Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 260–288

Parallel enumeration of degree sequences of simple graphs

Antal IVÁNYI
Eötvös Loránd University,
Faculty of Informatics
email: ivanyi.antal2@upcmail.h

Loránd LUCZ
Eötvös Loránd University,
Faculty of Informatics
email: lorand.lucz@gmail.com

Tamás MATUSZKA
Eötvös Loránd University,
Faculty of Informatics
email: matuszka1987@gmail.com

Shariefuddin PIRZADA
Kashmir University,
Department of Mathematics
email: sdpirzada@yahoo.co.in

Abstract. The problem of testing, reconstruction and enumeration of the degree sequences of simple graphs has rich bibliography. In this paper we report on the parallel enumeration of the degree sequences of simple graphs resulting the number of sequences for $n = 24, \ldots, 29$ vertices. We also present the linear test version of Havel-Hakimi algorithm and compare it with the earlier linear testing algorithms.

1 Introduction

In the practice an often appearing problem is the ranking of different objects (examples can be found e.g. in [13]), assignment of points to the objects and ranking of the objects on the base of the sum of the received points.

Especially great bibliography has the case when the results are represented by a simple graph and the problem is the test, reconstruction and enumeration of the degree sequences. Havel in 1955 [8], Erdős and Gallai in 1960 [5], Hakimi

Computing Classification System 1998: G.2.2.
Mathematics Subject Classification 2010: 05C85, 68R10
Key words and phrases: simple directed graphs, approximate filtering algorithms, approximate reconstruction algorithms, linear Havel-Hakimi algorithm

in 1962 [7], Tripathi et al. in 2010 [36] proposed a method to decide, whether a sequence of nonnegative integers can be the degree sequence of a simple graph. The running time of their algorithms in worst case is $\Omega(n^2)$. In 2007 Takahashi [32], in 2009 Hell and Kirkpatrick [9] and in 2011 Iványi et al. [13] independently proposed an algorithm, whose worst running time is $\Theta(n)$.

There are several new proofs for the classical Havel-Hakimi and Erdős-Gallai theorems [2, 18, 22, 34, 35, 36].

Extensions for $(0, b)$-graphs [3, 22] and (a, b)-graphs [10, 11, 12, 15, 24] are also known.

There are earlier parallel results, e.g. in [23, 31, 28]. As an application of our linear time algorithm we describe Erdős-Gallai-Enumerative algorithm and its parallel version used to enumerate the different degree sequences of simple graphs for 24, ..., 29 vertices. We also present the linear test version of Havel-Hakimi algorithm and compare it with the earlier linear algorithms.

Let $n \geq 1$. We call a sequence $\mathbf{s} = (s_1, \ldots, s_n)$ (l, u, n)-*bounded*, if $0 \leq s_i \leq n$ for $i = 1, \ldots, n$, n-bounded, if it is $(0, n-1, n)$-bounded, *n-regular*, if the conditions $n - 1 \geq s_1 \geq \cdots \geq s_n \geq 0$ hold, and n-*even*, if the sum of the elements of $\mathbf{s}$ is even. If there exists a graph with n vertices which has the degree sequence $\mathbf{s}$, then we say that $\mathbf{s}$ is n-*graphical*. If such graph does not exist, then we say that $\mathbf{s}$ is *nongraphical*. If n is not necessary, then we omit it in the terms n-bounded, n-regular, n-even and n-graphical. The first i elements of an n-regular $\mathbf{s}$ are called *the head*, and the last $n-i$ elements are called *the tail, belonging to the element i* of $\mathbf{s}$.

The main aim of this paper is to report on the parallel realization of the linear Erdős-Gallai algorithm. Although this problem is interesting in itself, for us the main motivation was our wish to answer the question formulated in the recent monograph [6, Research problem 2.3.1] of András Frank: "*Decide if a sequence of n integers can be the final score of a football tournament of n teams.*" During testing and reconstructing of potential football sequences important subproblem is the handling of sequences of draws. Since the questions "Is this sequence graphical?" and "Is this sequence a football draw sequence?" are equivalent (see [12, 16, 17, 19, 27]), the quick answer is vital for us.

The structure of the paper is as follows. After the introductory Section 1 in Section 2 we describe the linear test version of the classical Havel-Hakimi algorithm, then in Section 3 we present the enumerating version of the linear Erdős-Gallai algorithm. In Section 4 the parallel version of the enumerating Erdős-Gallai algorithm is analyzed, and finally in Section 5 we summarize the results.

2 Linear Havel-Hakimi algorithm (HHL)

In a previous paper [13] we described the classical Havel-Hakimi [7, 8] and Erdős-Gallai [5] algorithms and their some improvements as linear Erdős-Gallai (EGL) and jumping Erdős-Gallai (EGLJ) algorithms.

Here we present the linear version of Havel-Hakimi algorithm (HHL) [12] and compare it with the previous linear algorithms EGL and EGLJ [13]. It is important to remark that this linear version of HH only tests the investigated sequences without their reconstruction.

In the worst case the original Havel-Hakimi algorithm requires quadratic time to test the $(0, 1, n)$-regular sequences. Using the new concepts weight point and reserve we reduced the worst running time to $O(n)$.

Let $s = (s_1, \ldots, s_n)$ be a potential graphical sequence. The definition of the *weight point* w_i belonging to s_i was introduced in [13] in connection with ERDŐS-GALLAI-LINEAR: if $s_1 \geq i$, then w_i is the largest k $(1 \leq k \leq n)$ having the property $s_k \geq i$. But if $s_1 < i$, then $w_i = 0$. EGL exploits the property w_i ensuring that if $i \leq w_i$, then the key expression $\min j, s_k$ in the Erdős-Gallai theorem equals i, otherwise equals s_k.

In HHL the weight point w_i determines the increment of the tail capacity when we switch to the investigation of the next element of s.

The reserve r_i belonging to s_i is defined as the unused part of the actual tail capacity and can be computed by the formulas

$$r_1 = w_1 - 1 - s_1 \tag{1}$$

and

$$r_i = w_i + r_{i-1} - s_i \quad \text{for} \quad 2 \leq i \leq n-1. \tag{2}$$

The programs of this paper are written using the pseudocode described in [4].

Input. n: number of vertices $(n \geq 4)$;
$s = (s_1, \ldots, s_n)$: the investigated regular sequence.

Output. 0 or 1.

Work variable. i: cycle variable;
$r = (r_1, \ldots, r_n)$: r_i the reserve belonging to s_i;
$w = (w_1, \ldots, w_n)$: w_i the weight point belonging to s_i;
$H = (H_1, \ldots, H_n)$: H_i is the sum of the first i elements of s.

HAVEL-HAKIMI-LINEAR(n, s)

01 **if** $s_{s_1+1} == 0$ // lines 01–02: test of s_1 in constant time

```
02    return 0
03 if s_1 == 0      // lines 03–04: test of the sequence consisting of only zeros
04    return 1
05 H_1 = s_1                                          // line 05: initialization of H
06 for i = 2 to n                                     // lines 06–07: further H_i's
07      H_i = H_{i-1} + s_i
08 if H_n is odd                                      // lines 08–09: test of the parity
09    return L
10 w_1 = n   // lines 10–13: computation of the first weight point and reserve
11 while s_{w_1} < 1
12          w_1 = w_1 - 1
13 r_1 = w_1 - 1 - s_1
14 for i = 2 to n - 1                                 // lines 14–21: testing of s
15      if s_i ≤ i or s_{i+1} = 0
16         return 1
17      w_i = w_{i-1}
18      while s_{w_i} < i and w_i > 0
19              w_i = w_i - 1
20      if s_i > w_i - 1 + r_{i-1}                    // line 20: Is s graphical?
21         return 0                                   // line 21: s is not graphical
22      r_i = w_i + r_{i-1} - s_i                     // line 22: update of the reserve
23 return 1                                           // line 23: s is graphical
```

Theorem 1 *The running time of* HAVEL-HAKIMI-LINEAR *is in best case* $\Theta(1)$*, and in worst case it is* $\Theta(n)$**.**

Proof. If the condition in line 1 or 3 holds, then the running time is $\Theta(1)$. If not, then we decrease the actual w at most n times and the remaining operations require $O(1)$ operations for all reductions. □

The C++ code of HHL is as follows (in the original code [20] every & is substituted by \&, every _ by _, every < by $<$, every > by $>$.

```
//Linear Havel-Hakimi algorithm (HHL)
bool HHL(const int& n, const int s[], vector<vector<int> >& ops) {
  if (F[1] < 0) { return false; }
  vector<int>& v = ops.at(n);
  v.push_back(0);

  int w[n], r[n], H[n];
++v.back();
```

```
if (s[0] == 0) {                                    // line 1 of the pseudocode
  return true;                                      // line 2 of the pseudocode
}
++v.back(); // if (s[s[0]+1] == 0)
if (s[s[0]] == 0) {                                 // line 3 of the pseudocode
  return false;                                     // line 4 of the pseudocode
}
H[0] = s[0];                                        // line 5 of the pseudocode
++v.back(); // since H[0] = s[0]; miatt
++v.back(); // int i=1 miatt
for (int i=1; i¡n; ++i) {                           // line 6 of the pseudocode
  H[i] = H[i-1] + s[i];                             // line 7 of the pseudocode
  v.back() += 4; // i¡n, ++i, H[i] = H[i-1] + s[i] (2 operations)
}
   v.back() += 2;
if (H[n-1] %2 == 1) {                               // line 8 of the pseudocode
  return false;                                     // line 9 of the pseudocode


   w[0] = n-1;                                     // line 10 of the pseudocode
++v.back();
while (s[w[0]] ¡ 1) {                               // line11 of the pseudocode
  —w[0];                                           // line 12 of the pseudocode
  v.back() += 2;
}
r[0] = w[0] - s[0];                                 // line 13 of the pseudocode
v.back() += 2;

++v.back(); // i=1 miatt
for (int i=1; i¡n-2; ++i) {                         // line 14 of the pseudocode
  v.back() += 2;
  v.back() += 3;
  if (s[i]¡=i+1 —— s[i+1] == 0) {                   // line 15 of the pseudocode
    return true;                                    // line 16 of the pseudocode
 }
 w[i] = w[i-1];                                     // line 17 of the pseudocode
 ++v.back();
  while (s[w[i]]¡i+1 && w[i]¿0) {                   // line 18 of the pseudocode
   –w[i];                                           // line 19 of the pseudocode
```

```
    v.back() += 4;
 }
 if (s[i]¿w[i]+r[i-1] ) {                    // line 20 of the pseudocode
    v.back() += 2;
    return false;                            // line 21 of the pseudocode
 }
    r[i] = w[i] + r[i-1] - s[i];             // line 22 of the pseudocode
    v.back() += 3;
 }
 return true;                                // line 23 of the pseudocode
}
```

An even sequence $s = (s_1, \ldots, s_n)$ is called *zerofree,* if $s_n > 0$. Table 1 shows the number ($E_z(n)$) of the tested zerofree sequences, further the average testing time of one zerofree sequence in microseconds for EGL ($T_{EGL}(n)/E_z(n)$), EGLJ ($T_{EGLJ}(n)/E_z(n)$), and HHL ($T_{HHL}(n)/E_z(n)$), when $n = 10, \ldots, 19$. The values $n = 1, \ldots, 9$ are omitted from the table since our program rounds the running time to zero.

n	$E_z(n)$	$\frac{T_{EGL}(n)}{E_z(n)}$	$\frac{T_{EGLJ}(n)}{E_z(n)}$	$\frac{T_{HHL}(n)}{E_z(n)}$
10	21 942	0.683620	0.000000	0.000000
11	83 980	0.369136	0.190521	0.381083
12	323 554	0.336883	0.194712	0.287433
13	1 248 072	0.299662	0.213128	0.237967
14	4 829 708	0.319895	0.226101	0.222788
15	18 721 080	0.338281	0.241371	0.226643
16	72 714 555	0.348197	0.251665	0.233406
17	282 861 360	0.379355	0.255846	0.240789
18	1 101 992 870	0.377512	0.267014	0.249460
19	4 298 748 300	0.394319	0.281491	0.261416

Table 1: Number of zerofree sequences, further the average running time for a zerofree sequence in the case of EGL, EGLJ and HHL algorithms in microseconds.

Figure 1 shows the running times of EGL, EGLJ and HHL as the function of the number of vertices. On the figure (green) triangles show the $(n, T(n))$ pairs for the linear Erdős-Gallai algorithm (EGL), (red) squares for the linear jumping Erdős-Gallai algorithm (EGLJ) and (blue) diamonds for the linear Havel-Hakimi algorithm (HHL).

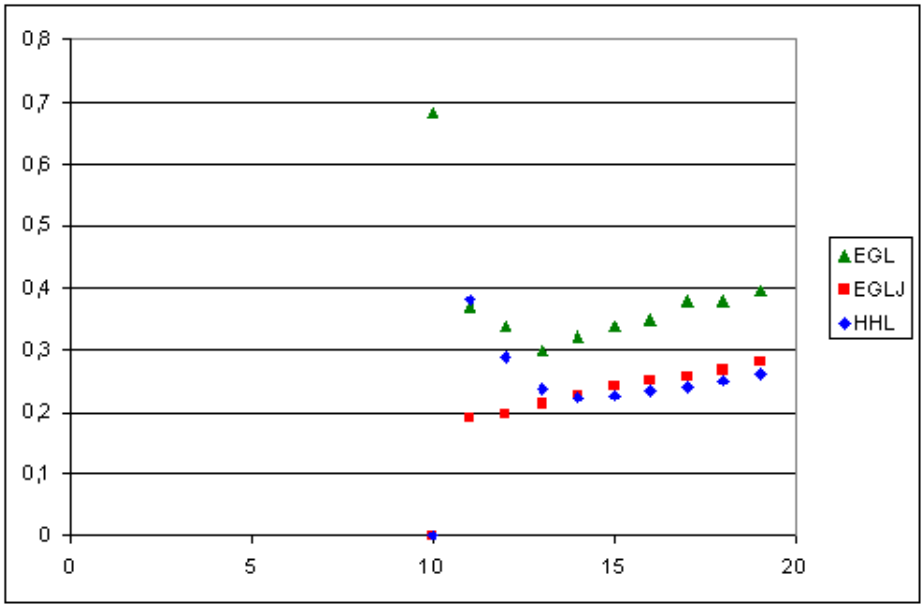

Figure 1: Average running time of EGL, EGLJ, and HHL.

Table 2 shows the average number of operations used to test one zerofree sequence in microseconds for EGL ($O_{EGL}(n)/E_z(n)$), EGLJ ($O_{EGLJ}(n)/E_z(n)$), and HHL ($O_{HHL}(n)/E_z(n)$), when $n = 10, \ldots, 19$. The values $n = 1, \ldots, 9$ are omitted from the table since our program rounds the corresponding running time to zero.

Figure 2 shows the running times of EGL, EGLJ and HHL as the function of the number of vertices. On the figure (green) triangles show the $(n, T(n))$ pairs for the linear Erdős-Gallai algorithm (EGL), (red) squares for the linear jumping Erdős-Gallai algorithm (EG) and (blue) diamonds for the linear Havel-Hakimi algorithm (HHL). The lines are drawn using the method of least squares.

As operations we counted comparisons, additions, subtractions, multiplications, divisions, residual divisions and assignments. The operations with indices are exceptions. For example the command $H[i] - i \cdot (i-1) > R$ requires three operations: the subtraction $H[i] - i \cdot (i-1)$, the multiplication $i \cdot (i-1)$, and the comparison $H[i] - i \cdot (i-1) > R$. The subtractions of type $i - 1$ are *not* counted when i is a cycle variable in the body of a cycle.

As an example we consider in details the testing of the zerofree input se-

n	$\frac{O_{EGL}(n)}{E_z(n)}$	$\frac{O_{EGLJ}(n)}{E_z(n)}$	$\frac{O_{HHL}(n)}{E_z(n)}$
2	35.000	13.000	14.000
3	55.000	26.500	18.000
4	73.000	37.667	29.889
5	91.000	51.429	39.357
6	101.609	61.473	48.591
7	123.495	72.480	57.553
8	139.162	82.042	66.123
9	154.944	91.751	74.552
10	170.421	100.929	82.749
11	185.885	110.047	90.824
12	201.209	118.930	98.758
13	212.177	124.720	106.591
14	231.659	136.373	114.739
15	246.785	144.939	121.976
16	261.846	153.411	129.552

Table 2: The average number of operations for a zerofree sequence in the case of EGL, EGLJ and HHL algorithms.

quence $(1, 1)$. This example is based on the C++ codes of the algorithms [20].

HHL (its pseudocode and C++ code see in this paper too) requires 14 operations: 1 comparison in line 1, 1 comparison in line 3, 1 assignment in line 5, 5 operations in lines 6 and 7 (1 assignment $i = 1$, 1 addition increasing i, 2 comparison $i < n$, 1 assignment $H_1 = s_1$), 1 residual division and 1 comparison in line 8, 1 assignment in line 10, 2 subtractions and 1 assignment in line 13 and 1 comparison in lines 14–22.

EGLJ requires 13 operations: 1 assignment in line 1, 5 operations in lines 2–3 (1 initialization of the cycle variable, 1 increasing of the cycle variable, 1 comparison, 2 assignment for H_i), 1 residual division and 1 comparison in lines 5–8, 1 assignment in line 9, 4 operations in lines 10–28 (1 initialization of the cycle variable, 1 increasing of the cycle variable, 1 comparison in line 11 and 1 comparison in line 17).

EGL requires 35 operations: 1 assignment in line 1, 9 operations in lines 2–3 (1 initialization of the cycle variable, 2 increasings of the cycle variable, 2 testing of the cycle variable, 2 additions for H_i, 2 assignments for H_i, 1 residual division and 1 comparison in line 4, 1 assignment in line 7, 7 operations in

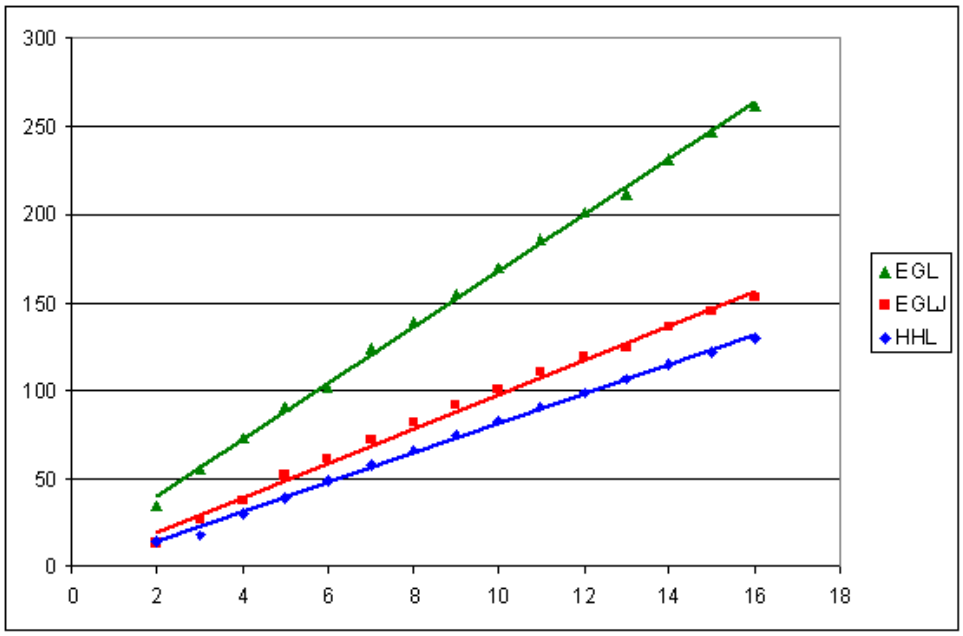

Figure 2: Amortized number of operations for EGL, EGLJ, and HHL.

lines 8–12 (1 initialization of the cycle variable, 2 increasings of the cycle variable, 2 comparisons, 2 tests of the branching), 4 operations in lines 13–14 (1 initialization of the cycle variable, 1 decreasing of the cycle variable, 1 comparison, 1 assignment), 11 operations in lines 15–23 (1 initialization of the cycle variable, 9 comparisons,1 increasing of the cycle variable).

Table 3 shows the number of the tested zerofree sequences ($E_z(n)$), further the average testing time of one tested sequence in microseconds for EGL ($o_{EGL}(n)/E_z(n)$), EGLJ ($o_{EGLJ}(n)/E_z(n)$), and HHL ($o_{HHL}(n)/E_z(n)$), when $n = 10, \ldots, 19$. The values $n = 1, \ldots, 9$ are omitted from the table since our computer rounds the running times to zero.

Figure 3 shows the running times of EGL, EGLJ and HHL as the function of the number of vertices. On the figure (green) triangles show the $(n, T(n))$ pairs for the linear Erdős-Gallai algorithm (EGL), (red) squares for the linear jumping Erdős-Gallai algorithm (EG) and (blue) diamonds for the linear Havel-Hakimi algorithm (HHL).

The most interesting data of Figure 3 are in the last three columns: they show that our algorithm is a CAT (Constant Time Amortized) algorithm (see [26]). In this columns the data show slowly decreasing character. The bases of

n	$G_z(n)$	$\frac{O_{EGL}(n)}{E_z(n)}$	$\frac{O_{EGLJ}(n)}{E_z(n)}$	$\frac{O_{HHL}(n)}{E_z(n)}$
2	1	17.500	6.500	7.000
3	2	18.333	8.833	6.000
4	7	18.250	9.417	7.472
5	20	18.200	10.286	7.781
6	71	16.935	10.246	8.099
7	240	17.642	10.154	8.222
8	871	17.395	10.255	8.265
9	3 148	17.216	10.195	8.284
10	11 655	17.042	10.093	8.275
11	43 332	16.899	10.004	8.257
12	162 769	16.767	9.911	8.230
13	614 718	16.321	9.593	8.199
14	2 330 537	16.547	9.741	8.196
15	8 875 768	16.452	9.663	8.132
16	33 924 858	16.365	9.588	8.097

Table 3: Number of zerofree graphical sequences ($G_z(n)$), further average number of operations for an element of a zerofree sequence in the case of EGL, EGLJ and HHL algorithms.

this decreasing tendency are Lemma 13 and Theorem 22 in [13]. According to these assertions $E(n) = \Theta(4^n/\sqrt{n})$ and $G(n) = O(4^n/((\log n)^C\sqrt{n}))$, where C is a positive constant. These assertions imply that $G(n)/E(n)$ tends to zero, when n tends to infinity, and so the limits of the sequences in the last three columns are determined by the average numbers of operations necessary to exclude the nongraphical sequences.

3 Enumerating Erdős-Gallai algorithm (EGE)

A classical problem of the graph theory is the enumeration of the degree sequences of different graphs—among others of simple graphs. For example *The On-Line Encyclopedia of Integer Sequences* [29] contains for $n = 1, \ldots, 29$ vertices the number of degree sequences of simple graphs (the values for $n = 20, \ldots, 23$ were set in July of 2011 by Nathann Cohen, and in November 15, 2011 for $24, \ldots, 29$ by us [13]).

We applied the new quick EGL to get these numbers for larger values of n.

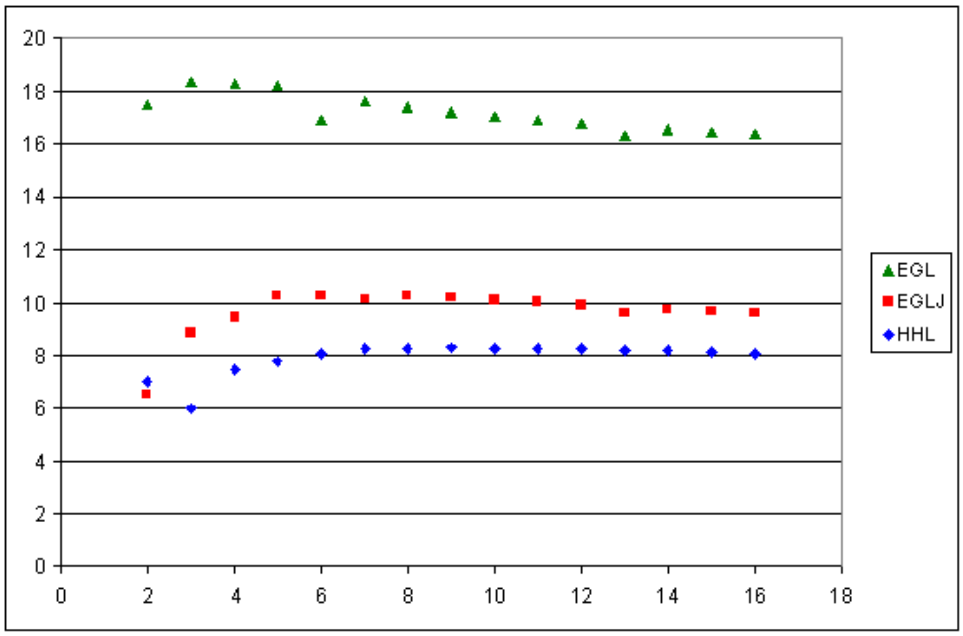

Figure 3: Average number of operations used for one element of zerofree sequences by EGL, EGLJ, and HHL.

Our starting point was to test all regular sequences and so enumerate the graphical ones. It is easy to see that there are

$$R(n) = \binom{2n-1}{n} \tag{3}$$

regular sequences. In 1987 Ascher derived the following explicit formula for the number of even sequences $E(n)$.

Lemma 2 (Ascher [1], Sloane, Pfoffe [30]) *If* $n \geq 1$, *then the number of even sequences* $E(n)$ *is*

$$E(n) = \frac{1}{2}\left(\binom{2n-1}{n} + \binom{n-1}{\lfloor n/2 \rfloor}\right). \tag{4}$$

Proof. See [1]). □

Using (3) and (4) we computed $R(n)$ and $E(n)$ for $i = 1, \ldots, 100$. The results for $n = 1, \ldots, 38$ were published in [13], for $n = 39, \ldots, 60$ are presented in

n	$R(n)$	$E(n)$
39	13608507434599516007800	6804253717317430635800
40	53753604366668088230810	2687680218336850574761 0
41	21239229042439586081442 0	1061961452122668536716 20
42	83945524310594554512366 0	4197276215531073370304 40
43	33187765425118777365354 00	1659388271256207997204920
44	13124252690842425594480900	6562126345421738821981380
45	51913710643776705684835560	2595685532188940489189964 0
46	20539772472102957466608852 0	1026988623605168456907261 60
47	81285057017258512527430776 0	4064252850862966793525176 80
48	32175335069331494542108015 50	1608766753466582789006321550
49	12738806129490428451365214 300	6369403064745230349484448700
50	50445672272782096667406248 628	2522283613639107993635473 3752
51	19980442743337222601600122 0056	9990221371668617621330382 8904
52	79153292406297458767877406 4068	3957664620314874178190202 69060
53	31362625293061257247649538 38760	1568131264653063110341743 393432
54	12428892245768720464809261 509160	6214446122884360719139487 166608
55	49263609265046928387789436 527216	2463180463252346516736443 1087664
56	19529502244357889468016526 6232892	9764751122178944925225528 3306556
57	77432763284647070522311140 6467256	3871638164232353564359010 03613848
58	30706095785291079689882004 04956360	1535304789264553992010916 827363440
59	12178349853827309571919303 301013360	6089174926913654800993284 900277200
60	48307454420181661301946569 760686328	2415372721009083068053943 0271558520

Table 4: Number of regular and even sequences for $n = 39, \ldots, 60$.

Table 4, and all values and the corresponding program can be found in [20]. The values of $R(n)$ for $n = 1, \ldots, 100$ are also contained in OEIS as sequence A001700 [21].

Due to the following lemma it is enough to test only the zerofree sequences.

Lemma 3 (Iványi, Lucz, Móri, Sótér [13]) *If $n \geq 2$, then the number of n-graphical sequences $G(n)$ can be computed from the number of $(n-1)$-graphical sequences $G(n-1)$ and the number of n-graphical zerofree sequences $G_z(n)$:*

$$G(n) = G(n-1) + G_z(n),$$

and if $n \geq 1$ then

$$G(n) = 1 + \sum_{i=2}^{n} G_z(i).$$

Proof. See [13]. □

Taking into account these results we have to test only about one fourth of the regular sequences. Table 5 shows the number of the zerofree sequences,

n	$G_z(n)$	$E_z(n)/R(n)$	$G_z(n)/R(n)$	$G(n)/R(n)$
1	0	0.000000	0.000000	1.000000
2	1	0.333333	0.333333	0.666667
3	2	0.200000	0.200000	0.400000
4	7	0.257143	0.200000	0.314286
5	20	0.222222	0.158730	0.246032
6	71	0.238095	0.153680	0.220779
7	240	0.230769	0.139860	0.199301
8	871	0.236053	0.135454	0.188500
9	3 148	0.235294	0.129494	0.179391
10	11 655	0.237524	0.126166	0.173375
11	43 332	0.238095	0.122852	0.168260
12	162 769	0.239188	0.120384	0.164278
13	614 198	0.245769	0.118108	0.160821
14	2 330 537	0.240783	0.116188	0.157882
15	8 875 768	0.241379	0.114439	0.155271
16	33 924 859	0.241946	0.112880	0.152950
17	130 038 230	0.242424	0.111448	0.150844
18	499 753 855	0.242860	0.101137	0.148926
19	1 924 912 894	0.243243	0.108920	0.147158
20	7 429 160 296	0.243590	0.107789	0.145521
21	28 723 877 732		0.106729	0.143997
22	111 236 423 288		0.105733	0.142569
23	431 403 470 222		0.104793	0.141228
24	1 675 316 535 350		0.103903	0.139961
25	6 513 837, 679 610		0.103058	0.138762
26	25 354 842 100 894		0.102254	0.137625
27	98 794 053 269 694		0.101486	0.136542
28	385 312 558 571 890		0.100752	0.135509
29	1 504 105 116 253 904		0.100049	0.134521

Table 5: The number of zerofree graphical sequences, further the number of zerofree, of zerofree graphical and of graphical sequences, divided by the number of regular sequences.

further the number of the zerofree, zerofree graphical and graphical sequences divided with the number of regular sequences.

Using the parallel version EGP (see the next section) of EGE we computed G_n till $n = 29$. These numbers can be found in Table 2 of [13].

We remark that $G_z(n)$ gives the number of degree sequences of simple

graphs, not containing isolated vertex. In 2006 Gordon Royle [25] posed the following problem: is it true that $G_z(n+1)/G_z(n)$ tends to 4?

Using the results of Tripathi and Vijay [13, Lemma 6 and Theorem 7] we can substantially decrease the average testing time of the zerofree even sequences. It is known that the expected number of checking points proposed by Tripathi and Vijay is about $n/2$ [13].

Using the following Lemma 4 later we will further fasten EGE. If $b = (b_1, \ldots, b_n)$ is a regular sequence, then $c = (c_1, \ldots, c_n)$ is called *lexicographically* i*-smaller, than* b if

$$c_j = b_j \quad \text{for} \quad j = 1, \ldots, i,$$

and

$$\sum_{j=i+1}^{n} c_j < \sum_{j=i+1}^{n} b_j.$$

Lemma 4 *If* $b = (b_1, \ldots, b_n)$ *is a nongraphical sequence and* $c = (c_1, \ldots, c_n)$ *is lexicographically* i*-smaller than* b*, then* c *is also nongraphical.*

The following algorithm Erdős-Gallai-Enumerating (EGE) is an enumerative version of EGL. This algorithm investigates the zerofree even sequences in lexicographical order, allowing to execute the majority of the basic operations in $O(1)$ average time.

- H_i (cumulated degrees): most of the time the only thing that is changing is the last element of the sequence b, so it is enough to update the last H value, according to the change of the value of b.
- C_i (checkpoints): if we modify the ith element of a sequence then the values before that point remain the same so all of the checkpoints before that remain the same, so we update only the first one before the ith index and all of them after it.
- W_i (weight points): every time the checking algorithm got a sequence to check we update the weight points, but we never start from 1 or n. We use the last value we used when we checked the sequence in that index. We have a distinct weight point for every i index and we just shift the value to left or right.

We suppose that n, b, H, c, C, and W are global variables, therefore their **return** does not require additional time.

Important property of EGE is that it solves in $\Theta(1)$ average time

- the generation of one zerofree even sequence;
- the updating of the sequence of the cumulated degrees H;
- the updating of the sequence of the checking points C;
- the updating of the sequence of the weight points W.

Although EGE solves the majority of the subproblems in $\Theta(1)$/sequence time, the work in the checking points requires more time, therefore the total running time $\Theta(E(n))$.

The following program is based on Theorem 9 of [13] and the properties just listed.

Input. n: number of vertices ($n \geq 4$);
$b = (b_1, \ldots, b_n)$: n-regular sequence.

Output. G_z: the number of n-length zerofree graphical sequences.

Work variables. i and j: cycle variables;
$H = (H_1, \ldots, H_n)$: H_i is the cumulated degree of the first i elements of the tested b;
$W = (W_1, \ldots, W_n)$: W_i the weight point of the actual b_i, that is the maximum of the indices of such elements of b, which are not smaller than i;
y: the cutting point of the actual b_i that is the maximum of i and w.

ERDŐS-GALLAI-ENUMERATING(n, G_z)

```
01 for i = 1 to n                                  // lines 01–09: initialization
02     b_i = n − 1
03     H_i = i(n − 1)
04     W_i = n
06     C_i = 0
07 G_z = 1
08 c = 0
09 b_{n+1} = −1
10 while b_2 ≥ 2 or b_1 ≥ 3                        // line 10: last sequence was?
11       if b_n ≥ 3              // lines 11–15: generating the next sequence
12          NEW3(n, b, H, c, C, W)
13       else if b_n = 2
14               NEW2(n, b, H, c, C, W)
15            else NEW1(n, b, H, c, C, W)
16       CHECK(n, b, H, c, W, L) // line 16: checks and updates the parameters
17       G_z = G_z + L                             // line 17: increasing of G_z
```

18 *print* G_z // line 18: final result

This algorithm uses four procedures. NEW1, NEW2, and NEW3 generate a new sequences (when b_n is 1, 2, resp. 3) and update the key parameters, while CHECK decides whether the actually investigated sequence is graphical or not.

In CHECK we use Theorem 8 of [13].

CHECK(n, b, H, c, C, W)

01 **for** $i = 1$ **to** c // lines 01–07: checking in checkpoints
02 $y = \max(W_{C_i}, i)$ // line 02: computation of the actual cutting point
03 **if** $H_i > i(y-1) + H_n - H_y$ // line 03–05: EG checking
04 $L = 0$
05 **return** L
06 $L = 1$ // line 06–07: b is graphical
07 **return** L

NEW3(n, b, H, c, C, W)

01 $b_n = b_n - 2$ // line 01–10: generation if $b_n = 3$
02 $H_n = H_n - 2$
03 **if** $b_n == b_{n-1} - 2$
04 $c = c + 1$
05 $C_c = n - 1$
06 $W_{b_n} = W_{b_n} - 1$
07 **if** $b_n \leq b_{n-1}$
08 $W_{b_n+1} = n + 1$
09 $W_{b_n} = n + 1$
10 **return** H, c, C, W

NEW2(n, b, H, c, C, W)

01 **if** $b_{n-1} == 2$ // line 01–53: generation if $b_n = 2$
02 $b_n = 1$ // line 01–09: generation if $b_{n-1} = 2$
03 $b_{n-1} = 1$
04 $H_{n-1} = H_{n-1} - 1$
05 $H_n = H_n - 2$
06 $W_2 = n - 2$
07 **if** $b_{n-2} == 2$ // line 07–09: generation if $b_{n-2} = 2$
08 $c = c + 1$

```
09        C_c = n − 1
10 else if b_{n−1} == 3                    // line 10–16: generation if b_{n−1} = 3
11           b_{n−1} = 2
12           b_n = 1
13           H_{n−1} = H_{n−1}
14           H_n = H_n − 2
15           W_3 = n − 2
16           W_2 = n − 1
17        else H_{n−1} = H_{n−1} − 1
18              if b_{n−2} == b_{n−1} and b_{n−1} is odd
19                 b_{n−1} = b_{n−1} − 1
20                 b_n = b_{n−1}
21                 H_n = H_n + b_{n−1} − b_n − 1
22                 C_c = C_c − 1
23                 W_{b_{n−2}} = n − 2
24                 for i = 1 to b_{n−2}
25                     W_i = n
26              if b_{n−2} == b_{n−1} and b_n − 1 is even
27                 b_{n−1} = b_{n−1} − 1
28                 b_n = b_{n−1} − 1
29                 H_n = H_n + b_{n−1} − b_n − 1
30                 C_c = C_c − 1
31                 c = c + 1
32                 C_c = n − 1
33                 W_{b_{n−2}} = n − 2
34                 W_{b_{n−1}} = n − 1
35                 for i = 1 to b_{n−2} − 2
36                     W_i = n
37              if b_{n−2} > b_{n−1} and b_{n−1} is odd
38                 b_{n−1} = b_{n−1} − 1
39                 b_n = b_{n−1}
40                 H_n = H_n + b_{n−1} − b_n − 1
41                 c = c − 1
42                 W_{b_{n−2}−1} = n − 2
43                 W_{b_{n−2}−1} = n − 1
44                 for i = 1 to b_{n−1} − 1
45                     W_i = n
46              if b_{n−2} > b_{n−1} and b_n − 1 is even
47                 b_{n−1} = b_{n−1} − 1
```

```
48              b_n = b_{n-1} − 1
49              H_n = H_n + b_{n-1} − b_n − 1
50              W_{b_{n-1}+1} = n − 1
51              for i = 1 to b_{n-1} − 1
52                  W_i = n
53 return H, c, C, W
```

New1 is similar to New2 (although more complicated, see Generate-New-Sequence in the following section), therefore it is omitted.

4 Parallel Erdős-Gallai algorithm (EGP)

The computing of $G(n)$ values lasts for a long time if we use a sequential program, so we used an accelerateded parallel version of EGE. The number of the used processors and the time we need to compute $G_z(n)$ are in inverse proportionality, therefore if we use more processors then we need less time.

In order to be able to use our new linear time algorithm on a bunch of sequences, we need an algorithm that can work on a part of all series we need to check.

Using our Erdős-Gallai-Parallel algorithm we computed this number till $n = 29$. These numbers can be found in Table 2 of [13].

Our application consists of two parts: server and client. The server has all the information to distribute jobs between client machines and to collect results from them. The client has the IP address and the PORT of the server too to ask for a job.

One of the most critical parts of the parallel algorithm is dividing the problem into jobs having almost the same sizes. The next equation helps us to give an approximation about the number of sequences starting with a fixed head. By knowing these numbers we can generate jobs with limited size, in other words, no job is largler than the given maximum.

It is easy to show that the number $Q(l, u, m)$ of the (l, u, m)-regular sequences is

$$Q(l, u, m) = \binom{u - l + m}{m}. \tag{5}$$

Based on (5) we get the next algorithm to generate jobs.

Input. n: the length of the sequences;
ms: maximal size of a job.
Output. M: the matrix containing the parameters of the jobs.

Working variables. i, j cycle variables;

GENERATE-MATRIX(n, ms, M)

```
01 for i = n downto 2                    // lines 01–03: filling up the matrix
02     for j = 1 to n − 1
03         M_{i,j} = C(i+j−2, i−1)
04 for j = n − 1 downto 1   // lines 04–05: filling up the first line in matrix
05     M_{1,j} = 1
06 GENERATE-NEW-SEQUENCES(M, n, n, 1, n − 1, ms, 0)  // line 06: new job
```

This algorithm gives us a matrix filled up with values computed by using the equation. Now, we can generate the sequences by reading out the last row from the matrix from left to right. In case of a value is too big and does not fit into a job, then we move one line above and read that line from the first column until the one that was too big we jumped here from and we can continue this technique until we get the size of parts we need. The next (recursive) algorithm reads out the last row with this method.

Input. n: the length of the sequences;
ms: maximal size of a job.

Output. M: the matrix containing the parameters of the jobs.

Working variables. i, j: cycle variables.

GENERATE-NEW-SEQUENCE(M, n, i, j, jm, ms, J)

```
01 S = 0                                  // line 01: setting the size of actual job
02 while j < j_m + 1
03     if S + M_{i,j} ≤ ms                // line 03: if we can add more sequences
04         S = S + M_{i,j}                          // line 04: add more sequences
05         if j ≤ j_m      // lines 05–06: line: move to next column in matrix
06             j = j + 1
07     else if S ≠ 0                                // line 07: job is not empty
08             for k = 2 to size(J, 2)              // lines 08–13: print result
09                 print(J_k)
10             for k = 1 to n − size(J, 2) + 1
11                 print(j − 1)
12             print   newline                                  // line 13: new line
13             S = 0
14         if M_{i,j} > ms and j ≤ j_m            // line 14: if decomposable
15             GENERATE-NEW-SEQUENCE(M, n, i − 1, 1, j, ms, [J, j])
```

```
16                  j = j + 1
17 if S ≠ 0                                   // line 18: last job is non empty
18    for k = 2 to size(J, 2)                    // lines 18–22: print last job
19        print (J_k)
20    for k = 1 to n − size(J, 2) + 1
21        print (J(size(J, 2)))
22    print   newline
```

Now we have divided the problem into smaller parts. So we can distribute them between multiple computers using our server program. In our next algorithm called DISTRIBUTING-JOBS we show how the server sends the jobs to the clients. In the algorithm we concentrate only on distributing the jobs so it does not contain code dealing with network communication, except for some very important network primitives (more on computer networks can be found in [33]).

Input. n: the length of the sequence;
N: estimated number of jobs;
M: matrix containing the parameters of jobs.

Output. G_z: number of n-regular zerofree graphical sequences.

Working variables. $S = (S_0, \ldots, S_n)$: vector containing the status of jobs;
fj: number of finished jobs;
aj: number of last job we sent to a client;
ji: index of job from incoming result;
cl: client identifier (used in network communication);
msg: message coming from client (important from network communication only);
S: the size of the actual job;
time: running time of the actual job in seconds;
al: lower bound;
upper bound: upper bound.

DISTRIBUTING-JOBS(n, N, M, G_z)

```
01 S_0 = true                          // lines 01–04: initializing job status vector
02 S_{N+1} = TRUE
03 for j = 1 to N + 1
04     S_j = FALSE
05 G_z = 0                                          // lines 05: initializing G_z
06 while fj < N                          // line 06: until all jobs are finished
```

```
07        accept(cl)                                  // line 07: accept client connection
08        recv(cl, msg)                               // line 08: receive message from client
09        if msg == 0                                 // line 09: client asks for a job
10          aj = aj + 1                               // line 10: increase index of last sent job
11          for i = M_{aj-1,0} to n                   // lines 11–12: update initial sequences
12              b_i = n + M_{aj-1,1}
13          while S_aj == TRUE or aj > N              // lines 13–22: unfinished job?
14                  aj = aj + 1
15                  if aj > N                         // line 14: we are over the maximal index
16                    aj = 1                          // line 15: set index to 1
17                  for i = M_{aj-1,0} to n           // line 19–21: update initial sequence
18                      b_i = n + M_{aj-1,1}
19          if aj < N // line 19–30: set parameters identifying last sequence
20            al = M_{aj,0}
21            b = n + M_{aj,1}
22          else al = 1
23                bu = 1
24          send(c, b, al, bu)                        // line 24: send job to client
25        else recv(c, ji, F_init, F_last, Z_{n,m}, time)  // line 25: receiving results
26            if S_ji == false                        // line 26: new result
27              S_j = true                            // line 27: set jobs status to finished
28              fj = fj + 1                           // line 28: increase number of finished jobs
29              G_z = G_z + Z_{n,m}                   // line 29: update G_z
30        close(cl)                                   // line 30: close network connection
31 return G_z                                         // line 31: return result
```

Our objective during implementing the client program was simplicity. We wanted to create a program the does not need any interaction from users. It is enough if the user starts it once and from that moment the program can work independently in the background. This is important because we wanted to distribute the program into as many parts as we can and use it in computer labs, where we do not have enough time and people to operate with the programs.

Another important idea was that we did not want to restart the programs when we change from computing $G_z(n)$ to $G_z(n+1)$. When the clients finish their jobs and the server cannot give them more, clients start to wait in the background—until they get new jobs—without using any significant resources.

A client program work as a thread. The reason for this is simple: we uploaded our program to a public homepage and anybody could join our computations.

By this our aim was to avoid loosing users only because our program use all the resources making the PC unable to respond their commands.

Our third objective was that we wanted to create a real fast program, because the running time can be really huge depending on the value of n. Because of this reason we used ANSI C language to implement our program. According to our experiments the ANSI C version of our program was one hundred times quicker, than our program written in MATLAB. For the network communication we used the Berkeley Sockets.

The client works as follows:

- After we create the network socket, we try to connect to the server. If it is not possible then we wait for an amount of time, and we double this amount every time we cannot connect and set to a default value when our attempt succeed. It is easy to see that the time we wait grows exponentially.
- After we connected to the server we ask for a job and disconnect after we got it.
- We compute a partial result of $G_z(n)$ and we send it back to the server using the same connection method as in the first step.

The program runs in clients called Parallel-Erdős-Gallai algorithm consisting of two parts: Check and Enumerating. The first one does the check of the sequences, but nothing else. The second generates sequences, H values and check points.

In Check we use a modified version of the linear Erdős-Gallai algorithm.

Input. d: input sequence;
$H = (H_1, \ldots, H_n)$: sums of the elements of d;
c: number of check points;
$C = (C_1, \ldots, C_{n-1})$: check points.

Output. L: Logical value. If the investigated sequence is graphical, then $L = 1$, otherwise $L = 0$.

Working values. p: actual checking point.

CHECK(b, H, c, C)

```
01 i = 1                                          // line 01: initialization of i
02 while i ≤ c and H_{C_i} > C_i(C_i − 1)          // lines 02–11: check sequences
03       p = C_i                                  // line 03: initial p value
04       while J_p < n and b_{J_{p+1}} > p        // lines 04–08: actualize p
05             J_p = J_p + 1
07       while J_p > p and b_{J_p} ≤ p
08             J_p = J_{p−1}
09       if H_p > H_n − H_{J_p} + p(J_p − 1)       // line 09: check
10          L = 0                                 // line 10: nongraphical sequence
11          return L
12 i = i + 1
13 L = 1                                          // lines 13–14: b is graphical
14 return L
```

In our checking algorithm we do not use the cases we proposed in the original algorithm. The reason is the following: if we don't let the weight points run under the current i index, then the second case will work fine and we do not need an additional condition to check if the weight point is smaller than the current index.

Input. n: length of sequences;
b: first sequence;
last_index: index of element we'll check if we reached the last sequence we need to check;
last_value: value of element we'll check if we reached the last sequence we need to check.

Output. G_z^p: number of n-regular zerofree graphical sequences between the first and the last checked sequences.

ENUMERATING$(n, b, \text{last_index}, \text{last_value})$

```
01 H_1 = b_1                                      // line 01: set H_1
02 for i = 2 to n                                 // lines 02–03: calculation of H
03      H_i = H_{i−1} + b_i
04 if b_n ≠ n − 1                                 // line 04: if it is not the full graph
05    if H_n odd                                  // lines 05–10: actualize series
06       b_n = b_n − 3
07       H_n = H_n − 3
```

```
08    else b_n = b_n − 2
09           H_n = H_n − 2
10  for i = 1 to n                          // lines 10–11: initialize weight points
11      J_i = n − 1
12  for i = 1 to n − 2                      // lines 12–15: calculate check points
13      if b_i ≠ b_{i+1} and b_i ≠ b_n
14          c = c + 1
15          C_c = i
16  L = CHECK(b, H, c, C)                   // line 16: check first sequence
17  G_z^p = G_z^p + L
18  while b_{last_index} > last_value       // line 18: till the last sequence in job
19          k = n                           // line 19: initialize working variable
20          if b_k == 1                     // line 20: if the last element of series is 1
21              j = n − 1
22              while b_j ≤ 1
23                  j = j − 1
24              if b_j == 2                 // line 24: if the 1 free part's last value is 2
25                  b_{j−1} = b_{j−1} − 1   // line 25: update sequence
26                  H_{j−1} = H_{j−1} − 1   // line 26: update H
27                  if j > 2                // line 27–36: update check points
28                      if (c ≤ 2 or (c > 2 and C_{c−2} ≠ j − 2)) and
            (c > 1 and C_{c−1} ≠ j − 2)
29                          if c > 1 and C_{c−1} > j − 2
30                              C_{c+1} = C_c
31                              C_c = C_{c−1}
32                              C_{c−1} = j − 2
33                              c = c + 1
34                          else C_{c+1} = C_c
35                              C_c = j − 2
36                              c = c + 1
37                  for k = j to n
38                      b_k = b_{j−1}       // line 39: update the last part of b
39                      H_k = H_{k−1} + b_k // line 40: update H
40                  while c > 1 and C_c > j − 1 // lines 42–43: update check points
41                      c = c − 1
42                  if H_n odd              // line 42: if parity is odd
43                      b_n = b_{n−1} − 1   // line 43: update b
44                      H_n = H_{n−1} + b_n // line 44: update H
45                      c = c + 1           // lines 45–46: update check points
```

```
46              C_c = n − 1
47          else b_j = b_j − 1                          // line 47: update b
48              H_j = H_j − 1                           // line 48: update H
49              if j > 1                    // line 49–50: update check points
50                 if (c == 1 and C_c ≠ j − 1) or (c > 1 and C_{c−1} ≠ j − 1)
51                    if c > 0 and C_c > j − 1
52                       C_{c+1} = C_c
53                       C_c = j − 1
54                       c = c + 1
55              for k = j + 1 to n
56                  b_k = b_j                           // line 56: update b
57                  H_k = H_{k−1} + b_k                 // line 57: update H
58              while c > 1 and C_c > j − 1
                                        // lines 58–59: update check points
59                     c = c − 1
60              if H_n odd                              // line 60: parity check
61                 b_n = b_n − 1                        // line 61: update b
62                 H_n = H_n − 1                        // line 62: update H
63                 c = c + 1                // line 63: update check points
64                 C_c = n − 1              // line 64: add new check point
65          else if b_k == 2
66              b_{k−1} = b_{k−1} − 1                   // line 66: update b
67              H_{k−1} = H_{k−1} − 1                   // line 67: update H
68              if (c == 1 and C_c ≠ n − 2)
                   or (c > 1 and C_{c−1} ≠ n − 2 and C_c ≠ n − 2)
                                        // lines 68–73: update check points
69                 if c > 0 and C_c > n − 2
70                    C_{c+1} = C_c
71                    C_c = n − 2
72                 else c = c + 1
73                        C_c = n − 2
74              if b_{k−1} odd                          // line 74: parity check
75                 b_k = b_{k−1}                        // line 75: update b
76                 if c > 0 and C_c == n − 1
77                    c = c − 1             // line 77: update checkpoints
78              else b_k = b_{k−1} − 1                  // line 78: update b
79              H_k = H_{k−1} + b_k                     // line 79: compute H
80           else b_k = b_k − 2                         // line 80: update b
81              H_k = H_k − 2                           // line 81: compute H
```

```
82          if c < 1 or C_c ≠ n − 1
                                        // lines 82–84: update check points
83              c = c + 1
84              C_c = n − 1
85      G_z^p = G_z^p + CHECK(b, H, c, C)          // line 85: update G_z^p
```

In *The On-Line Encyclopedia of Integer Sequences* [29] you can find numbers of degree sequences for simple graphs consisting of n vertices, that we uploaded $G(n)$ values from $n = 24$ to 29 on 16th of November.

To carry out the calculations we used more than two hundred computers and our theoretical maximal performance was over 6 TFLOPS based on the processors information we found on the home pages of the manufacturers.

The running time of computing the number of graphical series can be seen in Table 6. It is easy to see that the growing of the running time does not have the same ratio between the different n values. The reason for this is the type of processors we used. In our earlier computations (eg. when we considered $n = 25$ vertices) we had a few powerful machines, but as the complexity was larger in every time we increased n we had to use some less powerful machines. The total time of the calculations would be less if we used the more powerful machines, but the real running time would be more, because in total we had more than two hundred machines when we was working on G_{29}, so the real running time was under two weeks.

5 Summary

The paper reports on a linear version of the Erdős-Gallai testing algorithm [13], on its enumerative and parallel versions, further on enumerative results received using the new algorithms.

n	Running time (day)	Number of jobs
25	26	435
26	70	435
27	316	435
28	1130	2 001
29	6733	15 119

Table 6: Sum of running times measured during our calculations and number of jobs.

The number of different degree sequences of simple graphs on n vertices for $n = 24, \ldots, 29$ were accepted as new records by *The On-Line Encyclopedia of Integer Sequences* in November 15, 2011 [14].

The paper contains also the description and analysis of the linear test version of Havel-Hakimi algorithm which is about 10 percent quicker than the best version of the Erdős-Gallai algorithm.

The log files and source codes of our programs can be found at

http://people.inf.elte.hu/lulsaai/Holzhacker

and

http://people.inf.elte.hu/tomintt/DegreeSeq

Acknowledgements. The authors are indebted to Antal Sándor and his colleagues (Eötvös Loránd University, Faculty of Informatics), Ádám Mányoki (TFM World Kereskedelmi és Szolgáltató Kft.) and Zoltán Kása (Sapientia Hungarian University of Transylvania) for their help in running of our time-consuming programs.

References

[1] M. Ascher (1987) Mu torere: an analysis of a Maori game, *Math. Mag.* **60,** 2 1987 90–100. ⇒270

[2] S. A. Choudum, A simple proof of the Erdős-Gallai theorem on graph sequences, *Bull. Austral. Math. Soc.* **33** (1986) 67–70. ⇒261

[3] V. Chungphaisan, Conditions for sequences to be r-graphic, *Discrete Math.* **7** (1974) 31–39. ⇒261

[4] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* Third edition, The MIT Press/McGraw Hill, Cambridge/New York, 2009. ⇒262

[5] P. Erdős, T. Gallai, Graphs with vertices having prescribed degrees (Hungarian), *Mat. Lapok* **11** (1960) 264–274. ⇒260, 262

[6] A. Frank, *Connections in Combinatorial Optimization,* Oxford University Press, Oxford, 2011. ⇒261

[7] S. L. Hakimi, On the realizability of a set of integers as degrees of the vertices of a simple graph, *J. SIAM Appl. Math.* **10** (1962) 496–506. ⇒261, 262

[8] V. Havel, A remark on the existence of finite graphs (Czech), *Časopis Pěst. Mat.* **80** (1955), 477–480. ⇒260, 262

[9] P. Hell, D. Kirkpatrick, Linear-time certifying algorithms for near-graphical sequences, *Discrete Math.* **309,** 18 (2009) 5703–5713. ⇒261

[10] A. Iványi, Reconstruction of complete interval tournaments, *Acta Univ. Sapientiae, Inform.* **1,** 1 (2009) 71–88. ⇒261

[11] A. Iványi, Reconstruction of complete interval tournaments. II, *Acta Univ. Sapientiae, Math.* **2,** 1 (2010) 47–71. ⇒261

[12] A. Iványi, Degree sequences of multigraphs, *Annales Univ. Sci. Budapest., Sect. Comp.* **37** (2012) 195–214. ⇒261, 262

[13] A. Iványi, L. Lucz, T. F. Móri, P. Sótér, On the Erdős-Gallai and Havel-Hakimi algorithms, *Acta Univ. Sapientiae, Inform.* **3,** 2 (2011) 230–268. ⇒260, 261, 262, 269, 270, 271, 272, 273, 274, 275, 277, 285

[14] A. Iványi, L. Lucz, T. F. Móri, P. Sótér, The number of degree-vectors for simple graphs, in: ed. by N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences,* 2011. http://oeis.org/A004251 ⇒286

[15] A. Iványi, S. Pirzada, Comparison based ranking, in: *Algorithms of Informatics, Vol. 3* (ed. A. Iványi), AnTonCom, Budapest 2011, 1209–1258. ⇒261

[16] A. Iványi, J. E. Schoenfield, Deciding football sequences. *Acta Univ. Sapientiae, Inform.* **4,** 1 (2012) 130–183. ⇒261

[17] G. Zs. Kovács, N. Pataki, *Analysis of Ranking Sequences* (in Hungarian), Scientific student paper, Eötvös Loránd University, Faculty of Sciences, Budapest 2002. ⇒261

[18] M. D. LaMar, Algorithms for realizing degree sequences of directed graphs, arXiv, 2010. http://arxiv.org/abs/0906.0343. ⇒261

[19] L. Lucz, *Analysis of degree sequences of graphs* (Hungarian), MSc Thesis, Eötvös Loránd University, Faculty of Informatics, Budapest, 2012. http://people.inf.elte.hu/lulsaai/diploma. ⇒261

[20] T. Matuszka, *Programs and Results Connected with Degree Sequences*, http://people.inf.elte.hu/tomintt/DegreeSeq. ⇒263, 267, 271

[21] Noe, T. D., Table of n $a(n)$ for $n = 1, \ldots, 100$, in (ed. N. J. A. Sloane): *The On-Line Encyclopedia of the Integer Sequences*, 2010. http://oeis.org/A001700. ⇒271

[22] S. Özkan, Generalization of the Erdős-Gallai inequality, *Ars Combin.* **98** (2011) 295-302. ⇒261

[23] G. Pécsy, L. Szűcs, Parallel verification and enumeration of tournaments, *Stud. Univ. Babeş-Bolyai, Inform.* **45,** 2 (2000) 11–26. ⇒261

[24] S. Pirzada, *An Introduction to Graph Theory*, Orient BlackSwan, Hyderabad, 2012. ⇒261

[25] G. Royle, Is it true that $a(n+1)/a(n)$ tends to 4? in (ed. N. J. A.) Sloane): *The On-Line Encyclopedia of the Integer Sequences*, 2012. http://oeis.org/A095268 ⇒273

[26] F. Ruskey, F. R. Cohen, P. Eades, A. Scott, Alley CATs in search of good homes, *Congr. Numer.* **102** (1994) 97–110. ⇒268

[27] J. E. Schoenfield, The number of football score sequences, in: ed. by N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences,* 2012. http://oeis.org/A064626. ⇒261

[28] B. Siklósi, *Comparison of Sequential and Parallel Algorithms Solving Sport Problems* (in Hungarian). Master thesis. Eötvös Loránd University, Faculty of Sciences, Budapest, 2001. ⇒261

[29] N. J. A. Sloane, Number of graphical partitions (degree-vectors for simple graphs with n vertices, or possible ordered row-sum vectors for a symmetric 0-1 matrix with diagonal values 0), in: *The On-Line Encyclopedia of the Integer Sequences* (ed. by N. J. A. Sloane). http://oeis.org/A004251. ⇒269, 285

[30] N. J. A. Sloane, S. Plouffe, *The Encyclopedia of Integer Sequences,* Academic Press, 1995. ⇒270

[31] D. Soroker, *Optimal parallel construction of prescribed tournaments, Discrete Appl. Math.* **29,** 1 (1990) 113–125. ⇒261

[32] M. Takahashi, *Optimization Methods for Graphical Degree Sequence Problems and their Extensions*, PhD thesis, Graduate School of Information, Production and systems, Waseda University, Tokyo, 2007. http://hdl.handle.net/2065/28387. ⇒261

[33] A. S. Tanenbaum, D. J. Wetherall, *Computer Networks* (5th edition), Prentice Hall, 2010. ⇒279

[34] A. Tripathi, H. Tyagy, A simple criterion on degree sequences of graphs, *Discrete Appl. Math.* **156,** 18 (2008) 3513–3517. ⇒261

[35] A. Tripathi, S. Vijay, A note on a theorem of Erdős & Gallai, *Discrete Math.* **265,** 1-3 (2003) 417–420. ⇒261

[36] A. Tripathi, S. Venugopalan, D. B. West, A short constructive proof of the Erdős-Gallai characterization of graphic lists, *Discrete Math.* **310,** 4 (2010) 833–834. ⇒261

Received: October 2, 2012 • Revised: Decembet 30, 2012