ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
UNIVERSIDADE DE SÃO PAULO

Documentação sobre o Exercício-Programa 2 proposto para a avaliação da matéria:
ACH2024 - Algoritmos e Estruturas de Dados II
Professor Alexandre da Silva Freire

Caminhos mínimos de única origem

Autores
Guilherme Umemura (9353592)
Karina Duran Munhos (11295911)

São Paulo/2021

Sobre os testes

Para a elaboração dos testes de desempenho foi desenvolvido um algoritmo, contido na classe *Main.java* que se arquiteta da seguinte maneira: foram criadas duas *arrays*, uma de *doubles* que armazena um total de 14 elementos, sendo eles valores que variam de 0.05 até 1, que representam a densidade dos digrafos; e uma outra de inteiros que contêm valores que vão de 100 até 40.000.000 que representam a quantidade de entradas.

Foram desenvolvidos três métodos para resolver o problema de Caminhos Mínimos de Única Origem: Bellman-Ford para ambos os tipos de grafos, DAGmin para DAGs, e Dijkstra, que processa ambos os grafos. Esses são executados dentro de um comando *for* que itera pelos elementos do *array* de valores de entrada. Esse *for*, por sua vez, está dentro de um outro *for* que itera pelos elementos da *array* que armazena os valores de densidade.

O pseudocódigo, simplificado e representando apenas o algoritmo de testes, pode ser descrito da seguinte maneira:

```
For densidade in densidades{
        For entrada in entradas{
                dag = new Digrafo(densidade, entrada)
                digrafoAleatorio = new Digrafo(densidade, entrada)

                //Primeira parte
                BellmanFord (digrafoAleatorio)
                Dijkstra(digrafoAleatorio)

                //Segunda parte
                BellmanFord (dag)
                Dijkstra(dag)
                DAGmin(dag)
        }
}
```

Considerações sobre o algoritmo de BellmanFord

Como é sabido, "o algoritmo [de BellmanFord] parece muito simples, mas é lento e a análise da sua correção é difícil"[1], restando o fato comprovado nos testes que mais a frente serão apresentados. Por força dessa limitação, os testes desse algoritmo foram limitados somente às entradas menores ou iguais a 500.000 (mas com todas as densidades) sendo o intuito dessa limitação reduzir o tempo empregado pela máquina nos testes e permitir a condução dos experimentos uma vez que, apesar de ter sido objeto de menos testes, esses já são suficientes para traçar um paralelo entre os similares.

Isso dito, podemos atualizar o pseudocódigo acima descrito ao seguinte:

```
For densidade in densidades{
        For entrada in entradas{
                dag = new Digrafo(densidade, entrada)
                digrafoAleatorio = new Digrafo(densidade, entrada)

                //Primeira parte
                If (entrada <= 500000)
                        BellmanFord (digrafoAleatorio)
                Dijkstra(digrafoAleatorio)

                //Segunda parte
                If (entrada <= 500000)
                        BellmanFord (dag)
                Dijkstra(dag)
                DAGmin(dag)
        }
}
```

A *array* de densidades tem quatorze elementos, mas serão apresentados gráficos de apenas oito pois seus comportamentos são muito semelhantes; e a de entradas tem treze, sendo sete deles menores ou iguais a 500.000 (que incluem a execução dos métodos de BellmanFord), conclui-se que foram realizados um total de 742 testes, sendo 490 deles que incluem os resultados dos métodos de BellmanFord.

[1] Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bellman-ford.html <acessado em 31/07/2021>; página web da faculdade IME que discute sobe os algoritmos disponíveis para a solução de CPT (*cheapest-paths tree*); Nele o algoritmo BellmanFord é o primeiro a ser analisado pela sua simplicidade de implementação em detrimento de performance.

Dados do computador usado

Processador: Intel Core i5 – 7200U 2.50GHz – 2.70GHz

Memória 8,00 GB tipo DDR4

Detalhes da execução dos testes

Os testes estão contidos na classe Main. A execução dos testes é feita sob os seguintes comandos:

>> javac Main.java
>> java -Xms512m -Xmx5120m Main

Ou seja, estamos fornecendo à máquina virtual quase 5GB de memória RAM. Com esses recursos verificamos que o número máximo de entradas que o programa consegue lidar é de 40 milhões. Mais que isso é verificado o erro de *java.lang.OutOfMemoryError: Java heap space*.

Para sua execução, foram usados os seguintes valores para a densidade e entrada, respectivamente:

Densidades = {0.05, 0.1, 0.15, 0.20,0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 1}
Entradas = {100, 500, 1000, 5000, 10000, 100000, 500000, 2000000, 5000000, 10000000, 20000000, 30000000, 40000000}

Resultados observados e sua apresentação

Primeiramente serão apresentados dois conjuntos de gráficos: um para DAGmin e Dijkstra, e um segundo apenas para Bellman-Ford. Coloca-los todos juntos na mesma tabela está fora de questão haja visto a diferença abismal entre Bellman-Ford e os demais, que faz com que esses simplesmente desaparecem do gráfico.

Cada grupo trará dois gráficos distintos. Um deles apresentara as variações quando *n* (entradas) for constante e o outro quando *p* (densidade). Tais valores foram selecionados de forma aleatória.

O intuito de cada um desses é trazer uma análise sobre a variação da performance em diferentes situações e encontrar, se possível, o tipo de grafo ao qual o algoritmo melhor desempenha. Em cada eixo, caso haja valores em parêntesis, esse representará a quantidade de vértices do grafo que foi analisado pelo algoritmo (calculado por método apresentado pelo professor em vídeo do youtube).

Para representar as cinco operações analisadas serão usadas as seguintes legendas:

- DAGmin(DAG): representando o algoritmo de DAGmin que analisa um DAG
- BellmanFord(DAG): representando o algoritmo de BellmanFord que analisa um DAG
- Dijkstra(DAG): representando o algoritmo de Dijkstra que analisa um DAG
- BellmanFord(dig): BellmanFord que analisa digrafos aleatórios
- Dijkstra(dig): Dijkstra que analisa digrafos aleatórios

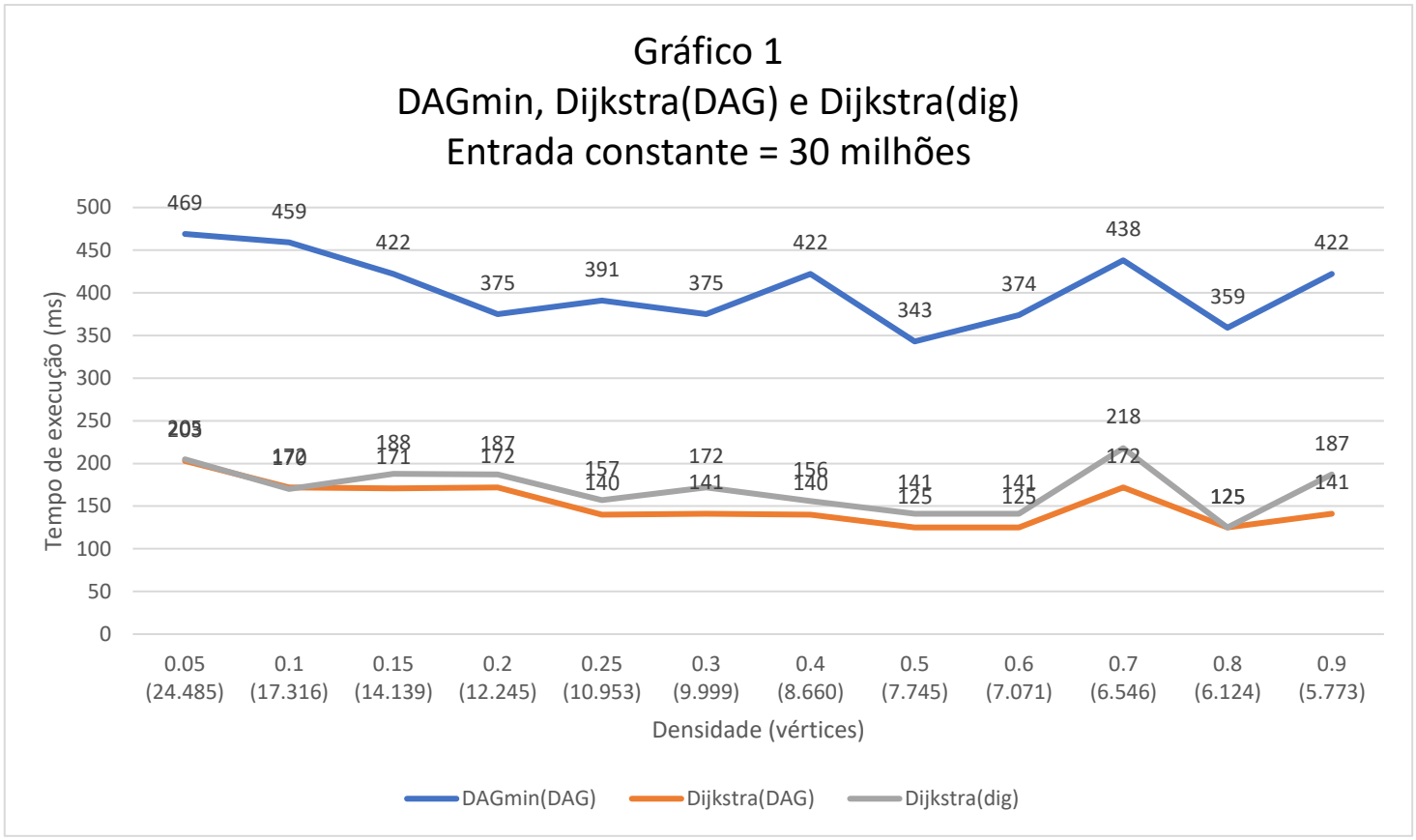


Gráfico 1
DAGmin, Dijkstra(DAG) e Dijkstra(dig)
Entrada constante = 30 milhões
Tempo de execução (ms)
Densidade (vértices)
0.05 (24.485)
0.1 (17.316)
0.15 (14.139)
0.2 (12.245)
0.25 (10.953)
0.3 (9.999)
0.4 (8.660)
0.5 (7.745)
0.6 (7.071)
0.7 (6.546)
0.8 (6.124)
0.9 (5.773)
DAGmin(DAG)
Dijkstra(DAG)
Dijkstra(dig)

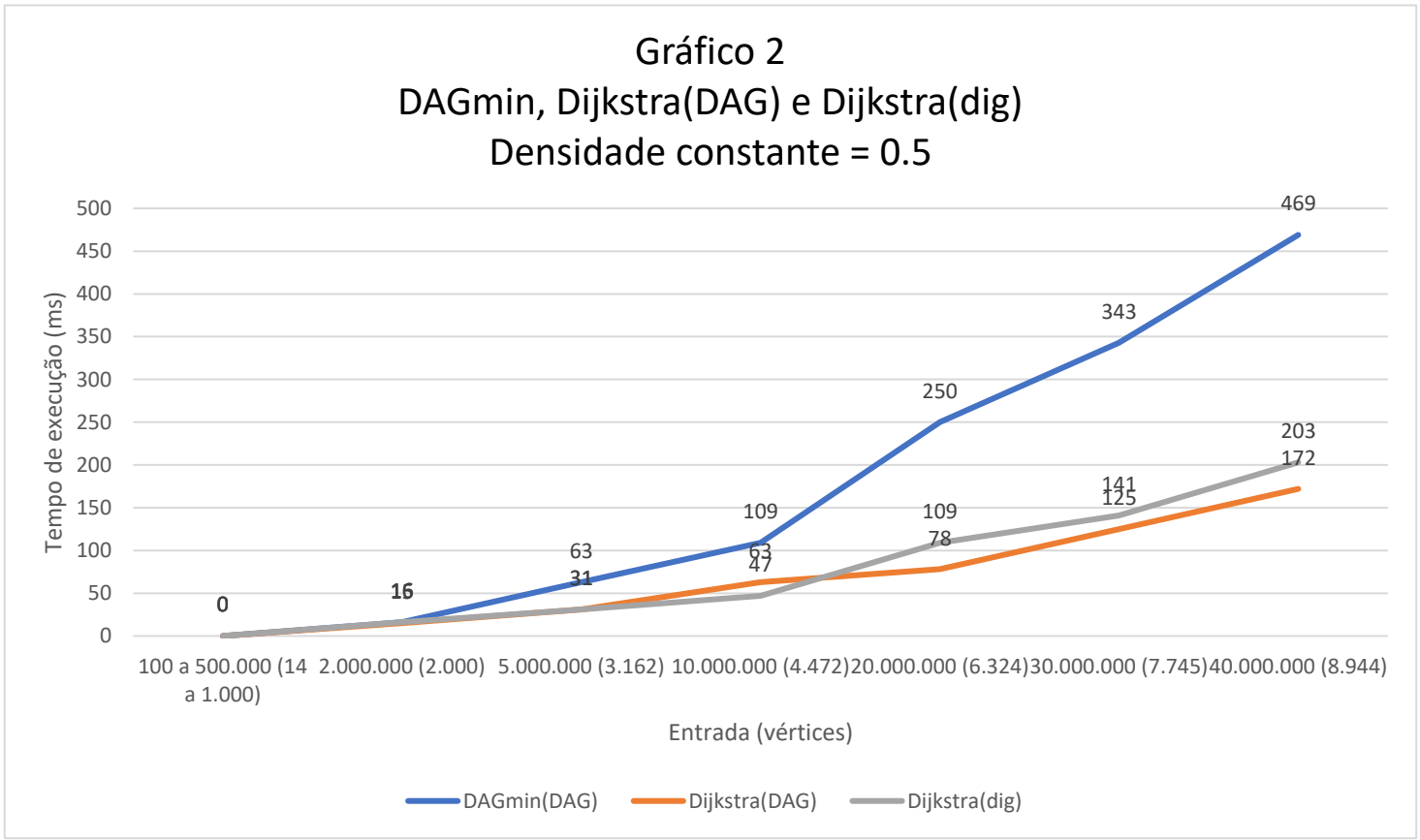


Gráfico 2
DAGmin, Dijkstra(DAG) e Dijkstra(dig)
Densidade constante = 0.5
Tempo de execução (ms)
Entrada (vértices)
100 a 500.000 (14 a 1.000)
2.000.000 (2.000)
5.000.000 (3.162)
10.000.000 (4.472)
20.000.000 (6.324)
30.000.000 (7.745)
40.000.000 (8.944)
DAGmin(DAG)
Dijkstra(DAG)
Dijkstra(dig)

Gráfico 3
Comparativo entre algoritmos Bellman-Ford
Entrada constante = 500.000

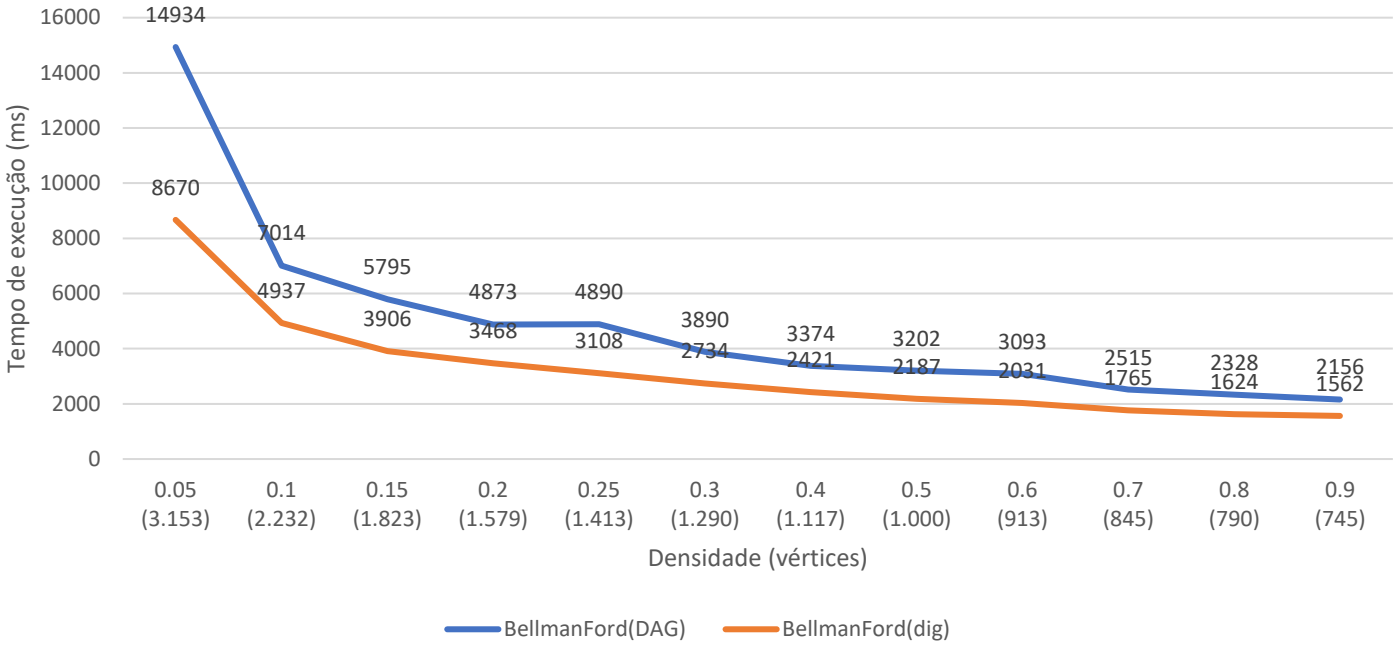
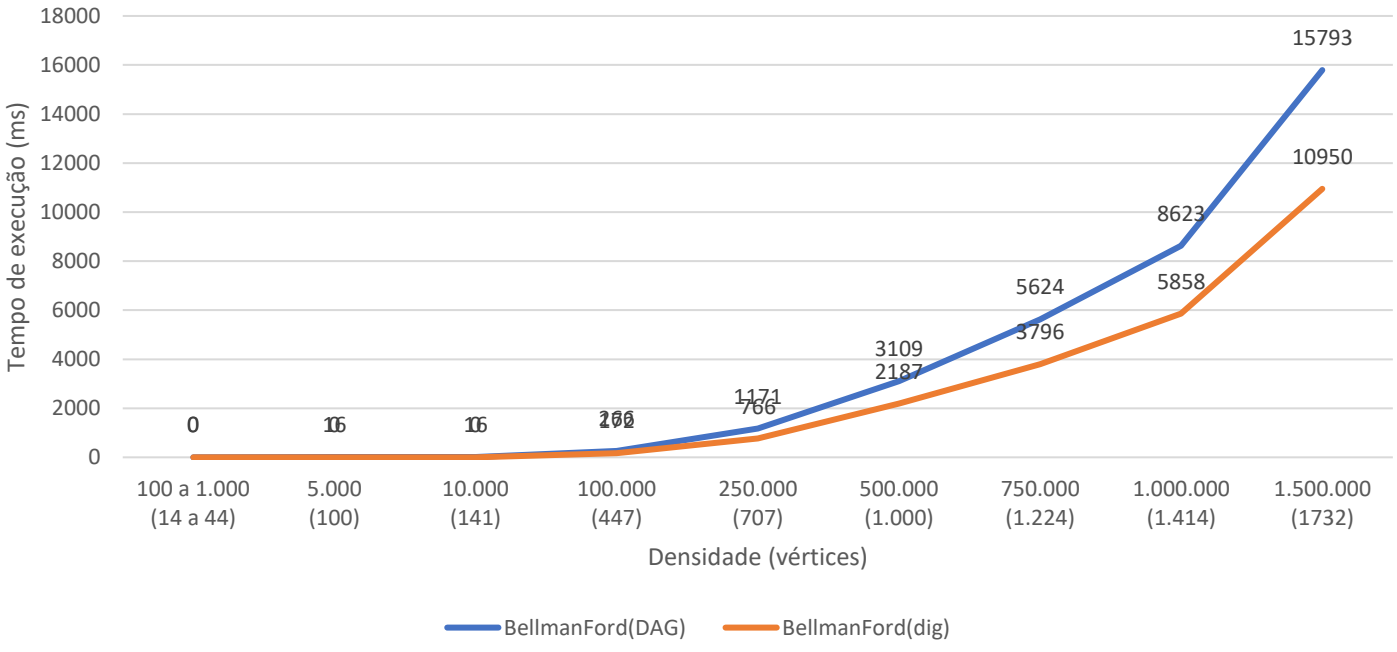


Gráfico 4
Comparativo entre algoritmos Bellman-Ford
Densidade constante = 0.5*



*após começarmos a elaborar o gráfico de Bellman-Ford com densidade constante, percebemos que a bateria de testes até então feita não era suficiente para aferir qualquer tipo de padrão ou comportamento. Por isso rodamos uma nova série de teste com as seguintes características:

Densidades = {0.5}
Entradas = {100, 500, 1000, 5000, 10000, 100000, 250000, 500000, 750000, 1000000, 1500000}

```
//Primeira parte
If (entrada <= 1500000)
        BellmanFord (digrafoAleatorio)

//Segunda parte
If (entrada <= 1500000)
        BellmanFord (dag)
```

Basicamente, aumentamos o limite para execução do algoritmo: de 500.000 para 1.500.000 pois com apenas uma densidade a execução passa a ser feita em tempo hábil, e tornamos mais esparsos os valores da *array* de entradas.

Conclusões

A partir dos gráficos acimas é possível aferir as seguintes conclusões:

- O tempo de execução dos algoritmos de DAGmin e Dijkstra independem da densidade, mas sim da quantidade de entradas (gráfico 1 e 2)

- DAGmin é mais custoso do que Dijkstra (gráfico 1 e 2)

- Para quantidade de entradas constantes, DAGmin, Dijkstra (DAG) e Dijkstra (dig), o tempo de execução é constante (gráfico 1)

- Para quantidade de entradas variáveis, DAGmin, Dijkstra (DAG) e Dijkstra (dig), o tempo de execução segue função linear (gráfico 2)

- Dijkstra mantém seu desempenho mesmo ao processar DAGs ou digrafos cíclicos (gráfico 1 e 2)

- DAGmin, Dijkstra (DAG) e Dijkstra (dig) são muito mais eficientes que Bellman-Ford (gráfico 1 e 3)

- Bellman-Ford opera melhor com grafos de poucos vértices (gráfico 3)

- Com densidade constante, o tempo de execução de Bellman-Ford segue função exponencial (gráfico 4)

- Para Bellman-Ford é mais custoso processar DAGs (gráfico 3 e 4)

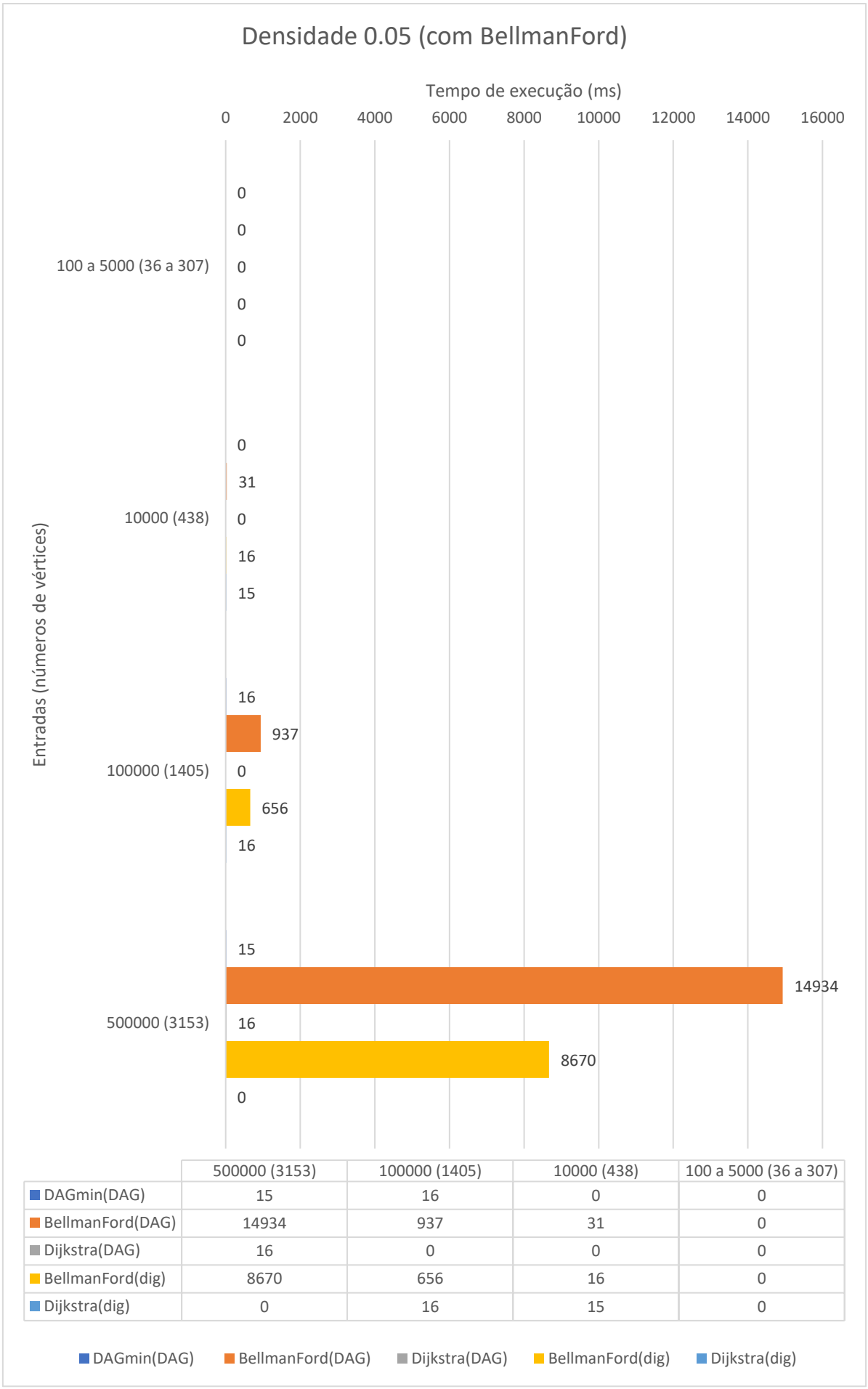
Gráficos remanescestes

Em seguida serão apresentados 16 gráficos agrupados por oito densidades escolhidas cujo intuito é por cada algoritmo lado a lado a fim traçar um paralelo entre eles.

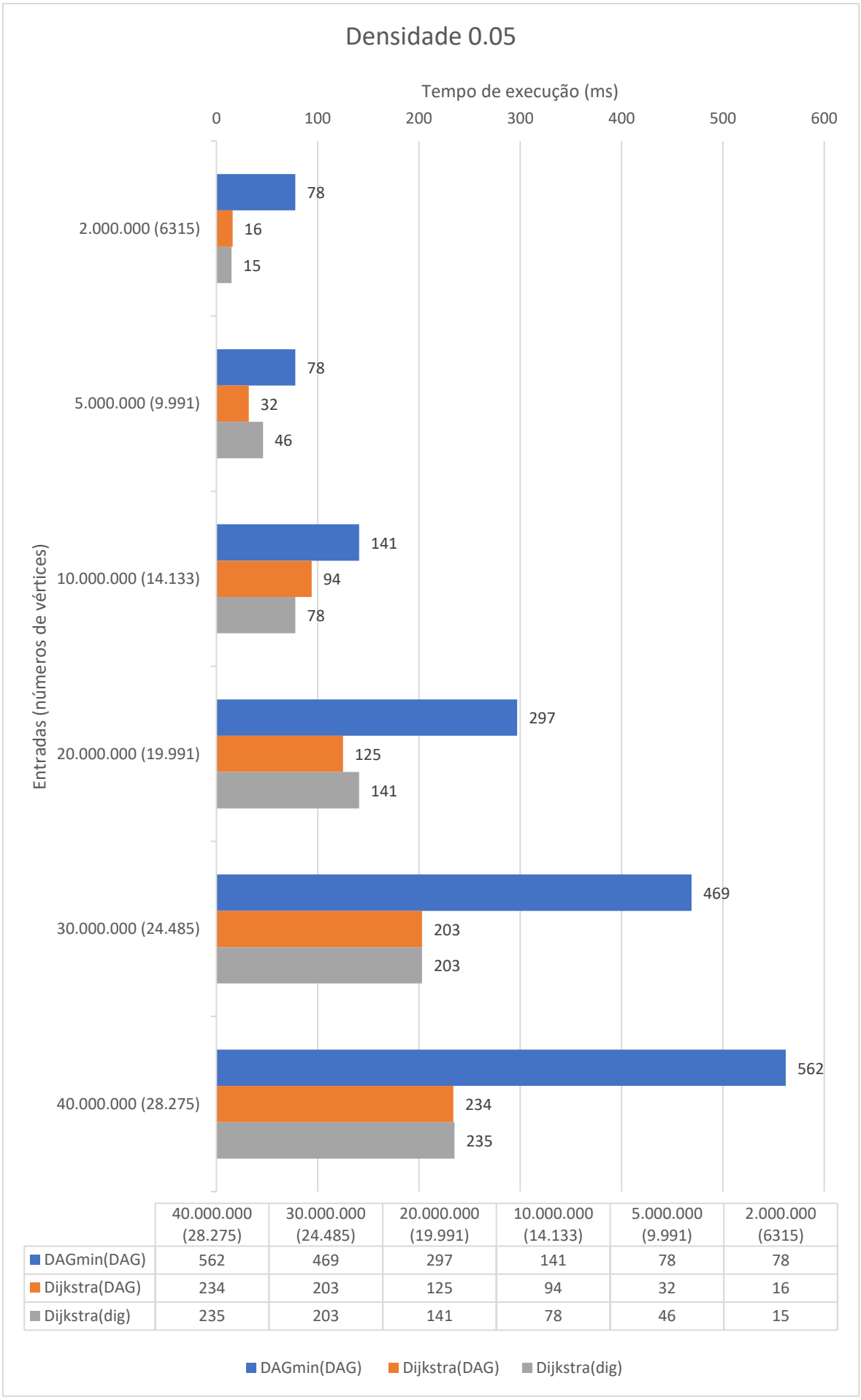
O eixo Y representa a quantidade de entradas, que variam de 100 a 40 milhões. Ainda no mesmo eixo o número entre parêntesis indica a quantidade de vértices assim como feito anteriormente. O eixo X indica o tempo gasto pelo algoritmo em milissegundos.

Os gráficos serão agrupados, como já dito acima, pelo grau de densidade. Cada grau, porém, terá dois gráficos: um com entradas até 500.000 e, portanto, com os resultados dos algoritmos Bellman-Ford (aqui é possível ver os demais desaparecem, restando justificada a limitação de sua execução até as 500.000 entradas), e o outro com os três testes restantes.

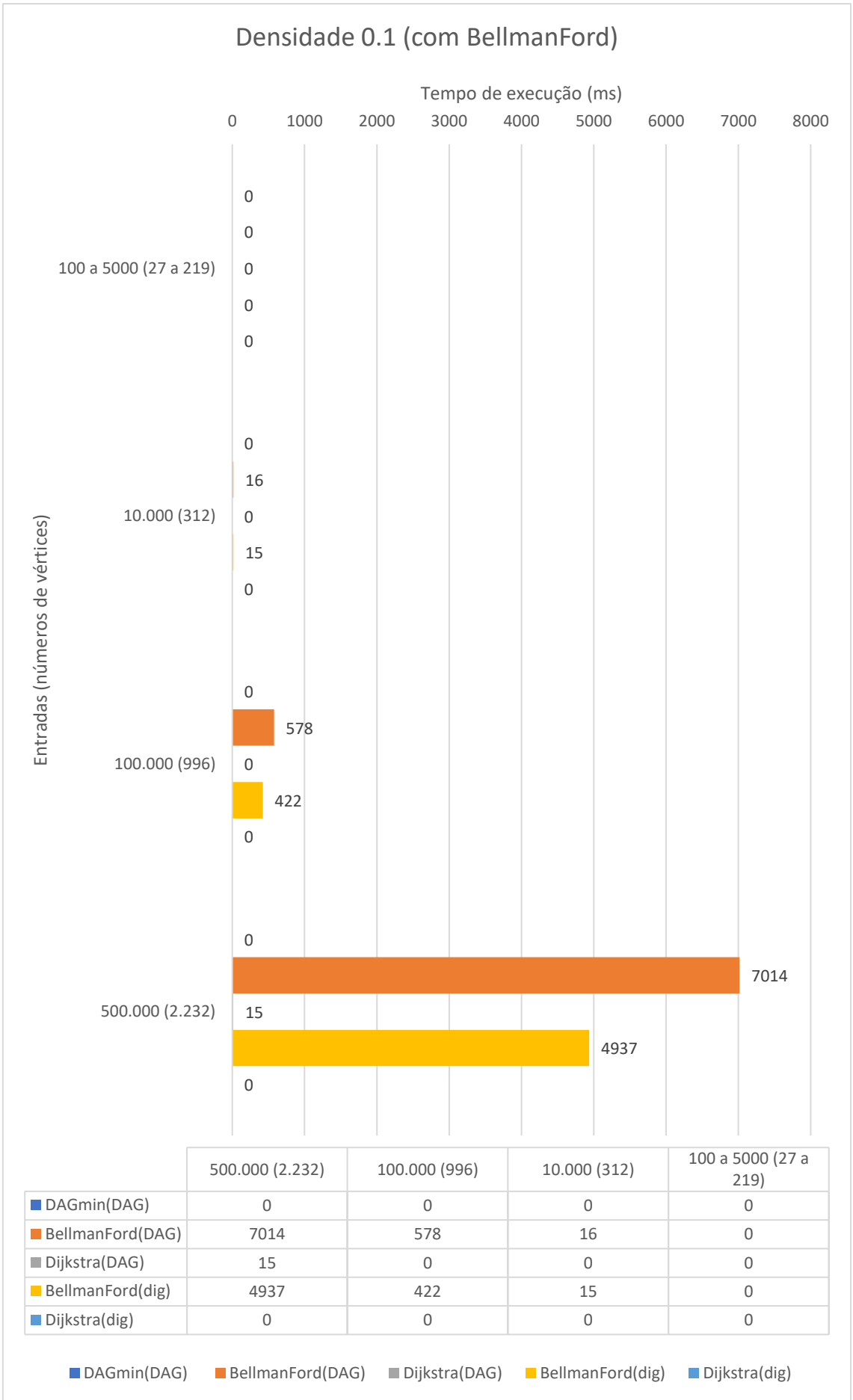
Observação: as tabelas que seguem ao fim de cada gráfico apresentam as variáveis em ordem decrescente, ao contrário do disposto nos gráficos. Assim foram desenvolvidas por conta de limitação da ferramenta Word.



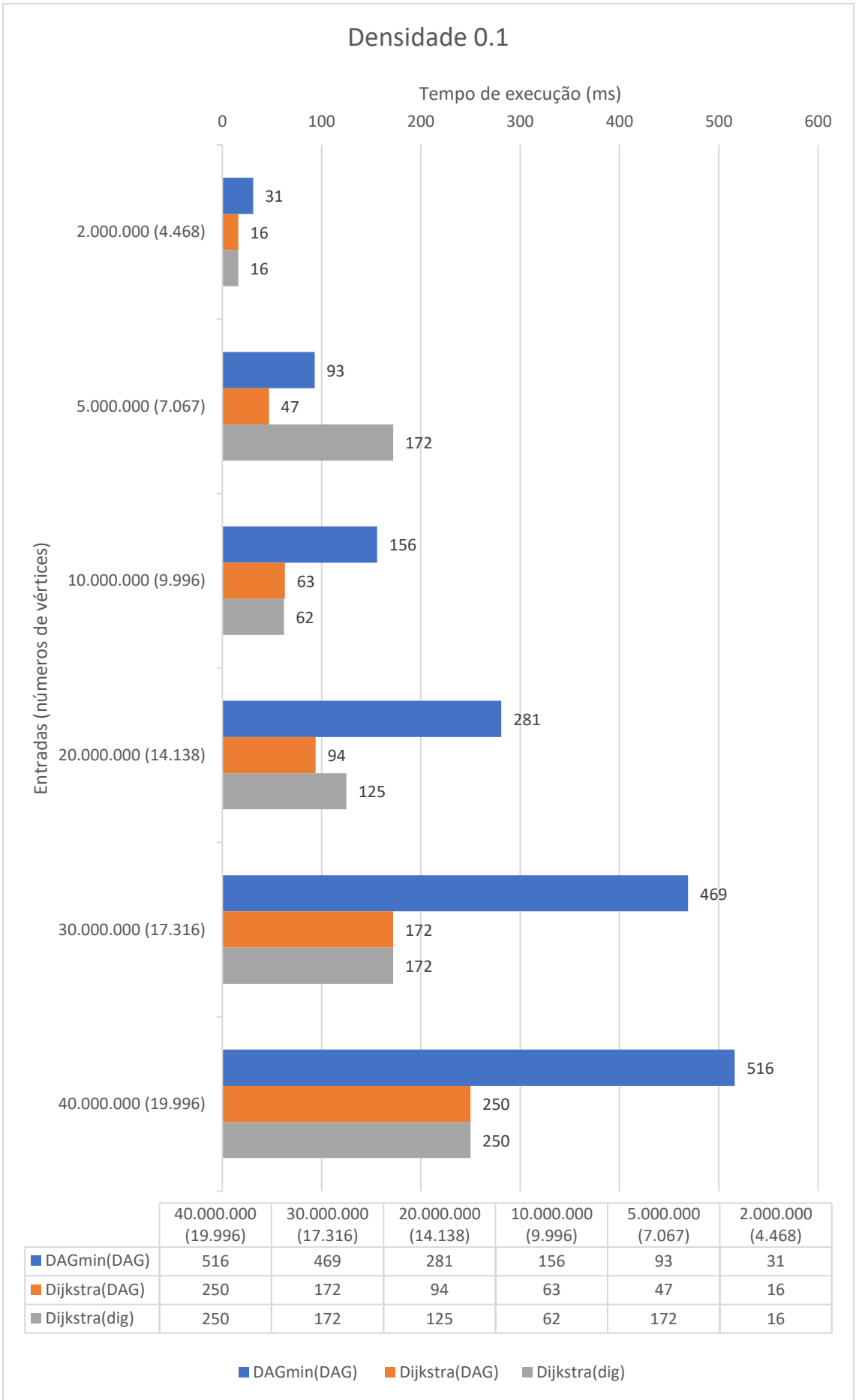
	500000 (3153)	100000 (1405)	10000 (438)	100 a 5000 (36 a 307)
DAGmin(DAG)	15	16	0	0
BellmanFord(DAG)	14934	937	31	0
Dijkstra(DAG)	16	0	0	0
BellmanFord(dig)	8670	656	16	0
Dijkstra(dig)	0	16	15	0



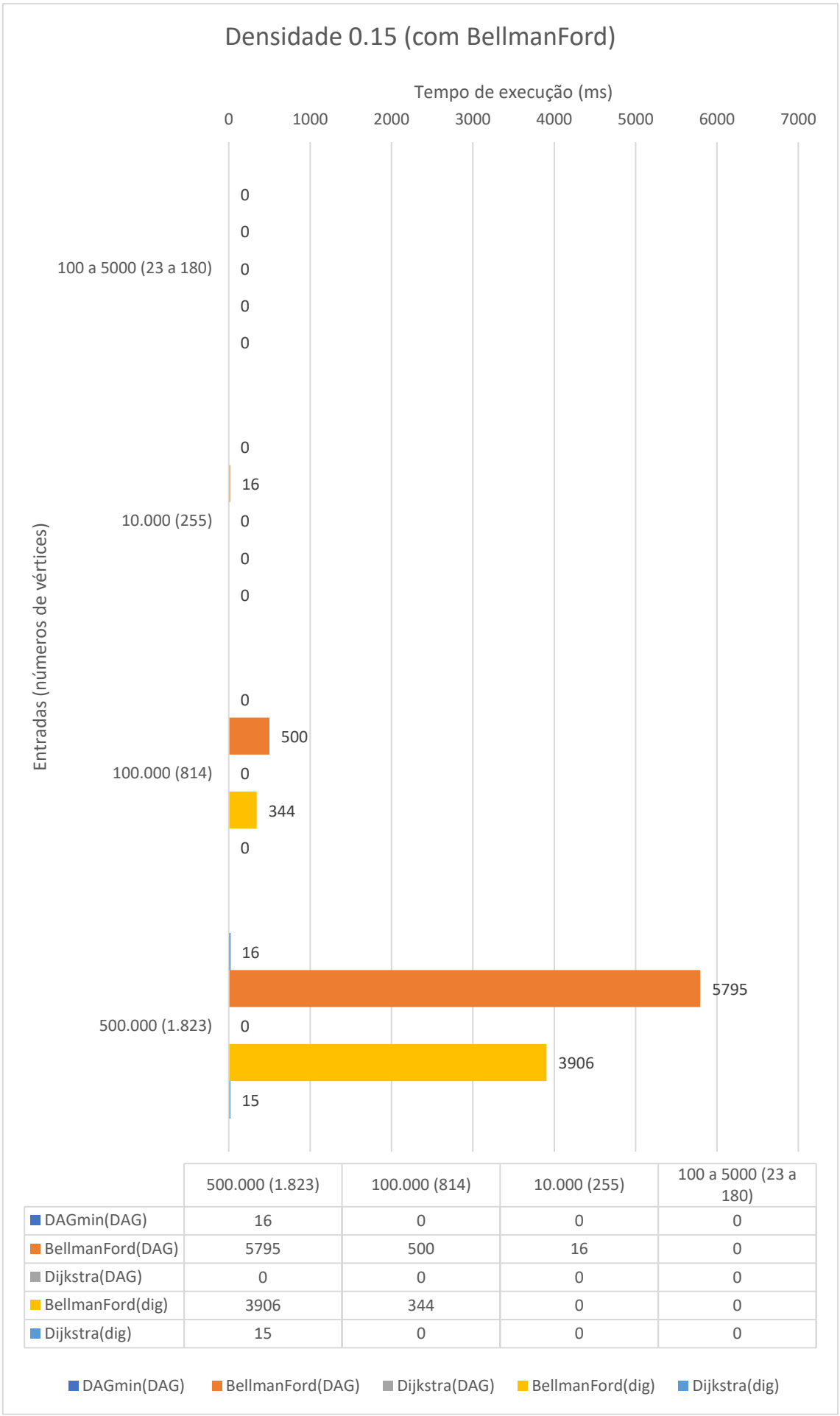
	40.000.000 (28.275)	30.000.000 (24.485)	20.000.000 (19.991)	10.000.000 (14.133)	5.000.000 (9.991)	2.000.000 (6315)
DAGmin(DAG)	562	469	297	141	78	78
Dijkstra(DAG)	234	203	125	94	32	16
Dijkstra(dig)	235	203	141	78	46	15



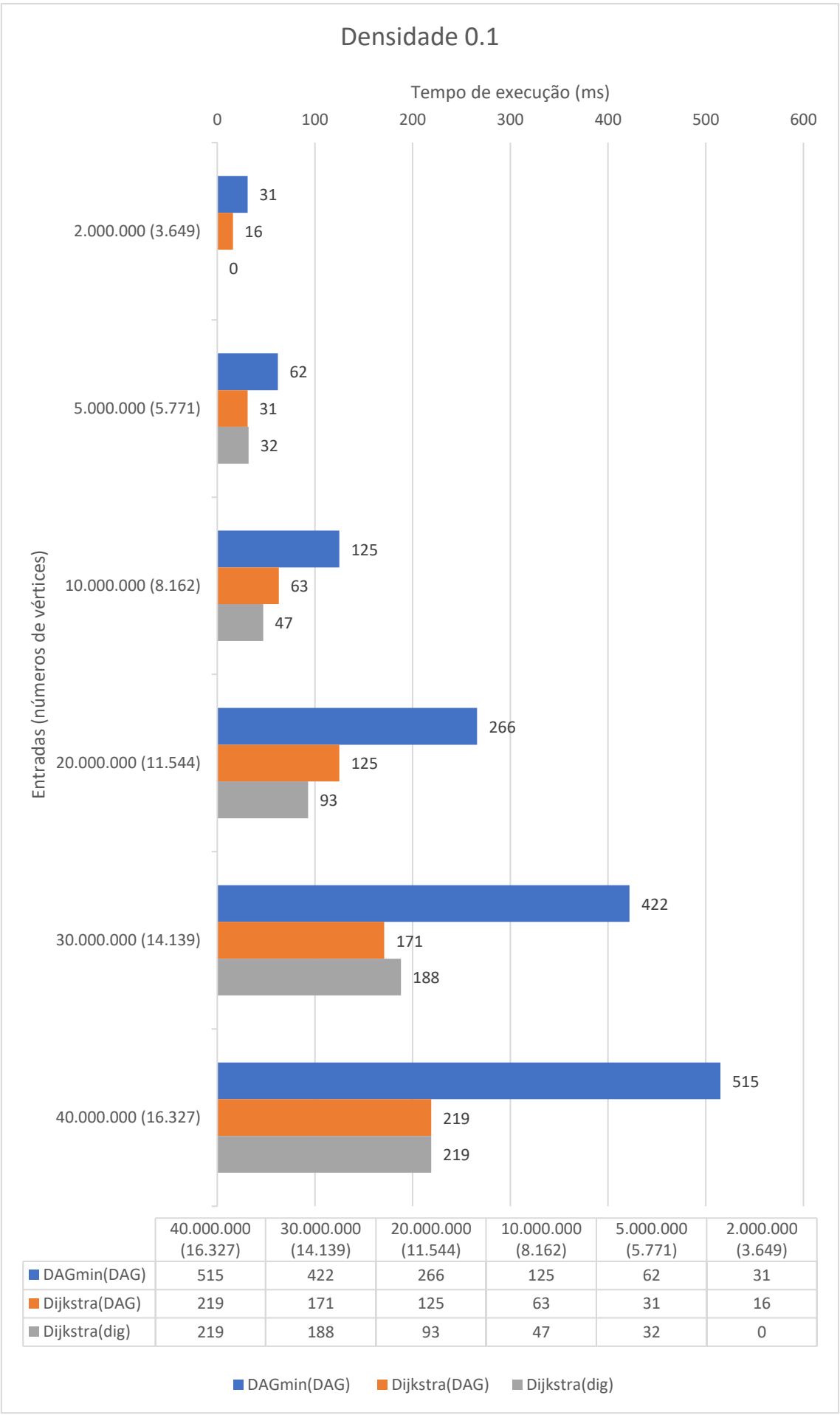
	500.000 (2.232)	100.000 (996)	10.000 (312)	100 a 5000 (27 a 219)
DAGmin(DAG)	0	0	0	0
BellmanFord(DAG)	7014	578	16	0
Dijkstra(DAG)	15	0	0	0
BellmanFord(dig)	4937	422	15	0
Dijkstra(dig)	0	0	0	0



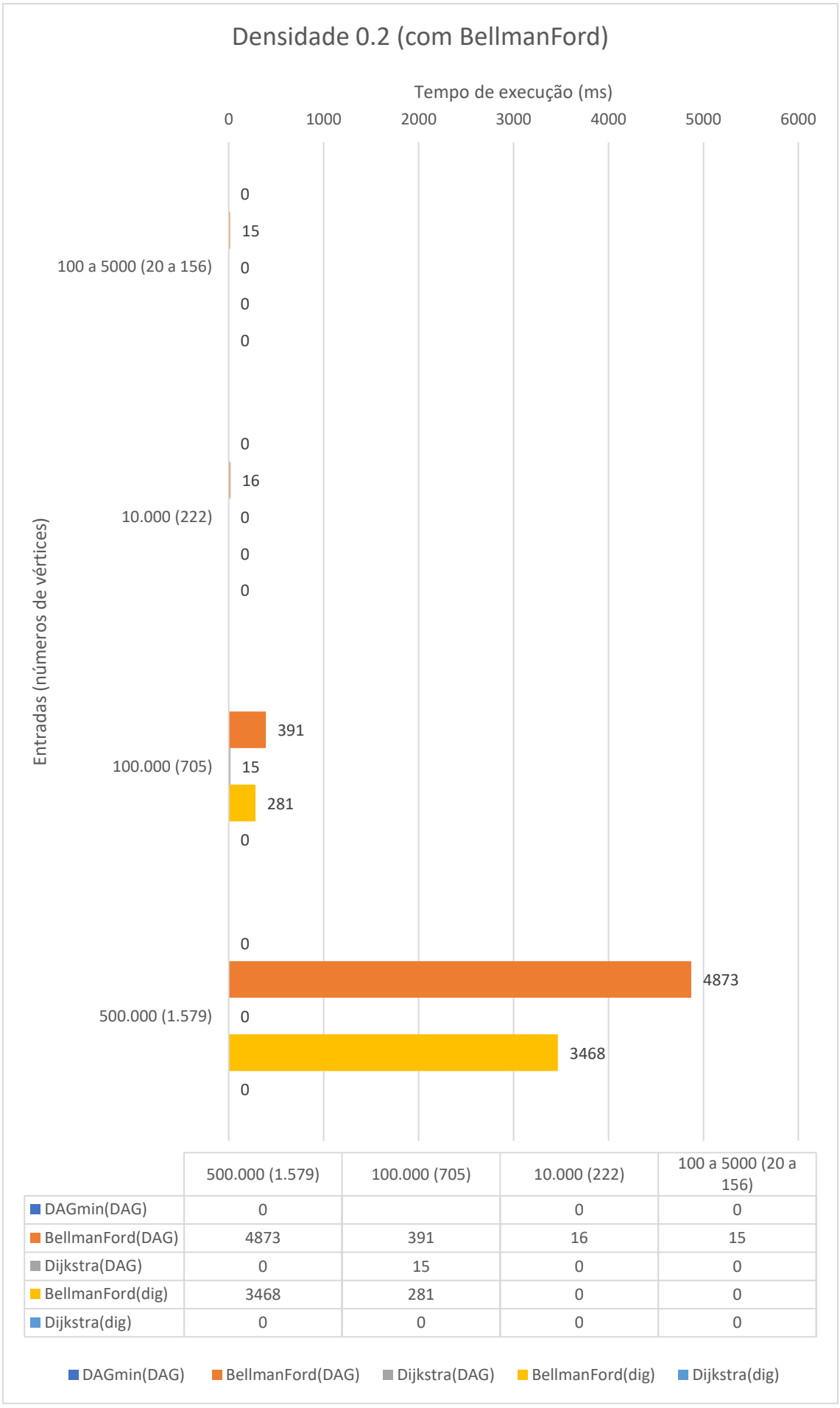
	40.000.000 (19.996)	30.000.000 (17.316)	20.000.000 (14.138)	10.000.000 (9.996)	5.000.000 (7.067)	2.000.000 (4.468)
DAGmin(DAG)	516	469	281	156	93	31
Dijkstra(DAG)	250	172	94	63	47	16
Dijkstra(dig)	250	172	125	62	172	16



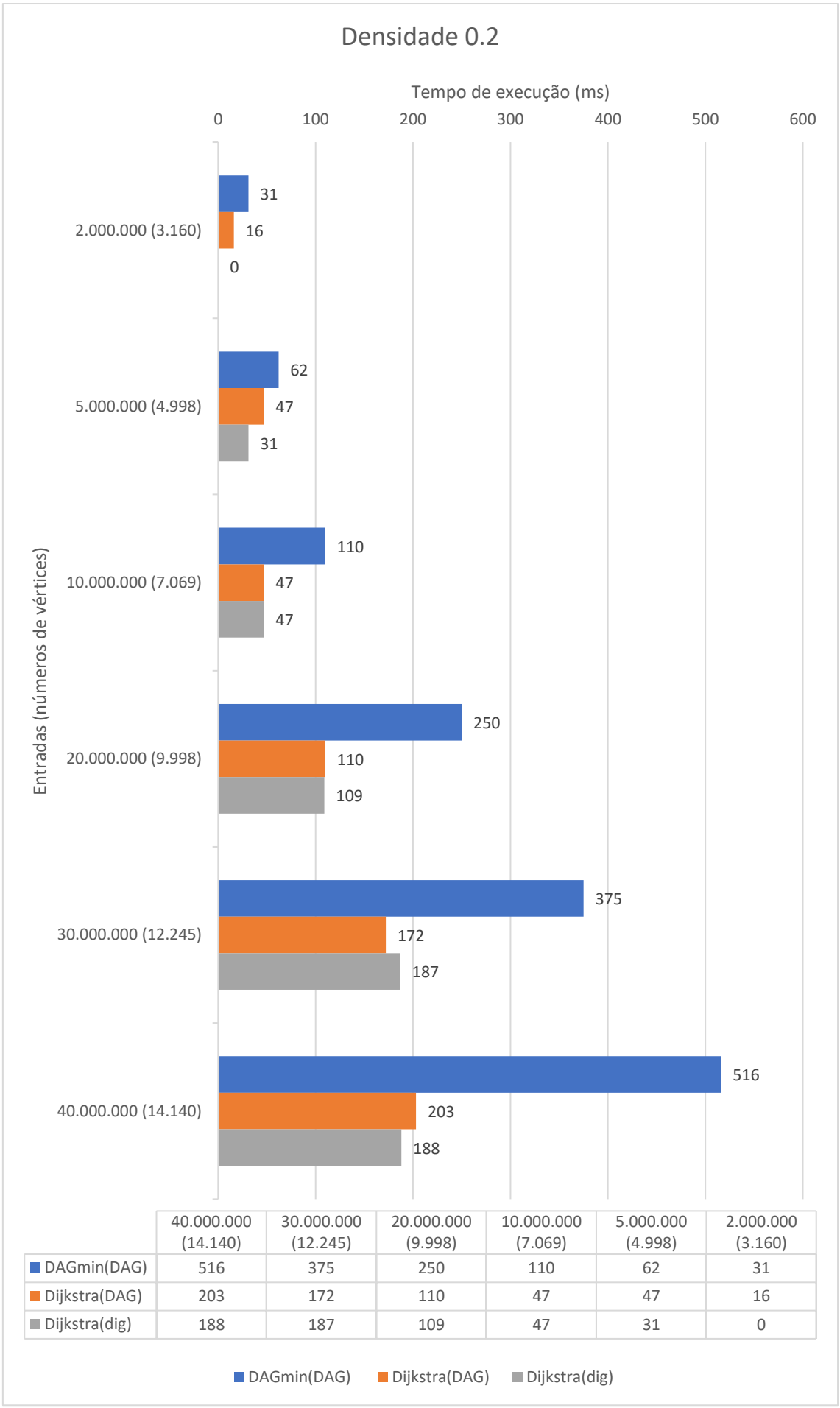
	500.000 (1.823)	100.000 (814)	10.000 (255)	100 a 5000 (23 a 180)
DAGmin(DAG)	16	0	0	0
BellmanFord(DAG)	5795	500	16	0
Dijkstra(DAG)	0	0	0	0
BellmanFord(dig)	3906	344	0	0
Dijkstra(dig)	15	0	0	0



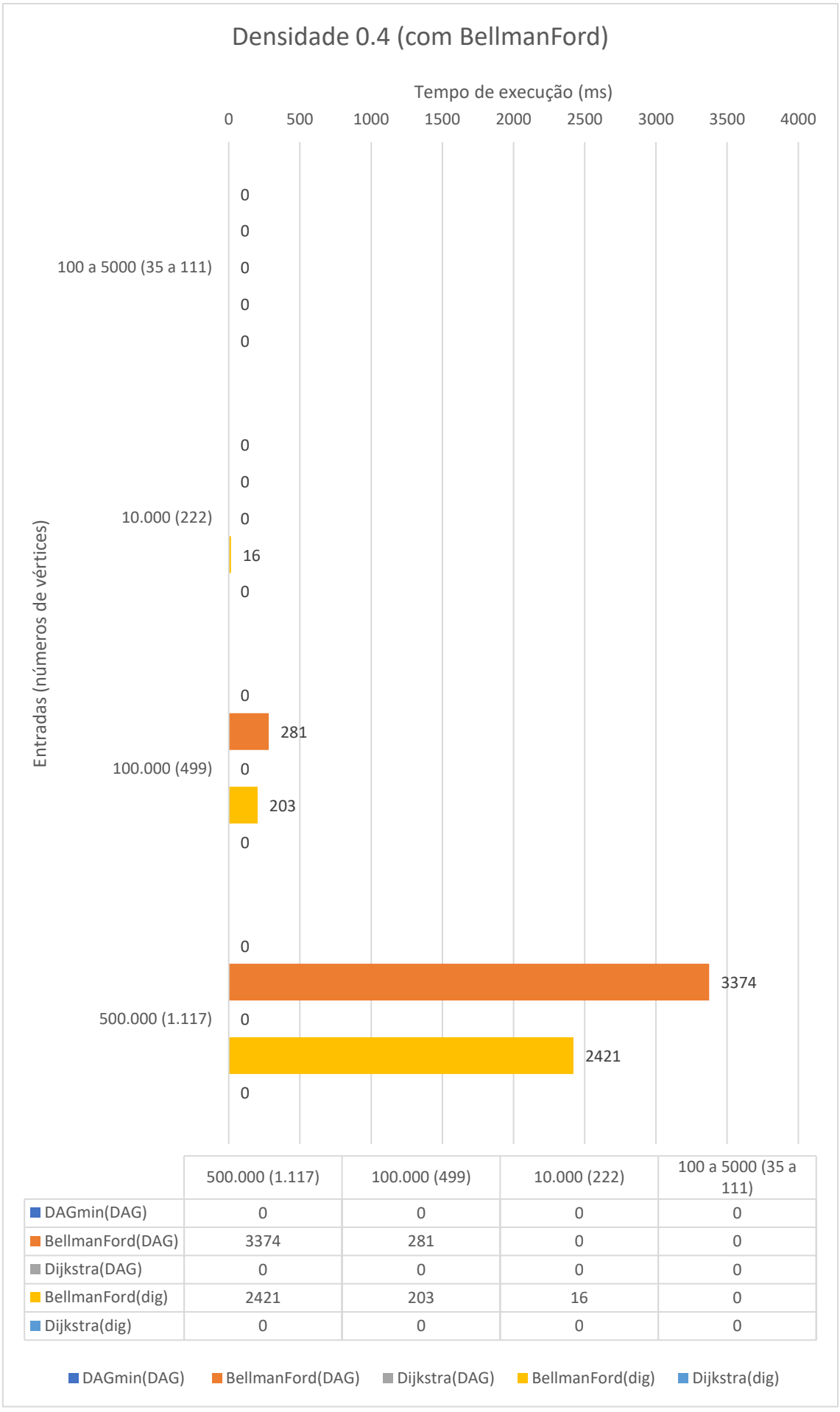
	40.000.000 (16.327)	30.000.000 (14.139)	20.000.000 (11.544)	10.000.000 (8.162)	5.000.000 (5.771)	2.000.000 (3.649)
DAGmin(DAG)	515	422	266	125	62	31
Dijkstra(DAG)	219	171	125	63	31	16
Dijkstra(dig)	219	188	93	47	32	0



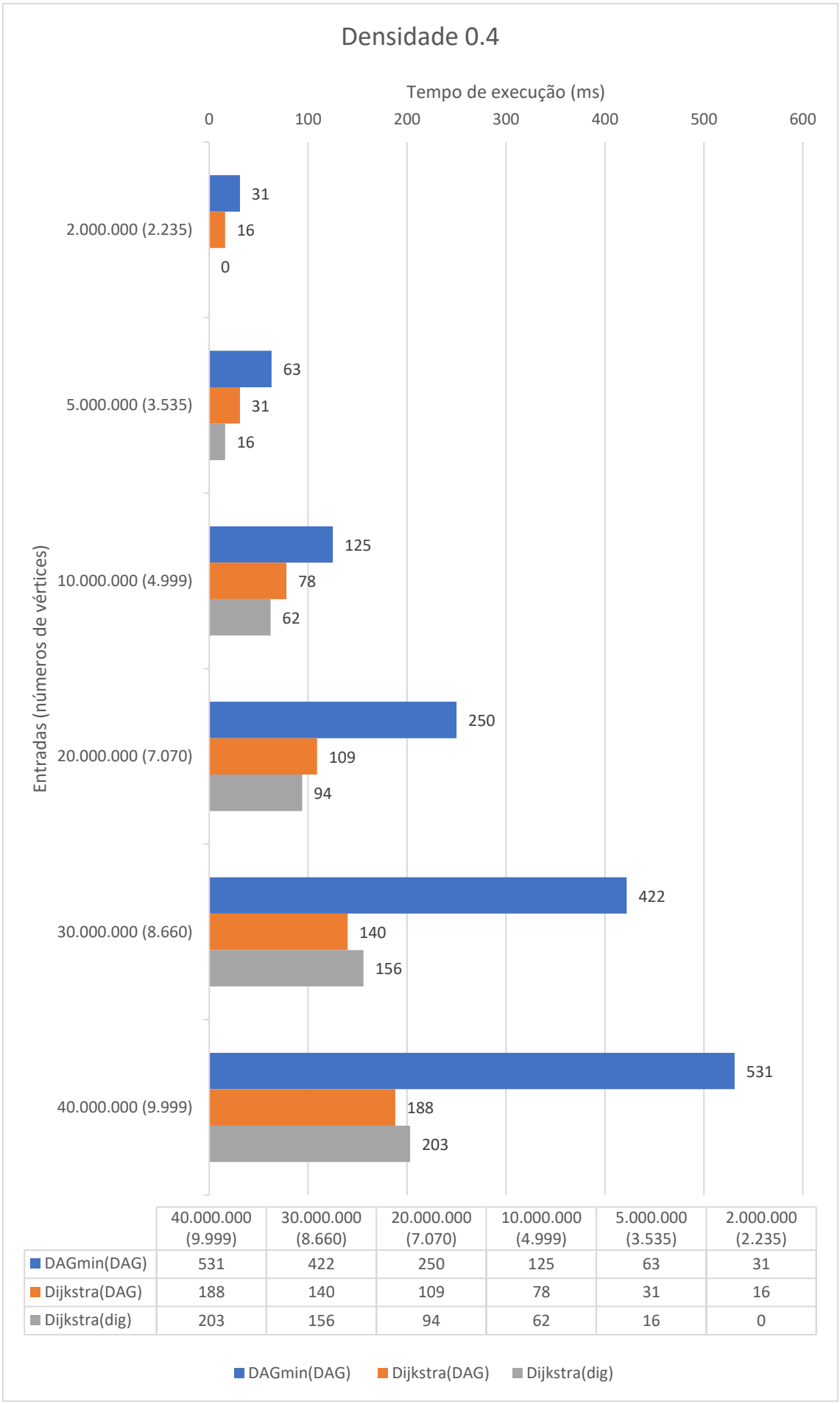
	500.000 (1.579)	100.000 (705)	10.000 (222)	100 a 5000 (20 a 156)
DAGmin(DAG)	0		0	0
BellmanFord(DAG)	4873	391	16	15
Dijkstra(DAG)	0	15	0	0
BellmanFord(dig)	3468	281	0	0
Dijkstra(dig)	0	0	0	0



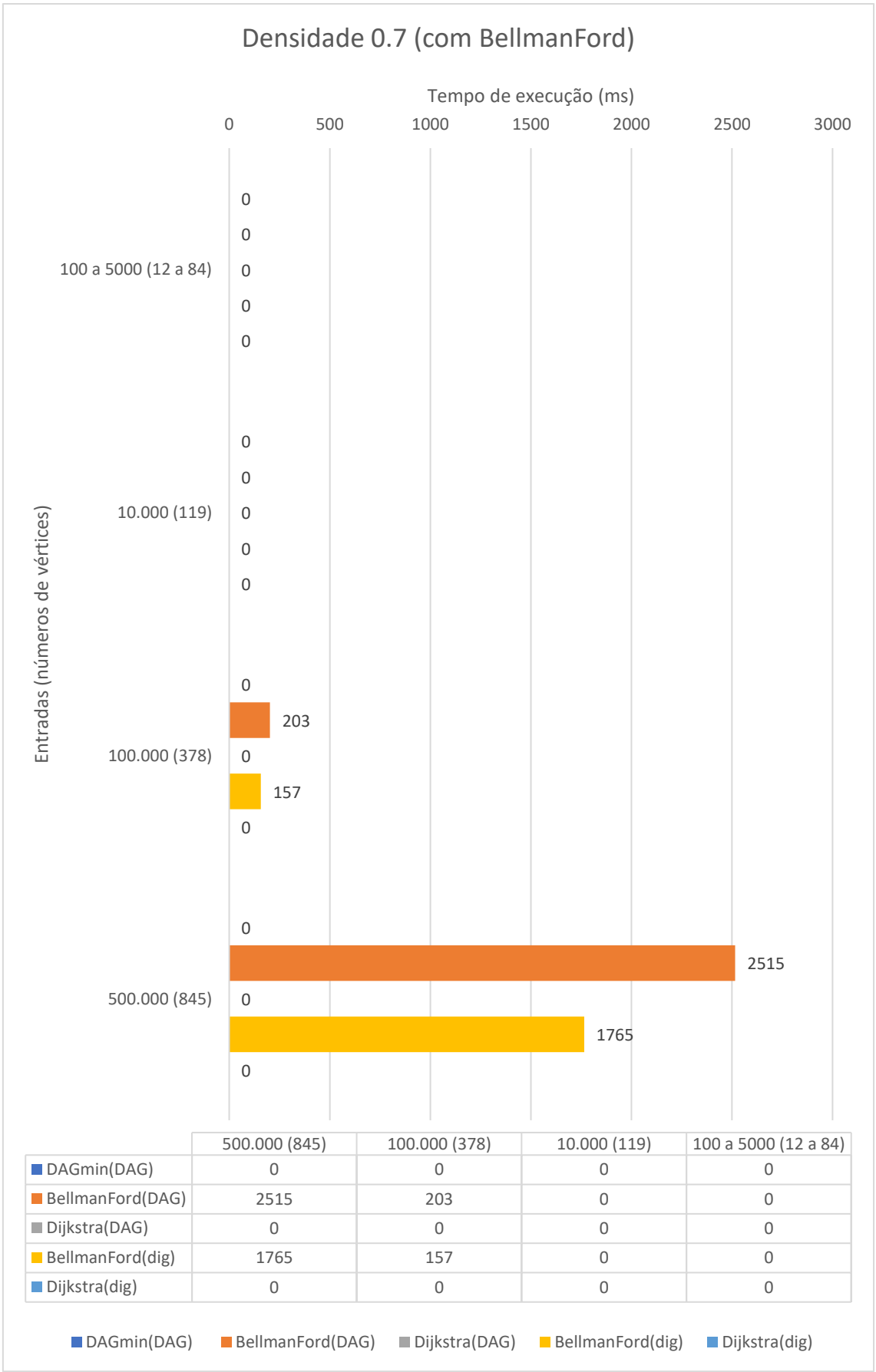
	40.000.000 (14.140)	30.000.000 (12.245)	20.000.000 (9.998)	10.000.000 (7.069)	5.000.000 (4.998)	2.000.000 (3.160)
DAGmin(DAG)	516	375	250	110	62	31
Dijkstra(DAG)	203	172	110	47	47	16
Dijkstra(dig)	188	187	109	47	31	0



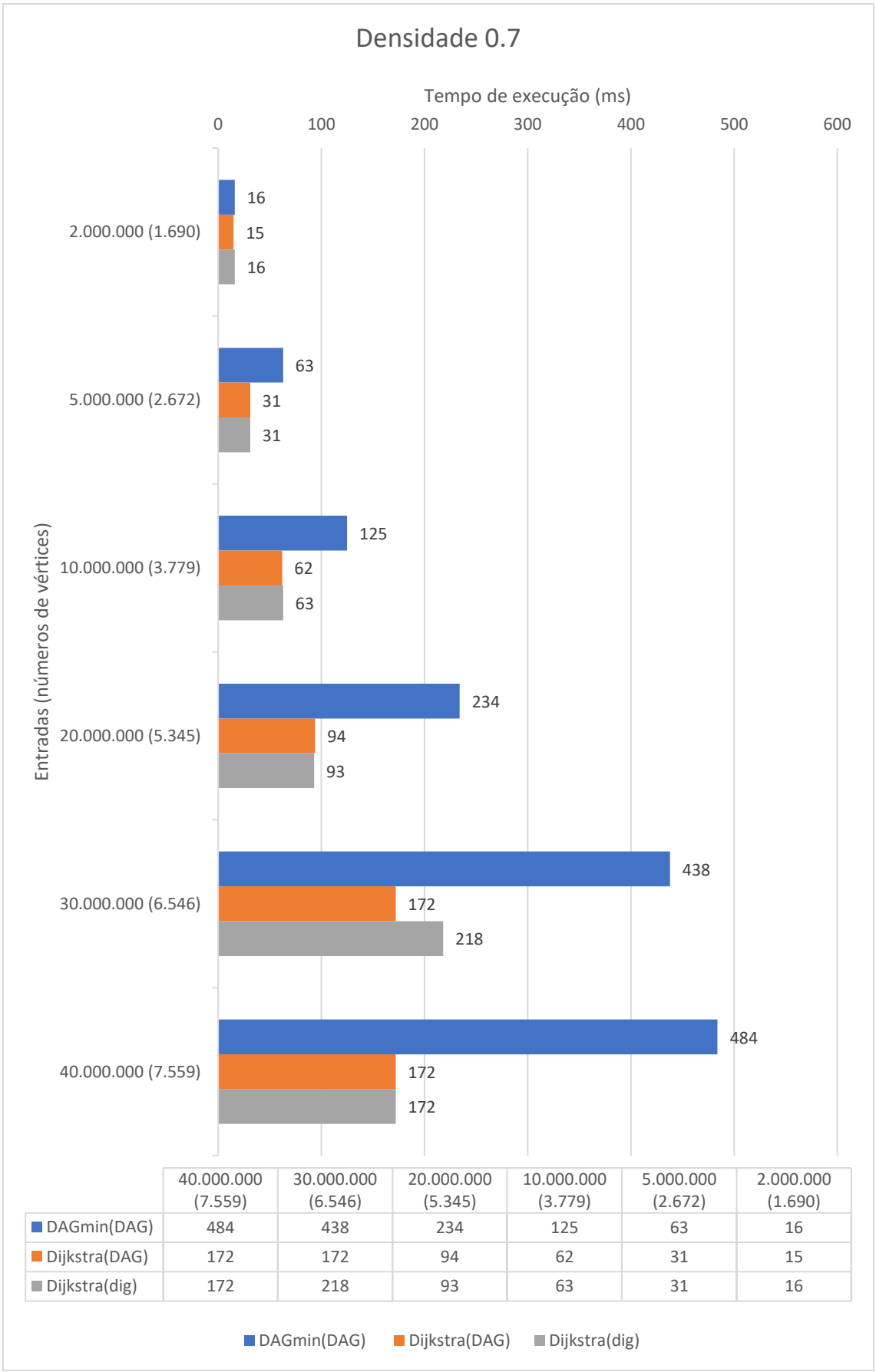
	500.000 (1.117)	100.000 (499)	10.000 (222)	100 a 5000 (35 a 111)
DAGmin(DAG)	0	0	0	0
BellmanFord(DAG)	3374	281	0	0
Dijkstra(DAG)	0	0	0	0
BellmanFord(dig)	2421	203	16	0
Dijkstra(dig)	0	0	0	0



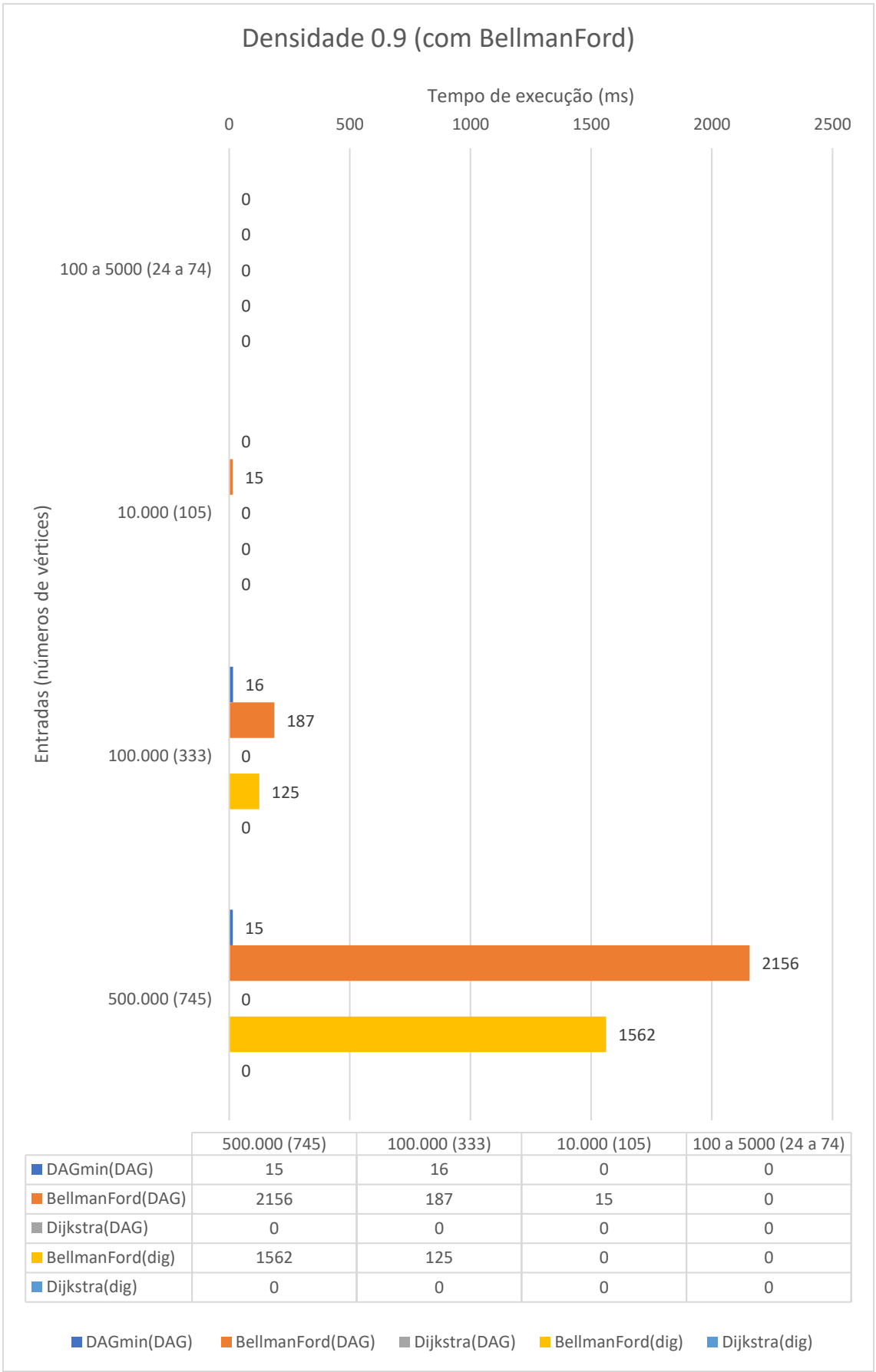
	40.000.000 (9.999)	30.000.000 (8.660)	20.000.000 (7.070)	10.000.000 (4.999)	5.000.000 (3.535)	2.000.000 (2.235)
DAGmin(DAG)	531	422	250	125	63	31
Dijkstra(DAG)	188	140	109	78	31	16
Dijkstra(dig)	203	156	94	62	16	0



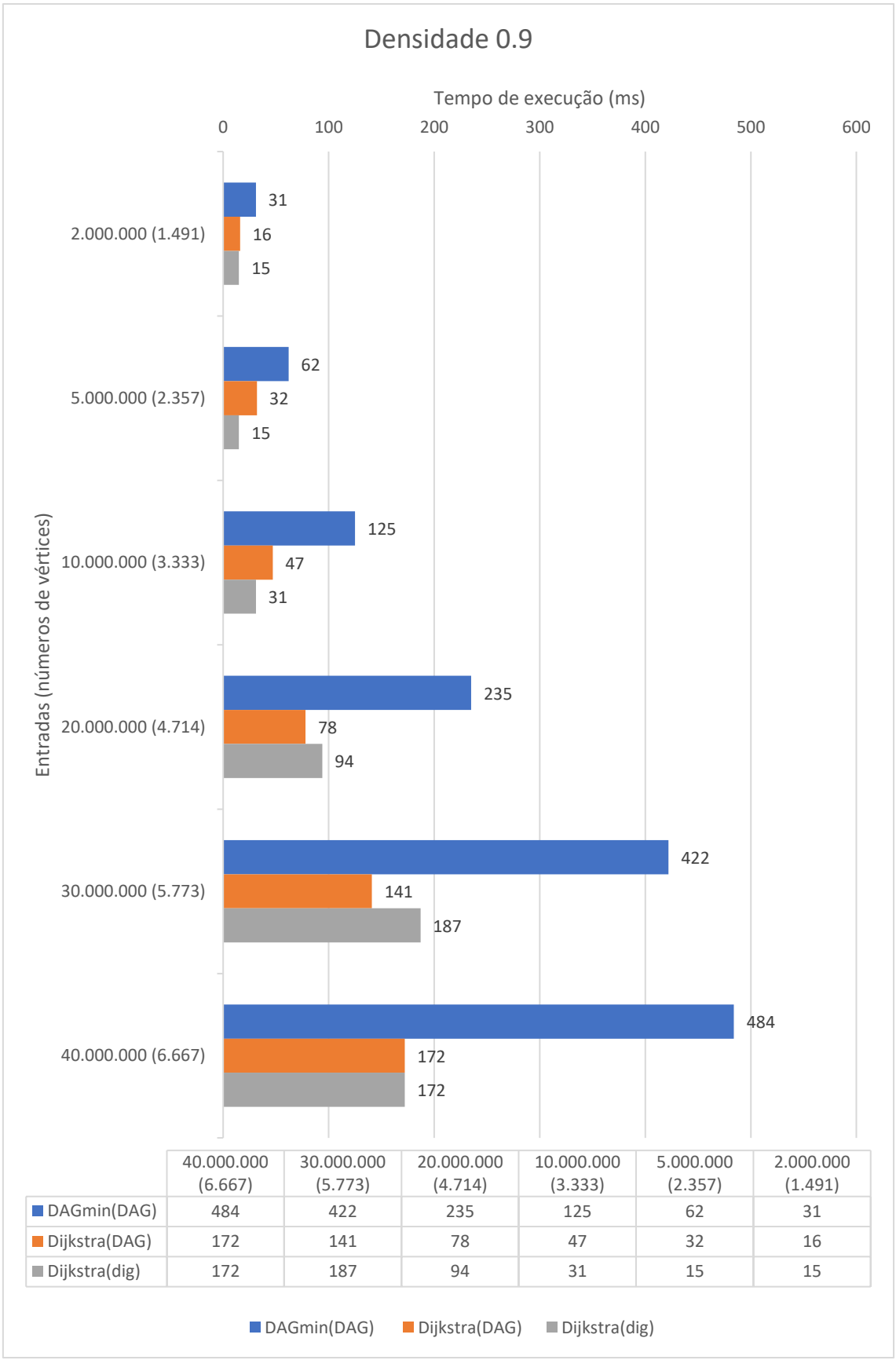
	500.000 (845)	100.000 (378)	10.000 (119)	100 a 5000 (12 a 84)
DAGmin(DAG)	0	0	0	0
BellmanFord(DAG)	2515	203	0	0
Dijkstra(DAG)	0	0	0	0
BellmanFord(dig)	1765	157	0	0
Dijkstra(dig)	0	0	0	0



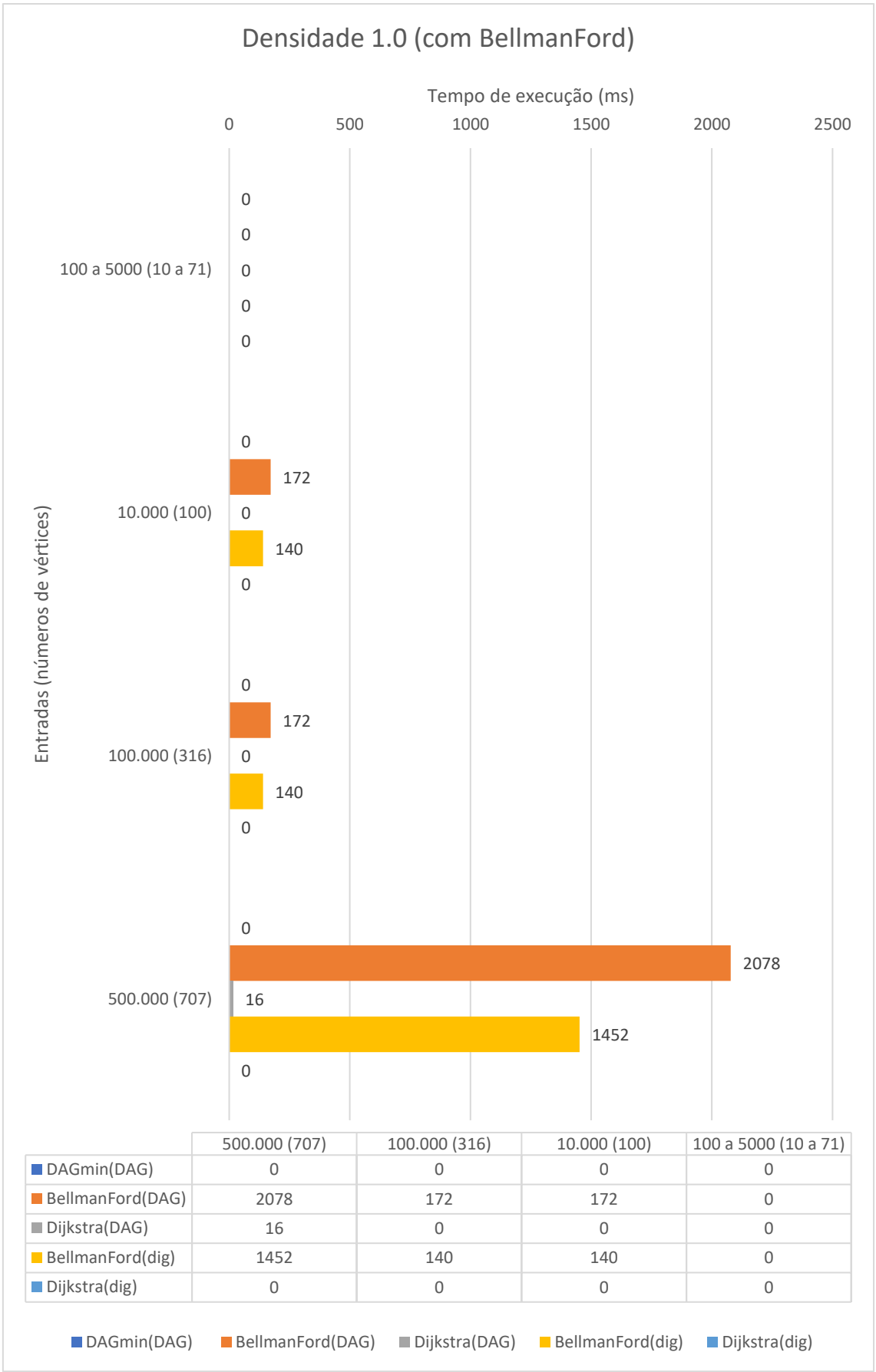
	40.000.000 (7.559)	30.000.000 (6.546)	20.000.000 (5.345)	10.000.000 (3.779)	5.000.000 (2.672)	2.000.000 (1.690)
DAGmin(DAG)	484	438	234	125	63	16
Dijkstra(DAG)	172	172	94	62	31	15
Dijkstra(dig)	172	218	93	63	31	16



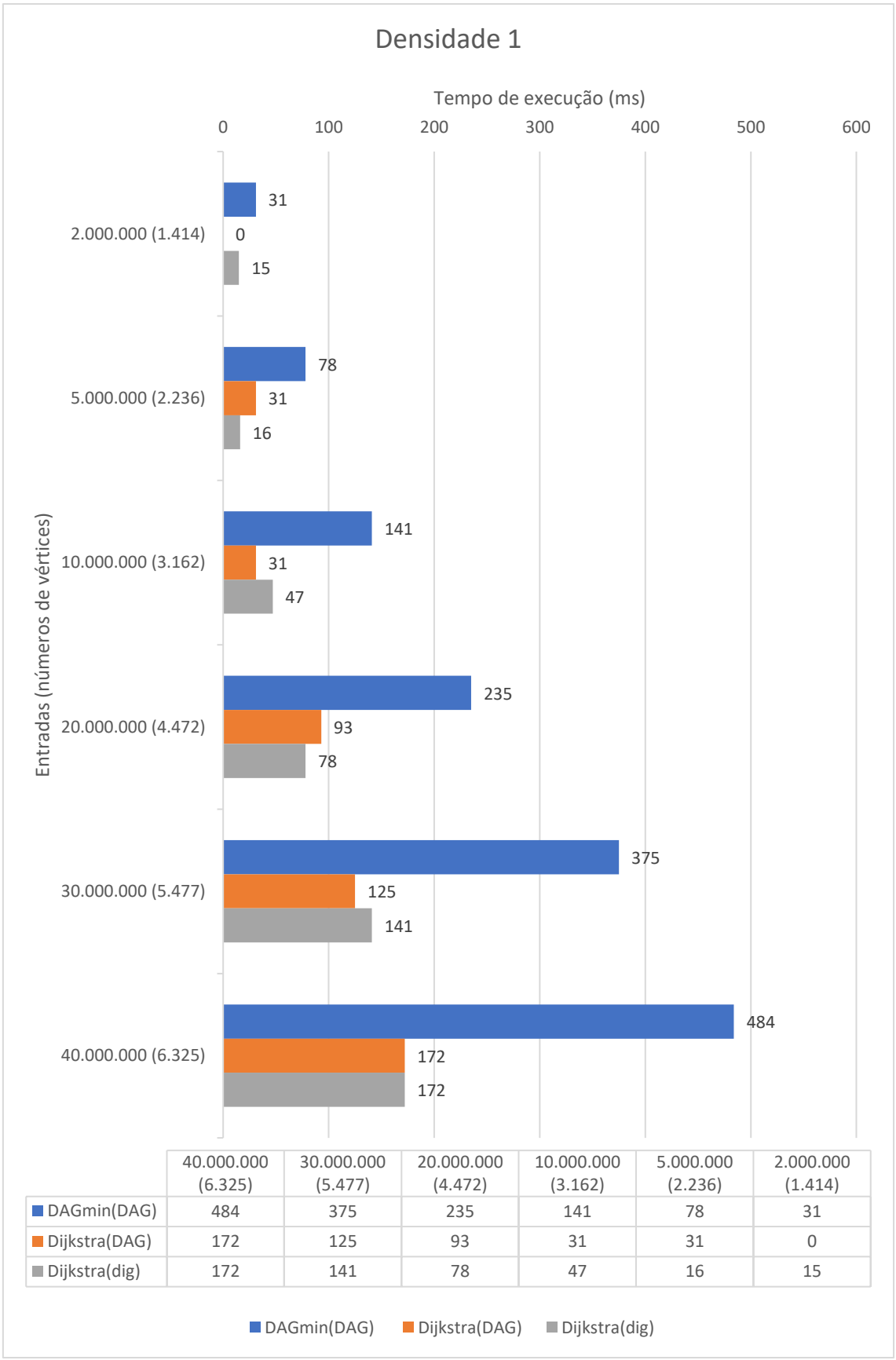
	500.000 (745)	100.000 (333)	10.000 (105)	100 a 5000 (24 a 74)
DAGmin(DAG)	15	16	0	0
BellmanFord(DAG)	2156	187	15	0
Dijkstra(DAG)	0	0	0	0
BellmanFord(dig)	1562	125	0	0
Dijkstra(dig)	0	0	0	0



	40.000.000 (6.667)	30.000.000 (5.773)	20.000.000 (4.714)	10.000.000 (3.333)	5.000.000 (2.357)	2.000.000 (1.491)
DAGmin(DAG)	484	422	235	125	62	31
Dijkstra(DAG)	172	141	78	47	32	16
Dijkstra(dig)	172	187	94	31	15	15



	500.000 (707)	100.000 (316)	10.000 (100)	100 a 5000 (10 a 71)
DAGmin(DAG)	0	0	0	0
BellmanFord(DAG)	2078	172	172	0
Dijkstra(DAG)	16	0	0	0
BellmanFord(dig)	1452	140	140	0
Dijkstra(dig)	0	0	0	0



	40.000.000 (6.325)	30.000.000 (5.477)	20.000.000 (4.472)	10.000.000 (3.162)	5.000.000 (2.236)	2.000.000 (1.414)
DAGmin(DAG)	484	375	235	141	78	31
Dijkstra(DAG)	172	125	93	31	31	0
Dijkstra(dig)	172	141	78	47	16	15