Capítulo 7

Resultados Experimentais

Atualmente, há um grande interesse em trabalhos relacionados à análise experimental de algoritmos. Em particular, no caso do algoritmo de Dijkstra, podemos citar os artigos de Cherkassky, Goldberg, Radzik e Silverstein [6, 21, 7]. Para algoritmos baseados em "decomposição hierárquica", como o de Thorup, podemos citar o artigo de Petti, Ramachandran e Sridhar [32].

O interesse em experimentação é devido ao reconhecimento de que os resultados teóricos, freqüentemente, não trazem informações referentes ao desempenho do algoritmo na prática. Porém, o campo da análise experimental é repleto de armadilhas, como comentado por Johnson [25]. Muitas vezes, a implementação do algoritmo é a parte mais simples do experimento. A parte difícil é usar, com sucesso, a implementação para produzir resultados de pesquisa significativos.

Segundo Johnson [25], pode-se dizer que existem quatro motivos básicos que levam a realizar um trabalho de implementação de um algoritmo:

(1) Para usar o código em uma aplicação particular, cujo propósito é descrever o impacto do algoritmo em um certo contexto;

(2) Para proporcionar evidências da superioridade de um algoritmo;

(3) Para melhor compreensão dos pontos fortes, fracos e do desempenho das operações algorítmicas na prática; e

(4) Para produzir conjecturas sobre o comportamento do algoritmo no caso-médio sob distribuições específicas de instâncias onde a análise probabilística direta é muito difícil.

Nesta dissertação estamos mais interessados no motivo (3).

7.1 Ambiente experimental

A plataforma utilizada nos experimentos é um PC rodando Linux RedHat 7.1 com um processador Pentium III de 733 MHz e 512MB de memória RAM. Os códigos estão compilados com o gcc versão 2.96 e opção de otimização 03.

Na implementação, as instâncias são obtidas utilizando-se os geradores de grafos aleatórios disponibilizados pelo "The Fifth DIMACS Challenge"[1] que estão acessíveis no endereço

`ftp://cs.amherst.edu/pub/dimacs`.

Essas instâncias são convertidas para o formato do SGB (seção 1.7), que é a plataforma utilizada nas implementações desta dissertação. Essa conversão é feita utilizando-se um "driver" também disponível pelo DIMACS, porém com algumas modificações. Em todos os testes, para cada valor de n e m, onde n é o número de vértices e m o número de arcos, foi obtido a média entre cinco valores. Não é utilizado o gerador do SGB, devido ao fato que o tempo gasto para se gerar um grafo com aproximadamente 2000 vértices e 1×10^6 arcos é em torno de 4 horas e o espaço ocupado por esse grafo em disco é de aproximadamente 50MB. Além disso, estamos interessados em grafos de dimensões um pouco maiores que este.

As implementações comparadas neste experimento são heap (Heap), D-heap (Dheap), fibonacci heap (Fheap), bucket heap (Bheap), radix heap (Rheap) e o algoritmo de thorup (Thorup).

A estimativa do tempo é calculada utilizando-se:

```
#include <time.h>
clock_t start, end;
double time;
start = clock();
/* implementação */
end = clock();
time = ((double) (end - start)) / CLOCKS_PER_SEC;
```

O tempo estimado para a construção da decomposição hierárquica (AGM), realizada pela implementação Thorup, é computado separadamente.

A estimativa da memória consumida por uma implementação é calculada utilizando-se o programa *memtime* da seguinte maneira:

memtime implementação

[1] O tópico era "Priority Queues" e "Dictionaries".

Esse programa está disponível em `http://www.update.uu.se/~johanb/memtime` e foi escolhido devido a facilidade na sua manipulação.

Na estimativa de memória, não foi possível computar separadamente a quantidade de memória gasta na construção da decomposição hierárquica, pois o programa *memtime* informa qual foi o máximo de memória utilizada pelo processo que executa a implementação Thorup.

Nas próximas seções, serão apresentados os desempenhos das implementações em grafos produzidos pelos geradores SPRAND, SPGRID e SPBAD.

7.2 SPRAND

SPRAND gera grafos aleatórios conexos. Inicialmente é criado um ciclo hamiltoniano, e depois são adicionados os demais arcos aleatoriamente. Os parâmetros de entrada para o gerador são especificados da seguinte maneira:

`sprand n m seed [parâmetros opcionais]`, onde

`n` é o número de vértices,
`m` é o número de arcos e
`seed` é a semente do gerador de números aleatórios.

Toda vez que o gerador é executado com os mesmos parâmetros, ele gera o mesmo grafo. Por default, os arcos pertencentes ao ciclo tem comprimento 1 e os demais têm comprimento em [0..10000].

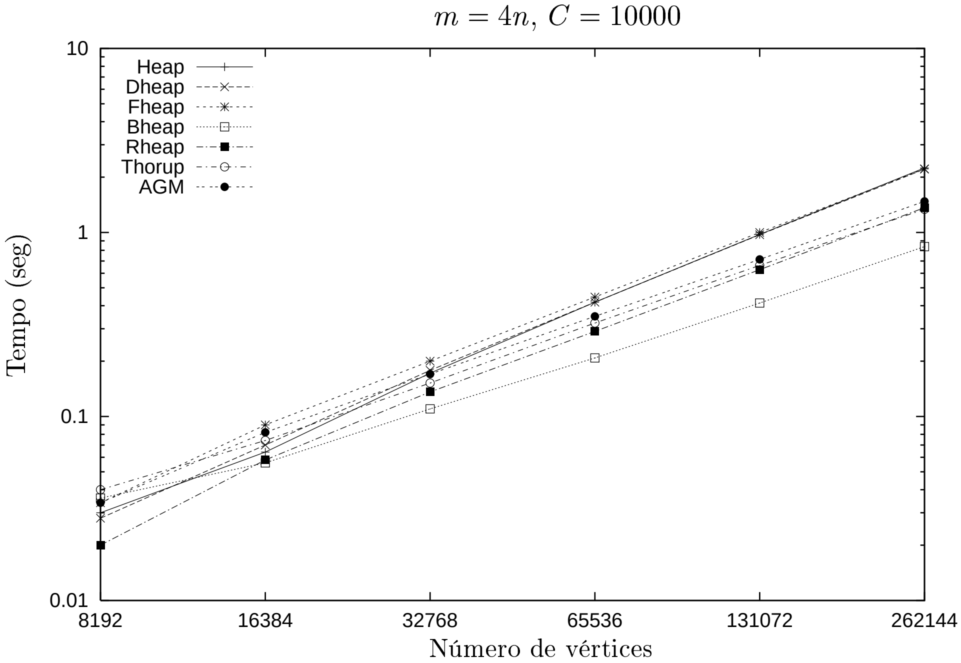
Os experimentos realizados com os grafos gerados por SPRAND se dividem em duas classes: grafos esparsos e grafos densos.

Grafos esparsos gerados por SPRAND

Os grafos desta classe possuem o número de arcos 4 vezes maior que o número de vértices, ou seja, $m = 4n$. A figura 7.1 exibe o tempo gasto pelos algoritmos e a figura 7.2 o quanto de memória foi consumida. Nestes experimentos, n varia de 8192 a 262144, e os comprimentos dos arcos estão em [1..10000].

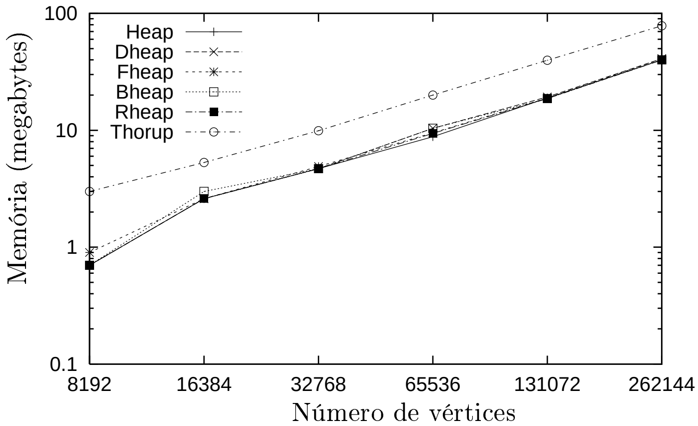
Pode-se observar que o Bheap teve o melhor desempenho. A implementação Thorup foi melhor do que as implementações Heap, Dheap e Fheap, lembrando que o tempo necessário para construir a decomposição hierárquica é computado separadamente. Além disso, nota-se que a correlação entre o tempo e o número de vértices é praticamente linear. A memória consumida por Thorup foi maior que a das demais implementações, mas apenas por um fator constante.

Na figura 7.3 é analisada a sensibilidade dos algoritmos em relação a C, que é o maior comprimento de um arco. Nota-se que Thorup teve um comportamento bem estável. As implementações Heap, Dheap e Fheap foram a mais sensíveis à variação de C.



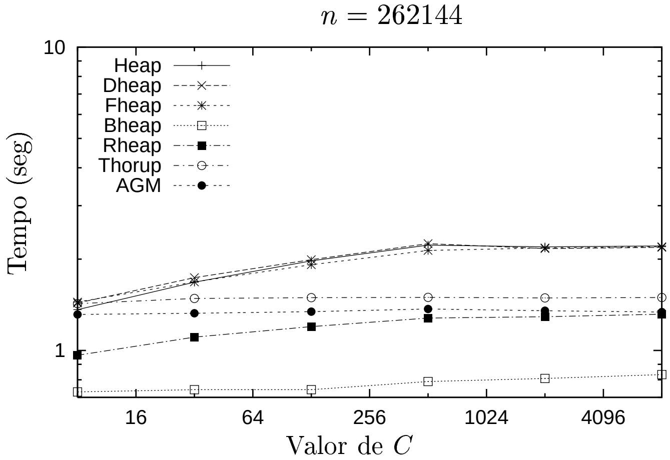
Tempos (seg)							
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8192	0.030	0.028	0.034	0.036	0.020	0.034	0.040
16384	0.064	0.070	0.090	0.056	0.058	0.082	0.074
32768	0.172	0.178	0.200	0.110	0.136	0.170	0.152
65536	0.418	0.418	0.446	0.208	0.290	0.350	0.322
131072	0.976	0.972	1.000	0.414	0.628	0.714	0.662
262144	2.242	2.210	2.218	0.838	1.364	1.474	1.336

Figura 7.1: Número de vértices em relação ao tempo em grafos esparsos gerados por SPRAND.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
8192	0.7	0.7	0.9	0.7	0.7	3.0
16384	2.6	2.6	2.6	3.0	2.6	5.3
32768	4.7	4.7	4.9	4.7	4.7	9.9
65536	8.8	10.4	9.5	10.4	9.4	20.0
131072	18.9	19.3	18.9	18.7	18.7	39.7
262144	40.1	41.0	40.9	40.1	39.9	78.4

Figura 7.2: Memória utilizada quando executado em grafos esparsos gerados por SPRAND.



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	1.362	1.434	1.444	0.730	0.964	1.426	1.314
32	1.678	1.738	1.682	0.742	1.106	1.484	1.326
128	1.974	1.992	1.916	0.742	1.198	1.492	1.342
512	2.224	2.250	2.136	0.790	1.278	1.496	1.370
2048	2.194	2.166	2.178	0.808	1.292	1.490	1.352
8192	2.210	2.186	2.198	0.832	1.316	1.494	1.338

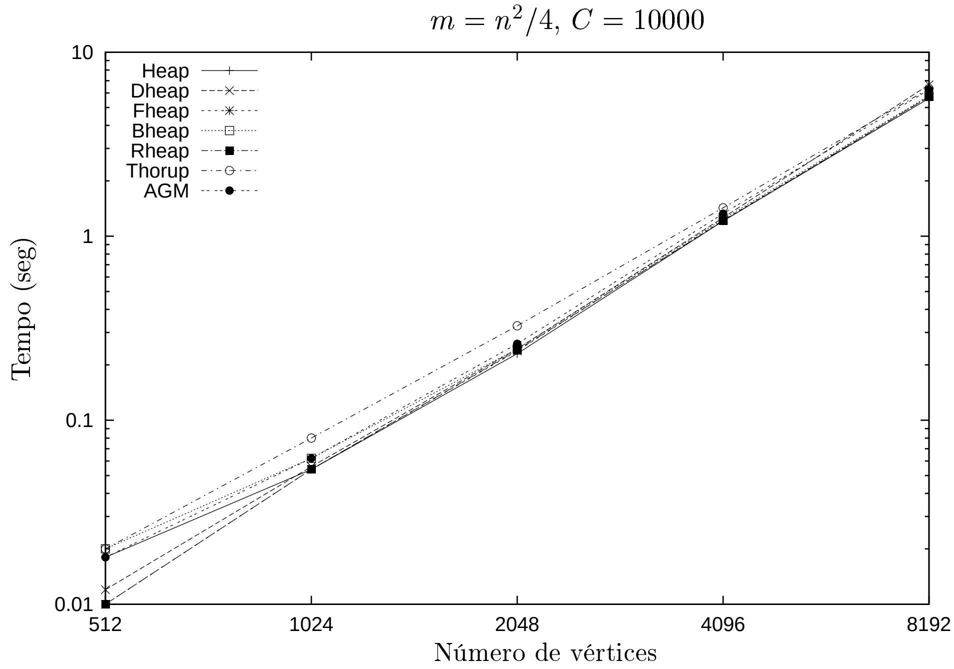
Figura 7.3: Valor de C em relação ao tempo em grafos esparsos gerados por SPRAND.

Grafos densos gerados por SPRAND

Os grafos desta classe possuem $n^2/4$ arcos. A figura 7.4 exibe o tempo gasto pelos algoritmos, e a figura 7.5 o quanto de memória foi consumida. Neste experimentos, n varia de 512 a 8192, e os comprimentos dos arcos são inteiros em [1..10000].

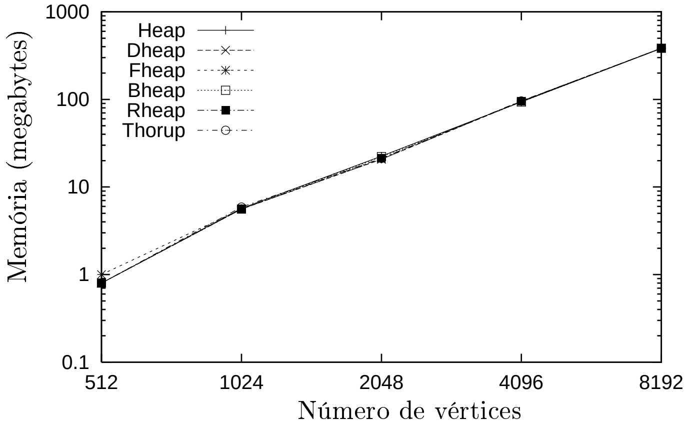
Observa-se que o desempenho das implementações foram muito parecidos, tanto no tempo, quanto no uso da memória.

Na figura 7.6 é analisada a sensibilidade do algoritmo em relação a C, que é o comprimento do maior arco. Nota-se que a construção da decomposição hierárquica (AGM) teve uma pequena variação.



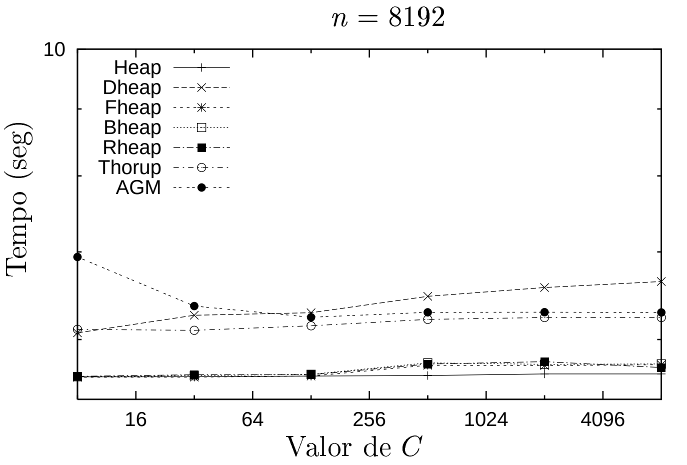
Tempos (seg)							
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.018	0.012	0.010	0.020	0.010	0.018	0.020
1024	0.054	0.056	0.054	0.062	0.054	0.062	0.080
2048	0.230	0.244	0.238	0.244	0.240	0.260	0.326
4096	1.210	1.262	1.220	1.238	1.210	1.326	1.426
8192	5.618	6.658	5.728	5.796	5.722	6.262	6.346

Figura 7.4: Número de vértices em relação ao tempo em grafos densos gerados por SPRAND.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	0.8	0.8	1.0	0.8	0.8	0.8
1024	5.7	5.6	5.6	5.6	5.6	5.9
2048	22.5	20.8	20.8	22.2	21.3	21.2
4096	93.8	94.9	95.4	93.4	95.9	96.0
8192	382.0	385.5	385.3	384.0	382.1	383.9

Figura 7.5: Memória utilizada quando executado em grafos densos gerados por SPRAND.



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	5.618	6.070	5.614	5.620	5.624	6.108	6.938
32	5.620	6.262	5.616	5.630	5.640	6.098	6.364
128	5.626	6.290	5.626	5.644	5.640	6.146	6.240
512	5.632	6.474	5.734	5.758	5.746	6.216	6.294
2048	5.648	6.574	5.732	5.744	5.770	6.236	6.296
8192	5.648	6.644	5.740	5.748	5.712	6.236	6.292

Figura 7.6: Valor de C em relação ao tempo em grafos densos gerados por SPRAND.

7.3 SPGRID

SPGRID gera grafos em forma de grade. Em casos simples, todos os arcos pertencem à grade. Contudo, é possível adicionar mais arcos, tornando a instância mais complicada. Os parâmetros de entrada para o gerador são especificados da seguinte maneira:

`spgrid X Y seed [parâmetros opcionais]`, onde

`X` é o tamanho horizontal da grade;

`Y` é o tamanho vertical da grade; e

`seed` é a semente do gerador de números aleatórios.

Um exemplo de um grafo gerado por SPGRID pode ser visto na figura 7.7.

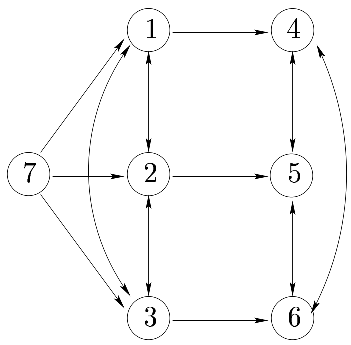


Figura 7.7: Grafo gerado com `spgrid 2 3 1`. Nesse exemplo, os comprimentos dos arcos foram omitidos. O vértice fonte é o $\mathtt{X} \times \mathtt{Y} + 1$.

Toda vez que o gerador é executado com os mesmos parâmetros, ele gera o mesmo grafo. Por default, os comprimento dos arcos verticais estão em [0..100] e dos arcos horizontais estão em [0..10000].

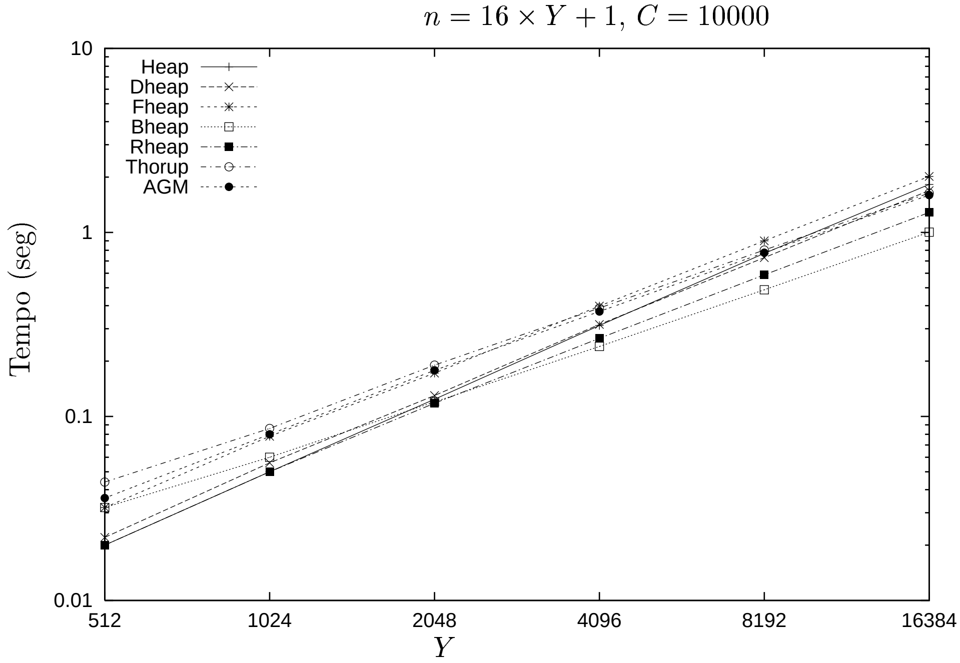
Os experimentos realizados com os grafos gerados por SPGRID se dividem em duas classes: (wide) grafos grade onde o comprimento horizontal é fixo e o vertical cresce com o tamanho da entrada e (long) grafos grade onde o comprimento vertical é fixo e o horizotal cresce com o tamanho da entrada. Observe que o número de arcos com ponta inicial no vértice origem é sempre determinado pelo valor de Y.

Grafos wide gerados com SPGRID

O número fixado de vértices na horizontal é 16, ou seja, $X = 16$. A figura 7.8 exibe o tempo gasto pelos algoritmos e a figura 7.9 o quanto de memória foi consumida. Nestes experimentos, o valor de Y varia de 512 a 16384, e os comprimentos dos arcos são inteiros em [1..10000]. O número de vértices do grafo é $X \times Y + 1$ e o número de arcos é $6(X \times Y)$.

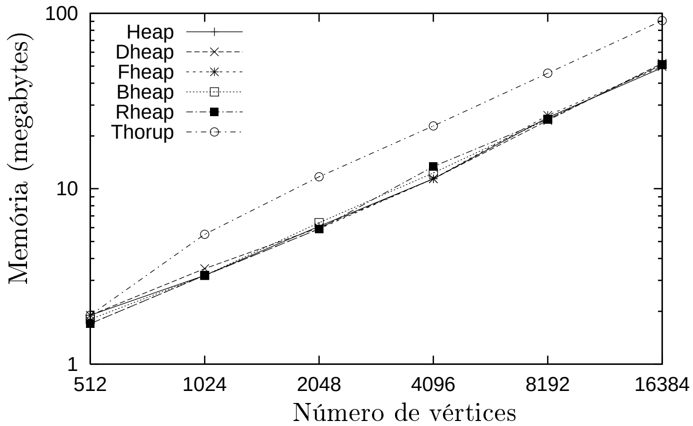
Observa-se que, para valores grandes de Y, o Bheap teve o melhor desempenho, enquanto o Fheap teve o pior. Em relação a memória consumida, novamente Thorup foi o maior consumidor.

Na figura 7.10 é analisada a sensibilidade do algoritmo em relação a C, que é o comprimento do maior arco. As implementações Heap e Dheap foram as mais sensíveis em relação à variação de C.



Tempos (seg)							
Y	heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.02	0.022	0.032	0.032	0.02	0.036	0.044
1024	0.05	0.056	0.078	0.06	0.05	0.08	0.086
2048	0.124	0.13	0.172	0.12	0.118	0.178	0.19
4096	0.312	0.316	0.398	0.24	0.266	0.372	0.388
8192	0.77	0.73	0.9	0.488	0.588	0.776	0.802
16384	1.822	1.69	2.016	1.004	1.284	1.596	1.636

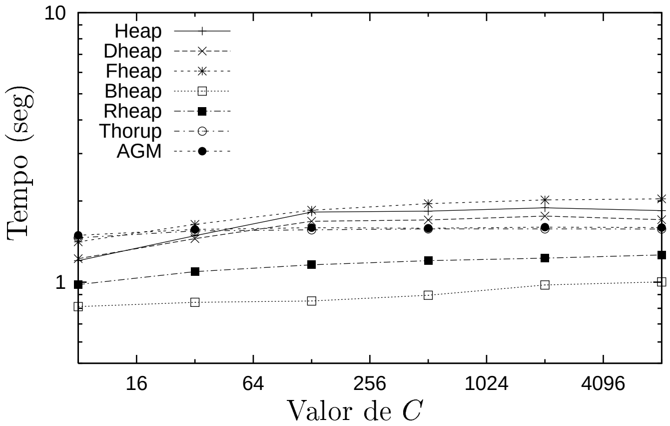
Figura 7.8: Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$.



Memória (megabytes)						
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	1.9	1.9	1.7	1.8	1.7	1.9
1024	3.2	3.5	3.2	3.2	3.2	5.5
2048	6.1	6.0	5.9	6.4	5.9	11.7
4096	11.4	11.4	11.4	12.3	13.4	22.8
8192	25.5	24.5	26.2	24.9	24.9	45.6
16384	49.0	51.9	49.8	51.0	50.9	90.9

Figura 7.9: Memória utilizada quando executado em grafos grade gerados por SPGRID com $X = 16$.

$n = 16 \times 16384 + 1$



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	1.202	1.222	1.412	0.812	0.980	1.460	1.492
32	1.488	1.450	1.638	0.842	1.094	1.550	1.570
128	1.820	1.682	1.850	0.852	1.160	1.564	1.592
512	1.834	1.702	1.956	0.894	1.202	1.576	1.586
2048	1.888	1.758	2.018	0.976	1.228	1.576	1.600
8192	1.844	1.706	2.038	1.002	1.262	1.574	1.594

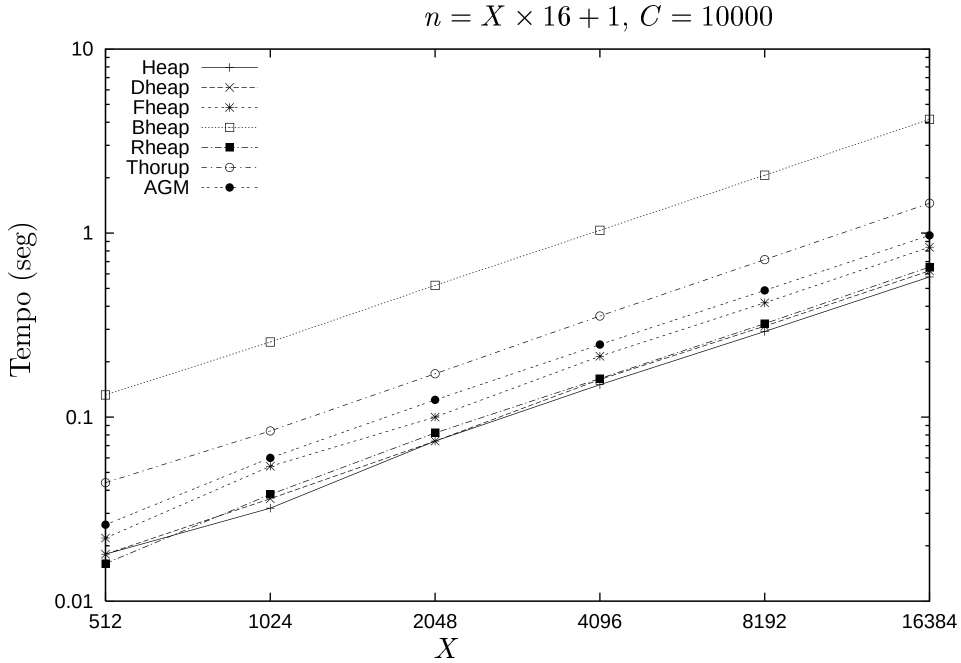
Figura 7.10: Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $X = 16$.

Grafos long gerados com SPGRID

O número fixado de vértices na vertical é 16, ou seja, $Y = 16$. Pode-se perceber que os grafos gerados dessa maneira tendem a ter poucos vértices na fila de prioridade. A figura 7.11 exibe o tempo gasto pelos algoritmos, e a figura 7.12 o quanto de memória foi consumida. Nestes experimentos, X varia de 512 a 16384, e os comprimentos dos arcos são inteiros em [1..10000]. O número de vértices do grafo é $X \times Y + 1$ e o número de arcos é $6(X \times Y)$.

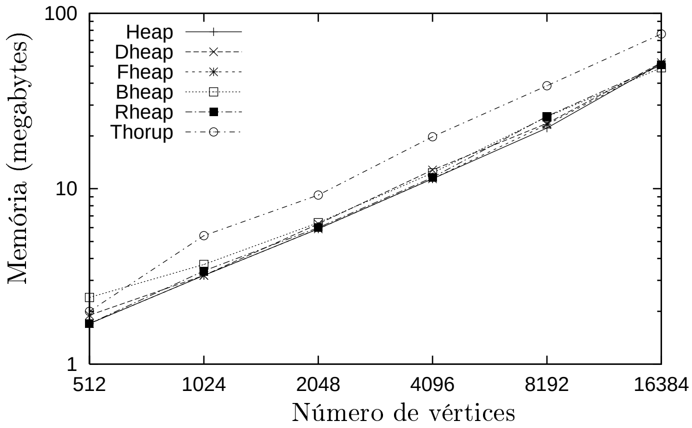
Nestes testes, a implementação Heap foi a vencedora, e a implementação Bheap teve um desempenho bem pior que as demais implementações. Mais uma vez, a implementação Thorup foi a que consumiu mais memória.

Na figura 7.13 é analisada a sensibilidade do algoritmo em relação a C, que é o maior comprimento de um arco. A implementação Bheap foi extremamente sensível. Pôde-se notar que a medida que C cresce, o tempo gasto por Bheap piora exponencialmente.



Tempos (seg)							
X	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.018	0.018	0.022	0.132	0.016	0.026	0.044
1024	0.032	0.036	0.054	0.256	0.038	0.060	0.084
2048	0.074	0.074	0.100	0.520	0.082	0.124	0.172
4096	0.150	0.160	0.214	1.036	0.162	0.248	0.354
8192	0.292	0.312	0.418	2.064	0.322	0.488	0.718
16384	0.578	0.624	0.838	4.152	0.654	0.972	1.454

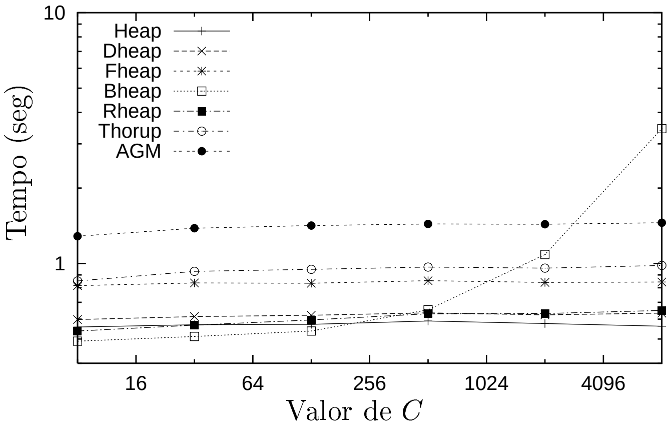
Figura 7.11: Número de vértices em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$.



Memória (megabytes)						
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	1.7	1.9	1.7	2.4	1.7	2.0
1024	3.2	3.2	3.2	3.7	3.4	5.4
2048	5.9	6.3	5.9	6.4	6.0	9.2
4096	11.4	12.8	11.4	12.3	11.6	19.8
8192	22.2	23.6	23.2	25.7	25.9	38.7
16384	51.9	52.4	51.0	49.0	50.7	76.4

Figura 7.12: Memória utilizada quando executado em grafos grade gerados por SPGRID com $Y = 16$.

$n = 16384 \times 16 + 1$



Tempos (seg)							
C	Heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
8	0.558	0.598	0.816	0.490	0.538	0.852	1.284
32	0.570	0.614	0.836	0.512	0.568	0.930	1.382
128	0.572	0.622	0.834	0.538	0.596	0.948	1.416
512	0.590	0.636	0.854	0.654	0.630	0.968	1.438
2048	0.576	0.624	0.840	1.086	0.632	0.958	1.434
8192	0.562	0.634	0.844	3.446	0.650	0.982	1.454

Figura 7.13: Valor de C em relação ao tempo em grafos grade gerados por SPGRID com $Y = 16$.

7.4 SPBAD

Assim como SPRAND, SPBAD gera grafos conexos. Porém, o comprimento dos arcos no ciclo hamiltoniano é sempre 1, e o comprimento dos outros arcos é calculado de forma a produzir um número maior de operações decrease-key (seção 1.4). A forma de se usar o gerador é a seguinte:

`spbad n d [parâmetros opcionais]`,

onde `n` é o número de vértices (deve ser dois ou mais) e `d` é o grau de saída de cada vértice (deve ser um ou mais). Um exemplo de um grafo desse tipo pode ser visto na figura 7.14.

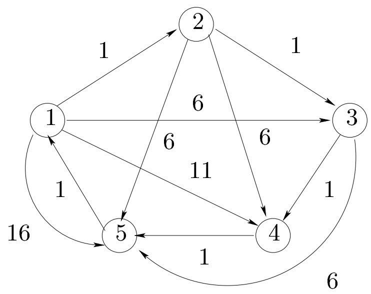
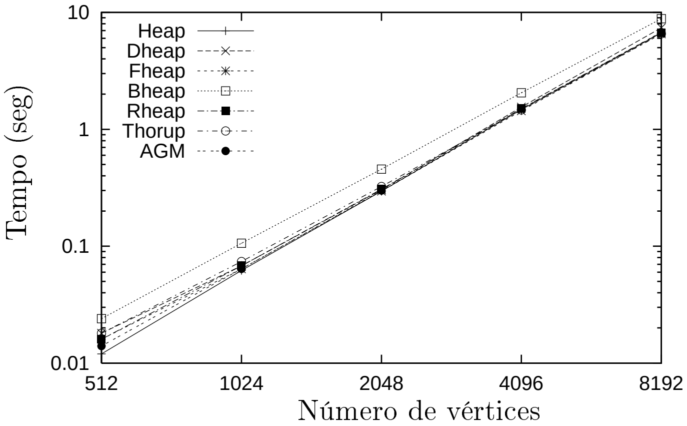


Figura 7.14: Grafo gerado com `spbad 5 4`. Nesse exemplo, alguns arcos foram omitidos, para simplificar a visualização.

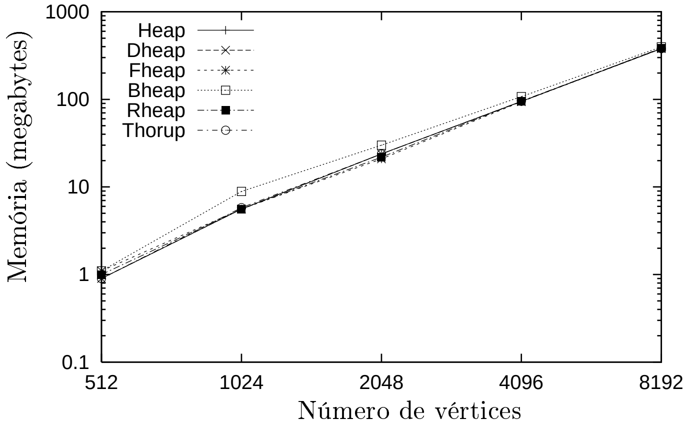
A figura 7.15 exibe o tempo gasto pelos algoritmos e a figura 7.16 o quanto de memória foi consumida. Nestes experimentos, n varia de 512 a 8192 e o número de arcos é $n^2/4$.

Nota-se que a implementação Bheap esteve pior em relação ao tempo e a memória consumida. As demais implementações tiveram desempenhos praticamente idênticos.



Tempos (seg)							
n	heap	Dheap	Fheap	Bheap	Rheap	Thorup	AGM
512	0.012	0.018	0.016	0.024	0.016	0.014	0.018
1024	0.062	0.068	0.064	0.106	0.068	0.064	0.074
2048	0.296	0.304	0.296	0.456	0.308	0.300	0.324
4096	1.476	1.550	1.440	2.056	1.504	1.496	1.466
8192	6.694	7.388	6.544	8.826	6.720	6.768	6.572

Figura 7.15: Número de vértices em relação ao tempo em grafos gerados por SPBAD.



Memória (megabytes)						
n	Heap	Dheap	Fheap	Bheap	Rheap	Thorup
512	0.9	0.9	1.1	1.1	1.0	0.9
1024	5.6	5.6	5.6	8.9	5.6	5.8
2048	23.9	23.8	20.9	30.0	21.9	23.8
4096	94.7	94.4	94.0	107.4	95.3	95.1
8192	380.6	384.7	380.6	397.0	383.3	380.5

Figura 7.16: Memória utilizada quando executado em grafos gerados por SPBAD.