,0001107
3-Heap	23.947.347	58.333.344	70.406	0,0001160
2-Heap	23.947.347	58.333.344	74.184	0,0001222
40-Iterations Average (US Graph)	***n***	***m***	**Duration (ms)**	**T/((n+m)log(n))**
4-Heap	23.947.347	58.333.344	57.383	0,0000945
20-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	**Duration (ms)**	**T/((n+m)log(n))**
5000 Nodes - 0.4 Edge Chance	5.000	10.000.000	101	0,0000027
5000 Nodes - 0.3 Edge Chance	5.000	7.500.000	77	0,0000028
5000 Nodes - 0.2 Edge Chance	5.000	5.000.000	57	0,0000031
5000 Nodes - 0.1 Edge Chance	5.000	2.500.000	39	0,0000042
5000 Nodes - 0.05 Edge Chance	5.000	1.250.000	22	0,0000047
5000 Nodes - 0.01 Edge Chance	5.000	250.000	14	0,0000148
40-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	**Duration (ms)**	**T/((n+m)log(n))**
40000 Nodes - 0.01 Edge Chance	40.000	16.000.000	346	0,0000047
20000 Nodes - 0.01 Edge Chance	20.000	4.000.000	106	0,0000061
10000 Nodes - 0.01 Edge Chance	10.000	1.000.000	43	0,0000106
5000 Nodes - 0.01 Edge Chance	5.000	250.000	14	0,0000148
50-Iteractions Average (Random Graph) - 4-Heap	***n***	***m***	**Duration (ms)**	**T/((n+m)log(n))**
2000 Nodes - 0.8 Edge Chance	2.000	3.200.000	28	0,0000026
1000 Nodes - 0.8 Edge Chance	1.000	800.000	7	0,0000029

2000 Nodes - 0.6 Edge Chance	2.000	2.400.000	20	0,0000025
1000 Nodes - 0.6 Edge Chance	1.000	600.000	6	0,0000033
2000 Nodes - 0.4 Edge Chance	2.000	1.600.000	20	0,0000038
1000 Nodes - 0.4 Edge Chance	1.000	400.000	7	0,0000058

O último conjunto de dados foi o menos estruturado, então mesmo mostrando os mesmos resultados, têm uma pior visualização.

Apesar disso, gostaria de apontar que o cálculo de **T/((n+m)log(n))** gerou valores consistentemente decrescentes. Não foram realizadas as contas para conferir se eles se aproximam corretamente de algum comportamento teórico, portanto deixaremos como trabalhos futuros calcular corretamente seu comportamento esperado e empírico para afirmar diretamente se os mesmos se aproximam de um comportamento 1/X ou de funções logarítmicas inversas.

Conclusão

Foi feita a construção de um algoritmo de Dijkstra e de uma fila de prioridade K-Heap na linguagem C++. A partir disso, foi feito o teste de processamento com base em diversas entradas de exemplo do desafio DIMACS 9, e por fim foi feita a execução do algoritmo com entradas geradas randomicamente para a constatação de seus comportamentos empíricos, e de que os mesmos respeitam os limites assintóticos teóricos.