Tendências em Matemática Aplicada e Computacional, **20**, N. 3 (2019), 457-476

www.scielo.br/tema
doi: 10.5540/tema.2019.020.03.0457

Uma Nova Proposta para a Obtenção da Complexidade de Pior Caso do ShellSort

R. M. SOUZA*, F. S. OLIVEIRA e P. E. D. PINTO

Recebido em 10 dezembro 2018 / Aceito em 2 maio 2019

RESUMO. A complexidade de pior caso do ShellSort, um algoritmo de ordenação por comparação, depende de uma sequência de passos dada de entrada. Cada passo consiste de um inteiro representando a diferença de índices dos pares de elementos que devem ser comparados durante a ordenação de um vetor de entrada. Tal complexidade é conhecida somente para algumas sequências específicas. Neste trabalho, usamos uma relação entre ShellSort e o número de Frobenius para apresentar um novo algoritmo que provê um limite superior no número de comparações que o ShellSort perfaz, para dados vetor e sequência de passos. Aplicamos este algoritmo, em conjunto com uma análise de complexidade empírica, para estudar sequências cujas complexidades de pior caso são conhecidas através do método analítico. Mostramos que a abordagem empírica foi bem sucedida em determinar tais complexidades. Baseado nestes resultados positivos, estendemos o estudo para sequências para as quais as complexidades de pior caso estão em aberto.

Palavras-chave: complexidade de algoritmos, método empírico, ShellSort.

1 INTRODUÇÃO

Um *algoritmo de ordenação* é um algoritmo que, tendo como entrada um vetor qualquer V com n elementos, permuta os elementos em V tal que, para todo $1 \leq i < n$, $V[i] \leq V[i+1]$. Existem algoritmos bem conhecidos de ordenação, tais como QuickSort, MergeSort, BucketSort, RadixSort, entre outros, cada um empregando uma estratégia distinta. Quando um algoritmo de ordenação utiliza somente como operação básica a comparação relativa entre dois elementos do vetor original, este algoritmo é classificado como *algoritmo de ordenação por comparação*. É esse o caso do QuickSort e do MergeSort, mas não do RadixSort. Numa ordenação por comparação, um elemento não pode ser manipulado pela sua representação binária parcial, como por exemplo, a leitura de um bit em particular de um elemento. Para algoritmos de ordenação por comparação,

*Autor correspondente: Raquel Marcolino de Souza – E-mail: raquelmarcolino25@gmail.com – https://orcid.org/0000-0002-8834-9172
Universidade do Estado do Rio de Janeiro (IME/UERJ) R. São Francisco Xavier, 524, CEP 20550-900 - Rio de Janeiro-RJ, Brasil – E-mail: raquelmarcolino25@gmail.com, fabiano.oliveira@ime.uerj.br, pauloedp@ime.uerj.br

então, é natural que as complexidades sejam representadas em função de n, o número de elementos presentes no vetor a ser ordenado. Além disso, assume-se que a complexidade assintótica do número de operações é a mesma daquela do número de comparações que o algoritmo executa. É sabido que o número de comparações realizadas no pior caso, por qualquer algoritmo de ordenação por comparação, é de $\Omega(n \log n)$ [3]. Além do tempo, a análise da complexidade de espaço também é importante. No caso de algoritmos de ordenação, esta se resume a determinar se a quantidade de memória adicional ao vetor de entrada, que o algoritmo requer, é constante ou proporcional ao número de elementos. A complexidade de tempo $\Theta(n \log n)$ do MergeSort é ótima em relação aos algoritmos de ordenação por comparação, mas o espaço extra utilizado pelo algoritmo é $\Theta(n)$, enquanto que para o InsertionSort este espaço é constante, mas a complexidade de tempo é $\Theta(n^2)$. Um algoritmo capaz de ordenar um vetor com espaço constante é classificado como um *algoritmo de ordenação local*.

ShellSort é um algoritmo de ordenação por comparação de um vetor V de n elementos que executa o algoritmo InsertionSort em diferentes subsequências de V [16]. Tais subsequências são definidas por uma sequência de passos (inteiros) $p_1, p_2, \ldots p_k$ com $p_1 = 1$, $p_i < p_{i+1}$ para todo $1 \leq i < k$ e $p_k < n$. O algoritmo foi proposto por Donald L. Shell visando obter a eficiência assintótica de tempo ótima de $\Theta(n \log n)$ e ser um algoritmo de ordenação local [16]. Embora atualmente existam algoritmos que atendam a tais propriedades procuradas por Shell, como é o caso do HeapSort por exemplo, à época, um algoritmo com tais propriedades era desconhecido.

Frank e Lazarus demonstraram, no ano seguinte à proposição do ShellSort, que o pior caso resultante da sequência de Shell é de $\Theta(n^2)$ [5]. Esta descoberta abriu caminho para que diversos trabalhos estudassem outras sequências de passos, que resultaram em complexidades de pior caso de tempo tais como $\Theta(n^{3/2})$, $\Theta(n \log^2 n)$ e $O(n^{4/3})$ [5, 14, 15]. Alguns dos estudos mais recentes [2, 6] focaram na busca de metodologias para obtenção de sequências ótimas, porém as complexidades de pior caso das novas sequências propostas ainda estão em aberto. A Tabela 1 apresenta as diversas sequências estudadas na literatura com suas complexidades de pior caso analíticas. Para estudos mais recentes sobre o ShellSort, veja [7, 17, 18, 20].

Dentre todas as sequências estudadas até hoje, a sequência de Pratt [14] é a que possui menor complexidade de pior caso e complexidade muito próxima ao limite inferior obtido por Cypher [4] (a ser visto na Seção 2).

Em [9] é introduzida uma relação do ShellSort com o problema de Frobenius. Neste trabalho, enunciamos uma relação similar, porém com um limite mais justo. Através desta nova relação, foi possível elaborar um algoritmo capaz de obter o limite superior do número de comparações que o ShellSort executa para certa sequência de passos. Com este algoritmo e com o uso do EMA [11], é possível determinar complexidades de pior caso para qualquer sequência do ShellSort. O EMA é uma ferramenta que, a partir das informações de consumo de recursos de um algoritmo, em função de uma variável de entrada, produz a notação assintótica da complexidade deste algoritmo. O objetivo deste trabalho foi o de avaliar tal metodologia de obtenção da complexidade de pior caso, aplicando-a para as sequências que possuem complexidades de pior caso conhecidas. Como os resultados foram positivos, ou seja, as complexidades experimentais e teóricas coinci-

Tabela 1: Complexidade analítica para algumas sequências de passos.

Autor, Ano	**Passos** $\{p_1,\ldots,p_k\}$	**Pior caso**
Shell, 1959 [16]	$\{\lfloor n/2^i \rfloor : 0 < i \leq \lfloor \log_2 n \rfloor\}$ Ex.: $\{1,2,5,10,21\}$, para $n = 43$	$\Theta(n^2)$
Frank e Lazarus, 1960 [5]	$\{2 \cdot \lfloor n/2^i \rfloor + 1 : 1 < i \leq \lfloor \log_2 n \rfloor\}$ Ex.: $\{1,3,5,11,21\}$, para $n = 43$	$\Theta(n^{3/2})$
Hibbard, 1963 [8]	$\{2^i - 1 : 0 < i \leq \lfloor \log_2 n \rfloor\}$ $\{1,3,7,15,31,\ldots\}$	$\Theta(n^{3/2})$
Papernov e Stasevich, 1965 [12]	$\{1\} \cup \{2^{i-1} + 1 : 1 < i \leq \lfloor \log_2 n \rfloor\}$ $\{1,3,5,9,17,\ldots\}$	$\Theta(n^{3/2})$
Pratt, 1972 [14]	$\{q = 2^r 3^s : r,s \geq 0 \mid q < n\}$ $\{1,2,3,4,6,8,9,12,\ldots\}$	$\Theta(n \log^2 n)$
Knuth, 1973 [10]	$\{q = (3^i - 1)/2 : i > 0 \mid q \leq \lceil n/3 \rceil\}$ $\{1,4,13,40,121,\ldots\}$	$\Theta(n^{3/2})$
Incerpi e Sedgewick, 1983 [9]	$\{p_1 = 1\} \cup \{p_{i+1} = p_{i-r+1} q_r :$ $i > 0$, onde r tal que $\binom{r}{2} < i \leq \binom{r+1}{2}$, onde $\binom{r}{s} = 0$, se $r < s$, e q_r é o menor inteiro $\geq (5/2)^r$ tal que $\text{MDC}(q_r, q_{r-1}, \ldots, q_1) = 1\}$ $\{q_1 = 3, q_2 = 7, q_3 = 16, q_4 = 41, \ldots\}$ $\{1,3,7,21,48,\ldots\}$	$O\left(n^{1+\sqrt{\frac{8\ln(5/2)}{\ln n}}}\right)$
Sedgewick, 1986 [15]	$\{1\} \cup \{q = 4^i + 3 \cdot 2^{i-1} + 1 :$ $i > 0 \mid q < n\}$ $\{1,8,23,77,281,\ldots\}$	$O(n^{4/3})$
Gonnet e Baeza-Yates, 1991 [6]	$\{\lfloor (0,45454)^i \cdot n \rfloor :$ $0 < i \leq \lfloor \log_{2,2} n \rfloor\}$ Ex.: $\{1,3,8,19\}$, para $n = 43$	em aberto
Tokuda, 1992 [19]	$\{q = \lceil (9^i - 4^i)/(5 \cdot 4^{i-1}) \rceil :$ $i > 0 \mid q < n\}$ $\{1,4,9,11,26,\ldots\}$	em aberto
Ciura, 2001 [2]	$\{1,4,10,23,57,132,301,701,1750\}$ (sequência finita)	em aberto

diram, os estudos foram estendidos para as sequências com complexidade ainda desconhecida. Desta maneira, as complexidades obtidas para este último grupo servem de conjecturas para o esforço futuro de obtenção da complexidade dessas sequências, via método analítico.

A estrutura do trabalho é apresentada a seguir. Na Seção 2, descrevemos o algoritmo ShellSort, com seus limites gerais conhecidos e estudados pela literatura. Na Seção 3, apresentam-se conceitos que são base para diversos estudos do algoritmo ShellSort. Na Seção 4, é mostrado o problema de Frobenius e sua relação com o algoritmo ShellSort. Na Seção 5, apresenta-se uma reformulação desta relação, além da elaboração do algoritmo que determina um limite superior para a complexidade de pior caso do ShellSort para qualquer sequência. A Seção 6 apresenta o EMA, ferramenta utilizada para os experimentos deste trabalho. A Seção 7 apresenta a me-

todologia para a escolha dos parâmetros de execução do EMA e os critérios de avaliação dos resultados, estes presentes na Seção 8. Por fim, a Seção 9 contém uma análise dos resultados obtidos e comentários finais.

2 O ALGORITMO SHELLSORT

O algoritmo ShellSort consiste de uma generalização do algoritmo InsertionSort (Algoritmo 1). Para ordenação de um vetor, o InsertionSort usa a ideia de, à medida que percorre esse vetor da esquerda para a direita, inserir cada novo elemento examinado no conjunto à sua esquerda, tal que esse conjunto fique ordenado. O ShellSort consiste de diversas execuções de uma versão generalizada de InsertionSort. Tal generalização é apresentada pelo Algoritmo 2, no qual, ao invés de se supor que todos os elementos devam ser ordenados, apenas aqueles em posições indexadas por inteiros em um vetor S de entrada devem ser ordenados. O ShellSort é descrito pelo Algoritmo 3. A Tabela 2 apresenta uma comparação entre o ShellSort, com a sequência de passos $p_1 = 1, p_2 = 2, p_3 = 4$, e do InsertionSort, para um vetor de tamanho 6. A ideia fundamental do ShellSort é trazer mais rapidamente elementos para suas posições finais na ordenação, fazendo com que os elementos não consecutivos no vetor sejam comparados (e eventualmente trocados), através da introdução da noção de passos. Note que o InsertionSort apenas compara elementos adjacentes no vetor. Note também que, quando o passo é o $p_1 = 1$, o ShellSort equivale ao InsertionSort ordinário. Apesar de existirem instâncias para as quais o ShellSort executa um número de comparações superior ao do InsertionSort, em geral ele consegue um número inferior delas, como pode ser visto na Tabela 2.

Algoritmo 1: Algoritmo InsertionSort.

Dados: $V[1..n]$ *de inteiros*
Resultado: $V[i] \leq V[i+1]$, *para todo* $1 \leq i < n$
para $i \leftarrow 2$ **até** n **faça**
    $j \leftarrow i-1$
    enquanto $j > 0$ **e** $V[j+1] < V[j]$ **faça**
        $V[j], V[j+1] \leftarrow V[j+1], V[j]$
        $j \leftarrow j-1$

Algoritmo 2: Algoritmo InsertionSort para Subsequências.

Dados: $V[1..n]$ *de inteiros,* $S[1..n']$ *com sequência crescente de índices de* V
Resultado: $V[S[i]] \leq V[S[i+1]]$, *para todo* $1 \leq i < n'$
para $i \leftarrow 2$ **até** n' **faça**
    $j \leftarrow i-1$
    enquanto $j > 0$ **e** $V[S[j+1]] < V[S[j]]$ **faça**
        $V[S[j]], V[S[j+1]] \leftarrow V[S[j+1]], V[S[j]]$
        $j \leftarrow j-1$

Algoritmo 3: Algoritmo ShellSort.

Dados: $V[1..n]$ *de inteiros, sequência de passos* $p_1, \ldots, p_k$
Resultado: $V[i] \leq V[i+1]$, *para todo* $1 \leq i < n$
para $j \leftarrow k$ **até** 1 **incremento** -1 **faça**
  para $i \leftarrow 1$ **até** p_j **faça**
    InsertionSort($S_{i,j}$), sendo $S_{i,j}$ o vetor que consiste dos elementos de V presentes nos índices $i, i+p_j, i+2p_j, \ldots, i+\left\lfloor \frac{n-i}{p_j} \right\rfloor p_j$

Tabela 2: Funcionamento dos Algoritmos 1 e 3, sendo este último para os passos $p_1 = 1$, $p_2 = 2$ e $p_3 = 4$. Comparações estão destacadas em sublinhado e trocas destacadas em negrito. Para o mesmo vetor de entrada V, o ShellSort faz 11 comparações e 3 trocas, enquanto InsertionSort faz 15 comparações e 12 trocas.

	ShellSort						**InsertionSort**					
$p_3 = 4$	**5**	6	1	4	**3**	2	5	6	1	4	3	2
	3	**6**	1	4	5	**2**	5	**6**	**1**	4	3	2
$p_2 = 2$	**3**	2	**1**	4	5	6	**5**	**1**	6	4	3	2
	1	2	3	4	5	6	1	5	**6**	**4**	3	2
	1	2	3	4	5	6	1	**5**	**4**	6	3	2
	1	2	3	4	5	6	1	4	5	6	3	2
$p_1 = 1$	1	2	3	4	5	6	1	4	5	**6**	**3**	2
	1	2	3	4	5	6	1	4	**5**	**3**	6	2
	1	2	3	4	5	6	1	**4**	**3**	5	6	2
	1	2	3	4	5	6	1	3	4	5	6	2
	1	2	3	4	5	6	1	3	4	5	**6**	**2**
							1	3	4	**5**	**2**	6
							1	3	**4**	**2**	5	6
							1	**3**	**2**	4	5	6
							1	2	3	4	5	6

O algoritmo ShellSort possui um limite inferior para sua complexidade de pior caso geral, enunciada por Cypher [4] e apresentada no Teorema 2.1, cuja demonstração não será mostrada por ser muito extensa.

Teorema 2.1 (Cypher [4]). *O número de comparações realizadas no pior caso pelo ShellSort é de* $\Omega(n \log^2 n / \log\log n)$.

Um limite superior para o pior caso geral também pode ser estabelecido através do Lema 2.1.

Lema 2.1 (Knuth [10]). *A complexidade de pior caso do algoritmo ShellSort, para qualquer sequência de passos $p_1, \ldots, p_k$ é de $O\Big(n^2 \sum_{i=1}^{k} \frac{1}{p_i}\Big)$.*

Proof. Para cada p_i, com $i = 1, \ldots, k$, há p_i execuções de InsertionSort em subsequências de tamanho $O(n/p_i)$. Como a complexidade de pior caso do InsertionSort é quadrática no número de elementos sendo ordenados, temos que, se $t(n)$ é a complexidade de pior caso do ShellSort, então

$$t(n) = \sum_{i=1}^{k} p_i O((n/p_i)^2) = O\Big(\sum_{i=1}^{k} p_i (n/p_i)^2\Big) = O\Big(n^2 \sum_{i=1}^{k} 1/p_i\Big)$$

□

Note que, para cada i = 1, ..., k, o número total de operações para executar as p_i execuções de InsertionSort é $\Omega(n)$. Assim, se o objetivo for o de obter complexidade de tempo $O(n \log n)$ para o ShellSort, é necessário que o número de passos seja tal que $k = O(\log n)$. Por isso, tal restrição no valor de k é usual. O Teorema 2.2 desenvolvido neste trabalho constitui um limite superior de tempo de pior caso para qualquer sequência do ShellSort com $O(\log^c n)$ passos. Observe que todas as sequências apresentadas na Tabela 1 possuem tal limite no número de passos.

Teorema 2.2. *A complexidade de pior caso do algoritmo ShellSort para qualquer sequência com $O(\log^c n)$ passos, para qualquer constante c, é de $O(n^2 \log\log n)$.*

Proof. Pelo Lema 2.1, a complexidade de pior caso $t(n)$ para qualquer sequência de ShellSort é de

$$t(n) = O\Big(n^2 \sum_{i=1}^{k} \frac{1}{p_i}\Big) = O\Big(n^2 \sum_{i=1}^{k} \frac{1}{i}\Big)$$

O somatório $\sum_{i=1}^{k} \frac{1}{i}$ corresponde ao número harmônico $H_k = \Theta(\log k)$. Substituindo na expressão anterior, temos

$$t(n) = O(n^2 \log k) = O(n^2 \log(\log^c n)) = O(n^2 \log\log n)$$

□

3 CONCEITOS FUNDAMENTAIS DO SHELLSORT

Algumas propriedades são importantes para as análises das complexidades de caso médio e pior caso de várias das sequências do ShellSort. O Teorema 3.3 descreve uma dessas propriedades mais relevantes. Os Teoremas 3.4 e 3.5 envolvem conceitos mais triviais amplamente utilizados na literatura, normalmente não justificados de maneira formal. Supriremos aqui essa lacuna. Dizemos que um vetor V está *k-ordenado* se $V[i] \leq V[i+k]$, para todo $1 \leq i < (n-k)$ e $k \geq 1$.

Teorema 3.3 (Poonen [13], adaptado). *Se um vetor V que está k-ordenado for h-ordenado pelo ShellSort pelo processamento do passo igual a h, V permanecerá k-ordenado.*

Proof. Considere V um vetor k-ordenado. Para a demonstração que se segue, supõe-se que o elemento do vetor V indexado fora dos limites de V é ∞. Como V é k-ordenado, para qualquer posição i, vale que $V[i] \leq V[i+k]$ e $V[i+h] \leq V[i+h+k]$, pela k-ordenação. Queremos mostrar que, a cada comparação (e potencialmente troca) durante a h-ordenação, V se mantém k-ordenado. De fato, se L é o vetor V corrente e L' é o vetor V após uma operação de comparação e eventual troca entre as posições i e $i+h$, temos que

$$\begin{array}{lllll} L'[i] & = \min\{L[i], L[i+h]\} & \leq \min\{L[i+k], L[i+h+k]\} & = L'[i+k] \\ L'[i+h] & = \max\{L[i], L[i+h]\} & \leq \max\{L[i+k], L[i+h+k]\} & = L'[i+h+k] \end{array}$$

Logo, após a h-ordenação, o vetor V permanece k-ordenado. □

Teorema 3.4. *Se um vetor está tanto k-ordenado quanto h-ordenado, então ele também está $(ak+bh)$-ordenado, para quaisquer $a, b \in \mathbb{N}^*$.*

Proof. Para um vetor V k-ordenado, temos a seguinte propriedade: $V[i] \leq V[i+k]$, para $i = 1, \ldots, n-k$. Para a mesma posição i, temos também que $V[i] \leq V[i+ak]$, para qualquer $a \in \mathbb{N}$. Sem perda de generalidade, considere $h < k$. Como V é k-ordenado e h-ordenado, temos que $V[i] \leq V[i+ak] \leq V[i+ak+bh]$, para $i = 1, \ldots, n-ak-bh$ e para $a, b \in \mathbb{N}^*$, ou seja, V é $(ak+bh)$-ordenado. □

O Teorema 3.5 justifica a definição de que $p_1 = 1$, ao invés de permitir valores arbitrários para p_1.

Teorema 3.5. *Toda sequência do algoritmo ShellSort necessita que $p_1 = 1$ para garantir que o vetor resultante esteja ordenado.*

Proof. Suponha que $p_1 > 1$ e tome como vetor inicial V a ser ordenado um vetor de naturais distintos de modo que os dois menores elementos estão nas duas primeiras posições e tal que $V[2] < V[1]$. Note que, seja qual for a sequência de passos, $V[1]$ e $V[2]$ não serão comparados. Além disso, como $V[1]$ e $V[2]$ serão mínimos em toda subsequência considerada por cada conjunto de passos onde tais elementos estão presentes, eles permanecerão em suas posições iniciais até o fim do algoritmo. Portanto, o passo $p_1 = 1$ para esta instância é necessário. □

4 PROBLEMA DE FROBENIUS

O problema de Frobenius é aquele de determinar o maior valor monetário que não pode ser gerado usando moedas de determinados valores, considerando disponibilidade infinita de moedas de cada valor [1]. Por exemplo, para moedas de 3 e 5 centavos, o valor máximo que não pode ser obtido com estas é de 7 centavos. Tal número é chamado de *número de Frobenius*. Formalmente, considere o conjunto $A = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{N}$, $|A| > 1$, tal que $a_1, a_2, \ldots, a_n > 1$ e $\text{MDC}(a_1, a_2, \ldots, a_n) = 1$. Seja $F(A) \subset \mathbb{N}$ o conjunto dos números x para os quais não existem $k_1, k_2, \ldots, k_n \in \mathbb{N}$ de modo que $k_1a_1 + k_2a_2 + \ldots + k_na_n = x$, isto é, x não pode ser representado

como uma combinação linear dos elementos de A. O *número de Frobenius*, representado por $g(A)$, equivale ao elemento máximo de $F(A)$. O problema de determinar $g(A)$ possui solução algébrica para $|A| = 2$, a saber, $g(\{a_1, a_2\}) = a_1a_2 - (a_1 + a_2)$, mas sua solução algébrica é um problema em aberto quando $|A| \geq 3$.

Elaboramos um algoritmo pseudo-polinomial para determinar $g(A)$ utilizando uma variação do algoritmo da mochila com itens infinitos (Algoritmo 4). Mais especificamente, dados um natural t e $A = \{a_1, a_2, \ldots, a_n\}$, é possível determinar $g(A,t) = \max\{x \in F(A) : x \leq t\}$. Para determinar $g(A)$, utilizamos o limite $t = g(\{a_1, a_2\})$. Um exemplo de execução do algoritmo é ilustrado pela Tabela 3, retornando $g(\{5,7,9\}, g(\{5,7\})) = g(\{5,7,9\}, 23) = 13$.

Algoritmo 4: Algoritmo para determinação de $g(A)$, com $g(A) \leq t$.

```
Dados: Conjunto A = {a_1, a_2, ..., a_n} e natural t ≥ g(A)
Resultado: g(A,t) = max{f ∈ F(A) : f ≤ t}
V[1..t] ← FALSO ; V[0] ← VERDADEIRO
para i ← 1 até n faça
    para j ← a_i até t faça
        se não V[j] e V[j − a_i] então
            V[j] ← VERDADEIRO
para i ← t até 1 incremento −1 faça
    se não V[i] então
        retorna i
```

Tabela 3: Funcionamento do Algoritmo 4 com $n = 3$, $t = 23$ e $A = \{5,7,9\}$.

	0	1	2	3	4	5	6	7	8	9	10	11	12
Início	V	F	F	F	F	F	F	F	F	F	F	F	F
5	V	F	F	F	F	V	F	F	F	F	V	F	F
7	V	F	F	F	F	V	F	V	F	F	V	F	V
9	V	F	F	F	F	V	F	V	F	V	V	F	V
	13	14	15	16	17	18	19	20	21	22	23		
Início	**F**	F	F	F	F	F	F	F	F	F	F		
5	**F**	F	V	F	F	F	F	V	F	F	F		
7	**F**	V	V	F	V	F	V	V	F	V	F		
9	**F**	V	V	V	V	V	V	V	V	V	V		
Retorno	**13**												

A relação entre o problema de Frobenius e o algoritmo ShellSort foi estudada pela primeira vez por Incerpi e Sedegwick [9]. No trabalho de Weiss, esta relação é explicitada através do Teorema 4.6 de [21]. O Lema 4.2, no qual este teorema se baseia, utiliza a notação $m_d(A)$, quantidade de múltiplos de d pertencentes ao conjunto $F(A)$.

Lema 4.2 (Incerpi e Sedgewick [9], adaptado). *Durante a execução do ShellSort, o número máximo de comparações para p_s-ordenar um vetor já $p_k, p_{k-1}, \ldots, p_{s+1}$-ordenado é $O(nm_{p_s}\{p_k, p_{k-1}, \ldots, p_{s+1}\})$.*

Proof. O número de operações necessário para inserir o elemento $V[i]$ é o número de elementos entre as posições $i-p_s, i-2p_s, \ldots$ que são maiores que $V[i]$. Qualquer elemento $V[i-x]$, onde $x \notin F(\{p_k, p_{k-1}, \ldots, p_{s+1}\})$ deve ser menor que $V[i]$, já que o vetor já está $p_k, p_{k-1}, \ldots, p_{s+1}$-ordenado. Logo, o número máximo de comparações para inserir $V[i]$ é o número dos múltiplos de p_s pertencentes ao conjunto $F(\{p_k, p_{k-1}, \ldots, p_{s+1}\})$. □

Lema 4.3 (Incerpi e Sedgewick [9]). *Para qualquer $A \subset \mathbb{N}$, $\mathrm{MDC}(A) = 1$, $m_d(A) \leq g(A)/d$.*

Proof. Todo número maior que $g(A)$ pode ser representado como combinação linear de A. No pior caso, todos os múltiplos de d menores que $g(A)$ não podem. □

Teorema 4.6 (Weiss [21]). *O número máximo de comparações para p_s-ordenar um vetor, onde p_s é um dos passos da sequência $p_1, \ldots, p_k$ de passos, que já está $p_k, p_{k-1}, \ldots, p_{s+1}$-ordenado é $O(ng(\{p_k, p_{k-1}, \ldots, p_{s+1}\}))$.*

Proof. Consequência direta dos Lemas 4.2 e 4.3. □

5 ALGORITMO DE LIMITE SUPERIOR PARA O NÚMERO DE COMPARAÇÕES

Nesta secção, apresentamos o Teorema 5.7, similar ao Teorema 4.6, porém trocando a notação assintótica pelo número preciso de comparações. Isso possibilitará derivar um limite superior mais justo para o Algoritmo 5, apresentado na sequência, que determina o número máximo de comparações do ShellSort usando qualquer sequência de passos.

Para o Teorema 5.7, será útil considerar outra variação da função $g(A)$. Definimos

$$g_d(A,t) = \max\{x \in \mathbb{N} : x \in F(A), x \leq t, x \bmod d = 0\}$$

Desta forma, enquanto $m_d(A)$ representa a cardinalidade dos elementos múltiplos de d em $F(A)$, $g_d(A,t)$ representa o maior elemento de $F(A)$ múltiplo de d e menor que t.

A relação entre o ShellSort e o número de Frobenius foi apresentada pela primeira vez em [9] e estudada mais aprofundadamente em [21]. Neste último, um teorema semelhante ao Teorema 5.7 é apresentado. Neste trabalho, modificamos tal teorema trocando a notação assintótica por um limite de comparações numérico para utilização no Algoritmo 5, um algoritmo que calcula um número máximo de comparações em um vetor de tamanho n a ser ordenado por ShellSort com uma sequência de passos $p_1, \ldots, p_k$.

Teorema 5.7. *Durante a ordenação do vetor V pelo ShellSort com sequência de passos $p_1, \ldots, p_k$, o número de comparações realizadas na iteração relativa ao passo p_s para cada elemento de V é, no máximo, $g_{p_s}(\{p_k, p_{k-1}, \ldots, p_{s+1}\}, n)/p_s + 1$.*

Proof. Uma consequência do Teorema 3.4 é que se um vetor está $p_k, p_{k-1}, \ldots, p_{s+1}$-ordenado, então também está $(m_1 p_k + m_2 p_{k-1} + \ldots + m_{k-s} p_{s+1})$-ordenado, para quaisquer $m_1, m_2, \ldots, m_{k-s} \in \mathbb{N}^*$. Portanto, para qualquer $j \notin F(A)$, com $A = \{p_k, p_{k-1}, \ldots, p_{s+1}\}$, temos que $V[i] \leq V[i+j]$, para qualquer posição i. Para $j \in F(A)$, a relação $V[i] \leq V[i+j]$ pode não valer. Logo, no ordenação com o passo p_s, são comparados apenas os pares $(V[i], V[i+tp_s]), t \in \mathbb{N}^*$, para qualquer posição i, ou seja, são vistos intervalos de elementos de posições com distância múltipla de p_s. Com isto, o maior valor x de modo que a relação $V[i] \leq V[i+x]$ possa ser falsa é $g_{p_s}(\{p_k, p_{k-1}, \ldots, p_{s+1}\}, n)$. Portanto, o número máximo de comparações a serem realizadas para um determinado elemento de V utilizando o passo p_s é $g_{p_s}(\{p_k, p_{k-1}, \ldots, p_{s+1}\}, n)/p_s + 1$. □

O Algoritmo 5 utiliza o Teorema 5.7 para determinar um limite superior para o número de comparações realizadas em um vetor, para qualquer sequência de passos.

Algoritmo 5: Determinação de um limite superior de pior caso do ShellSort.

Dados: *Sequência de passos* $p_1, p_2, \ldots, p_k$ *e tamanho n do vetor V*
Resultado: *Número máximo de comparações realizadas em um vetor de tamanho n com os passos* $p_1, p_2, \ldots, p_k$

```
ncomp ← 0
para s ← k até 1 incremento −1 faça
    limite ← ⌊g_{p_s}(p_{s+1}, ..., p_k, n)/p_s⌋ + 1
    para i ← 1 até n faça
        limite_i ← ⌊(n − i)/p_s⌋
        ncomp ← ncomp + min{limite, limite_i}
retorna ncomp
```

Para cada passo da sequência considerada, o algoritmo calcula o limite das comparações de cada elemento do vetor, de acordo com o Teorema 5.7. Tal limite leva em conta todos os passos anteriores já executados. Para cada elemento do vetor é calculado, também, um limite natural de comparações, com os elementos à sua direita e considerando o valor do passo. Desta maneira, toda vez que um elemento do vetor é considerado, adota-se o menor dos dois limites. Portanto, com a utilização deste algoritmo, é possível conseguir um limite superior da complexidade de pior caso para qualquer sequência aplicada ao algoritmo ShellSort. Não se tem garantia, entretanto, de que o limite seja exatamente o pior caso.

Este algoritmo possui complexidade $O(kn)$, onde k é o número de passos da sequência utilizada. Como as sequências estudadas são tais que $k = O(\log^c n)$, para $c \in \mathbb{N}$, então a complexidade do algoritmo é de $O(n \log^c n)$.

6 EMA

A ferramenta de análise de algoritmos EMA [11] tem como objetivo analisar empiricamente a complexidade de algoritmos, tomando como base execuções iterativas sobre a implementação

deste algoritmo, armazenando os respectivos tempos de execução, memórias alocadas e as quantidades de outros recursos personalizados que a aplicação requeira monitoramento, em função de uma variável de entrada do algoritmo.

Para a análise de um algoritmo pela ferramenta EMA, precisa-se implementar dois programas: (i) o algoritmo a ser analisado e (ii) um *script* de geração de entradas para tal algoritmo. Desta maneira, o EMA pode executar primeiramente o *script*, para gerar a entrada para a execução do programa cujo algoritmo está sendo analisado. Tais execuções são feitas iterativamente variando-se valores de um ou mais parâmetros de entrada do algoritmo, os quais são chamados de *variáveis*. No caso do ShellSort, apenas a variável n foi definida, correspondendo ao número de elementos no vetor. Uma vez construída uma base de dados dos recursos utilizados pelo algoritmo de acordo com cada valoração das variáveis, a ferramenta atribui funções que estima o consumo de cada recurso (tempo, espaço, ou outro particular) em função das variáveis. Uma execução da ferramenta pode ser dividida em três etapas: calibração, simulação e análise, que são descritas a seguir.

Na etapa de calibração, a ferramenta recebe um *script* $\mathscr{A}$ que tem o propósito de gerar entradas para o algoritmo a ser testado. A entrada de $\mathscr{A}$, por sua vez, corresponde a uma valoração específica das variáveis, e sua saída corresponde a uma entrada do algoritmo em análise que possui os valores informados das variáveis. Por exemplo, no caso do ShellSort, $\mathscr{A}$ recebe um dado valor de n e a sequência de passos $p_1, \ldots, p_k$ e simplesmente escreve um arquivo cujo conteúdo consiste desses valores, para que o algoritmo de obtenção do limite superior de número de comparações possa ser executado e retornar o valor calculado. Note que, apesar de não aparentar agregar muito, a função de $\mathscr{A}$ é permitir que o EMA gere entradas para qualquer programa, sem conhecer os detalhes de como o programa de fato exige a formatação de suas entradas (via arquivo, via parâmetro de linha de comando, via entrada padrão, etc.). Além de $\mathscr{A}$, especificam-se limites inferior e superior tanto do tempo de execução, quanto dos valores das variáveis e do consumo de memória. Com estes limites e com $\mathscr{A}$, a ferramenta está apta a fazer experimentações de execução automáticas e determinará, como resultado da calibração, um intervalo de valores de variáveis tais que todas as execuções satisfaçam os limites especificados de tempo, valores de variáveis e consumo de memória. Determina-se assim o intervalo $[n_{\min}; n_{\max}]$ no qual qualquer execução de simulação com $n_{\min} \leq n \leq n_{\max}$ respeita as restrições impostas (e para valorações próximas aos extremos do intervalo, alguma restrição de mínimo ou máximo está próxima de ser violada). Dentro deste intervalo, a ferramenta escolherá q pontos a serem simulados com o objetivo de se armazenar dados de consumo de recursos para posterior análise.

Na etapa de simulação, o conjunto de q pontos escolhidos na etapa anterior são processados em sequência para a coleta de dados, medindo o consumo dos recursos. A ferramenta EMA mede, automaticamente, o tempo de execução e a memória alocada pelo programa. Para recursos personalizados, o programa escreve um arquivo com tais contadores de recursos personalizados, que são importados pela ferramenta. Para a simulação, especifica-se o número de amostras de cada valor de variável, que pode ser fixo ou baseado num fator de convergência da média amostral. Neste trabalho, o critério de avaliação dos resultados obtidos é o número máximo

de comparações. Como tal parâmetro é determinístico, então apenas uma amostra é necessária. Para outras medições não determinísticas entretanto, a ferramenta armazena, para cada valor de variável simulada, como estatística para cada recurso, os valores máximo e mínimo, a média amostral e desvio padrão da amostra.

Finalmente, há a etapa de análise. Para esta etapa, especifica-se o recurso a ser analisado e o tipo de análise (melhor, pior ou caso médio). Dadas tais especificações, a partir da base de execuções, seja $f(x)$ a função de consumo de tal recurso no tipo de análise escolhida. A função $f(x)$ tem por domínio o conjunto de q pontos escolhidos para a simulação e como imagem a medição dos respectivos recursos consumidos. A fase de análise objetiva estimar $f(x)$ por uma função contínua $\hat{f}(x)$, de modo que seja possível fazer previsões de consumo para outras valorações de variáveis.

A metodologia geral do EMA para obter $\hat{f}$ é como se segue. Primeiro, assume-se que tal função deva ter o formato geral, onde a_0 e $a_2 > 0$,

$$f_{\text{geral}}(x) = a_0 x^{a_1} a_2^{x^{a_3} \log^{a_4} x} \log^{a_5} x + a_6$$

onde x é a variável do algoritmo (n, no caso do ShellSort) e a_i uma constante real para todo $1 \leq i \leq 6$. Informalmente, o que se procura é ajustar o valor de tais parâmetros via um método numérico de modo que o erro seja pequeno (não necessariamente mínimo, como veremos adiante). Note que a hipótese de que este é o formato geral da função contínua da qual $f(x)$ é um subconjunto está fundamentada em dois pontos: (i) o que se procura é determinar, ao invés da função que estima de maneira precisa o consumo em qualquer ponto de execução, a função de consumo de recursos para valores da variável suficientemente grandes e, desde modo, a função procurada é aproximada por apenas um termo (o de maior crescimento assintótico), e (ii) este formato abrange um amplo espectro de complexidades de algoritmos comumente encontrados na literatura.

Para fazer tal ajuste de constantes (as quais são chamadas de *parâmetros* neste contexto), definiremos o *erro* $e(\hat{f}, f)$ associado a um conjunto de valores para tais parâmetros (que definem uma estimação $\hat{f}$) em relação a uma função medida f como

$$e(\hat{f}, f) = \sum_{(x,y) \in f} (\hat{f}(x) - y)^2$$

O bem conhecido método dos mínimos quadrados é uma classe de algoritmos numéricos que visa ajustar os parâmetros de modo a minimizar o valor da função e. A aplicação deste método diretamente para se obter a complexidade de algoritmos não conduz a resultados satisfatórios pois, em geral, todos os parâmetros assumem valores de modo a minimizar o erro, falhando em detectar oscilações de consumo de recursos que são peculiares aos algoritmos. Como exemplo, suponha que estejamos tentando medir a complexidade de tempo de um algoritmo de tempo $\Theta(n^2)$, para alguma variável n de entrada deste problema. Neste caso, pequenas oscilações no tempo de execução fazem com que os tempos de execução estejam ligeiramente fora da parábola ideal que determinaria tal valor, resultando que o ajuste de parâmetros falhe em detectar exatamente o conjunto de parâmetros que resultam na complexidade correta (a quadrática). Porém, é esperado

que, se e_P e e_G são, respectivamente, os erros associados à função da parábola e f_{geral}, então e_P estaria próximo de e_G. No entanto, o último, por possuir uma quantidade maior de parâmetros, faria com que $e_G < e_P$. Portanto, na metodologia do EMA, o formato geral f_{geral} é simplificado de todas as possíveis formas, entendendo-se por simplificar a eliminação de ao menos um parâmetro do formato geral. Cada um destes formatos são ajustados via aplicação de método de mínimos quadrados e as funções correspondentes estimadas são consideradas candidatas à solução, classificadas em três grupos:

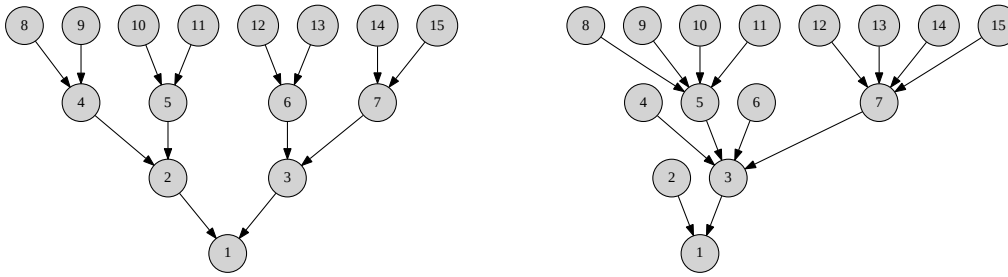
- **Função de erro mínimo:** função, dentre todas as candidatas, com o menor erro;
- **Funções equivalentes:** funções cujos erros não ultrapassam um limite percentual do erro da função de erro mínimo (tipicamente, 0.5%). Funções deste grupo são consideradas, para efeitos práticos, equivalentes por não diferirem significativamente para o conjunto de pontos testados;
- **Melhor-palpite:** função que faz parte do grupo de funções equivalentes e classificada, dentro dos critérios do EMA, como a mais provável de corresponder à real. A ideia que motiva os critérios do EMA nesta escolha é conhecida como navalha de Occam (*"Among competing hypotheses, the one with the fewest assumptions should be selected."*). No exemplo do algoritmo de complexidade de tempo quadrática anterior, a função ajustada a partir do formato da quadrática seria escolhida, ao invés daquela ajustada a partir de f_{geral} mesmo com erro maior, desde que suficientemente próximo.

Após a etapa de análise, a ferramenta apresenta as funções retornadas dentro de cada grupo. Ao final da apresentação das funções, o EMA encerra seu funcionamento, cabendo ao usuário analisar os resultados. Para a simulação do ShellSort, decidimos que o recurso de interesse é "Número de Comparações entre Elementos" e que devemos dali extrair a função classificada como melhor-palpite. Na Seção 8, veremos os retornos do EMA para cada sequência e uma análise sobre os resultados obtidos.

7 DESCRIÇÃO DOS EXPERIMENTOS

Os experimentos foram conduzidos com o EMA. Como foi visto na Seção 6, uma execução no EMA consiste de três etapas: calibração, simulação e análise. As etapas de simulação e análise são utilizadas integralmente como descrito. Contudo, a etapa de calibração, que é a determinação do conjunto de valorações de entrada a serem simuladas, é feita por método próprio devido às particularidades do problema sendo estudado.

A escolha dos valores do número n de elementos do vetor de entrada, na substituição da etapa de calibração do EMA, é feita conforme o tipo da sequência de passos a ser simulada. Uma sequência de passos é dita *uniforme* se esta pode ser definida por uma série infinita [4]. A sequência de valores de n escolhida para as sequências uniformes coincide com a própria sequência de passos. Por exemplo, para a sequência de Hibbard [8], cujos passos são



(a) Shell [16] (b) Frank e Lazarus [5]

Figura 1: Árvore de possibilidades para as sequências (a) Shell e (b) Frank e Lazarus.

$1,3,7,15,31,\ldots$, tais valores serão também os valores para n a serem simulados. Isto é feito desta forma, pois essa escolha faz com que o número de passos efetivamente utilizados (aqueles que são inferiores a n) também incrementem com o incremento de n. Por exemplo, para $n = 7$ o conjunto de passos efetivamente utilizados da sequência de Hibbard será $\{1,3\}$, enquanto para $n = 15$ tal conjunto será $\{1,3,7\}$. Para a sequência de Ciura, cujo número de termos é finito, utilizamos a recursão $p_k = \lfloor 2,25p_{k-1} \rfloor$, para todo $k > 9$, a fim de transformá-la em uma sequência uniforme.

Para sequências não-uniformes, nas quais os passos podem ser definidos em função dos passos seguintes e o maior passo em função de n, a escolha será feita através das *árvores de possibilidades* para determinada sequência. Nestas árvores, que são digrafos, cada vértice é um natural e (i, j) é uma aresta se o passo i é imediatamente precedido pelo passo j em sequências daquele tipo. As Figuras 1 e 2 ilustram as árvores de possibilidades para três sequências não-uniformes. A propriedade principal de tais árvores é que, se o maior passo é certo valor p_k, o caminho de p_k até 1 é uma sequência $p_k, p_{k-1}, \ldots, p_1$, com $p_1 = 1$, dentro desta classe de sequências. Por exemplo, para a sequência de Shell, se o maior passo é 10, então a sequência inteira é $1,2,5,10$. Se o maior passo for 13, a sequência completa é $1,3,6,13$.

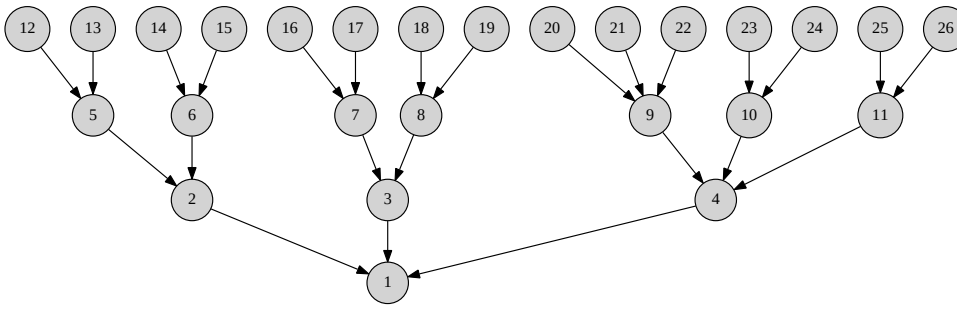


Figura 2: Árvore de possibilidades para a sequência de Gonnet e Baeza-Yates [6].

Nos experimentos, a escolha dos valores de n para as sequências não-uniformes foram tais que as sequências geradas consistem dos ramos mais à esquerda e mais à direita de sua árvore de possibilidades. Por exemplo, no caso de Shell, os valores de n utilizados foram $n = 2,4,8,16,\ldots$ e $n = 3,7,15,31,\ldots$ ($n = 1$ oferece resultados triviais). Estes ramos foram escolhidos de forma arbitrária. Entretanto, o cálculo destes ramos tende a ser mais previsível que os de outros.

Com os valores para n escolhidos, como explicado acima, o Algoritmo 5 retorna, para cada valor de n, o número máximo de comparações entre elementos em um vetor de tamanho n. Tal número de comparações é contabilizado no EMA como um recurso personalizado. Como este algoritmo é determinístico, isto é, o número de comparações de saída é invariante sob diferentes execuções com o mesmo valor de n, foi configurada apenas uma amostra no EMA por valor de n.

Na etapa de análise, o EMA determina as curvas que melhor se ajustam (como explicado na Seção 6) ao conjunto de pontos que representam o número de comparações por valor de n, para cada sequência. No trabalho, será considerada como função reportada pelo EMA aquela classificada como melhor-palpite. Na Seção 8, são apresentados os resultados dos experimentos para cada sequência e analisam-se as complexidades empíricas obtidas, em comparação com as conhecidas da literatura.

8 ANÁLISE DOS RESULTADOS

A Tabela 4 apresenta as complexidades analíticas e experimentais de pior caso do algoritmo ShellSort para cada sequência estudada. Para as complexidades experimentais, as comparações são aquelas retornadas pelo Algoritmo 5 considerando o conjunto selecionado de valores n. Em caso de sequências não-uniformes, as quais admitem mais de uma possibilidade de sequência de passos, a complexidade empírica considerada para tal sequência é a maior associada a cada um dos conjuntos. Por exemplo, a sequência de Shell, que é não-uniforme, foi testada para dois conjuntos de passos, que resultaram nas complexidades de $O(n^2)$ e $O(n^{1,5})$. A complexidade empírica de pior caso então resultante é considerada ser $O(n^2)$.

O resultado para a sequência de Shell concorda com o da literatura, descrevendo um comportamento quadrático do algoritmo para n como potências de 2. Para as sequências de Frank e Lazarus, Hibbard, Papernov e Stasevich, Pratt e Knuth, os resultados também concordam com o estudo analítico.

A complexidade de pior caso do ShellSort para a sequência de Incerpi e Sedgewick é $O\left(n^{1+\sqrt{8\cdot\frac{\ln 5/2}{\ln n}}}\right)$. Como tal função não se encaixa no modelo de função apresentado na Seção 6, tal complexidade foi simplificada para $O(n^{1,698})$, substituindo-se no expoente o maior valor de n considerado. Observe que, como a literatura apresenta apenas um limite superior para a complexidade e que o Algoritmo 5 também, o resultado sugere que a complexidade da literatura não está muito folgada em relação à real, embora permaneça em aberto seu valor exato. Para a sequência de Sedgewick, o EMA retornou complexidade de pior caso com uma casa decimal de precisão em

Tabela 4: Tabela de complexidades retornadas pelo EMA para cada sequência.

Sequência	**Analítico**	**Empírico**
Shell, 1959 [16]	$\Theta(n^2)$	
$\{1,2,4,8,16,32,64,128,256,512,\ldots\}$		$O(n^2)$
$\{1,3,7,15,31,63,127,255,511,1023,\ldots\}$		$O(n^{1,5})$
Frank e Lazarus, 1960 [5]	$\Theta(n^{1,5})$	
$\{1,3,5,9,17,33,65,129,257,513,\ldots\}$		$O(n^{1,5})$
$\{1,3,7,15,31,63,127,255,511,1023,\ldots\}$		$O(n^{1,5})$
Hibbard, 1963 [8]	$\Theta(n^{1,5})$	
$\{1,3,7,15,31,63,127,255,511,1023,\ldots\}$		$O(n^{1,5})$
Papernov e Stasevich, 1965 [12]	$\Theta(n^{1,5})$	
$\{1,3,5,9,17,33,65,129,257,513,\ldots\}$		$O(n^{1,5})$
Pratt, 1972 [14]	$\Theta(n\log^2 n)$	
$\{1,2,3,4,6,8,9,12,16,18,\ldots\}$		$O(n\log^2 n)$
Knuth, 1973 [10]	$\Theta(n^{1,5})$	
$\{1,4,13,40,121,364,1093,3280,9841,29524,\ldots\}$		$O(n^{1,5})$
Incerpi e Sedgewick, 1983 [9]	$O(n^{1,597})$	
$\{1,3,7,21,48,112,336,861,1968,4592,\ldots\}$		$O(n^{1,411})$
Sedgewick, 1986 [15]	$O(n^{1,333\ldots})$	
$\{1,8,23,77,281,1073,4193,16577,65921,\ldots\}$		$O(n^{1,329})$
Gonnet e Baeza-Yates, 1991 [6]	em aberto	
$\{1,2,5,12,27,60,133,293,645,1420,\ldots\}$		$O(n^{1,145})$, $O(n\log^3 n)$
$\{1,4,11,26,59,132,292,644,1419,3124,\ldots\}$		$O(n^{1,129})$ $O(n\log^3 n)$
Tokuda, 1992 [19]	em aberto	
$\{1,4,9,20,46,103,233,525,1182,2660,\ldots\}$		$O(n^{1,255})$, $O(n\log^{5,5} n)$
Ciura, 2001 [2]	em aberto	
$\{1,4,10,23,57,132,301,701,1750,3937,\ldots\}$		$O(n^{1,372})$

relação àquela da literatura. Levando-se em conta que trata-se de um método empírico, podemos dizer que a complexidade empírica concorda com a analítica também para esta sequência.

Com resultados em grande parte em conformidade com os resultados conhecidos, apresentamos conjecturas para as complexidades de pior caso ainda em aberto, que são os três últimos da Tabela 4. Mais precisamente, para a sequência de Gonnet e Baeza-Yates, as complexidades obtidas pelo método empírico foram $O(n^{1,145})$ e $O(n\log^3 n)$. Aqui, optou-se em obter do EMA não só a função melhor-palpite, mas também uma outra dentro do grupo de funções equivalentes. A razão para esta escolha é que estes dois formatos funcionais estão presentes nas complexidades de pior

caso conhecidas analiticamente. Assim, para aquelas com complexidade desconhecida, optamos por relatar as duas formas colocadas como alternativas pelo EMA. Para a sequência de Tokuda, as complexidades obtidas pelo método empírico foram $O(n^{1,255})$ e $O(n\log^{5,5} n)$. Finalmente, para a sequência de Ciura, a complexidade obtida pelo método empírico foi $O(n^{1,372})$.

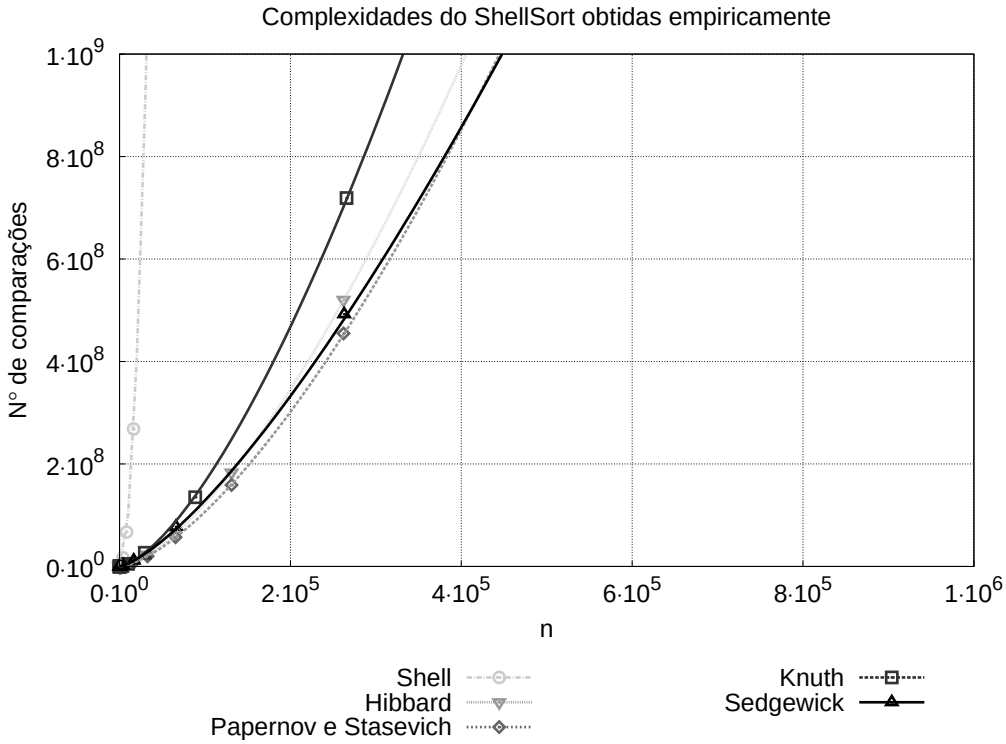


Figura 3: Número de comparações realizadas pelo algoritmo ShellSort.

A Figura 3 apresenta as curvas das complexidades de pior caso obtidas pelo EMA para as sequências de Shell, Hibbard, Papernov e Stasevich, Knuth e Sedgewick, sequências mais antigas e com complexidades de pior caso de mesma forma funcional. Como as sequências obtidas pela árvore de sequências de Frank e Lazarus são similares a outras sequências estudadas (o lado esquerdo equivalente à sequência de Papernov e Stasevich e o direito à de Hibbard), então sua representação foi omitida no gráfico. Pode-se observar que o algoritmo de Shell é aquele consistentemente com mais comparações e que, portanto, os métodos propostos após o mesmo realmente o melhoraram.

A Figura 4 apresenta as curvas das complexidades de pior caso obtidas pelo EMA para as sequências de Pratt, Incerpi e Sedgewick, Gonnet e Baeza-Yates, Tokuda e Ciura. A sequência de Pratt, proposta como uma alternativa às sequências da época realmente provou-se ter o melhor desempenho téorico em relação até a sequências posteriores a ela, sendo até hoje conhecida como a sequência mais próxima do limite ótimo de pior caso do ShellSort [13] (ver Teorema 2.2). En-

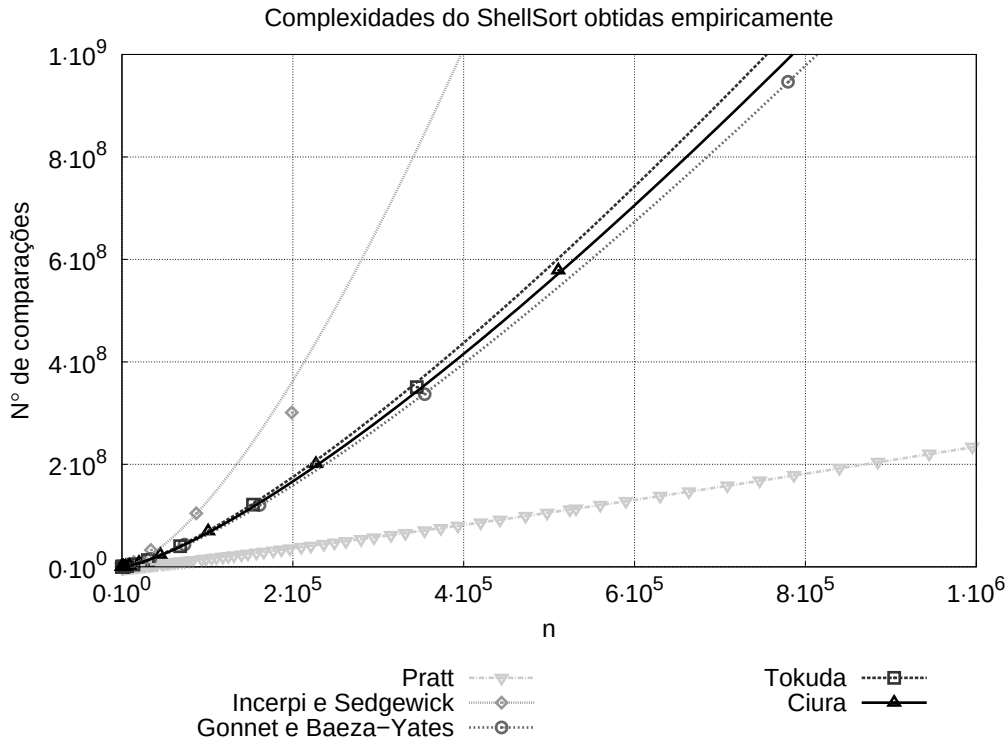


Figura 4: Número de comparações realizadas pelo algoritmo ShellSort.

tretanto, embora os resultados obtidos nas execuções das sequências mais recentes não tenham alcançado a eficiência da sequência de Pratt, tais sequências ainda são mais eficientes do que todas as outras propostas anteriormente.

9 CONCLUSÃO

Este trabalho considerou a análise de pior caso do algoritmo de ordenação ShellSort de forma empírica, apoiada em métodos para a determinação experimental de pior caso na ferramenta EMA, que auxilia na determinação experimental da complexidade de algoritmos. Os resultados obtidos mostram que a metodologia proposta neste trabalho obteve empiricamente a complexidade analítica para todas as sequências com complexidade exata de pior caso conhecida. Estendemos o estudo para três sequências cujo pior caso permanece em aberto, fornecendo conjecturas para as complexidades de tais sequências, no intuito de apontar novos caminhos para o estudo analítico da complexidade de pior caso do ShellSort para estas sequências. Por fim, observamos que o melhor limite superior analítico para sequências gerais, dado pelo Teorema 2.2, não é justo para nenhuma sequência específica, enquanto que a abordagem empírica proposta neste trabalho obteve limites justos para todas elas, o que ressalta a importância desta abordagem.

AGRADECIMENTOS

Agradecemos à CAPES e à FAPERJ pelo financiamento deste trabalho.

ABSTRACT. The worst-case time-complexity of the ShellSort, a comparison sorting algorithm, depends on a sequence of steps given as input. Each step consists of an integer representing an index offset of the pairs of elements that should be compared during the sorting of an input array. Such a complexity is known only to some specific sequences. In this work, we use a relation between the ShellSort and Frobenius number to present a new algorithm that provides an upper bound on the number of comparisons that ShellSort performs, for any given array and sequence of steps. We apply this algorithm, together with an empirical complexity analysis, to study sequences whose worst-case complexities are known through analytical method. We show that the empirical approach succeeded in determining such complexities. Based on such positive results, we extend the study to sequences for which the worst-case complexities are unknown.

Keywords: complexity of algorithms, empirical method, ShellSort.

REFERÊNCIAS

[1] J.L.R. Alfonsín. "The diophantine Frobenius problem", volume 30. Oxford University Press on Demand (2005).

[2] M. Ciura. Best increments for the average case of shellsort. In "Fundamentals of Computation Theory". Springer (2001), pp. 106–117.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein. "Introduction to algorithms". Editora MIT (2009).

[4] R. Cypher. A lower bound on the size of Shellsort sorting networks. *SIAM Journal on Computing*, **22**(1) (1993), 62–71.

[5] R.M. Frank & R.B. Lazarus. A high-speed sorting procedure. *Communications of the ACM*, **3**(1) (1960), 20–22.

[6] G.H. Gonnet & R.A. Baeza-Yates. "Handbook of Algorithms and Data Structures: in Pascal and C". Addison-Wesley, EUA (1991).

[7] M.T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *Journal of the ACM (JACM)*, **58**(6) (2011), 27.

[8] T.N. Hibbard. An empirical study of minimal storage sorting. *Communications of the ACM*, **6**(5) (1963), 206–213.

[9] J. Incerpi & R. Sedgewick. Improved upper bounds on Shellsort. In "Anais do 24º Annual Symposium on Foundations of Computer Science, 1983". IEEE (1983), pp. 48–55.

[10] D.E. Knuth. "The Art of Computer Programming 3: Sorting and Searching". Addison-Wesley, EUA (1998).

[11] F.S. Oliveira. EMA - WebPage. `http://fabianooliveira.ime.uerj.br/ema` (2016). [Último acesso: 22 de Outubro de 2019].

[12] A.A. Papernov & G.V. Stasevich. A method of information sorting in computer memories. *Problemy Peredachi Informatsii*, **1**(3) (1965), 81–98.

[13] B. Poonen. The worst case in Shellsort and related algorithms. *Journal of Algorithms*, **15**(1) (1993), 101–124.

[14] V.R. Pratt. "Shellsort and sorting networks". Ph.D. thesis, Universidade de Stanford (1972).

[15] R. Sedgewick. A new upper bound for ShellSort. *Journal of Algorithms*, **7**(2) (1986), 159–173.

[16] D.L. Shell. A high-speed sorting procedure. *Communications of the ACM*, **2**(7) (1959), 30–32.

[17] R. Souza, F. Oliveira & P. Pinto. Um Limite Superior para a Complexidade do ShellSort. In "III Encontro de Teoria da Computação". SBC, Natal, Brasil (2018), pp. 49–52.

[18] R.M. Souza, F.S. Oliveira & P.E.D. Pinto. O Algoritmo ShellSort e o Número de Frobenius. In "XXXVIII Congresso Nacional de Matemática Aplicada e Computacional". SBMAC, Campinas, Brasil (2018).

[19] N. Tokuda. An improved Shellsort. *Anais do 12º IFIP World Computer Congress on Algorithms, Software, Architecture-Information Processing*, **1** (1992), 449–457.

[20] P. Vitányi. On the average-case complexity of Shellsort. *Random Structures & Algorithms*, **52**(2) (2018), 354–363.

[21] M.A. Weiss & R. Sedgewick. More on shellsort increment sequences. *Information Processing Letters*, **34**(5) (1990), 267–270.