Acta Universitatis Sapientiae

Informatica

Volume 4, Number 2, 2012

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 189–200

Diamond-free degree sequences

Alice MILLER
School of Computing Science
University of Glasgow, Scotland
email: alice.miller@glasgow.ac.uk

Patrick PROSSER
School of Computing Science
University of Glasgow, Scotland
email: pat@dcs.gla.ac.uk

Abstract. While attempting to classify partial linear spaces produced during the execution of an extension of Stinson's hill-climbing algorithm a new problem arises, that of generating all graphical degree sequences that are diamond-free (i.e. have no diamond as subgraph) and satisfy additional constraints. We formalize this new problem, propose a constraint programming solution and list all satisfying degree sequences of length 8 to 16 inclusive.

1 Introduction

We introduce a new problem, CSPLib number 50 [1], to generate all degree sequences that have a corresponding diamond-free graph with secondary properties. This arises naturally from a problem in mathematics to do with partial linear spaces; we devote Section 2 to this. The motivation described in Section 3 is the challenge of the necessary computational effort arising from the large number of symmetries within the models (see Section 4). We introduce two constraint programming models. The second model is an improvement on the first, and this improvement largely consists of breaking the problem into three stages: the first stage produces degree sequences that satisfy arithmetic constraints, the second stage tests that a given degree sequence is graphical and if it is the third stage determines if there exists a graph with that degree sequence that is diamond-free. We now present the problem in detail and give

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 68R10
Key words and phrases: diamond-free graphs, degree sequences, constraint programming, BIBD, hill-climbing algorithm

motivation for it. In Section 4 two models, in Section 5 a list of solutions are presented. Finally in Section 6 we conclude and suggest future work.

2 Problem definition

Given a simple undirected graph $G = (V, E)$, V is the set of vertices and E the set of undirected edges. The edge $\{u, v\} \in E$ if and only if vertex u is adjacent to vertex v in G. The graph is simple in that there are no loop edges, i.e. $\forall_{v \in V} [\{v, v\} \notin E]$. Each vertex v in V has a degree $\delta(v) = |\{\{v, w\} : \{v, w\} \in E\}|$, i.e. the number of edges incident on that vertex. A diamond is a set of four vertices in V such that there are five edges between those vertices (see the diamond in Figure 1).

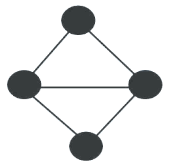


Figure 1: The diamond graph (four vertices and five edges)

Conversely, a graph is diamond-free if it has no diamond as a subgraph, i.e. for every set of four vertices the number of edges between those vertices is at most four. Determining whether a graph is diamond-free is a polynomial-time problem. E.g. checking every four vertices for a diamond is at worst case $\Theta(n^4)$. Note that a diamond is sometimes referred to as a $K_4 - e$ graph. Our definition of a diamond-free graph agrees with that of [14] which addresses a different, but related problem. That is, identifying degree sequences for which there is a realisation containing a diamond as a subgraph. Others [6, 7] use the term diamond-free to denote a graph which has no diamond as an *induced* subgraph (in which case a K_4 is an allowable subgraph, unlike in our case). A further definition of a diamond-free graph [2] is a graph G with no diamond as a *minor*, i.e. a graph (isomorphic to one that can be) obtained from a subgraph of G by zero or more edge contractions.

In our problem we have additional properties required of the degree sequences of the graphs, in particular that the degree of each vertex is greater than zero (i.e. isolated vertices are disallowed), the degree of each vertex is divisible by 3 and the sum of the degrees is divisible by 12 (i.e. $|E|$ is divisible by 6).

The problem is then for a given value of n, such that $|V| = n$, produce all degree sequences $\delta(1) \geq \delta(2) \geq \ldots \geq \delta(n)$ such that there exists a diamond-free graph with that degree sequence, each degree is non-zero and divisible by 3, and the number of edges is divisible by 6. In Figure 2 we give as an example the unique degree sequence for $n = 8$ that satisfies our arithmetic constraints, a corresponding diamond-free graph and its adjacency matrix.

```
3 3 3 3 3 3 3 3

0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 1
0 0 0 1 1 0 0 1
0 0 1 0 1 0 1 0
0 0 1 1 0 1 0 0
1 1 0 0 1 0 0 0
1 1 0 1 0 0 0 0
1 1 1 0 0 0 0 0
```

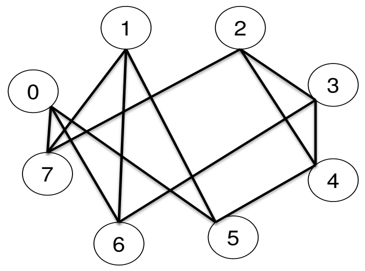


Figure 2: Unique degree sequence for $n = 8$ with a corresponding diamond-free graph and its adjacency matrix

3 Motivation

The problem is a byproduct of attempting to classify partial linear spaces that can be produced during the execution of an extension of Stinson's hill-climbing algorithm [3, 4, 5, 15] for block designs with block size 4. First we need some definitions.

Definition 1 *A Balanced Incomplete Block Design (BIBD) is a pair* (V, B) *where* V *is a set of* n *points and* B *a collection of subsets of* V *(blocks) such that each element of* V *is contained in exactly* r *blocks and every* 2*-subset of* V *is contained in exactly* λ *blocks.*

Variations on BIBDs include *Pairwise Balanced Designs* (PBDs) in which blocks can have different sizes, and *linear spaces* which are PBDs in which every block has size at least 2. It is usual to refer to the blocks of a linear space as a *line*. A partial linear space is a set of lines in which every pair appears in *at most* λ blocks. Here we refer to a BIBD with $\lambda = 1$ as a *block design* and to a partial linear space with $\lambda = 1$, having s_i lines of size i, where

$i \geq 3$ and $s_i > 0$ as a $3^{s_3}4^{s_4}\ldots$ structure. For example, a block design on 7 points with block size 3 is given by the following set of blocks:

$$\{(1,2,3),(1,6,7),(1,4,5),(2,5,6),(3,4,6),(3,5,7),(2,4,7)\}$$

and a $3^4 4^1$ structure on 8 points by the following set

$$\{(1,2,3,4),(1,5,6),(1,7,8),(2,5,7),(2,6,8)\}$$

Note that in the latter case we do not list the lines of size 2. Block designs with block size 3 are known as Steiner Triple Systems (STSs). These exist for all n for which $n \equiv 1,3 \pmod 6$ [12]. For example the block design given above is the unique STS of order 7 (STS(7)). Similarly block designs with block size 4 always exist whenever $n \equiv 1,4 \pmod{12}$.

```
Stinson($n$)
 1 $LivePairs \leftarrow \{(i,j) : 1 \leq i < j \leq n\}$
 2 $Blocks \leftarrow \emptyset$
 3 while $LivePairs \neq \emptyset$
 4       choose pairs $(x,y)$ and $(y,z)$ from $LivePairs$
 5       $LivePairs \leftarrow LivePairs \setminus \{(x,y)\}$
 6       $LivePairs \leftarrow LivePairs \setminus \{(y,z)\}$
 7       if $(x,z) \in LivePairs$
 8            $LivePairs \leftarrow LivePairs \setminus \{(x,z)\}$
 9       else $Blocks \leftarrow Blocks \setminus \{(w,x,z) : (w,x,z) \in Blocks\}$
10            $LivePairs \leftarrow LivePairs \cup \{(w,x)\}$
11            $LivePairs \leftarrow LivePairs \cup \{(w,z)\}$
12       $Blocks \leftarrow Blocks \cup \{(x,y,z)\}$
13 return $Blocks$
```

Algorithm Stinson above allows us to generate an STS for any n and is due to Stinson [13]. This algorithm always works, i.e. it never fails to terminate due to reaching a point where the STS is not created and there are no suitable pairs (x,y) and (y,z).

A natural extension to this algorithm, for the case where block size is 4, is proposed in algorithm Stinson4. Note that the triples in set $WeightedTriples$ are all initially assigned weight 0 in line 1. Triples can only be selected to make a new block if they have weight zero. If S is a set of triples and X a set of points then the algorithms IncreaseWeight(X,S) and DecreaseWeight(X,S) (lines 6 and 9) increment (decrement) the weight of every element of S that contains π, for all pairs π of distinct points from X.

```
STINSON4($n$)
1 $WeightedTriples \leftarrow \{\langle(i,j,k),0\rangle : 1 \le i < j < k \le n\}$
2 $\mathsf{Blocks} \leftarrow \emptyset$
3 while $\langle(w,x,y),0\rangle \in WeightedTriples \ \wedge\ \langle(x,y,z),0\rangle \in WeightedTriples$
4         choose $\langle(w,x,y),0\rangle$ and $\langle(x,y,z),0\rangle$ from $WeightedTriples$
5         for $(u,v,w,z) \in \mathsf{Blocks}$
6              DECREASEWEIGHT($\{u,v,w,z\}, WeightedTriples$)
7              $\mathsf{Blocks} \leftarrow \mathsf{Blocks} \setminus \{(u,w,x,z)\}$
8         $\mathsf{Blocks} \leftarrow \mathsf{Blocks} \cup \{(w,x,y,z)\}$
9         INCREASEWEIGHT($\{w,x,y,z\}, WeightedTriples$)
10 return $\mathsf{Blocks}$
```

Algorithm STINSON4 does not always work. It is possible for a situation to be reached from which one pair of triples is constantly swapped with another, in which case the algorithm fails to terminate. It is also possible for the algorithm to terminate but fail to create a block design due to reaching a point at which *WeightedTriples* contains elements of weight zero but does not contain suitable triples (w,x,y) and (x,y,z) with weight zero. In this case the algorithm produces a 4^{s_4} structure (where s_4 is less than the number of blocks in the corresponding block design) for which the complement has no pair of triples (w,x,y), (x,y,z), with weight zero. I.e. the complement graph is diamond-free. When $n = 13$ the algorithm either produces a block design or a 4^8 structure whose complement graph consists of 4 non-intersecting triangles.

The next open problem therefore is for $n = 16$. If the algorithm terminates but does not produce a block design, what is the nature of the structure it does produce? To do this, we need to classify the 4^{r_4} structures whose complement graph is diamond-free.

The cases for which the 4^{s_4} structure has at least 2 points that are in the maximum number of blocks (5) are fairly straightforward. (There are fewer cases as this number increases.). However if the number of such points is 0 or 1, there is a large number of sub-cases to consider. The problem is simplified if we can dismiss potential 4^{s_4} structures because the degree sequences of their complements can not be associated with a diamond-free graph. This leads us to the problem outlined in this report: to classify the degree sequences of diamond-free graphs of order 15 and 16. Note that each point that is not in 5 blocks is either in no blocks or is in blocks with some number of points, where that number is divisible by 3. Thus for every point there is a vertex in the complement graph whose degree is also divisible by 3. In addition, since the number of pairs in both a block design on 16 points and a 4^{s_4} structure are divisible by 6, the number of edges in the complement graph must be divisible by 6. We can immediately eliminate some cases via the following lemmas. In all cases G is a diamond-free graph with n vertices for which every vertex has degree greater than 0 and divisible by 3.

Lemma 2 *If* $n = 16$ *then no vertex has degree* 15.

Proof. Suppose that u be a vertex that has degree 15. Then all other vertices are adjacent to u. Let v be such a vertex. Since v has degree at least 3, there are two vertices, w and x that are adjacent to both u and v. There is a diamond on vertices u, v, w and x. This is a contradiction. □

Lemma 3 *If* $n = 15$ *and* $\delta(1) = 12$ *then the degree sequence is either*

1. $(12, 12, 12, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$*, or*
2. $(12, 6, 6, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$.

Proof. Let u be a vertex that has degree 12 and $N(u)$ the set of vertices that are in the neighbourhood of (i.e. adjacent to) u. Then there are two vertices, v and w that are not u and are not in $N(u)$. No element of $N(u)$ can have degree greater than 3, for then it would have degree at least 6 and must be adjacent to at least two other elements of $N(u)$, and we would have a diamond. Let $\delta(v)$ and $\delta(w)$ be the degrees of v and v respectively. Without loss of generality we can assume that $\delta(v) \geq \delta(w)$. Then G has degree sequence $(12, \delta(v), \delta(w), 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$, where $\delta(v)+\delta(w)$ is divisible by 12. Hence $(\delta(v), \delta(w))$ is $(12, 12)$, $(9, 3)$ or $(6, 6)$.

If $(\delta(v), \delta(w)) = (12, 12)$ there is a solution. In this case every element of $N(u)$ is adjacent to both v and w.

If $(\delta(v), \delta(w)) = (9, 3)$, then suppose that v and w are adjacent. None of the 8 vertices in $N(u)$ that are adjacent to v can be adjacent to w (or we have a diamond), so they must all be adjacent to one of the 4 remaining vertices in $N(u)$. Hence some vertices in $N(u)$ are adjacent to more than one other vertex in $N(u)$, and there is a diamond. A similar argument holds if v and w are not adjacent.

If $(\delta(v), \delta(w)) = (6, 6)$ there is a solution and it can be constructed as follows. Divide the 12 vertices in $N(u)$ into two disjoint sets of equal cardinality $\alpha = \{a_1 \ldots a_6\}$ and $\beta = \{b_1 \ldots b_6\}$. Connect vertex a_i to b_i for $1 \leq i \leq 6$. Now connect vertex v to all vertices in α and connect w to all vertices in β. Such a graph is shown in Figure 3. □

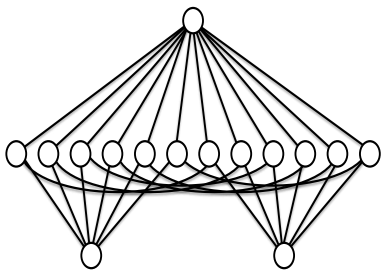


Figure 3: A diamond-free graph with 15 vertices and degree sequence $(12, 6, 6, 6, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$

Lemma 4 *If* $n = 16$ *and* $\delta(1) = 12$ *then the degree sequence is either*

1. $(12, 12, 9, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$*, or*
2. $(12, 12, 6, 6, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$*, or*
3. $(12, 9, 9, 6, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$*, or*
4. $(12, 6, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$*.*

Proof. Let u and $N(u)$ be defined as above, and let v, w and x be vertices that are not u and are not in $N(u)$, with corresponding degrees $\delta(v) \geq \delta(w) \geq \delta(x)$. By an argument similar to the above, G has degree sequence

$$(12, \delta(v), \delta(w), \delta(x), 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$$

where $\delta(v)+\delta(w)+\delta(x)$ is divisible by 12. Then we must have $(\delta(v), \delta(w), \delta(x)) = (12, 12, 12)$, $(12, 9, 3)$, $(12, 6, 6)$ or $(9, 9, 6)$ or $(6, 3, 3)$. If $(\delta(v), \delta(w), \delta(x)) = (12, 12, 12)$ then there are at least 3 vertices in $N(u)$ that are adjacent to all of u, v, w and x, which is impossible since, as before, vertices in $N(u)$ must have degree 3. In all other cases there are solutions (which we do not include here). $\square$

4 Constraint programming models

We present two constraint models for the diamond-free degree sequence problem. The first model we call model A, the second model B. In many respects the two models are very similar but what is different is how we solve them. In the subsequent descriptions we assume that we have as input the integer n, where $|V| = n$ and vertex $i \in V$. All the constraint models were implemented using the choco toolkit [9]. Further we assume that a variable x has a domain of values $\mathsf{dom}(x)$.

4.1 Model A

Model A is based on the adjacency matrix model of a graph. We have a 0/1 constrained integer variable A_{ij} for each potential edge in the graph such that $A_{ij} = 1 \iff \{i, j\} \in E$. In addition we have constrained integer variables deg_1 to deg_n corresponding to the degrees of each vertex, such that

$$\forall_{i \in [1..n]} \ \mathsf{dom}(\mathsf{deg}_i) = [3 \ .. \ n-1]. \tag{1}$$

We then have constraints to ensure that the graph is simple:

$$\forall_{i \in [1..n]} \forall_{j \in [i..n]} \ A_{i,j} = A_{j,i} \tag{2}$$

$$\forall_{i \in [1..n]} \ A_{i,i} = 0. \tag{3}$$

Constraints are then required to ensure that the graph is diamond-free:

$$\forall_{i<j<k<l \in [1..n]} [A_{i,j} + A_{i,k} + A_{i,l} + A_{j,k} + A_{j,l} + A_{k,l} \leq 4]. \tag{4}$$

Finally we have constraints on the degree sequence:

$$\forall_{i \in [1 \ .. \ n]} \ \ \mathsf{deg}_i = \sum_{j=1}^{j=n} A_{i,j} \tag{5}$$

$$\forall_{i \in [1 \ .. \ n-1]} \ \ \mathsf{deg}_i \geq \mathsf{deg}_{i+1} \tag{6}$$

$$\forall_{i \in [1 \ .. \ n]} \ \ \mathsf{deg}_i \ \mathbf{mod} \ 3 = 0 \tag{7}$$

$$\left(\sum_{i=1}^{i=n} \mathsf{deg}_i \right) \ \mathbf{mod} \ 12 = 0. \tag{8}$$

The vertex degree variables deg_1 to deg_n are the decision variables. The constraint model uses $O(n^2)$ constrained integer variables and $O(n^4)$ constraints.

4.2 Model B

Model B is essentially model A broken into three parts, each part solved separately. The first part is to produce a degree sequence that meets the arithmetic constraints. The second part tests if that degree sequence is graphical and if it is the third part determines if there exists a diamond-free graph with that degree sequence. Therefore solving proceeds as follows.

Step 1 Generate the next degree sequence $\pi = d_1, d_2, \ldots, d_n$ that meets the arithmetical constraints. If no more degree sequences exist then terminate the process.

Step 2 If the degree sequence π is not graphical return to Step 1.

Step 3 Determine if there is a diamond-free graph with degree sequence π.

Step 4 Return to **Step 1**.

The first part of model B is then as follows. Integer variables $\mathtt{deg}_1$ to $\mathtt{deg}_n$ correspond to the degrees of each vertex and satisfy constraints (1), (6), (7), and (8) to generate a degree sequence.

Each valid degree sequence produced is then tested to determine if it is graphical (Step 2 above) using the Havel-Hakimi algorithm. We have used the $\Theta(n^2)$ algorithm [8] although the linear Erdős–Gallai type [10] or linear Havel–Hakimi type [11] algorithms could equally well be used and would have been more efficient.

If the degree sequence is graphical (Step 3) we create an adjacency matrix with properties (2) and (3) and post the constraints (4) and (5) (diamond free with given degree sequence) where the variables $\mathtt{deg}_1$ to $\mathtt{deg}_n$ have already been instantiated (in Step 1). Finally we are in a position to post static symmetry breaking constraints. If we are producing a graph and $\mathtt{deg}_i = \mathtt{deg}_j$ then these two vertices are interchangeable. Consequently we can insist that row i in the adjacency matrix is lexicographically less than or equal to row j. Therefore we post the symmetry breaking constraints:

$$\forall_{i \in [1\ ..\ n-1]} [\mathtt{deg}_i = \mathtt{deg}_{i+1} \Rightarrow A_i \preceq A_{i+1}] \tag{9}$$

where $\preceq$ means lexicographically less than or equal. In this second stage of solving the variables $A_{1,1}$ to $A_{n,n}$ are the decision variables.

5 Solutions

Our results are tabulated in Table 1 for $8 \leq n \leq 16$. All our results are produced using model B run on a machine with 8 Intel Zeon E5420 processors running at 2.50 GHz, 32Gb of RAM, with version 5.2 of linux. The longest run time was for $n = 16$ taking about 5 minutes cpu time. Included in Table 1 is the cpu time in seconds to generate all degree sequences for a given value of n.

All our results were verified. For each degree sequence the corresponding adjacency matrix was saved to file and verified to correspond to a simple

n	time	degree sequence
8	0.1	3 3 3 3 3 3 3 3
9	0.1	6 6 6 3 3 3 3 3 3
10	0.5	6 6 3 3 3 3 3 3 3 3
11	0.8	6 3 3 3 3 3 3 3 3 3 3
12	1.4	3 3 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 6
		9 6 6 3 3 3 3 3 3 3 3 3
13	3.7	6 6 6 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 3 3
		9 6 3 3 3 3 3 3 3 3 3 3 3
14	14.0	6 6 3 3 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 6 6 6
		9 3 3 3 3 3 3 3 3 3 3 3 3 3
		9 6 6 6 6 3 3 3 3 3 3 3 3 3
		9 9 6 6 3 3 3 3 3 3 3 3 3 3
		9 9 9 3 3 3 3 3 3 3 3 3 3 3
15	107.7	6 3 3 3 3 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 6 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 6 6 3 3
		9 6 6 6 3 3 3 3 3 3 3 3 3 3 3
		9 6 6 6 6 6 6 6 3 3 3 3 3 3 3
		9 6 6 6 6 6 6 6 6 6 6 6 3 3 3
		9 9 6 3 3 3 3 3 3 3 3 3 3 3 3
		9 9 6 6 6 6 6 3 3 3 3 3 3 3 3
		9 9 6 6 6 6 6 6 6 6 6 3 3 3 3
		9 9 9 6 6 6 3 3 3 3 3 3 3 3 3
		9 9 9 9 9 9 6 6 6 6 6 6 6 6 6
		12 6 6 3 3 3 3 3 3 3 3 3 3 3 3
		12 12 12 3 3 3 3 3 3 3 3 3 3 3 3
16	339.8	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3
		6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
		9 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3
		9 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3
		9 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3
		9 9 3 3 3 3 3 3 3 3 3 3 3 3 3 3
		9 9 6 6 6 6 3 3 3 3 3 3 3 3 3 3
		9 9 6 6 6 6 6 6 6 6 3 3 3 3 3 3
		9 9 6 6 6 6 6 6 6 6 6 6 6 6 3 3
		9 9 9 6 6 3 3 3 3 3 3 3 3 3 3 3
		9 9 9 6 6 6 6 6 6 6 6 6 6 3 3 3
		9 9 9 9 3 3 3 3 3 3 3 3 3 3 3 3
		9 9 9 9 6 6 6 6 6 6 6 6 3 3 3 3
		9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 6
		9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 3
		9 9 9 9 9 9 6 6 6 6 6 6 6 6 3 3
		12 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3
		12 9 9 6 3 3 3 3 3 3 3 3 3 3 3 3
		12 12 6 6 3 3 3 3 3 3 3 3 3 3 3 3
		12 12 9 3 3 3 3 3 3 3 3 3 3 3 3 3

Table 1: Degree sequences, of length n, that meet the arithmetic constraints and have a simple diamond-free graph. Tabulated is n, cpu time in seconds to generate all sequences of length n and those sequences.

diamond-free graph that matched the degree sequence and satisfied the arithmetic constraints and this is an $\Theta(n^4)$ process. The verification software did not use any of the constraint programming code.

6 Conclusion

We have presented a new problem, the generation of all degree sequences for diamond free graphs subject to arithmetic constraints. Two models have been presented, A and B. Model A is impractical whereas model B is two stage and allows static symmetry breaking.

There are two possible improvements. The first is to model A. We might add the lexicographical constraints, as used in model B, conditionally during search. The second improvement worthy of investigation is to employ a mixed integer programming solver for the second stage of model B.

We are currently using the lists of feasible degree sequences for diamond-free graphs with 15 or 16 vertices to simplify our proofs for the classification of 4^{s_4} structures with diamond-free complements, when the number of points in the maximum number of blocks is 1 or 0 respectively. The degree sequence results for a smaller number of points will also help to simplify our existing proofs for cases where more points are in the maximum number of blocks. Ultimately we would like to use our classification to modify the extension of Stinson's algorithm for block size 4 to ensure that a block design is always produced.

In the more distant future, we would like to analyse the structures produced using our algorithm when $n > 16$. The next case is $n = 25$ and the corresponding diamond-free graphs would have up to 25 vertices.

Acknowledgements

We would like to thank Ian Miguel for his help in setting up the original CSPLib problem [1], and Mike Codish and Brendan McKay for pointing out errors in an earlier draft of this paper.

References

[1] CSPLib: a problem library for constraints. ⇒189, 199

[2] J. Barát, M. Stojaković, On winning fast in avoider-enforcer games. *Electron. J. Comb.*, **17** (2010) #56. ⇒190

[3] C. J. Colbourn, J. H. Dinitz (eds.), *Handbook of Combinatorial Designs*. CRC Press, New York, USA, 1996. ⇒191

[4] J. H. Dinitz, D. R. Stinson, A fast algorithm for finding strong starters, *SIAM J. Alg. Discrete Math.,* **2,** 1 (1981) 50–56. ⇒191

[5] J. H. Dinitz, D. R. Stinson, A hill-climbing algorithm for the construction of one-factorizations and room squares, *SIAM J. Alg. Discrete Math.,* **8** (1987) 430–438. ⇒191

[6] E. Eschen, C. Hoànga, J. Spinrad, R. Sritharan, On graphs without a C_4 or a diamond, *Discrete Appl. Math.*, **159,** 7 (2011) 581–587. ⇒190

[7] J.-L. Guo, T.-M. Wang, Y.-L. Wang, Unique intersectability of diamond-free graphs, *Discrete Appl. Math.*, **159,** 8 (2011) 774–778. ⇒190

[8] 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. ⇒197

[9] Intellij IDEA, JProfiler, CHOCO Solver. Current version: Choco-2.1.5. Downloaded: October 14, 2012. ⇒196

[10] 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. ⇒197

[11] A. Iványi, Degree sequences of multigraphs. *Annales Univ. Sci. Budapest., Sect. Comp.* **37** (2012) 195–214. ⇒197

[12] T. P. Kirkman, On a problem in combinatorics, *Cambridge Dublin Math. J.,* **2** (1847) 191–204. ⇒192

[13] D. L. Kreher, D. R. Stinson, *Combinatorial Algorithms*, The CRC Press Series on Discrete Mathematics and its Applications, CRC Press, Boca Raton, Florida, USA, 1999. ⇒192

[14] C. Lai, A note on potentially $K_4 - e$ graphical sequences. *Australas. J. Comb.,* **24** (2001) 123–127. ⇒190

[15] D. R. Stinson, Hill-climbing algorithms for the construction of combinatorial designs, *Annals of Discrete Math.* (Algorithms in Combinatorial Design Theory, Vol. 26) **114** (1985) 321–334. ⇒191

Received: September 6, 2012 • Revised: December 7, 2012

Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 201–214

The complexity of an exotic edge coloring of graphs

Sándor SZABÓ
University of Pécs, Hungary
email: sszabo7@hotmail.com

Abstract. Coloring the nodes of a graph is a commonly used preprocessing method to speed up clique search procedures. For the very same purpose we propose coloring the edges of the graph. It will be shown that the recommended type of edge coloring leads to an NP-complete problem. Therefore in practical computations we should rely on some approximate algorithm.

1 Introduction

Let $G = (V, E)$ be a finite simple graph. In other words G has finitely many nodes and G does not have any double edge or loop. In this situation an edge of G can be identified with a two element subset of V. Consequently the set of edges E of G forms a family of two element subsets of V. A subgraph Δ of G is a clique if each two distinct nodes in Δ are adjacent. A clique with k nodes is called a k-clique. The number of the nodes of a clique sometimes referred as the size of the clique. A k-clique in G is a maximal clique if it is not a subgraph of any $(k+1)$-clique in G. A k-clique Δ in G is a maximum clique if G does not have any $(k+1)$-clique. The size of a maximum clique in G is called the clique size of G and it is denoted by $\omega(G)$.

Computing the clique size of a given graph has many important applications inside and outside of mathematics. Many of these applications are described in [1]. It was pointed out in [2] that the performance of their algorithms to

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 05C15, 03D15
Key words and phrases: maximal clique, maximum clique, k-clique, graph coloring, coloring the nodes, coloring the edges, NP-complete problems

determine the clique size of a given graph is critically depend on efficiently computable upper estimates of clique sizes. The most commonly used method to estimate clique size is via coloring the nodes of the graph. Although there is an interesting additional technique based on dynamic programming presented in [5] and further developed in [4] in this paper we will restrict our attention to the coloring idea.

Suppose that the nodes of a finite simple graph G are colored using k given colors such that

(1) each node of G receives exactly one color,

(2) adjacent nodes never receive the same color.

This type of coloring is the most commonly encountered coloring of the nodes of a graph. We will refer to it saying that the vertices of G have an L type coloring with k colors. The letter L stands for the expression legal coloring. The connection between the coloring and the clique size is the following. If the nodes of the graph G have an L type coloring with k colors then $\omega(G) \leq k$.

It should not came to us as a surprise that coloring the edges of a graph can provide upper estimates for the clique size. We color the edges of a graph G with k colors such that

(1) each edge of G receives exactly one color,

(2) if x, y, z are nodes of a 3-clique in G, then the edges $\{x, y\}$, $\{x, z\}$, $\{y, z\}$ receive three distinct colors.

For the sake of easier reference we call this type of coloring of the edges of a graph an S type edge coloring. The coloring could be called a rainbow triangle coloring. (The letter S stands for the initial letter of the word "rainbow" in Hungarian.) The minimum number of colors k for which the edges of an n-clique have an S type coloring is denoted by $\chi_S(n)$. A possible connection between the edge coloring and the clique size of a graph is the following. If the edges of a graph G have an S type coloring with k colors and $\omega(G) = t$, then $\chi_S(t) \leq k$ must hold. In other words if the edges of G have an S type coloring with k colors and $\chi_S(t) > k$, then $\omega(G) < t$.

It is not hard to construct an S type coloring of the edges of a given graph in greedy fashion. A greedy S type edge coloring together with the next lemma provide a practical way to estimate the clique size.

Lemma 1 $\chi_S(n) \geq n - 1$ *for each positive integer* n.

Proof. Let $\Delta = (V, E)$ be an n-clique and suppose that the edges of Δ have an S type coloring with k colors. Let $f : E \to \{1, \ldots, k\}$ be the coloring of the edges of Δ. Finally let $v_1, \ldots, v_n$ be all the vertices of Δ. Note that $f(\{v_1, v_i\}) \neq f(\{v_1, v_j\})$ holds for each i, j, $1 \leq i < j \leq n$ since the edges $\{v_1, v_i\}$, $\{v_1, v_j\}$, $\{v_i, v_j\}$ receive three distinct colors. In particular the edges $\{v_1, v_i\}$, $2 \leq i \leq n$ receive distinct colors. It follows that $\chi_S(n) \geq k \geq n - 1$, as required. □

The reader will notice that the exact value of $\chi_S(n)$ can be determined. Namely,

$$\chi_S(n) = \begin{cases} n, & \text{if } n \text{ is odd,} \\ n-1, & \text{if } n \text{ is even} \end{cases}$$

holds for each $n \geq 2$. Suppose that the edges of an n-clique Δ have an S type coloring. The edges of Δ receiving color c form the c-color class of the coloring. Notice that the edges in the c-color class must form a matching in Δ. A maximum matching is a 1-factor of Δ. It is a known result that an n-clique can be decomposed into $n-1$ 1-factors if n is even. Further, an n-clique cannot be decomposed into $n-1$ 1-factors when n is odd. In the next section we will see that the cruder result stated in Lemma 1 will suffice for our purposes.

By Lemma 1, if the edges of a given graph G have an S type coloring with k colors and $n - 1 > k$, then $\omega(G) < n$. By the main result of this note the problem to decide if the edges of a given graph have an S type coloring with k colors is an NP-complete problem for $k \geq 3$. This result loosely can be interpreted such that determining the minimum value of k for which the edges of G have an S type coloring with k colors is a computationally demanding problem.

2 Numerical experiments

The main motivation of this paper is to explore the possibility of utilizing edge coloring in clique search algorithms. It is relatively straightforward to construct S type edge coloring for a given graph in a greedy fashion. The greedy algorithm does not provide the optimum number of colors but it is computationally feasible.

Let $G = (V, E)$ be a given graph and suppose that we want to find an S type coloring of the edges of G. We locate a clique Δ in G. The clique Δ is not necessarily a largest clique in G. For our purposes any suboptimal clique is suitable. Let $e_1, \ldots, e_m$ be a fixed list of the edges of the given graph G such that we list first the edges of Δ then we list the remaining edges of G. Decomposing Δ into 1-factors and using the 1-factors as color classes we can

Name	$\|V\|$	$\|E\|$	L	S
MON03	27	189	6	9
MON04	64	1296	12	20
MON05	125	5500	20	35
MON06	216	17550	30	57
MON07	343	46305	42	79
MON08	512	106624	56	108
MON09	729	221616	72	141
MON10	1000	425250	90	178
MON11	1331	765325	110	218
MON12	1728	1306800	132	261
MON13	2197	2135484	156	309
MON14	2744	3362086	182	361
MON15	3375	5126625	210	418

Table 1: Graphs associated with monotonic matrices.

color the edges of Δ and we end up with a partial coloring of the edges of G. Suppose $C_1, \dots, C_r$ are the existing color classes and e_i is the first uncolored edge of G. The edge e_i can be placed into the colors class C_1 if C_1 does not contain any edge e_j such that e_i and e_j are edges of a 3-clique in G. If e_i can be placed into C_1, then we put e_i into C_1. If e_i does not fit into C_1, then we try to place it into C_2. Continuing in this way either e_i fits into one of the colors classes $C_1, \dots, C_r$ or we open a new color class C_{r+1} for e_i. When all the edges on the list $e_1, \dots, e_m$ are colored, then we have an S type coloring of the edges of G.

We carried out a large scale numerical experiment to compare the upper estimates for the clique size of the given graph G provided by the ordinary L type node coloring and the proposed S type edge coloring of G. The results are summarized in Tables 1, 2, and 3. We considered $13+10+13 = 36$ graphs. These graphs are coming from coding theory. They are related to monotonic matrices, deletion error detecting, and error correcting codes, respectively. Using sequential greedy coloring algorithms we constructed an L type coloring of the nodes and an S type coloring of the edges for each graph. In the tables we listed the number of colors, the number of nodes and the number of edges of the graphs. From the results it is fairly clear that the greedy node coloring provides tighter estimates for the clique sizes of the graphs than the edge coloring does. Therefore in a clique search algorithm we do not recommend to

Name	$\lvert V\rvert$	$\lvert E\rvert$	L	S
DEL03	8	9	2	1
DEL04	16	57	4	4
DEL05	32	305	8	11
DEL06	64	1473	14	24
DEL07	128	6657	26	53
DEL08	256	28801	50	114
DEL09	512	121089	101	236
DEL10	1024	499713	199	492
DEL11	2048	2037761	395	995
DEL12	4096	8247297	782	2024

Table 2: Graphs associated with deletion error correcting codes.

Name	$\lvert V\rvert$	$\lvert E\rvert$	L	S
JOHNSON06	15	45	4	3
JOHNSON07	35	385	10	11
JOHNSON08	70	1855	20	26
JOHNSON09	126	6615	35	52
JOHNSON10	210	19425	56	85
JOHNSON11	330	49665	84	131
JOHNSON12	495	114345	120	197
JOHNSON13	715	242385	165	279
JOHNSON14	1001	480480	220	377
JOHNSON15	1365	900900	286	496
JOHNSON16	1820	1611610	364	646
JOHNSON17	2380	2769130	455	813
JOHNSON18	3060	4594590	560	1008

Table 3: Graphs associated with Johnson error correcting codes.

replace greedy sequential L type coloring of the nodes by greedy sequential S type coloring of the edges. We suggest to use the edge coloring in a different fashion. It can be used as a preconditioning tool.

We color the edges of the given graph G before the clique search starts. One can store the colors of the edges of G in an n by n matrix M conveniently. Here n is the number of the nodes of G. The rows and columns of M are labeled by the nodes of G and $m_{u,v}$ is the entry of M in the row labeled by node u and column labeled by node v. If c is the color of the edge $\{u, v\}$, then we set $m_{u,v}$ to be c. In the course of a clique search we can read off the colors of the edges from the matrix M with relatively low cost. Let H be a subgraph of G and suppose we are looking for a k-clique Δ in H. Note that if the edges of G have an S type coloring, then by inheritance the edges of H have an S type coloring too. The edges joining to a node v of Δ must have pair-wise distinct colors. Therefore if the edges of H joining to the node v are colored with less than $k-1$ colors, then v can be deleted from H. Deleting nodes from H reduces the size of the search space and might help in speeding up the computation.

3 A complexity result

Let $\Gamma = (V, E)$ be a finite simple graph. Using Γ we construct a new graph $G' = (V', E')$. We try to establish the following facts.

(1) If the nodes of Γ have an L type coloring with 3 colors, then the edges of G' have an S type coloring with 3 colors.

(2) If the edges of G' have an S type coloring with 3 colors, then the nodes of Γ have an L type coloring with 3 colors.

Let $v_1, \ldots, v_n$ be all the nodes of Γ. We assign a graph H_i to v_i for each i, $1 \leq i \leq n$. The constructions of H_i and G' are guided by the structure of the incidence matrix of Γ. The incidence matrix of Γ has $n = |V|$ rows and $m = |E|$ columns. The rows are labeled by the nodes $v_1, \ldots, v_n$ and the columns are labeled by the edges of Γ. If $e_k = \{v_i, v_j\}$ is an edge of Γ, then the two cells at the intersection of rows v_i, v_j and column e_k both contain a bullet.

We illustrate the construction working out the details in connection with a toy example. The graph Γ in the example can be seen in Figure 1 and the incidence matrix of this graph is in Table 4.

To vertex v_i of Γ we assign a graph H_i which has $4m$ nodes, where $m = |E|$. Let $K = (V'', E'')$ be a 4-clique such that $V'' = \{a, b, c, d\}$. We take m isomorphic copies $K_{i,1}, \ldots, K_{i,m}$ of K. We choose the notation such that $K_{i,j} =$

	e_1	e_2	e_3	e_4	e_5
v_1	●	●	●		
v_2	●			●	
v_3		●		●	●
v_4			●		●

Table 4: The node edge incidence matrix of of the graph Γ in the toy example.

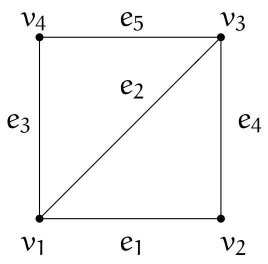


Figure 1: A geometric representation of the graph Γ in the toy example.

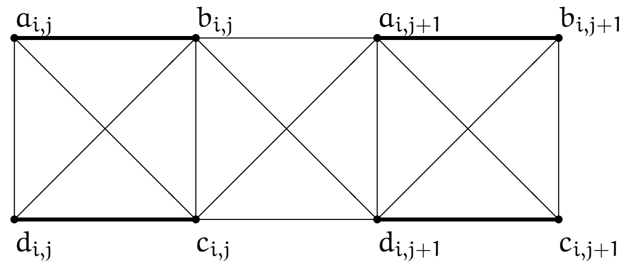


Figure 2: A step of the construction of H_i. The 1st square is $K_{i,j}$ and the 3rd square is $K_{i,j+1}$.

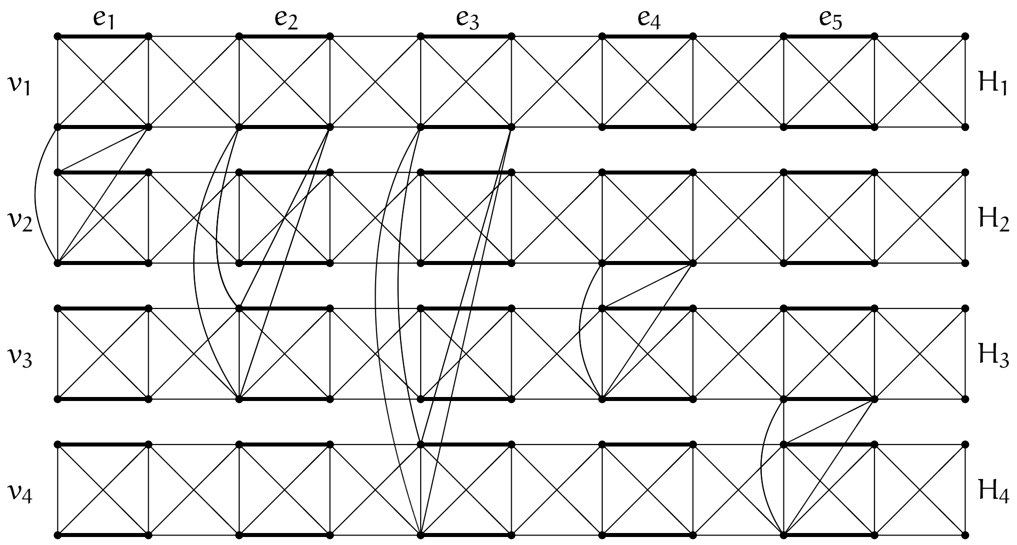


Figure 3: The graphs $H_1, \ldots, H_4$ in the toy example.

$(V''_{i,j}, E''_{i,j})$ and $V''_{i,j} = \{a_{i,j}, b_{i,j}, c_{i,j}, d_{i,j}\}$. We add the edges

$$\{b_{i,j}, a_{i,j+1}\}, \quad \{b_{i,j}, d_{i,j+1}\}, \quad \{c_{i,j}, a_{i,j+1}\}, \quad \{c_{i,j}, d_{i,j+1}\},$$

for each j, $1 \le j \le m-1$. This step of the construction is depicted in Figure 2. Finally we add the edges

$$\{b_{i,m}, a_{i,1}\}, \quad \{b_{i,m}, d_{i,1}\}, \quad \{c_{i,m}, a_{i,1}\}, \quad \{c_{i,m}, d_{i,1}\}.$$

We encourage the reader to visualize H_i as a long narrow paper strip divided into $2m$ squares. The two opposite short sides of the rectangle are united to form a closed strip. However, we draw H_i as an open flattened strip in Figure 3 in order not to clutter the diagram. Figure 3 exhibits the geometric representations of the graphs $H_1, \ldots, H_4$ associated with the vertices $v_1, \ldots, v_4$ of the toy example Γ.

If v_i and v_j are adjacent edges in Γ such that $i < j$, then to represent the edge $e_k = \{v_i, v_j\}$ of Γ in G' we add the edges

$$\{a_{j,k}, c_{i,k}\}, \quad \{a_{j,k}, d_{i,k}\}, \quad \{d_{j,k}, c_{i,k}\}, \quad \{d_{j,k}, d_{i,k}\}$$

to G'. (The reader may follow the flow of the argument in Figure 6.) If v_i and v_j are not adjacent edges in Γ, then we do not add any extra edges to G'. The

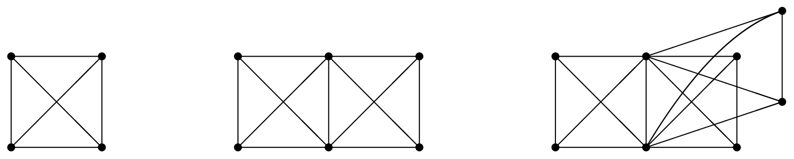


Figure 4: The subgraphs spanned by $N(u, v)$ in the proof of Lemma 2.

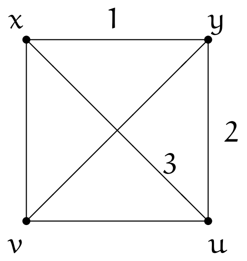


Figure 5: Coloring the edges of a 4-clique in the proof of Lemma 3.

toy example Γ has five edges

$$e_1 = \{v_1, v_2\}, \quad e_2 = \{v_1, v_3\}, \quad e_3 = \{v_1, v_4\},$$
$$e_4 = \{v_2, v_3\}, \quad e_5 = \{v_3, v_4\}.$$

The reader can spot five modifications corresponding to these edges in Figure 3.

When we analyze the graph G' we will use the following two lemmas.

Lemma 2 *If* Γ *has at least one edge, then the clique number of* G' *is equal to* 4*. In symbols* $\omega(G') = 4$*.*

Proof. Since Γ has an edge, it follows that G' contains a 4-clique. Consequently $\omega(G') \geq 4$. It remains to show that $\omega(G') \leq 4$.

Let Δ be a maximum clique in G' and let $\{u, v\}$ be an edge in Δ. Let $N(u, v)$ be the set of the next nodes of G'.

(1) The nodes u and v.

(2) All the nodes adjacent to both u and v.

We call $N(u, v)$ the neighborhood set of the edge $\{u, v\}$.

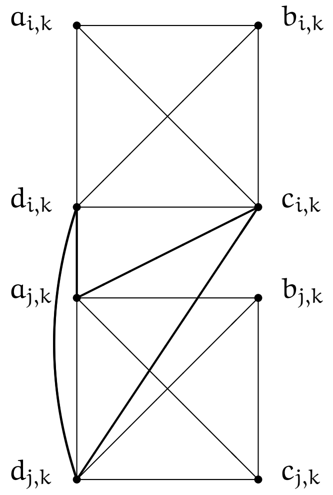


Figure 6: The connecting device in the first construction.

An inspection shows that the subgraph of G' spanned by the neighborhood set $N(u, v)$ can only be one of the three graphs shown in Figure 4. Since Δ must be a subgraph of the graph spanned by $N(u, v)$, it follows that $\omega(G') \leq 4$. □

Lemma 3 *Let Δ be a 4-clique with nodes x, y, u, v. If the edges of Δ have an S type coloring with 3 colors, then the "opposite" edges $\{x, y\}$ and $\{u, v\}$ must receive the same color.*

Proof. The edges of the 3-clique whose nodes are x, y, u must receive three distinct colors. We may assume that the edges $\{x, y\}$, $\{y, u\}$, $\{x, u\}$ receive colors 1, 2, 3 respectively since this is only a matter of rearranging the colors 1, 2, 3 among each other. Edge $\{x, v\}$ cannot receive color 1 because $\{x, v\}$ and $\{x, y\}$ are edges of the 3-clique with nodes x, y, v. Edge $\{x, v\}$ cannot receive color 3 since $\{x, v\}$ and $\{x, u\}$ are edges of the 3-clique with nodes x, u, v. Thus edge $\{x, v\}$ must receive color 2. Finally, edge $\{y, v\}$ has to be colored with color 3 and edge $\{u, v\}$ must be colored with color 1. (The reasoning can be followed in Figure 5.) □

Suppose now that the nodes of Γ have an L type coloring with 3 colors. Let $f : V \to \{1, 2, 3\}$ be the coloring. Let us consider the subgraph H_i of G' assigned to node v_i of Γ. We color the edge $\{a_{i,1}, d_{i,1}\}$ of G' with color $f(v_i)$. We know from Lemma 3 that the edge $\{b_{i,1}, c_{i,1}\}$ must be colored with color $f(v_i)$ in order to define an S type coloring of the edges of G' with 3 colors. Therefore we color all the "vertical" edges

$$\{a_{i,j}, d_{i,j}\}, \quad \{b_{i,j}, c_{i,j}\}, \quad 1 \leq j \leq m$$

of H_i with color $f(v_i)$. In a fixed 4-clique in H_i we color the opposite "horizontal" edges with the same color. Similarly in a fixed 4-clique in H_i we color the "diagonal" edges with the same color. If the colors used for the vertical, horizontal, and diagonal edges are pair-wise distinct, then the edges of H_i have an S type coloring with 3 colors.

There are further edges in G' which play the role of connecting devices between H_i and H_j when v_i and v_j are adjacent nodes of Γ. Let $e_k = \{v_i, v_j\}$ be the edge of Γ connecting the vertices v_i and v_j. The color of the edge $\{a_{j,k}, d_{j,k}\}$ has already been assigned to be $f(v_j)$. This forces us to color the edge $\{d_{i,k}, c_{i,k}\}$ with color $f(v_j)$. But in the 4-clique $K_{i,k}$ with edges $a_{i,k}$, $b_{i,k}$, $c_{i,k}$, $d_{i,k}$ only the color of the vertical edges are fixed to be $f(v_i)$ and so we have a freedom to choose the color of the horizontal edges.

Summing up our considerations we may say that the edges of the graph G' have an S type coloring with 3 colors provided that the nodes of Γ have an L type coloring with 3 colors.

Suppose now that the edges of G' have an S type coloring with 3 colors. Let $f' : E' \to \{1, 2, 3\}$ be this coloring. In particular the edges of the subgraph H_i of G' have an S type coloring with 3 colors for each i, $1 \leq i \leq n$. By Lemma 3, in an S type coloring of the edges of H_i the vertical edges must receive the same color. This colors is $f'(\{a_{i,1}, d_{i,1}\})$. We color the node v_i of Γ with this color. In other words we define a map $f : V \to \{1, 2, 3\}$ by setting $f(v_i)$ to be $f'(\{a_{i,1}, d_{i,1}\})$.

We claim that $f(v_i) = f(v_j)$ implies that v_i and v_j are not adjacent nodes of Γ.

In order to verify the claim assume on the contrary that v_i and v_j are adjacent nodes of Γ and $f(v_i) = f(v_j)$ holds. Let $e_k = \{v_i, v_j\}$ be the edge of Γ that connects the nodes v_i and v_j. Let us consider the 4-clique $K_{i,j,k}$ of G' whose vertices are $c_{i,k}$, $d_{i,k}$, $a_{j,k}$, $d_{j,k}$. Since the edges of G' have an S type coloring with 3 colors, it follows that the edges of the 4-clique $K_{i,j,k}$ have an S type coloring with 3 colors. Lemma 3 is applicable to $K_{i,j,k}$ and gives that the edge $\{a_{j,k}, d_{j,k}\}$ of H_j and the edge $\{d_{i,k}, c_{i,k}\}$ of H_i are colored with the same color. This common color is $f(v_j)$. The vertical edge $\{a_{i,k}, d_{i,k}\}$ of H_i is colored with color $f(v_i)$. This implies $f(v_i) = f(v_j)$. From $f(v_i) = f(v_j)$, it follows that two edges of the 3-clique with nodes $a_{i,k}$, $d_{i,k}$, $c_{i,k}$ are colored with the same

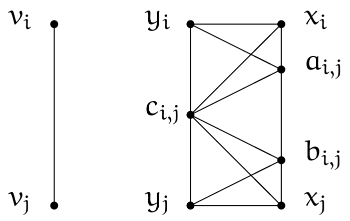


Figure 7: The graph assigned to the edge $\{v_i, v_j\}$.

color. Namely the edges $\{a_{i,k}, d_{i,k}\}$ and $\{c_{i,k}, d_{i,k}\}$ are receiving the same color. This contradiction proves our claim.

Theorem 4 *The problem to decide if the edges of a finite simple graph have an S type coloring with 3 colors is an NP-complete problem.*

Proof. For the proof we should recall the known result that the problem of deciding if the nodes of a finite simple graph have an L type coloring with 3 colors is an NP-complete problem. The result on the coloring of the nodes can be found for example in [3] or [6]. □

4 An alternative construction

In this section we give a second proof for Theorem 4 using a new construction.

Proof. Let $\Gamma = (V, E)$ be a finite simple graph. Using Γ we construct a new graph $G' = (V', E')$. We try to show that the following requirements hold.

(1) If the nodes of Γ have an L type coloring with 3 colors, then the edges of G' have an S type coloring with 3 colors.

(2) If the edges of G' have an S type coloring with 3 colors, then the nodes of Γ have an L type coloring with 3 colors.

Let $v_1, \ldots, v_n$ be all the nodes of Γ. We assign two points x_i and y_i to node v_i for each i, $1 \leq i \leq n$. We choose the points $x_1, \ldots, x_n$, $y_1, \ldots, y_n$ to be pair-wise distinct. We connect the nodes x_i and y_i in G' with an edge.

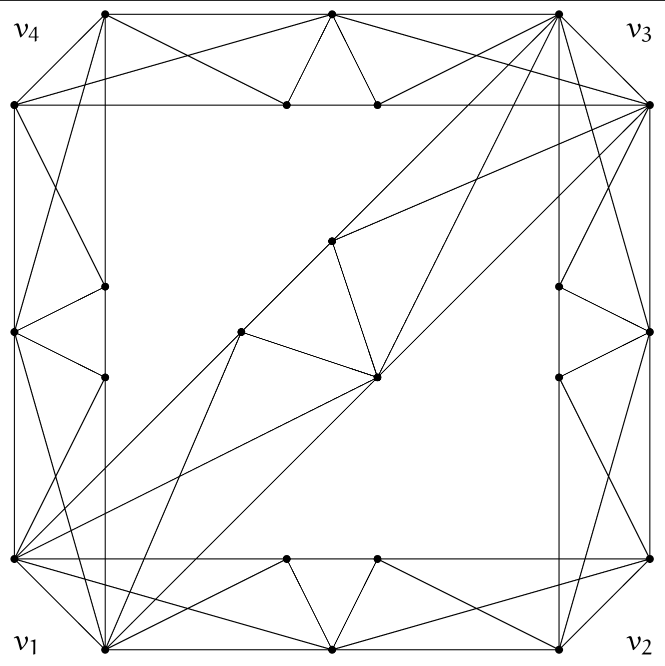


Figure 8: The graph G' associated with the toy example.

If ν_i and ν_j are adjacent nodes in Γ, then we add three new nodes $a_{i,j}$, $b_{i,j}$, $c_{i,j}$ and eleven new edges

$$\begin{array}{llll} \{x_i, a_{i,j}\}, & \{x_i, c_{i,j}\}, & \{y_i, a_{i,j}\}, & \{y_i, c_{i,j}\}, \\ \{x_j, b_{i,j}\}, & \{x_j, c_{i,j}\}, & \{y_j, b_{i,j}\}, & \{y_j, c_{i,j}\}, \\ \{a_{i,j}, b_{i,j}\}, & \{a_{i,j}, c_{i,j}\}, & \{b_{i,j}, c_{i,j}\}. & \end{array}$$

If ν_i and ν_j are not adjacent in Γ, then we do not add any new node or new edge to G'. This step of the construction is illustrated in Figure 7. The graph G' associated with the toy example is in Figure 8.

Suppose first that the nodes of Γ have an L type coloring with 3 colors. Let $f : V \to \{1, 2, 3\}$ be such a coloring. We define an edge coloring $f' : E' \to \{1, 2, 3\}$ of G'. To do so we set $f'(\{x_i, y_i\})$ to be $f(\nu_i)$ and we set

$$f'(\{a_{i,j}, c_{i,j}\}) = f(\nu_i), \quad f'(\{b_{i,j}, c_{i,j}\}) = f(\nu_j).$$

Let us consider the 3-clique Δ in G' whose nodes are $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. Two edges of Δ has already been colored. So we color the edge $\{a_{i,j}, b_{i,j}\}$ with the only color in the set $\{1, 2, 3\} \setminus \{f(\nu_i), f(\nu_j)\}$. We color the edges $\{x_i, a_{i,j}\}$, $\{y_i, c_{i,j}\}$ with one of the colors in the set $\{1,2,3\} \setminus \{f(v_i)\}$ and we color the edges $\{x_i, c_{i,j}\}$, $\{y_i, a_{i,j}\}$ with the remaining last color. Similarly, we color the edges $\{x_j, b_{i,j}\}$, $\{y_j, c_{i,j}\}$ with one of the colors in the set $\{1,2,3\} \setminus \{f(v_j)\}$ and we color the edges $\{x_j, c_{i,j}\}$, $\{y_j, c_{i,j}\}$ with the remaining last color.

An inspection shows that the coloring $f' : E' \longrightarrow \{1,2,3\}$ is an S type coloring of the edges of G'.

Next suppose that the edges of G' have an S type coloring with 3 colors. Let $f' : E' \longrightarrow \{1,2,3\}$ be such a coloring. Using the edge coloring f' of G' we define a coloring $f : V \longrightarrow \{1,2,3\}$ of the nodes of Γ by setting $f(v_i)$ to be $f'(\{x_i, y_i\})$. We claim that $f(v_i) = f(v_j)$ implies that v_i and v_j are not adjacent in Γ.

In order to prove the claim assume on the contrary that $f(v_i) = f(v_j)$ and the nodes v_i and v_j are adjacent in Γ. Since f' is an S type coloring of the edges of G', it follows that

$$\begin{aligned} f'(\{a_{i,j}, c_{i,j}\}) &= f'(\{x_i, y_i\}) &= f(v_i), \\ f'(\{b_{i,j}, c_{i,j}\}) &= f'(\{x_j, y_j\}) &= f(v_j). \end{aligned}$$

Let us watch the 3-clique Δ in G' whose nodes are $a_{i,j}$, $b_{i,j}$, $c_{i,j}$. (The reader may consult with Figure 7.) We get the contradiction that two edges of Δ are colored with the same color. □

References

[1] I. Bomze, M. Budinich, P. M. Pardalos, M. Pelillo, The Maximum Clique Problem, in: *Handbook of Combinatorial Optimization* Vol. 4 (eds. D.-Z. Du and P. M. Pardalos), Kluwer Academic Publisher, Boston, MA 1999, pp. 1–74. ⇒201

[2] R. Carraghan, P. M. Pardalos, An exact algorithm for the maximum clique problem, *Oper. Res. Lett.* **9** (1990) 375–382. ⇒201

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, New York, NY, 2003. ⇒212

[4] D. Kumlander, *Some Practical Algorithms to Solve the Maximal Clique Problem*, PhD. Thesis, Tallin University of Technology, 2005. ⇒202

[5] P. R. J. Östergård, A fast algorithm for the maximum clique problem, *Discrete Appl. Math.* **120,** 1-3 (2002) 197–207. ⇒202

[6] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley Publishing Company, Inc., Reading, MA 1994. ⇒212

Received: October 5, 2012 • Revised: November 5, 2012

Towards optimal sorting of 16 elements

Marcin PECZARSKI
Institute of Informatics, University of Warsaw,
ul. Banacha 2, 02-097 Warszawa, Poland
email: marpe@mimuw.edu.pl

Abstract. One of the fundamental problem in the theory of sorting is to find the pessimistic number of comparisons sufficient to sort a given number of elements. Currently 16 is the lowest number of elements for which we do not know the exact value. We know that 46 comparisons suffices and that 44 do not. There is an open question if 45 comparisons are sufficient. We present an attempt to resolve that problem by performing an exhaustive computer search. We also present an algorithm for counting linear extensions which substantially speeds up computations.

1 Introduction

We consider sorting by comparisons. One of the fundamental problem in that area is to find the pessimistic number $S(n)$ of comparisons sufficient to sort n elements. Steinhaus posed this problem in [8]. Knuth considered it in [4]. From the *information-theoretic lower bound*, further denoted by ITLB, we know that $S(n) \geq \lceil \log_2 n! \rceil = C(n)$. Ford and Johnson discovered [2] an algorithm, further denoted by FJA, which nearly and sometimes even exactly matches $C(n)$. Let $F(n)$ be the worst case number of comparisons in the FJA. It holds $S(n) = F(n) = C(n)$ for $n \leq 11$ and $n = 20, 21$. The FJA does not achieve the ITLB for $12 \leq n \leq 19$ and infinitely many $n \geq 22$. Carrying an exhaustive computer search, Wells discovered in 1965 [9, 10] that the FJA is optimal for 12 elements and $S(12) = F(12) = C(12) + 1 = 30$. Kasai et al. [3] computed $S(13) = F(13) = C(13) + 1 = 34$ in 1994, but that result was not widely

Computing Classification System 1998: F.2.2
Mathematics Subject Classification 2010: 06A07, 68P10
Key words and phrases: optimal sorting, poset, computer aided proof

known. It was discovered again a few years later [5], independently, extending the Wells method. Further improvement of the method led to show in years 2003–2004 [6, 7] that it holds $S(n) = F(n) = C(n) + 1$ for $n = 14, 15, 22$, similarly.

In this paper we consider the case $n = 16$. This is now the lowest number of elements for which we do not know the exact value of $S(n)$. The previous results could suggest that $S(16) = F(16) = C(16) + 1 = 46$. However Knuth conjectures that $S(16) = C(16) = 45$. He does not believe that the FJA is optimal for 16 elements. He wrote [4]: "There must be a way to improve upon this!" We present recently obtained results[1] aiming to compute the value of $S(16)$. It is very unlikely that someone will find it by pure theoretical consideration. It seems that the only promising way leads by performing an exhaustive computer search supported by cleaver heuristics.

The paper is organized as follows. In Section 2 we introduce notation used throughout the paper. In Section 3 we briefly describe the algorithm we use to resolve if there exists a sorting algorithm for a given number of elements and comparisons. We analyse why the ITLB is not achieved for 13, 14 and 15 elements in Section 4. We present the newest results for 16 elements in Section 5. In Section 6 we compare the computation complexity of the previous cases and the case of 16 elements. Finally, in Section 7, we present the algorithm for counting linear extensions which substantially improves the algorithm from Section 3.

2 Notation

We denote by $U = \{u_0, u_1, \ldots, u_{n-1}\}$ an n-element set to be sorted. Sorting of the set U is represented as a sequence of posets $(P_c = (U, R_c))_{c=0,1,\ldots,C}$, where R_c is a partial order relation over a set U. Sorting starts from the total disorder $P_0 = (U, R_0)$, where $R_0 = \{(u, u) : u \in U\}$. After performing c comparisons we obtain a poset $P_c = (U, R_c)$. Sorting should end with a linear order P_C. Assume that elements u_j and u_k are being compared in step c. Without loss of generality we can assume that $(u_j, u_k) \notin R_{c-1}$ and $(u_k, u_j) \notin R_{c-1}$. Suppose the answer to the comparison is that *element* u_j *is less than element* u_k. Then we obtain the next poset $P_c = (U, R_c)$, where the relation R_c is the transitive closure of the relation $R_{c-1} \cup \{(u_j, u_k)\}$. We denote this by $P_c = P_{c-1} + u_j u_k$.

By $e(P)$ we denote the number of linear extensions of a poset $P = (U, R)$.

[1]The results presented in this paper are obtained using computer resources of the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM), University of Warsaw.

We assume that $e(P + u_j u_k) = e(P)$ and $e(P + u_k u_j) = 0$ if elements u_j, u_k are in relation, i.e., if $(u_j, u_k) \in R$.

3 The algorithm

In this section we remember briefly the algorithm which answers if sorting of a given poset P_0 can be finished in C comparisons. The algorithm was invented in [9, 10] and improved in [5] and later in [6]. We present the next improvement to the algorithm in Section 7. The algorithm has two phases: forward steps and backward steps.

In the forward steps we consider a sequence of sets $(\mathcal{S}_c)_{c=0,1,\ldots,C}$. The set $\mathcal{S}_0$ contains only the poset P_0. In step c we construct the set $\mathcal{S}_c$ from the set $\mathcal{S}_{c-1}$. Every poset $P \in \mathcal{S}_{c-1}$ is examined for every unrelated pair (u_j, u_k) in order to verify whether it can be sorted in the remaining $C - c + 1$ comparisons. As the result of the comparison of u_j and u_k one can get one of two posets $P_1 = P + u_j u_k$ or $P_2 = P + u_k u_j$. If the number of linear extensions of P_1 or P_2 exceeds 2^{C-c} then by the ITLB it cannot be sorted in the remaining $C - c$ comparisons. It follows that in this case, in order to finish sorting in $C - c + 1$ comparisons, elements u_j and u_k should not be compared in step c. If the number of linear extensions of both P_1 and P_2 do not exceed 2^{C-c} then we store one of them in the set $\mathcal{S}_c$, namely that with greater number of linear extensions. If both have the same number of linear extensions we choose P_1 arbitrarily. We do not store isomorphic posets or a poset which dual poset is isomorphic to some already stored poset.

If some set $\mathcal{S}_c$ in the sequence appears to be empty then we conclude that the poset P_0 cannot be sorted in C comparisons. Such results are received for 12 and 22 elements and $C = C(n)$ [6], where the set $\mathcal{S}_{23}$ and $\mathcal{S}_{40}$ is empty, respectively. Wells reported [10] that for $n = 12$ only the set $\mathcal{S}_{24}$ is empty. Those results mean that $S(n) > C(n)$ for $n = 12, 22$. If the set $\mathcal{S}_C$ is not empty after performing forwards steps, we cannot conclude about sorting of the poset P_0. In that case we continue with backward steps.

In the backward steps we consider the sequence of sets $(\mathcal{S}^*_c)_{c=0,1,\ldots,C}$. We start with the set $\mathcal{S}^*_C = \mathcal{S}_C$ which contains only a linear order of the set U. In step c, where $c = C - 1, C - 2, \ldots, 0$, we construct the set $\mathcal{S}^*_c$ from the set $\mathcal{S}^*_{c+1}$. The set $\mathcal{S}^*_c$ is a subset of the set $\mathcal{S}_c$ and contains only posets which can be sorted in the remaining $C - c$ comparisons. Poset $P \in \mathcal{S}_c$ is stored in $\mathcal{S}^*_c$ iff there exists in P a pair of unrelated elements (u_j, u_k) such that poset $P_1 = P + u_j u_k$ or poset $P_2 = P + u_k u_j$ belongs to the set $\mathcal{S}^*_{c+1}$ (as previously we identify isomorphic and dual posets) and both posets are sortable in $C - c - 1$

comparisons. Therefore we store the poset P in the set $\mathcal{S}^*_c$ iff both $P_1, P_2 \in \mathcal{S}^*_{c+1}$ or $P_1 \in \mathcal{S}^*_{c+1}$ and P_2 is sortable in $C - c - 1$ comparisons or $P_2 \in \mathcal{S}^*_{c+1}$ and P_1 is sortable in $C - c - 1$ comparisons. Sortability of P_1 or P_2 can be checked recursively using the same algorithm.

If some set $\mathcal{S}^*_c$ in the sequence appears to be empty then we conclude that the poset P_0 cannot be sorted in C comparisons. On the other hand, if the set $\mathcal{S}^*_0$ is not empty, it contains the poset P_0 and we conclude that the poset P_0 can be sorted in C comparisons. For $n = 13, 14, 15$ and $C = C(n)$ we received that the set $\mathcal{S}^*_{15}$ is empty [5, 6, 7], which means that $S(n) > C(n)$ for $n = 13, 14, 15$. We analyze those results in detail in the next section.

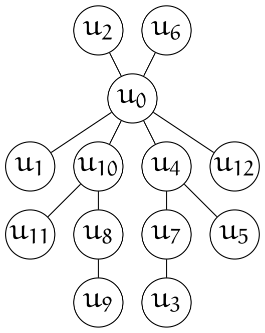
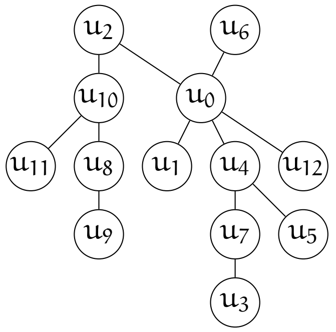


Figure 1: The poset P_{16}, $e(P_{16}) = 113400$

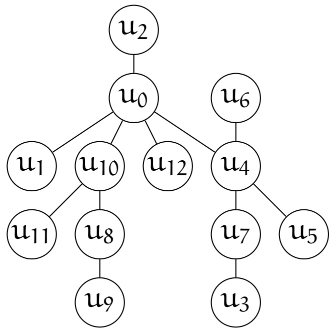
4 The previous cases

The computer experiment for $n = 13$ and $C = C(n)$ returns that the set $\mathcal{S}^*_{15}$ is empty, which means that $S(13) = F(13) = C(13) + 1 = 34$ [5]. In that experiment the set $\mathcal{S}^*_{16}$ contains only one poset P_{16}, whose Hasse diagram is shown in Figure 1. The poset P_{16} can be obtained from a poset contained in the file $\mathcal{S}_{15}$ in two ways:

- we compare elements u_0 and u_{10} in the poset $P'_{15} \in \mathcal{S}_{15}$ shown in Figure 2(a); if $u_0 > u_{10}$ we obtain the poset P_{16}; if $u_0 < u_{10}$ we obtain the poset Q'_{16} shown in Figure 3(a);
- we compare elements u_0 and u_6 in the poset $P''_{15} \in \mathcal{S}_{15}$ shown in Figure 2(b); if $u_0 < u_6$ we obtain the poset P_{16}; if $u_0 > u_6$ we obtain the poset Q''_{16} shown in Figure 3(b).



(a) $e(P'_{15}) = 222750$



(b) $e(P''_{15}) = 238140$

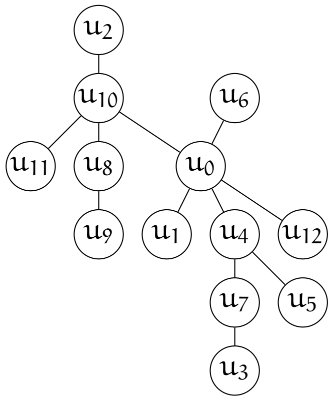
Figure 2: The posets P'_{15} and P''_{15}

Neither the poset P'_{15} nor the poset P''_{15} can be stored in the file $\mathcal{S}^*_{15}$, because neither the poset Q'_{16} nor the poset Q''_{16} can be sorted in the remaining $C-16 = 17$ comparisons. It is quite surprising that the posets Q'_{16}, Q''_{16} cannot be sorted. The poset Q'_{16} has less linear extensions than the poset P_{16}, which intuitively should make it easier to sort. Indeed, the poset Q''_{16} has more linear extensions than the poset P_{16}, which intuitively makes it harder to sort. On the other hand, there are known the two largest elements of the poset Q''_{16}, which intuitively makes it easier to sort. The poset P_{16} is sortable in 17 comparisons because of its symmetry.

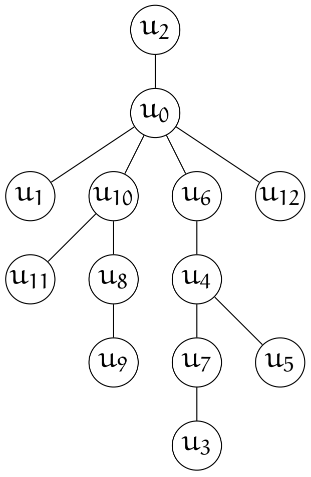
Similar results were received in the computer experiments for $n = 14, 15$ and $C = C(n)$, i.e., $S(14) = F(14) = C(14) + 1 = 38$ and $S(15) = F(15) = C(15) + 1 = 42$. In both cases the file $\mathcal{S}^*_{16}$ contains only one poset, namely the poset P_{16} extended by one isolated element u_{13} (for $n = 14$) or two isolated elements u_{13}, u_{14} (for $n = 15$), respectively. In both cases the file $\mathcal{S}^*_{15}$ is empty and the reason is the same. The posets P'_{15}, Q''_{15}, Q'_{16}, Q''_{16} extended by u_{13} or u_{13}, u_{14} are observed, respectively, and neither Q'_{16} nor Q''_{16} is sortable in the remaining $C - 16$ comparisons. Note that for $n = 14$ we have $C - 16 = C(n) - 16 = 21$ and for $n = 15$ we have $C - 16 = C(n) - 16 = 25$.

5 The case of 16 elements

In this section we describe an attempt to find for $n = 16$ a sorting algorithm better than the FJA or to exclude the existence of such algorithm. Before starting a long time computation it was checked if the scenario from



(a) $e(Q'_{16}) = 109350$



(b) $e(Q''_{16}) = 124740$

Figure 3: The posets Q'_{16} and Q''_{16}

the previous section repeats for $n = 16$. The posets Q'_{16}, Q''_{16} were extended by three isolated elements u_{13}, u_{14}, u_{15}. As previously, the experiment returned that neither the poset Q'_{16} nor the poset Q''_{16} can be sorted in the remaining $C - 16 = C(n) - 16 = 29$ comparisons. Of course, this result does not exclude the existence of the desired algorithm.

To find the exact value of $S(16)$ the algorithm from Section 3 with improvement from Section 7 is applied. Because the search space is very reach, the problem is divided into smaller subproblems. Let $T(k)$ be the number of elements which were compared (touched) by a sorting algorithm in the first k comparisons. Observe that $T(k_1) \leq T(k_2)$ for $k_1 < k_2$. A sorting algorithm for 16 elements, using at most $C(16) = F(16) - 1 = 45$ comparisons, is examined for possible values of $T(k)$.

The first experiment returned that if $S(16) = 45$ then it holds $T(15) < 16$. Note that for the FJA we have $T(k) = 16$ for $k \geq 8$. Hence a hypothetical algorithm, using for 16 elements pessimistically less comparisons than the FJA, must be complete different from the FJA. It must differ from the FJA already before the 9th comparison. This is quite surprising, when we look at regular structure of the first 15 comparisons in the FJA. The next experiment showed that if $S(16) = 45$ then $T(15) > 11$, which is already not surprising.

n	Pentium II 233 MHz 2002	Pentium III 650 MHz 2003	Opteron 246 2 GHz 2004	Core 2 Duo 2.13 GHz 2007
13	10 hr. 30 min.	41 min.	10 min. 44 sec.	46 sec.
14		391 hr. 37 min.	44 hr. 10 min.	4 hr. 31 min.
15			17554 hr.	

Table 1: Computation times

6 Computation complexity

Computation complexity of the method groves exponentially. The case $S(13)$ needed in year 2002 [5] more than 10 hours of CPU time. The value of $S(14)$ was computed one year later (published in 2004 [5]) and took about 392 hours on faster computer and using improved algorithm, which could solve $S(13)$ in about 40 minutes. Further progress in hardware allowed to compute the value of $S(15)$ in year 2004 (published only in 2007 [7]) using about 17500 hours of CPU time. Each next case required significant improvements in the algorithm or hardware. The progress is presented in Table 1. One can argue that the comparison is not fair, because the machines used in the experiments are different. The purpose of this table is to show an overall improvement in software and hardware, and to give a filling, how difficult the case of 16 elements could be. The about 10 times improvement observed between the second last and the last column is due mainly to the algorithm described in the next section. Note that for Core 2 Due processor both cores were used in parallel. A few years of CPU time was used up to now to search for an algorithm achieving the ITLB for $n = 16$. The computation that $T(15) < 16$ and $T(15) > 11$ took about 20000 and 7000 hours, respectively. Computation for the next case $T(15) = 12$ is currently in progress. It used up to now more than 25000 hours.

7 Counting linear extensions

The most time consuming part of the algorithm presented in Section 3 is counting linear extensions of a given poset. In this section we describe the algorithm for counting linear extensions which is inspired by [1] and which substantially improves computations. For a given poset $P = (U, \prec)$ the algorithm computes $e(P)$ and the table $t[j, k] = e(P + u_j u_k)$ for $j \neq k$.

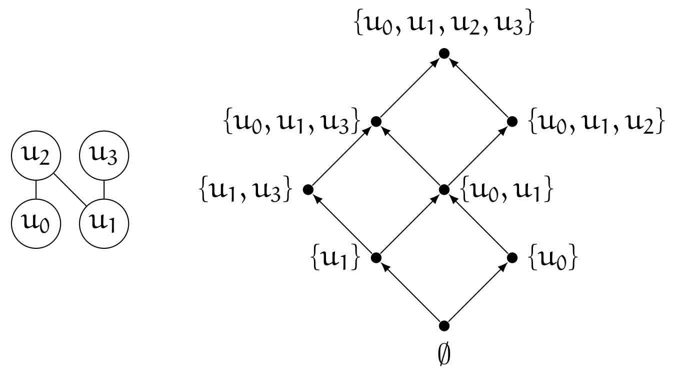


Figure 4: A poset and the graph of its downsets

Let $P = (U, \prec)$ be a poset. A subset $D \subseteq U$ is called a down set of the poset P if for each $x \in D$ all elements $y \in U$ preceding x (i.e., $y \prec x$) also belong to D. We consider a directed acyclic graph $\mathcal{G}$ whose nodes are all downsets of P. For two nodes D_1 and D_2 there is an edge (D_1, D_2) if there exists $x \in U \setminus D_1$ such that $D_2 = D_1 \cup \{x\}$. An example of a poset and its graph of downsets is shown in Figure 4, where $U = \{u_0, u_1, u_2, u_3\}$.

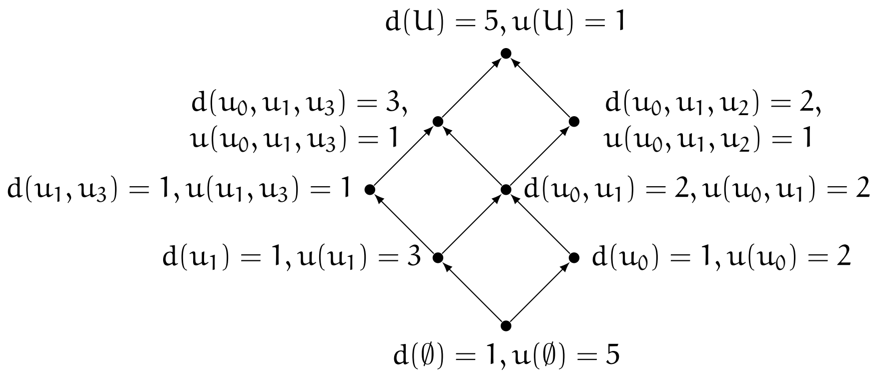
Let $d(D)$ denote the number of linear extensions of the poset $(D, \prec)$ which is the poset P reduced to the down set D. Let $u(D)$ denote the number of linear extensions of the poset $(U \setminus D, \prec)$ which is the poset P reduced to the complementary set of the down set D. We have [1]

$$d(D) = \sum_{(X,D)} d(X), \tag{1}$$

where the sum is taken over all edges (X, D) in the graph $\mathcal{G}$ incoming to the node D. We assume $d(\emptyset) = 1$. Observe that $d(U) = e(P)$. All values of $d(D)$ are computed using the DFS in the graph $\mathcal{G}$, starting at the node U and going down, i.e., in the opposite direction to the edges. Similarly, it holds [1]

$$u(D) = \sum_{(D,X)} u(X), \tag{2}$$

where the sum is taken over all edges (D, X) in the graph $\mathcal{G}$ outgoing from the node D. We assume $u(U) = 1$. Observe that $u(\emptyset) = e(P)$. All values of $u(D)$ are computed using the second DFS in the graph $\mathcal{G}$, starting at the node $\emptyset$



$d(U) = 5, u(U) = 1$
$d(u_0, u_1, u_3) = 3,$ $u(u_0, u_1, u_3) = 1$
$d(u_0, u_1, u_2) = 2,$ $u(u_0, u_1, u_2) = 1$
$d(u_1, u_3) = 1, u(u_1, u_3) = 1$
$d(u_0, u_1) = 2, u(u_0, u_1) = 2$
$d(u_1) = 1, u(u_1) = 3$
$d(u_0) = 1, u(u_0) = 2$
$d(\emptyset) = 1, u(\emptyset) = 5$

Figure 5: The numbers of linear extensions of the downsets and they complementary sets

		k			
		0	1	2	3
j	0	–	2	5	4
	1	3	–	5	5
	2	0	0	–	2
	3	1	0	3	–

Table 2: The values of $t[j, k]$

and going up. Values of $d(D)$ and $u(D)$ for the graph in Figure 4 are shown in Figure 5. The curly braces are omitted for clarity, e.g., instead of $d(\{u_0\})$ we write $d(u_0)$. The table t can be computed from the equation

$$t[j, k] = \sum_{(V,W)} d(V)u(W), \tag{3}$$

where the sum is taken over all edges (V, W) in the graph $\mathcal{G}$ such that $W = V \cup \{u_j\}$ and $u_k \in U \setminus W$. For a proof see [1]. This computation is done altogether with the second DFS. For the graph in Figure 4 the values $t[j, k]$ are included in Table 2.

For a given poset on an n-element set its graph of downsets can have up to 2^n nodes. We implemented the graph as a table of the size 2^n. The table is indexed by downsets. The index is the characteristic function of the set D, i.e., the index is the n-bit number, where bit j is set iff $u_j \in D$. Graph $\mathcal{G}$ is not constructed explicitly. When we proceed a node D all incoming and outgoing edges are easily computable from a poset representation. We hold at position D in the table only two numbers $d(D)$, $u(D)$ and visited time stamp $v(D)$ needed to implement the DFS. We initialize the table only once at the beginning of the program by setting all $v(D) = 0$. We also hold the global visited time stamp v_t initialized to 0. Starting a new DFS we increment the time stamp v_t. If we proceed a node D and $v_t > v(D)$ then it means that the node D was not yet visited in the current DFS run. If $v_t = v(D)$ then the node was already visited. We do not need to reinitialize the table before the next DFS. This is very important and decreases running time. The algorithm is very efficient for small n, because with a high probability the whole graph resides in a processor cache memory.

References

[1] K. De Loof, H. De Meyer, B. De Baets, Exploiting the lattice of ideals representation of a poset, *Fund. Inform.* **71** (2006) 309–321. ⇒221, 222, 223

[2] L. Ford, S. Johnson, A tournament problem, *Amer. Math. Monthly* **66** (1959) 387–389. ⇒215

[3] T. Kasai, S. Sawato, S. Iwata, Thirty four comparisons are required to sort 13 items, in: *Logic, Language, and Computation: Festschrift in Honor of Satoru Takasu* (eds. N. D. Jones, M. Hagiya, M. Sato), *Lecture Notes in Comput. Sci.* **792** (1994) 260–269. ⇒215

[4] D. E. Knuth, *The Art of Computer Programming. Vol. 3. Sorting and Searching* (second edition), Addison–Wesley, Reading, MA, 1998. First edition: Addison–Wesley, Reading, MA, 1973. ⇒215, 216

[5] M. Peczarski, Sorting 13 elements requires 34 comparisons, in: *Proceedings of the 10th Annual European Symposium on Algorithms* (eds. R. Möhring, R. Raman), *Lecture Notes in Comput. Sci.* **2461** (2002) 785–794. ⇒216, 217, 218, 221

[6] M. Peczarski, New results in minimum-comparison sorting, *Algorithmica* **40** (2004) 133–145. ⇒216, 217, 218

[7] M. Peczarski, The Ford–Johnson algorithm still unbeaten for less than 47 elements, *Inform. Process. Lett.* **101** (2007) 126–128. ⇒216, 218, 221

[8] H. Steinhaus, *Mathematical Snapshots*, Dover Publications, Mineola, NY, 1981. First edition: Oxford University Press, New York, NY, 1950. ⇒215

[9] M. Wells, Applications of a language for computing in combinatorics, in: *Proceedings of the 1965 IFIP Congress, Information Processing* **65**, North-Holland, Amsterdam, 1966, 497–498. ⇒215, 217

[10] M. Wells, *Elements of Combinatorial Computing*, Pergamon Press, Oxford, 1971. ⇒215, 217

Received: November 22, 2012 • Revised: December 9, 2012

Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 225–236

Scattered subwords and composition of natural numbers

Zoltán KÁSA
Sapientia Hungarian Univerity of Transylvania
Department of Mathematics and Informatics, Târgu Mureş
email: kasa@ms.sapientia.ro

Zoltán KÁTAI
Sapientia Hungarian Univerity of Transylvania
Department of Mathematics and Informatics, Târgu Mureş
email: katai_zoltan@ms.sapientia.ro

Abstract. Special scattered subwords in which the length of the gaps are bounded by two natural numbers are considered. For rainbow words the number of such scattered subwords is equal to the number of special restricted compositions of natural numbers in which the components are natural numbers from a given interval. Linear algorithms to compute such numbers are given. We also introduce the concepts of generalized scattered subword (duplex-subword) and generalized composition.

1 Introduction

We define a special scattered subword [5] as a generalization of the d-subword [2] and supper-d-subword [4].

Definition 1 *Let n, $d_1 \le d_2$ and s be positive natural numbers, and let $u = x_1x_2\dots x_n \in \Sigma^n$ be a word over an alphabet Σ. A word $v = x_{i_1}x_{i_2}\dots x_{i_s}$, where*

$i_1 \ge 1$,
$d_1 \le i_{j+1} - i_j \le d_2$, *for* $j = 1, 2, \dots, s-1$,
$i_s \le n$,

*is a (d_1, d_2)-**subword** of length s of u.*

Computing Classification System 1998: G2.1, F2.2, F2.1
Mathematics Subject Classification 2010: 68R15, 05A17, 05A05
Key words and phrases: scattered subwords, composition of integers, complexity of words

For example, in the word *aabcade* the subwords abd, ace, ad are $(2,4)$-subwords.

Definition 2 *The number of different (d_1, d_2)-subwords of a word w is the (d_1, d_2)-**complexity** of w.*

The $(1,d)$-complexity was studied in [2] and [3], the (d,n)-complexity in [4], while the (d_1, d_2)-complexity in [5].

2 Computing the (d_1, d_2)-complexity by digraphs

The graph method was defined for the general case of scattered subwords in [5], and for this particular case can be used as follows to compute the (d_1, d_2)-complexity of a rainbow word.

Let $G = (V, E)$ be a digraph attached to the rainbow word $a_1a_2\ldots a_n$ and positive integers $d_1 \le d_2$, where

$V = \{a_1, a_2, \ldots, a_n\}$,

$E = \{(a_i, a_j) \mid d_1 \le j - i \le d_2,\ i = 1, 2, \ldots, n, j = 1, 2, \ldots, n\}$.

The adjacency matrix $A = \big(a_{ij}\big)_{\substack{i=\overline{1,n}\\ j=\overline{1,n}}}$ of the digraph is defined by:

$$a_{ij} = \begin{cases} 1, & \text{if } d_1 \le j - i \le d_2, \\ 0, & \text{otherwise,} \end{cases} \qquad i = 1, 2, \ldots, n, j = 1, 2, \ldots, n.$$

To compute the (d_1, d_2)-complexity we use a Floyd-Warshall-type algorithm [5]:

```
FW($A, n$)

1  $W \leftarrow A$
2  for $k \leftarrow 1$ to $n$
3      do for $i \leftarrow 1$ to $n$
4             do for $j \leftarrow 1$ to $n$
5                    do $w_{ij} \leftarrow w_{ij} + w_{ik}w_{kj}$
6  return $W$
```

If $R = I + W$, where I is the unity matrix, then the (d_1, d_2)-complexity is:

$$K(n; d_1, d_2) = \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij}.$$

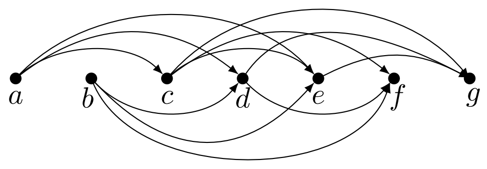


Figure 1: Graph for $n = 7, d_1 = 2, d_2 = 4$.

Example 3 *For digraph in Figure 1 we have the following adjacency matrix:*

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The above algoithm give us the matrix W, and by completing with 1's on the first diagonal we obtain the corresponding matrix R:

$$W = \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 4 \\ 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad R = \begin{pmatrix} 1 & 0 & 1 & 1 & 2 & 2 & 4 \\ 0 & 1 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

The complexity $K(7; 2, 4) = 30$ (the sum of all entries of the matrix R). The corresponding (2,4)-subwords are:

```
{a} {ac} {ad} {ace,ae} {adf,acf} {aeg,aceg,adg,acg}
{b} {bd} {be} {bdf,bf} {beg,bdg}
{c} {ce} {cf} {ceg,cg}
{d} {df} {dg}
{e} {eg}
{f}
{g}.
```

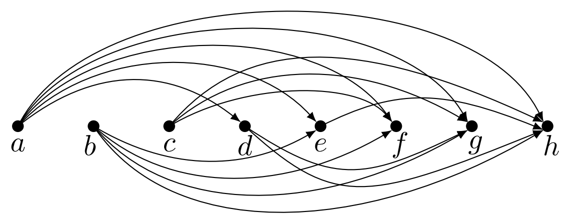


Figure 2: Graph for $n = 8, d_1 = 3, d_2 = 7$.

The Floyd–Warshall-type algorithm combined with the Latin square method can be used to obtain all nontrivial (with length at least 2) (d_1, d_2)-subwords of a given rainbow word $a_1a_2 \dots a_n$ of length n [5]. Let us consider a matrix $\mathcal{A}$ with entries A_{ij} which are sets of words. Initially this entries are defined as:

$$A_{ij} = \begin{cases} \{a_ia_j\}, & \text{if } d_1 \le j - i \le d_2, \\ \emptyset, & \text{otherwise,} \end{cases} \qquad \text{for } i = 1, 2, \dots, n,\ j = 1, 2, \dots, n.$$

If $\mathcal{A}$ and $\mathcal{B}$ are sets of words, $\mathcal{AB}$ is the set of concatenation of each word from $\mathcal{A}$ with each word from $\mathcal{B}$:

$$\mathcal{AB} = \big\{ab \,\big|\, a \in \mathcal{A}, b \in \mathcal{B}\big\}.$$

If $s = s_1s_2 \dots s_p$ is a word, let us denote by $'s$ the word obtained from s by erasing the first character: $'s = s_2s_3 \dots s_p$. Let us denote by $'A_{ij}$ the set A_{ij} in which we erase from each element the first character. In this case $'\mathcal{A}$ is a matrix with elements $'A_{ij}$.

Starting with the matrix $\mathcal{A}$ defined as before, the algorithm to obtain all nontrivial (d_1, d_2)-subwords is the following [5]:

```
FW-Latin($\mathcal{A}, n$)

1  $\mathcal{W} \leftarrow \mathcal{A}$
2  for $k \leftarrow 1$ to $n$
3      do for $i \leftarrow 1$ to $n$
4             do for $j \leftarrow 1$ to $n$
5                    do if $W_{ik} \ne \emptyset$ and $W_{kj} \ne \emptyset$
6                          then $W_{ij} \leftarrow W_{ij} \cup W_{ik}\,'W_{kj}$
7  return $\mathcal{W}$
```

The set of (d_1, d_2)-subwords is $\displaystyle\bigcup_{i,j\in\{1,2,\ldots,n\}} W_{ij}$.

Example 4 *For the digraph in Figure 2, when $n = 8$, $d_1 = 3, d_2 = 7$, the initial matrix A is:*

$$\begin{pmatrix}
\emptyset & \emptyset & \emptyset & \{ad\} & \{ae\} & \{af\} & \{ag\} & \{ah\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \{be\} & \{bf\} & \{bg\} & \{bh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{cf\} & \{cg\} & \{ch\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{dg\} & \{dh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}.$$

The result of the algorithm is:

$$\begin{pmatrix}
\emptyset & \emptyset & \emptyset & \{ad\} & \{ae\} & \{af\} & \{ag, adg\} & \{ah, adh, aeh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \{be\} & \{bf\} & \{bg\} & \{bh, beh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{cf\} & \{cg\} & \{ch\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{dg\} & \{dh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}.$$

3 Linear algorithm for the (d_1, d_2)-complexity

In this section linear algorithms to obtain (d_1, d_2)-complexity and (d_1, d_2)-subwords of rainbow words are given.

The following dynamic programming algorithm Complexity computes the (d_1, d_2)-complexity of rainbow word $a_1a_2\ldots a_n$. The element x_i $(i = 1, 2, \ldots, n)$ of the array X stores the number of (d_1, d_2)-subwords ending in letter a_i. Letter a_i can be added to all (d_1, d_2)-subwords ending in letters from interval $a_i - d_1 \ldots a_i - d_2$ (assuming that this interval exists). Consequently, value x_i can be computed as the sum of values $x_i - d_1, \ldots, x_i - d_2$. (lines 4, 5, 6) Since letter a_i itself is considered as a (d_1, d_2)-subword, element x_i is initialized with 1 (line 3). The (d_1, d_2)-complexity of a rainbow word of length n is the sum of values x_i $(i = 1, 2, \ldots, n)$ (lines 1 and 7).

Complexity(n, d_1, d_2)

```
$k \leftarrow 0$
for $i \leftarrow 1$ to $n$
    do $x_i \leftarrow 1$
       for $j \leftarrow i - d_2$ to $i - d_1$
           do if $j > 0$
                 then $x_i \leftarrow x_i + x_j$
       $k \leftarrow k + x_i$
return $X, k$
```

Time complexity of this algorithm is $\Theta(n)$, because the inner loop is executed $n(d_2 - d_1 + 1)$ times.

The following algorithm generates the (d_1, d_2)-subwords of rainbow word $a_1a_2 \ldots a_n$. Bidimensional array (of strings) Y stores the generated (d_1, d_2)-subwords. Array y_i has x_i element and stores the (d_1, d_2)-subwords ending in letter a_i. Operation $s \circ l$ (line 8) means that letter l is added to the end of string s.

Subwords(n, d_1, d_2, A, X)

```
1 for $i \leftarrow 1$ to $n$
2     do $y_{i1} \leftarrow a_i$
3        $p \leftarrow 1$
4        for $j \leftarrow i - d_2$ to $i - d_1$
5            do if $j > 0$
6                  then for $k \leftarrow 1$ to $x_j$
7                           do $p \leftarrow p + 1$
8                              $y_{ip} \leftarrow y_{jk} \circ a_i$
9 return $Y$
```

The inner loop is executed $n(d_2 - d_1 + 1)\max\limits_i x_i$ times, so this is a pseudo-linear algorithm.

4 Generalized scattered subwords

Definition 1 can be generalized for rainbow words as we choose letters not only going ahead in the word, but back too at every step.

Definition 5 *Let n, $d_1 \leq d_2$ and s be positive natural numbers, and let*

$u = x_1x_2 \dots x_n \in \Sigma^n$ be a rainbow word over an alphabet Σ. A rainbow word $v = x_{i_1}x_{i_2} \dots x_{i_s}$, where

$i_1 \geq 1$,
$d_1 \leq |i_{j+1} - i_j| \leq d_2$, *for* $j = 1, 2, \dots, s-1$,
$i_s \leq n$,

is a ***duplex*** *(d_1, d_2)-**subword** of length s of u.*

It is worth to note that the definition is given only for rainbow words, and the duplex subwords are rainbow words too.

For example $acfbe$ and $beadfc$ both are duplex (2,4)-subwords of the word $abcdef$.

Definition 6 *The number of all duplex (d_1, d_2)-subwords of a word is the* ***duplex*** *(d_1, d_2)-**complexity** of that rainbow word.*

We denote the duplex (d_1, d_2)-complexity of a rainbow word of length n by $D(n; d_1, d_2)$, and let x_i, y_i and z_i be the numbers of subwords starting, ending and starting or ending in letter a_i, respectively. Notice that $0 \leq d_2 - d_1 \leq n-1$. For the minimum cases, $d_2 - d_1 = 0$ ($d_1 = d_2 = d$, $d = 1, \dots, n-1$) we have the following recursive formulas (where the sequences X and Y are identical):

$$\begin{aligned} &x_i = 1, && \text{if } 0 < i \leq d \\ &x_i = x_{i-d} + 1, && \text{if } d < i \leq n \\ &y_i = 1, && \text{if } 0 < i \leq d \\ &y_i = y_{i-d} + 1, && \text{if } d < i \leq n \\ &v_i = x_i + y_i - 1. && \end{aligned}$$

The duplex complexity $D(n; d, d)$ is

$$D(n; d, d) = \sum_{i=1}^{n} v_i.$$

By a simple computation we can obtain the generating functions $F_d(z) = \sum_{n\geq 1} x_n z^n$, $G_d(z) = \sum_{n\geq 1} v_n z^n$, $H_d(z) = \sum_{n\geq 1} D(n; d, d) z^n$:

$$F_d(z) = \frac{z}{(1-z)(1-z^d)}, \quad G_d(z) = \frac{z(1+z^d)}{(1-z)(1-z^d)},$$

$$H_d(z) = \frac{1}{1-z} G_d(z) = \frac{z(1+z^d)}{(1-z)^2(1-z^d)}.$$

Based on Proposition 4 in [5] it is easy to prove the following

Proposition 7 *For integers $n, d \geq 1$, where $n = hd + m$, $0 \leq m < d$,*

$$D(n; d, d) = hn + (h+1)m.$$

The graph method for the case of duplex (d_1, d_2)-subwords can be defined as follows. Let $G = (V, E)$ be a graph (with V the set of vertices, E the set of edges) attached to the rainbow word $a_1 a_2 \ldots a_n$ and integers d_1, d_2, where
$V = \{a_1, a_2, \ldots, a_n\}$,
$E = \{\{a_i, a_j\} \mid d_1 \leq |j-i| \leq d_2,\ i = 1, 2, \ldots, n, j = 1, 2, \ldots, n\}$.
Using the attached graph, we can prove the following

Proposition 8

$$D(n; 1, n-1) = n! \sum_{k=0}^{n-1} \frac{1}{k!}.$$

Proof. In this case the attached graph is a complete graph on n vertices, and $D(n; 1, n-1) - n$ is equal to the number of all paths in this graph. In [9] the sequence A007526 is defined by the formula $a_n = n\big(a_{n-1} + 1\big)$, and "for $n \geq 1$, $a(n)$ is the number of non-empty sequences with n or fewer terms, each a distinct element of $\{1, 2, \ldots, n\}$". So, $a(n)$ is the sum of the number of all paths and the number of all vertices. For a_n in [9] the following formula is given too: $a_n = n! \sum_{k=0}^{n-1} \frac{1}{k!}$. From this:

$$D(n; 1, n-1) = n! \sum_{k=0}^{n-1} \frac{1}{k!}.$$ □

The duplex (d_1, d_2)-complexity of a rainbow word of length n is equal to the number of paths in the attached graph ($NumberOfPaths$). The following modified DFS algorithm (based on [8]) generates all paths in the attached graph G and prints the corresponding duplex subwords. Global arrays DEGREE, PREVIOUS and COLOR have elements indexed from 1 to n. Element $degree_i$ stores the degree of vertex i (corresponding to letter i). Arrays PREVIOUS and COLOR are initiated with zero. Element $previous_i$ stores the previous vertex of vertex i on the current path. We use array COLOR to avoid cycles. Bi-dimensional array NEIGHBOUR stores the neighbor-lists of the vertices of graph G. Element $neighbor_{ik}$ stores the k-th neigbour of vertex i. Procedure DFS(i) prints all distinct paths starting with root-vertex a_i (line

		n								
d_1	d_2	2	3	4	5	6	7	8	9	10
1	1	*4*	*9*	*16*	*25*	*36*	*49*	*64*	*81*	*100*
1	2		**15**	42	101	224	469	944	1849	3552
1	3			**64**	227	716	2111	6058	16971	46546
1	4				**325**	1434	5707	21244	76487	273580
1	5					**1956**	10437	50624	229541	1000106
1	6						**13699**	86114	495161	2662784
1	7							**109600**	794607	5299996
1	8								**986409**	8110482
1	9									**9864100**
2	2		*5*	*8*	*13*	*18*	*25*	*32*	*41*	*50*
2	3			16	45	106	225	474	983	2000
2	4				69	264	853	2432	6683	18560
2	5					378	1855	7708	28209	97200
2	6						2497	14832	75865	343674
2	7							19184	133497	812746
2	8								167513	1334960
2	9									1635970
3	3			*6*	*9*	*12*	*17*	*22*	*27*	*34*
3	4				17	36	91	194	389	756
3	5					62	243	912	2783	7390
3	6						345	1914	9405	37448
3	7							2524	17295	103560
3	8								21901	174694
3	9									214930
4	4				*7*	*10*	*13*	*16*	*21*	*26*
4	5					18	37	64	153	306
4	6						63	186	699	2580
4	7							290	1559	8832
4	8								2075	15794
4	9									19660
5	5					*8*	*11*	*14*	*17*	*20*
5	6						19	38	65	100
5	7							64	187	482
5	8								291	1204
5	9									1722
6	6						*9*	*12*	*15*	*18*
6	7							20	39	66
6	8								65	188
6	9									292

Table 1: Duplex (d_1, d_2)-complexity for rainbow words of length n

3). Procedure PrintCurentPathTo(i) prints the currently generated path from the current root-vertex to vertex a_i (line 8).

```
DuplexSubwords()

 1  $NumberOfPaths \leftarrow 0$
 2  for $i \leftarrow 1$ to $n$
 3      do DFS($i$)
 4  PRINT($NumberOfPaths$)
```

```
DFS($i$)

 1  $color_i \leftarrow 1$
 2  for $k \leftarrow 1$ to $degree_i$
 3      do $j \leftarrow neighbor_{ik}$
 4          if $color_j = 0$
 5             then $previous_j \leftarrow i$
 6                  DFS($j$)
 7  $NumberOfPaths \leftarrow NumberOfPaths + 1$
 8  PrintCurrentPathTo($i$)
 9  $color_i \leftarrow 0$
10  $previous_i \leftarrow 0$
```

```
PrintCurrentPathTo($i$))

 1  if $previous_i \leftarrow 0$
 2     then PrintCurrentPathTo($previous_i$)
 3  PRINT($a_i$)
```

5 (d_1, d_2)-complexity and (d_1, d_2)-compositions

Compositions [1, 6, 7] are partitions in which the order of the summands (components) does matter. A (d_1, d_2)-***composition*** is a restricted composition in which the components are natural numbers from the interval $[d_1, d_2]$.

For example, for the word *abcdefg* the (2,4)-subwords, which begin in *a* and end in *g* are: *aeg*, *aceg*, *adg*, *acg*, which correspond to the following compositions in which the components are the distances between the letters in the original word:

$$6 = 4 + 2 = 2 + 2 + 2 = 3 + 3 = 2 + 4.$$

In general, if $a_1 a_{i_1} \cdots a_{i_s} a_{n+1}$ is a (d_1, d_2)-subword of the rainbow word $a_1 a_2 \cdots a_{n+1}$, then this subword corresponds to a composition:

$$n = (i_1 - 1) + (i_2 - i_1) + \ldots + (i_s - i_{s-1}) + (n + 1 - i_s).$$

Let us consider the following (2,4)-subwords: $a_1 a_5 a_7$, $a_1 a_3 a_5 a_7$, $a_1 a_4 a_7$, and $a_1 a_3 a_7$ of the word $a_1 a_2 a_3 a_4 a_5 a_6 a_7$. Then the corresponding (2,4)-compositions are

$$6 = 4 + 2 = 2 + 2 + 2 = 3 + 3 = 2 + 4.$$

So, each (d_1, d_2)-subword of a rainbow word of length $n + 1$, which begins by the first, and ends by the last letter of the rainbow word, corresponds to a (d_1, d_2)-composition of n.

By a simple reasoning we can obtain a formula between the (d_1, d_2)-complexity and (d_1, d_2)-compositions. Let us denote the (d_1, d_2)-composition of n by $C(n; d_1, d_2)$.

Proposition 9 *If $n \geq 1$, then*

$$K(n; d_1, d_2) = n + \sum_{i=1}^{n-1} iC(n - i; d_1, d_2)$$

Example 10 *If $n = 7$, $d_1 = 2$ and $d_2 = 4$, then $K(7; 2, 4) = 30$.*
$C(6; 2, 4) = 4$, *because* $6 = 2 + 2 + 2 = 2 + 4 = 3 + 3 = 4 + 2$.
$C(5; 2, 4) = 2$, *because* $5 = 2 + 3 = 3 + 2$.
$C(4; 2, 4) = 2$, *because* $4 = 2 + 2 = 4$.
$C(3; 2, 4) = 1$, *because* $3 = 3$.
$C(2; 2, 4) = 1$, *because* $3 = 2$.
$C(1; 2, 4) = 0$,
and $K(7; 2, 4) = 7 + 1 \cdot 4 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 1 + 5 \cdot 1 + 6 \cdot 0 = 30$.

Similarly, if we extend the compositions to the case when instead of the natural numbers, we consider nonzero integers, a relation with the duplex subwords can be given.

Definition 11 *A **generalized composition** of a natural number is one way of writing this number as an ordered sum of nonzero integers, such that all partial sums*[1] *are positive and different. If the absolute value of the summands are from an interval $[d_1, d_2]$, we call this a **generalized** (d_1, d_2)**-compositions**.*

[1]Partial sums are $\sum_{i=1}^{k} s_i$.

Example 12 *The generalized (2,4)-compositions of 6 are:*

$2+2-3+2+3$	$3+2-4+3+2$
$2+2-3+4-2+3$	$3+2-3+2+2$
$2+2+2$	$3+2-3+4$
$2+3-4+2+3$	$3+3$
$2+3-4+2+2$	$4-3+2+2-3+4$
$2+3-2-2+3+2$	$4-3+2+3$
$2+3-2+3$	$4-3+4-3+4$
$2+4$	$4-3+4-2+3$
$3-2+3-2+4$	$4-2+3-4+2+3$
$3-2+3+2$	$4-2+3-2+3$
$3-2+4-3+2+2$	$4-2+4$
$3-2+4-3+4$	$4+2.$
$3+2-4+3-2+4$	

If $a_1a_{i_1}\cdots a_{i_s}a_{n+1}$ is a duplex (d_1,d_2)-subword of the rainbow word $a_1a_2\cdots a_{n+1}$, then this subword corresponds to a generalized (d_1,d_2)-composition:

$$n=(i_1-1)+(i_2-i_1)+\ldots+(i_s-i_{s-1})+(n+1-i_s).$$

References

[1] S. Heubach, A. Knopfmacher, M. E. Mays, A. Munagi, Inversions in compositions of integers, *Quaest. Math.* **34**, 2 (2011) 187–202. ⇒234

[2] A. Iványi, On the d-complexity of words, *Ann. Univ. Sci. Budapest. Sect. Comput.*, **8** (1987) 69–90. ⇒225, 226

[3] Z. Kása, On the d-complexity of strings, *Pure Math. Appl.*, **9,** 1–2 (1998) 119–128. ⇒226

[4] Z. Kása, Super-d-complexity of finite words, *8th Joint Conf. on Math. and Comput. Sci., Selected Papers*, Komárno, July 14–17, 2010. Novodat, 2011 (eds. H. F. Pop, A. Bege), pp. 257–266. ⇒225, 226

[5] Z. Kása, On scattered subword complexity, *Acta Univ. Sapientiae, Inform.* **3,** 1 (2011) 127–136. ⇒225, 226, 228, 231

[6] C. Kimberling, Enumeration of paths, compositions of integers, and Fibonacci numbers, *Fibonacci Quart.* **39,** 5 (2001) 430–435. ⇒234

[7] C. Kimberling, Path-counting and Fibonacci numbers, *Fibonacci Quart.* **40,** 4 (2002) 328–338. ⇒234

[8] R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, Addison Wesley Professional, 3rd ed., 2001. ⇒232

[9] N. J. A. Sloane, The on-line encyclopedia of integer sequences, `http://www.research.att.com/~njas/sequences/`. ⇒232

Received: March 31, 2012 • Revised: November 30, 2012

Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 237–246

Pareto-optimal Nash equilibrium detection using an evolutionary approach

Noémi GASKÓ
Babes-Bolyai University
email: gaskonomi@cs.ubbcluj.ro

Mihai SUCIU
Babes-Bolyai University
email: mihai.suciu@ubbcluj.ro

Rodica Ioana LUNG
Babes-Bolyai University
email: rodica.lung@econ.ubbcluj.ro

D. DUMITRESCU
Babes-Bolyai University
email: ddumitr@cs.ubbcluj.ro

Abstract. Pareto-optimal Nash equilibrium is a refinement of the Nash equilibrium. An evolutionary method is described in order to detect this equilibrium. Generative relations induces a non-domination concept which is essentially in the detection method. Numerical experiments with games having multiple Nash equilibria are presented. The evolutionary method detects correctly the Pareto-optimal Nash equilibria.

1 Introduction

Equilibrium detection in non-cooperative games is an essential task. Decision making processes can be analysed and predicted using equilibrium detection. The most known equilibrium concept is the Nash equilibrium [9]. Unfortunately this equilibrium has some limitations: if a game has multiple Nash equilibria, a selection problem can appear. Solution to this problem are the refinements of Nash equilibria such as: Aumann equilibrium [1], coalition proof Nash equilibrium [2], modified strong Nash equilibrium [6],[10] (detection of this equilibrium is described in [5]), etc.

Computing Classification System 1998: I.2.8
Mathematics Subject Classification 2010: 91A10
Key words and phrases: game theory, Pareto-optimal Nash equilibrium, evolutionary algorithm

The Pareto-optimal Nash equilibrium is one of the most important refinements of the Nash equilibrium that selects the NE that is Pareto non-dominated with respect to the other NE's of the game. The evolutionary method, presented in this paper is the first one, for the best of our knowledge.

In the next part of the paper some basic notions from game theory and a computationally method are described in order to detect the Pareto-optimal Nash equilibrium. The advantage of the proposed method is that the equilibrium can be obtained in one step, and there is no need for two different selection steps.

A finite strategic game is a system $G = ((N, S_i, u_i), i = 1, \dots, n)$, where:

- N represents a set of players, and n is the number of players;
- for each player $i \in N$, S_i is the set of available actions,

  $$S = S_1 \times S_2 \times \cdots \times S_n$$

  is the set of all possible situations of the game and $s \in S$ is a strategy (or strategy profile) of the game;
- for each player $i \in N$, $u_i : S \to R$ represents the payoff function (utility) of the player i.

One of the most important solving concept in non-cooperative game theory is the Nash equilibrium [9]. The Nash equilibrium (NE) of the game G means that no player can increase his/her payoff by unilateral deviation.

Let us denote by (s_i, s^*_{-i}) the strategy profile obtained from s^* by replacing the strategy of player i with s_i :

$$(s_i, s^*_{-i}) = (s^*_1, \dots, s_i, \dots, s^*_n).$$

Formally we have the next definition:

Definition 1 *A strategy profile $s^* \in S$ is a Nash equilibrium if the inequality*

$$u_i(s_i, s^*_{-i}) \leq u_i(s^*),$$

holds $\forall i = 1, \dots, n, \forall s_i \in S_i, s_i \neq s^*_i$.

2 Pareto-optimal Nash equilibrium

To describe the Pareto-optimal Nash equilibrium first we introduce the notion of the Pareto optimality:

Definition 2 *A strategy profile $s^* \in S$ is Pareto efficient when it does not exist a strategy $s \in S$, such that*

$$u_i(s) \geq u_i(s^*),\ i \in N,$$

with at least one strict inequality.

Pareto-optimal Nash equilibrium [8] is a refinement of the Nash equilibrium. The Pareto-optimal Nash equilibrium is a Nash equilibrium for which there is no other state in which every player is better off.

Formally:

Definition 3 *Let $s^* \in S$ be a Nash equilibrium. s^* is a Pareto-optimal Nash equilibrium, there exists no $s \in S$ such that:*

$$u_i(s) \geq u_i(s^*), \forall i \in N.$$

Let us denote the Pareto-optimal Nash equilibrium by PNE.

Remark 4 *The Pareto-optimal Nash equilibrium is a subset of the Nash equilibrium:*

$$PNE \subseteq NE.$$

3 Generative relation for Pareto-optimal Nash equilibrium

Generative relations are used for equilibrium detection by inducing a domination concept. Two strategy profiles can be dominated, non-dominated, or indifferent with respect to a generative relation. An evolutionary algorithm with a generative relation will approximate a certain equilibrium type.

First generative relation has been introduced in [7] for detecting the Nash equilibrium.

To obtain the Pareto-optimal Nash equilibrium a new generative relation is introduced next.

Consider two strategy profiles s^* and s from S. Denote by $pn(s^*, s)$ the number of strategies, for which some players can benefit deviating.

We may express $pn(s^*, s)$ as:

$$\begin{aligned} pn(s^*, s) &= card\{i \in N,\ u_i(s) > u_i(s^*), s \neq s^*\} \\ &\quad + card\{i \in N, u_i(s_i, s^*_{-i}) > u_i(s^*)), s^*_i \neq s_i\}, \end{aligned}$$

where $card\{R\}$ denotes the cardinality of the set R.

Definition 5 *Let* $s^*, s \in S$. *We say the strategy* s^* *is better than the strategy* s *with respect to Pareto-optimal Nash equilibrium, and we write* $s^* \prec_{PN} s$, *if the inequality*

$$pn(s^*, s) < pn(s, s^*)$$

holds.

Definition 6 *The strategy profile* $s^* \in S$ *is a Pareto-optimal Nash non-dominated strategy, if there is no strategy* $s \in S$, $s \neq s^*$ *such that* s *dominates* s^* *with respect to* $\prec_{PN}$ *i.e.*

$$s \prec_{PN} s^*.$$

Denote by PNS the set of all non-dominated strategies with respect to the relation $\prec_{PN}$.

We may consider relation $\prec_{PN}$ as a candidate for generative relation of the Pareto-optimal Nash equilibrium. This means the set of the non-dominated strategies with respect to the relation $\prec_{PN}$ equals to the set of Pareto-optimal Nash equilibria.

It can be considered that the set of all Pareto-optimal Nash equilibrium strategies as representing the Pareto-optimal Nash equilibrium (PNE) of the game.

4 Evolutionary equilibria detection

Differential Evolution [11] is an evolutionary algorithm designed for continuous function optimization. It is a simple but very efficient algorithm, these two advantages have made DE one of the most popular optimization technique for real value single objective optimization. It has also been extended to multi-objective optimization. For the trial vector generation we use the strategy *rand/1/bin* proposed in [11]. This version of DE has only four parameters: n – population size, stopping criterion – number of generations, $F \in [0, 1]$ – mutation factor, and $Cr \in [0, 1]$ – crossover rate. High values for F assure the exploration of the search space while high Cr values assure the space exploitation. The DE procedure is given by Algorithm 1.

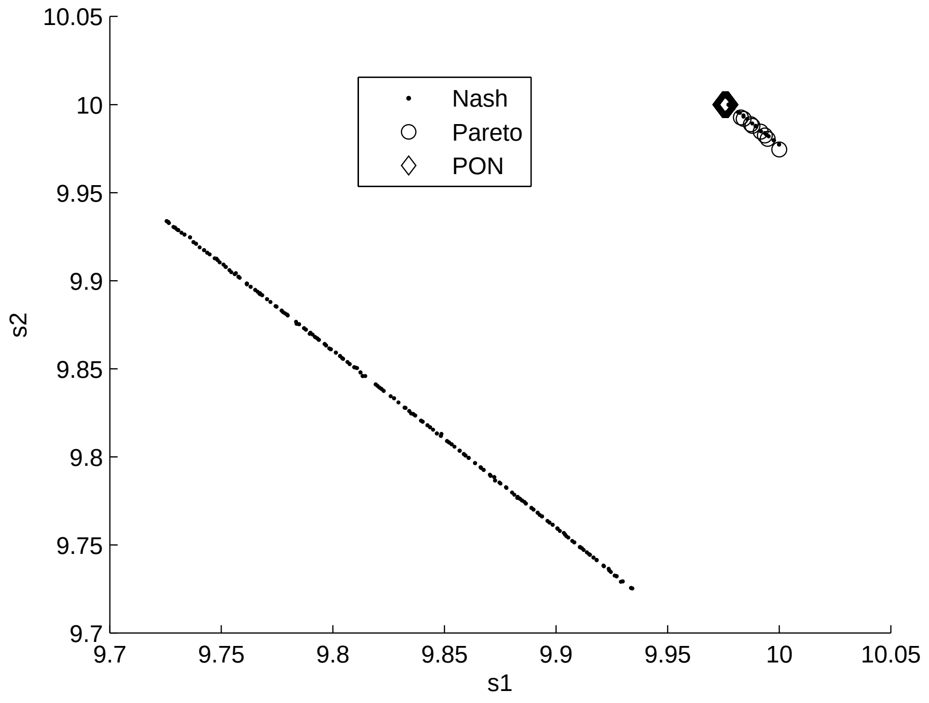


Figure 1: Detected Pareto front, Nash equilibrium, and Pareto-optimal Nash equilibrium strategies for Game 1

Algorithm 1: DE/rand/1/bin

```
1: Evaluate fitness
2: for $i = 0 \to max - iterations$ do
3:     Create difference offspring by mutation and recombination
4:     Evaluate fitness
5:     if the offspring is better than the parent then
6:         Replace the parent by offspring in the next generation
7:     end if
8: end for
```

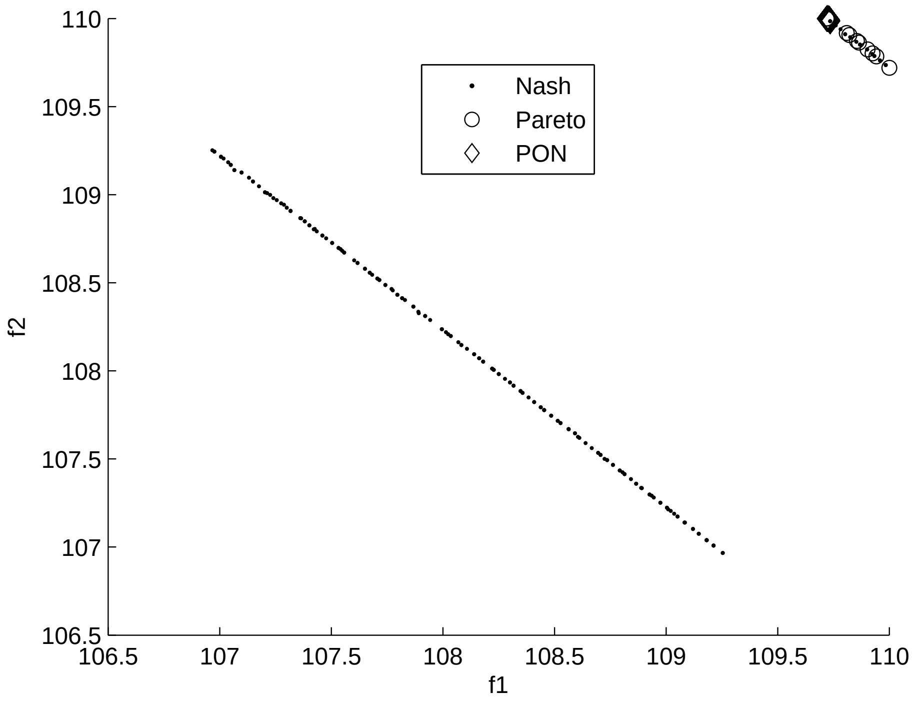


Figure 2: Detected Pareto front, Nash equilibrium, and Pareto-optimal Nash equilibrium payoff for Game 1

5 Numerical experiments

Parameter settings used in numerical experiments are the following: population size $n = 100$, number of generation $=1000$, $Cr = 0.7$, $F = 0.25$.

5.1 Game 1

Let us consider the two person game G_1 [4], having the following payoff functions:

$$u_1(s_1, s_2) = s_1(10 - \sin(s_1^2 + s_2^2)),$$

$$u_2(s_1, s_2) = s_2(10 - \sin(s_1^2 + s_2^2)),$$

$$s_1, s_2 \in [0, 10].$$

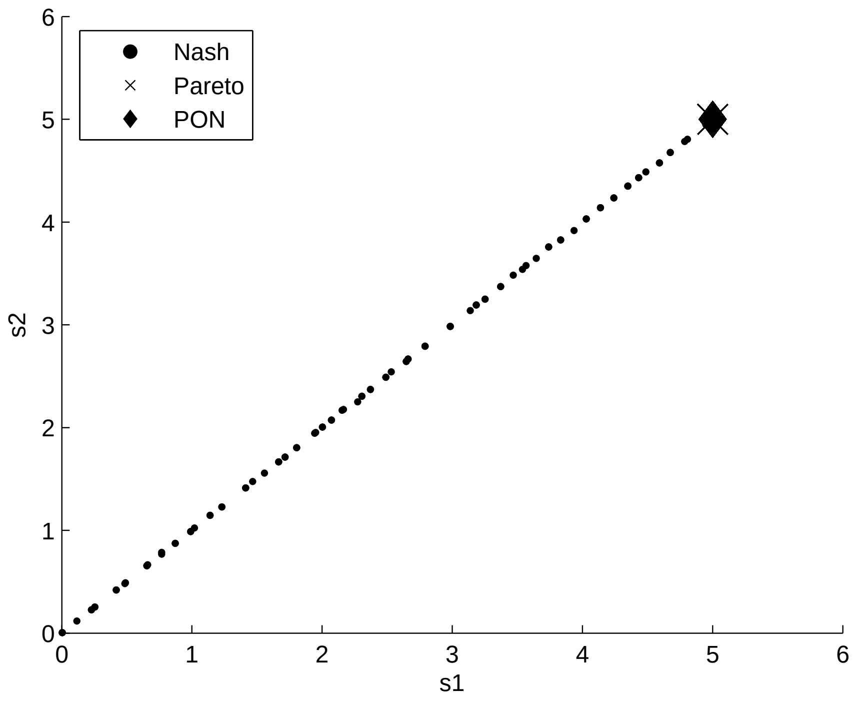


Figure 3: Detected Pareto front, Nash equilibrium, and Pareto-optimal Nash equilibrium strategies for the two-player version of Game 2

Obtained strategies are depicted in Figure 3, obtained payoffs are depicted in Figure 2. The Pareto-optimal Nash equilibrium reduces the set of Nash equilibria.

5.2 Game 2

Let us consider the Bryant game [3], where the payoff function is the following:

$$u_i(s) = c_i - s_i,$$

where $c_i = \alpha \min\{s_1, s_2, \dots, s_n\}$, $\alpha = 2$, and $s_i \in [0, 5], i = 1, \dots, n$.

Experiments with the two-person version of the game are presented in Figure 3 (strategies) and 4 (corresponding payoff). The set of the Nash equilibrium is the whole interval $s^* = (s_1^*, s_2^*) \in [0, 5]$, but the Pareto-optimal Nash equilibrium is only the point $(5, 5)$.

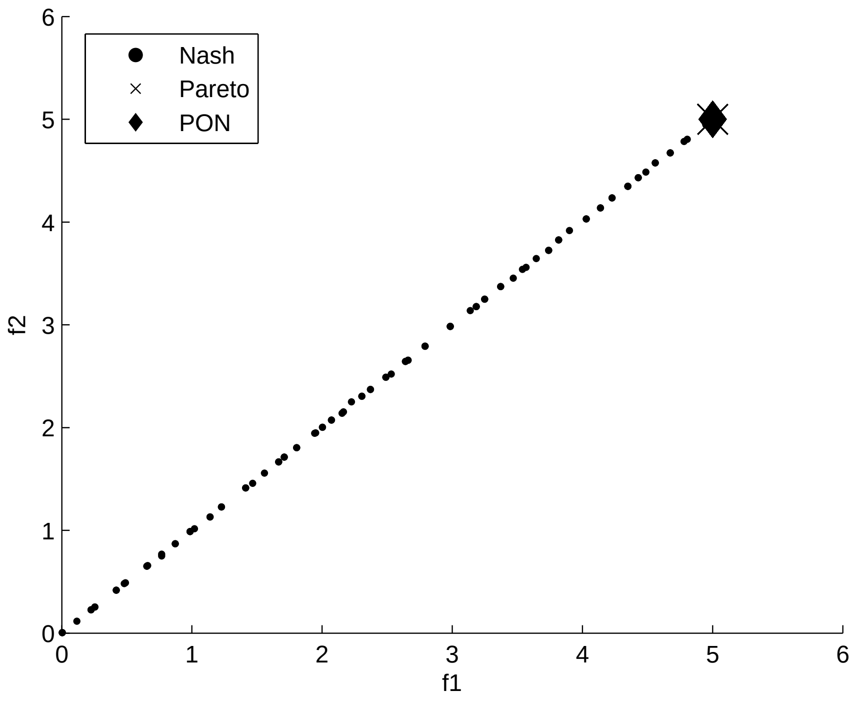


Figure 4: Detected Pareto front, Nash equilibrium, and Pareto-optimal Nash equilibrium payoff for the two-player version of Game 2

Figure 5 presents the same game with three players. The results are the same, the algorithm approximates well the Pareto-optimal Nash equilibrium. Numerical experiments were also conducted for 4, 5 players, and the algorithm finds correctly the Pareto-optimal Nash equilibrium of the game.

6 Conclusions

Nash equilibrium is not always the best solution in all non-cooperative games. In games having several Nash equilibria a selection problem can appear, which Nash equilibrium is best from the detected equilibria.

Pareto-optimal Nash equilibrium is a refinement of the Nash equilibrium. An evolutionary method based on generative relations is considered in order to detect this equilibrium. For the best of our knowledge this is the first evolutionary method for detecting the Pareto optimal Nash equilibrium.

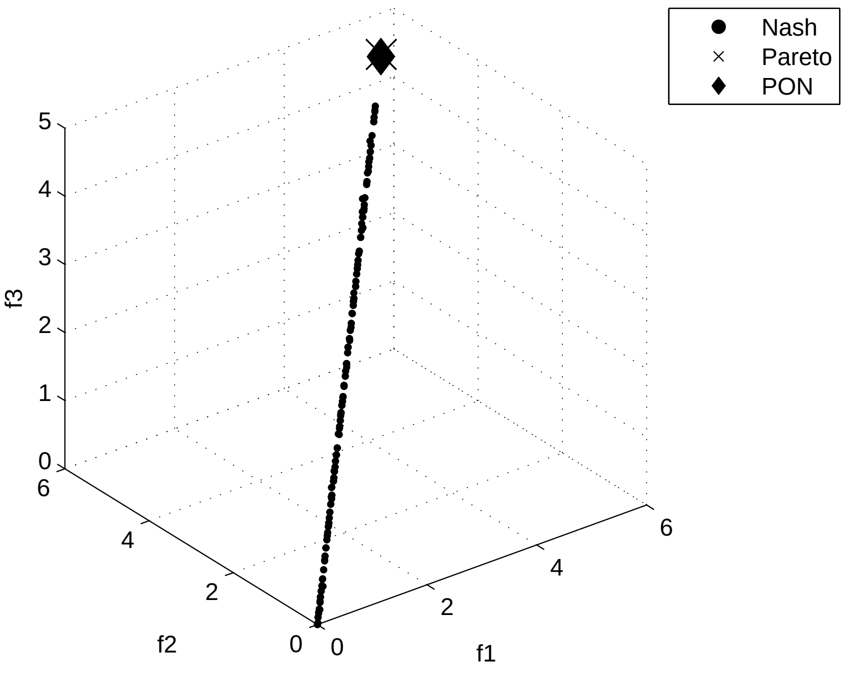


Figure 5: Detected Pareto front, Nash equilibrium, and Pareto-optimal Nash equilibrium payoff for the three-player version of Game 2

Numerical experiments show the potential of the proposed method. The method can deal with continuous and also discrete games. Further work will address games with multiple players.

Acknowledgements

This project was supported by the national project code TE 252 financed by the Romanian Ministry of Education and Research CNCSIS-UEFISCSU. The first author wishes to thank for the support of "Collegium Talentum". The second author wishes to thank for the financial support of the Sectoral Operational Programme for Human Resources Development 2007-2013, co-financed by the European Social Fund, under the project number POS-DRU/107/1.5/S/76841 with the title Modern Doctoral Studies: Internationalization and Interdisciplinarity. This publication was made possible through the support of a grant from the John Templeton Foundation. The opinions expressed in this publication are those of the authors and do not necessarily reflect the views of the John Templeton Foundation. This research is supported by Grant TE 320 - Emergence, auto-organization and evolution: New computational models in the study of complex systems, funded by CNCSIS, Romania.

References

[1] R. Aumann, Acceptable points in general cooperative n person games, *Contributions to the theory of games, Vol IV, Ann. of Math. Stud.*, **40** (1959) 287–324. ⇒237

[2] B. D. Bernheim, B. Peleg, M. D. Whinston, Coalition-proof equilibria. I. Concepts., *J. Econ. Theory* **42,** 1 (1987) 1–12. ⇒237

[3] J. Bryant, A simple rational expectations Keynes type model, *Quarterly J. Economics*, **98,** (1983) 525-529. ⇒243

[4] D. Dumitrescu, R. I. Lung, N. Gaskó, T. D. Mihoc, Evolutionary detection of Aumann equilibrium, *Genetic and Evolutionary Computation Conference, GECCO 2010*, pp. 827–828, 2010. ⇒242

[5] N. Gaskó, R. I. Lung, D. Dumitrescu, Modified strong and coalition proof Nash equilibria. An evolutionary approach, *Studia Univ. Babeş-Bolyai, Inform.* **61,** 1 (2011) 3–10. ⇒237

[6] J. Greenberg, *The core and the solution as abstract stable sets*, Mimeo, University of Haifa, 1987. ⇒237

[7] R. I. Lung, D. Dumitrescu, Computing Nash equilibria by means of evolutionary computation, *Int. J. Comput., Comm. & Control,* **3,** Suppl.(2008) 364–368. ⇒239

[8] K. Mordecai, S. Hart, Pareto-optimal Nash equilibria are competitive in a repeated economy, *J. Econ. Theory*, **28,** (1982) 320–346. ⇒239

[9] J. F. Nash, Non-cooperative games, *Ann. of Math.*, **54,** (1951) 286–295. ⇒237, 238

[10] D. Ray, Credible coalitions and the core, *Internat. J. Game Theory*, **18,** 2 (1989) 185–187. ⇒237

[11] R. Storn, K. Price, Differential evolution – A simple and efficient adaptive scheme for global optimization over continuous spaces, *J. Global Optim.*, **11,** (1997) 341–359. ⇒240

Received: November 1, 2012 • Revised: December 18, 2012

Acta Univ. Sapientiae, Informatica, **4,** 2 (2012) 247–259

Abstract levels of programming theorems

Tibor GREGORICS
Eötvös Loránd University, Faculty of Informatics
Budapest
email: gt@inf.elte.hu

Abstract. In this paper three abstract levels of programming theorems are introduced. These levels depend on the form of the sequence of the elements that are produced by a programming theorem. We are going to investigate the difference between the solutions of the same problem if these solutions are derived from altering abstract levels of the same programming theorem. One of the famous programming theorems, the maximum selection, is chosen as an example, all of its three versions will be presented, and their usage will be shown in a case study.

1 Introduction

Programming theorems are used frequently to plan algorithms. A programming theorem is a pattern, a problem-program (task-algorithm) pair where a program can solve the problem. All the tasks that are similar to the problem of a theorem can be solved on the basis of the algorithm of the theorem. Programming theorems (summation, counting, maximum selection, and linear search, etc.) [2, 4] are well-known by all programmers but only a few of them know that these theorems can be expressed in multiple ways. Most programmers consider programming theorems as sample solutions. When they want to solve a task that is similar to the problem of a theorem, they try to repeat the same activities that created the program of the theorem. Thus programming theorems support their algorithmic way of thinking that is used to construct

Computing Classification System 1998: F.3.1
Mathematics Subject Classification 2010: 68N30
Key words and phrases: analogous programming, programming theorem, enumerator, object-oriented programming

the algorithm of their task. However, there exists another method to create programs. This is derivation [1, 4]. Starting from the exact comparison of the task to be solved and of the problem of the candidate programming theorem, the program of the theorem has to be updated according to the differences between the task and the problem. Thus, without algorithmic way of thinking, the program by which the new task is solved can be produced almost automaticly. This method is faster and guarantees the correctness of the algorithm but it requires the formal description of the task. The whole of this paper can be articulated from this single point of view, i.e. when algorithms are planned with derivation.

The quality (efficiency and compactness) and, very often, the success of the solution depend on the degree of the universality of the programming theorem. According to the way that the problem of the theorem is generalized, different versions of the theorem can be obtained. It is obvious that a good programming theorem should be adequately universal so that the class of the tasks to be solved is wide enough. But the theorem must preserve some specialty in order that it can be identified in a simple way. The reason for this, for example, is that counting is a separate theorem; nevertheless, it is a special case of summation.

One of the common properties of programming theorems is that they process a sequence of elementary values. The way these values are produced may differ. Programming theorems may be distinguished according to these three levels. Henceforth these levels are going to be defined, the versions of maximum selection are going to be fully given, and various solutions of the same task are going to be produced by using different levels of the same programming theorem.

2 Different forms of the sequence of elements are processed

A sequence of elementary values can be placed into a container such as a sequential file or a linked list. The most widely-known container, however, is the one-dimensional array. Most programmers use the programming theorems processing the elements of an array. This is the lowest level of the programming theorems.

A higher level is the one when an appropriate function gives the elements that must be proccessed. The domain of this function is always an interval of integers. (Hereafter $[n..m]$ denotes the integer interval $[n, m] \cap \mathbb{Z}$ for all

$n, m \in \mathbb{Z}$.) This function is more universal than an array: each array can be interpreted as a function over integer interval.

The third level is when the elements are provided by a special activity, an enumeration. The enumerator is an object that disposes the four enumeration operators: `First(), Next(), End(), Current()` [5]. These operators permit iterating the elements that must be processed. The elements of an array can be iterated like the proper divisors of a natural number. This point of view gives more universal definitions of programming theorems.

3 Different levels of the programming theorems

Now the maximum selection is going to be defined in three different forms. Other programming theorems can be defined in this same way.

3.1 Maximum selection in an array

An array over the non-empty integer interval $[m..n]$ is given where the elements on the array form a totally ordered set, set H (notation: $H^{m..n}$). The greatest element of the array is sought, and one of the indexes should also be given where this element occurs.

Specification:

In the formal specification used below the letter A denotes the state space that enumerates the variables of the problem with their types. The letter Q is the precondition and R is the postcondition of the problem. If v is a variable of the state space, then the notation v' is an arbitrary, initial value of the variable v. The variable i is the index variable of the `for` statement.

$$\begin{aligned} A &= (x : H^{m..n}, max : H, ind : \mathbb{Z}) \\ Q &= ((x = x') \land (n \leq m)) \\ R &= ((x = x') \land (ind \in [m..n]) \land (max = x[ind] = \underset{i=m}{\overset{n}{MAX}}\, x[i])) \end{aligned}$$

Algorithm:

```
$max, ind := x[m], m;$
**for** $i = m+1 \dots n$ **do**
    **if** $x[i] > max$ **then**
        $max, ind := x[i], i;$
    **endif**
**endfor**
```

3.2 Maximum selection over interval

There are many problems, the solution of which can not be derived from a programming theorem in an array but from the same programming theorem over an interval. For example, suppose the average temperatures of successive days are fixed in an array (its elements are indexed by the integer interval $[1..n]$) and the neighboring pairs of temperature must be counted where the first value of the pair is under freezing point and the second one is above it. This task cannot be derived from the counting in an array because the elements that must be checked in the counting are not elements of an array. These elements are logical values provided by a logical function (condition) that is defined over the integer interval $[2..n]$, and these values depend on the pairs of the original array of the task. Sometimes there is no array at all in a problem. For example, if the proper divisors of a given natural number have to be counted, then the function $f(i) = i$ *can divide* n (which is defined over the integer interval $[2..n/2]$) should be checked. Anyway programming theorems on array are looked upon as special cases of programming theorems over interval because each array can be interpreted as a function over integer interval.

There is a non-empty integer interval $[m..n]$ and a function $f : [m..n] \to H$, where H is a totally ordered set. The greatest value of the function is sought, and one of its arguments should also be given.

Specification:

$$\begin{aligned} A &= (m:\mathbb{Z}, n:\mathbb{Z}, max:H, ind:\mathbb{Z}) \\ Q &= ((m=m') \wedge (n=n') \wedge (n \le m)) \\ R &= ((m=m') \wedge (n=n') \wedge (ind \in [m..n]) \\ &\quad \wedge (max = f(ind) = \underset{i=m}{\overset{n}{MAX}} f(i))) \end{aligned}$$

The postcondition can be written in a shorter form. In this notation, MAX is a multi-valued function mapping from an interval to H and $\mathbb{Z}$.

$$R = ((m=m') \wedge (n=n') \wedge ((max, ind) = \underset{i=m}{\overset{n}{MAX}} f(i)))$$

Algorithm:

```
$max, ind := f(m), m;$
**for** $i = m+1 \dots n$ **do**
    **if** $f(i) > max$ **then**
        $max, ind := f(i), i;$
    **endif**
**endfor**
```

3.3 Maximum selection on enumerator

There is an enumerator that can iterate the elements of a finite non-empty sequence which belongs to set E ($enor(E)$ notates the type of this enumerator). A function is given $f : E \to H$ where H is a totally ordered set. The greatest value over the values mapped from the elements of the enumerator by the function f is sought, and one element should also be given where this value occurs.

Specification:

$$\begin{aligned} A &= (t : enor(E), max : H, ind : E) \\ Q &= ((t = t') \wedge (|t| \neq 0)) \\ R &= ((e \in t') \wedge (max = f(ind) = \underset{e \in t'}{MAX} f(e))) \end{aligned}$$

Algorithm:

```
t.First();
max, ind := f(t.Current()), t.Current();
t.Next();
while ¬t.End() do
    if f(t.Current()) > max then
        max, ind := f(t.Current()), t.Current();
    endif
endwhile
```

4 Case study

Let us solve the following problems. There is a plan where n points are given. Which is the greatest distance between pairs of points?

The points on the plan can be represented by their coordinates if there is a fixed coordinate system. These coordinates are saved in two one-dimensional arrays: x and y. The coordinates of the i^{th} point are $x[i]$ and $y[i]$. The distance between the i^{th} and j^{th} points is $\sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$ but the greatest distance is not wanted, so it will be enough to use their squares.

Let $d(i, j)$ denote the square distance $(x[i] - x[j])^2 + (y[i] - y[j])^2$ $(i = 1..n,$ $j = 1..n)$. These values can be arranged in an $n \times n$ symmetrical matrix. Our task is to select the maximal element of the lower triangular part of this matrix without its diagonal.

It seems that the solution can be created with a maximum selection but the theorem of maximum selection can investigate only one-dimensional forms and here the values of $d(i,j)$ $\big(i,j \in [1..n]\big)$ are in a two-dimensional shape.

4.1 First solution: numerous maximum selections in an array

The rows of the virtual lower triangular matrix can be seen as many one-dimensional arrays with increasing size. If the greatest element is selected from every row and put into an auxiliary array, then a new maximum selection in this array can solve the original problem. More precisely, this auxiliary array indexed by the interval $[2..n]$ (denoted by z) should store value-index pairs ($rec(m:\mathbb{R}, k:\mathbb{N})$) and the i^{th} element of this array can show the greatest value and its index in the i^{th} row. By definition $z[i] > z[j]$ if $z[i].m > z[j].m$.

Specification:

$$\begin{aligned}
A &= (x,y:\mathbb{R}^n, z:rec(m:\mathbb{R},k:\mathbb{N})^{2..n}, max:\mathbb{R}, ind, jnd:\mathbb{N})\\
Q &= ((x=x') \wedge (y=y') \wedge (n \geq 2))\\
R &= ((x=x') \wedge (y=y') \wedge (\forall i \in [2..n]: z[i] = \underset{j=1}{\overset{i-1}{MAX}}\, d(i,j))\\
&\quad \wedge((max, jnd), ind) = \underset{i=2}{\overset{n}{MAX}}\, z[i]))
\end{aligned}$$

This problem can be solved with n maximum selections. The first n–1 maximum selections fill an auxiliary array z and the last one selects the maximal elements of this array.

Each of the first n–1 maximum selections works in one of the rows of the virtual lower triangular matrix. The i^{th} row of this matrix is considered as an array indexed by the interval $[1..i–1]$, and the j^{th} element of this array is the value $d(i,j)$. Obviously these maximum selections can be derived from the programming theorem in an array. The assignment $z[i] := (d(i,j), j)$ is a shorter form of the assignments $z[i].m, z[i].k := d(i,j), j$

```
for $i = 2 \ldots n$ do
    $z[i] := (d(i,1), 1)$;
    for $j = 2 \ldots i–1$ do
        if $d(i,j) > z[i].m$ then
            $z[i] := (d(i,j), j)$;
        endif
    endfor
endfor
```

The last maximum selection is also derived from the programming theorem in an array because it uses the auxiliary array z. The variables i and j are the index-variables of the $\mathtt{for}$ statements.

```
$(\mathtt{max}, \mathtt{jnd}), \mathtt{ind} := z[2], 2;$
for $i = 3 \ldots n$ do
    if $z[i].m > \mathtt{max}$ then
        $(\mathtt{max}, \mathtt{jnd}), \mathtt{ind} := z[i], i;$
    endif
endfor
```

Finally, the whole solution is repeated with minor modifications. Hence the second row of the lower triangular matrix contains only one element, this is the $z[2]$ which can be calculated without maximum selection: $z[2] := (d(2, 1), 1)$. Moreover the assigment $(\mathtt{max}, \mathtt{jnd}) := z[i]$ is equivalent to the assigment $\mathtt{max}, \mathtt{jnd} := z[i].m, z[i].k$, and the value of the variable $\mathtt{jnd}$ is enough to be set at the end of the algorithm. At the end the local auxiliary variable s is introduced to contain the value of $d(i, j)$.

```
$z[2] := (d(2, 1), 1);$
for $i = 3 \ldots n$ do
    $z[i] := (d(i, 1), 1);$
    for $j = 2 \ldots i{-}1$ do
        $s := d(i, j);$
        if $s > z[i].m$ then
            $z[i] := (s, j);$
        endif
    endfor
endfor
$\mathtt{max}, \mathtt{ind} := z[2].m, 2;$
for $i = 3 \ldots n$ do
    if $z[i].m > \mathtt{max}$ then
        $\mathtt{max}, \mathtt{ind} := z[i].m, i;$
    endif
endfor
$\mathtt{jnd} := z[\mathtt{ind}].k;$
```

Note that the auxiliary array can be eliminated from this program if the two outsider loops (where i goes from 3 to n) are combined. However, this solution can be produced in a simpler way if a more generalized programming theorem is used.

4.2 Second solution: several maximum selections over interval

Let us follow the previous line of thought but instead of the auxiliary array, a function is going to be defined which gives the greatest element and its index from every row of the virtual lower triangular matrix.

Specification:

$$\begin{aligned} A &= (x, y : \mathbb{R}^n, max : \mathbb{R}, ind, jnd : \mathbb{N}) \\ Q &= ((x = x') \wedge (y = y') \wedge (n \geq 2)) \\ R &= ((x = x') \wedge (y = y') \wedge ((max, jnd), ind) = \underset{i=2}{\overset{n}{\mathrm{MAX}}}\, g(i))) \end{aligned}$$

where $g : [2..n] \to \mathbb{R} \times \mathbb{N}$ and $g(i) = \underset{j=1}{\overset{i-1}{\mathrm{MAX}}}\, d(i,j))$

This problem can be solved with the maximum selection over interval 2..n with the function g. By definition, $g(i) > g(j)$ if $g(i)_1 > g(j)_1$. The variable i is the index-variable of the for statement, the variable m and k are local auxiliary variables.

```
$(max, jnd), ind := g(2), 2;$
for $i = 3 \ldots n$ do
    $(m, k) := g(i);$
    if $m > max$ then
        $max, jnd, ind := m, k, i;$
    endif
endfor
```

The subproblem $(m, k) := g(i)$ is also a maximum selection but its interval is $[1..i\text{–}1]$ and its function is $d(i, j)$. The variable j is the index-variable of the for statement. The variable s is a local auxiliary variable. The main program calls this subprogram twice.

```
$m, k := d(i, 1), 1;$
for $j = 2 \ldots i\text{–}1$ do
    $s := d(i, j);$
    if $s > m$ then
        $m, k := s, j;$
    endif
endfor
```

We can compare this solution with the previous one if this program is combined. The value of $g(2)$ can be calculated as $d(2, 1)$ and 1; hence, the initial assignment will be $max, jnd, ind := d(2, 1), 1, 2$.

```
$max, jnd, ind := d(2,1), 1, 2;$
for $i = 3 \dots n$ do
    $m, k := d(i,1), 1;$
    for $j = 2 \dots i–1$ do
        $s := d(i,j);$
        if $s > m$ then
          $m, k := s, j;$
        endif
    endfor
    if $m > max$ then
       $max, jnd, ind := m, k, i;$
    endif
endfor
```

4.3 Third solution: one maximum selection over interval

Let us imagine that the elements of the lower triangular matrix are in a sequence. The first element of this sequence is the single element of the second row (it is indexed with $(2,1)$), the 2^{nd} element is the one on the $(3,1)$ position, the 3^{rd} is the $(3,2)$, 4^{th} is $(4,1)]$ and so on. The size of this sequence is $n(n–1)/2$. How can the i^{th} element of this sequence be found in the matrix?

It is easy to see that the $(i,j)^{\text{th}}$ element of the lower triangular matrix $(j < i)$ is the $((i–1)(i–2)/2 + j)^{\text{th}}$ element of the sequence because there are $(i–1)(i–2)/2$ elements in front of the i^{th} row in the lower triangular matrix. But where can the k^{th} element of the sequence be found in the matrix?

Lemma 1 *The* k^{th} *element of the sequence is the* $(i,j)^{\text{th}}$ *element of the matrix where* $j = 2k-(i-1)(i-2)$ *and if* $2k > \lceil\sqrt{2k}\rceil(\lceil\sqrt{2k}\rceil-1)$*, then* $i = \lceil\sqrt{2k}\rceil+1$*, otherwise* $i = \lceil\sqrt{2k}\rceil$.

Proof. Because of $k = (i-1)(i-2)/2 + j(j < i)$, we get

$$(i-1)(i-2) < 2k \leq i(i-1). \qquad (1)$$

It follows that $(i-2) < \sqrt{2k} < i$, so the value $\lceil\sqrt{2k}\rceil$ (upper integer part) may be i or $i-1$. If $2k > \lceil\sqrt{2k}\rceil(\lceil\sqrt{2k}\rceil-1)$, then $\lceil\sqrt{2k}\rceil = i-1$ because supposing $\lceil\sqrt{2k}\rceil = i$ we get $2k > i(i-1)$ that is a contradiction of (1). If $2k \leq \lceil\sqrt{2k}\rceil(\lceil\sqrt{2k}\rceil-1)$, then $i = \lceil\sqrt{2k}\rceil$ because supposing $\lceil\sqrt{2k}\rceil = i-1$, we get $2k \leq (i-1)(i-2)$ that is also a contradiction. $\square$

Based on this lemma, the following function can be defined:

$h : [1..n(n–1)/2] \to \mathbb{N} \times \mathbb{N}$

$$h(k) = \begin{cases} (\lceil\sqrt{2k}\rceil + 1, 2k\lceil\sqrt{2k}\rceil(\lceil\sqrt{2k}\rceil - 1)) & \text{if} \quad 2k > (\lceil\sqrt{2k}\rceil - 1)\lceil\sqrt{2k}\rceil \\ (\lceil\sqrt{2k}\rceil, 2k(\lceil\sqrt{2k}\rceil - 1)(\lceil\sqrt{2k}\rceil - 2) & \text{if} \quad 2k \leq (\lceil\sqrt{2k}\rceil - 1)\lceil\sqrt{2k}\rceil \end{cases}$$

Now the problem can be re-specified. We introduce the auxiliary variable knd.

$$\begin{aligned} A &= (x, y : \mathbb{R}^n, max : \mathbb{R}, ind, jnd, knd : \mathbb{N}) \\ Q &= ((x = x') \wedge (y = y') \wedge (n \geq 2)) \\ R &= ((x = x') \wedge (y = y') \wedge ((max, knd) = \underset{k=1}{\overset{n(n-1)/2}{MAX}} d(h(k))) \\ &\quad \wedge((ind, jnd) = h(knd))) \end{aligned}$$

This problem can be derived to the maximum selection over the interval $[1..n(n–1)/2]$ with the function h. In the initial assignment, the expression $d(h(1))$ can be changed to $d(2, 1)$. The variable k is the index-variable of the for statement, the variable s is a local auxiliary variable.

```
$max, knd := d(2, 1), 1;$
**for** $k = 2 \ldots n(n–1)/2$ **do**
    **if** $d(h(k)) > max$ **then**
        $max, knd := s, k;$
    **endif**
**endfor**
$(ind, jnd) := h(knd)$
```

Let us take some minor modifications. The knd auxiliary variable can be eliminated but the local auxiliary variables i, j and s are introduced.

```
$max, ind, jnd := d(2, 1), 2, 1;$
**for** $k = 2 \ldots n(n–1)/2$ **do**
    $(i, j) := h(k);$
    $s := d(i, j);$
    **if** $s > max$ **then**
        $max, ind, jnd := s, i, j;$
    **endif**
**endfor**
```

4.4 Fourth solution: one maximum selection on enumerator

The specification of the problem can be rewritten:

$$\begin{aligned}
A &= (x, y : \mathbb{R}^n, max : \mathbb{R}, ind, jnd : \mathbb{N})\\
Q &= ((x = x') \wedge (y = y') \wedge (n \geq 2))\\
R &= ((x = x') \wedge (y = y') \wedge ((max, (ind, jnd)) = \underset{i=2}{\overset{n}{\mathrm{MAX}}}(\underset{j=1}{\overset{i-1}{\mathrm{MAX}}}\, d(i, j))))\\
&= ((x = x') \wedge (y = y') \wedge ((max, (ind, jnd)) = \underset{i=2,j=1}{\overset{n,j-1}{\mathrm{MAX}}}\, d(i, j))).
\end{aligned}$$

The last expression of this specification resembles a two-dimensional enumeration [5]. This enumeration should traverse the elements of a virtual lower triangular matrix, i.e. the sequence of index pairs $(2, 1), (3, 1), (3, 2), (4, 1), \dots, (n, 1), (n, 2), \dots, (n, n–1)$ should be enumerated. Let us take this enumerator into the state space.

$$\begin{aligned}
A &= (t : enor(\mathbb{N} \times \mathbb{N}), max : \mathbb{R}, ind, jnd : \mathbb{N})\\
Q &= ((t = t') \wedge (|t| \neq 0))\\
R &= ((max, (ind, jnd)) = \underset{(i,j)\in t'}{\mathrm{MAX}}\, d(i, j))).
\end{aligned}$$

The enumerator handles two indexes: i and j. The operator $First()$ set them to the pair $(2, 1)$, the operator $Next()$ increases the variable j if $j < i–1$, otherwise $(j = i–1)$ increases the variable i and set j to 1. The operator $End()$ gives $true$ if $i > n$. (This process produces the same sequence of index pairs as in the previous solution.) This enumeration can be implemented with a double loop and can be combined with the maximum selection [5]. The local auxiliary variable s is also introduced.

```
$max, ind, jnd := d(2, 1), 2, 1;$
for $i = 3 \dots n$ do
    for $j = 2 \dots i–1$ do
        $s := d(i, j);$
        if $s > max$ then
            $max, ind, jnd := s, i, j;$
        endif
    endfor
endfor
```

5 Discussion

The first two solutions in the case study are very similar. It is easy to see that the second algorithm can be received from the first one through applying equivalent transformations [3]. But the second solution, which is based on a more universal programming theorem, avoids these transformations. We emphasize that using a more universal programming theorem results in a more efficient algorithm (it does not require an auxiliary array, so its memory space is smaller).

The second and third solutions are based on the programming theorem over interval but the third one uses a function abstraction. Therefore, in the third solution, it is enough to apply the theorem of maximum selection only once. Here the structure of the solving algorithm is simpler than the algorithm of the second solution; moreover, it is the simplest structure among all solutions.

The relationship between the third and fourth algorithm can be seen clearly. Both of them are founded on the same idea, that is, they traverse the elements of the lower triangular matrix row by row. Actually the third solution is complicated. It uses a function abstraction but this function is not trivial. The fourth solution uses a data abstraction when it defines and implements an appropiate enumerator. The structure of the algorithm of the fourth solution is a loop in the loop which is more difficult than the single loop of the third algorithm, but this double loop is the routine algorithm among matrices. Nevertheless, the enumeration could have been implemented in other ways (as we have pointed out) and in this case the algorithm would be a simple loop. The cost of the production of the fourth solution is surely cheaper than that of the third one.

On the whole we can deduct that the more universal programming theorem is used the cheaper the solution is. The cost of production may be cheaper, the structure of the result algorithm may be simpler or its efficient may be better. Then again, to learn and to use an advanced tool is always more difficult than a simple one. Bescause of this in the teaching of programming, gradation must be followed: firstly, programming theorems in an array are to be taught, then the ones over interval, and at the end the theorems on enumerator.

Acknowledgements

This paper is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003).

References

[1] T. Gregorics, S. Sike, Generic algorithm patterns, *Proc. Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS 2008*, Budapest, Hungary, March 29, 2008, 141–150. ⇒248

[2] Á. Fóthi, *Bevezetés a programozáshoz*, ELTE Eötvös Kiadó. 2005. (Introductory programming, in Hungarian) ⇒247

[3] Á. Fóthi, Z. Horváth, J. Nyéky-Gaizler, A relational model of transformation in programming, *Proc. 3th International Conference on Applied Informatics*, Eger-Noszvaj, Hungary, August 24–28. 1997. ⇒258

[4] Sz. Csepregi, A. Dezső, T. Gregorics, S. Sike, Automatic implementation of service required by components, *PROVECS'2007 Workshop*, Zurich, Switzerland, ETH Technical Report, 567. 2007. ⇒247, 248

[5] T. Gregorics, Programming theorems on enumerator, *Proc. Teaching Mathematics and Computer Science*, Debrecen, Hungary, **8**/1 (2010), 89–108. ⇒249, 257

Received: August 29, 2012 • Revised: November 30, 2012

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 $s = (s_1, \dots, s_n)$ 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 $(n \geq 4)$;
$s = (s_1, \dots, s_n)$: the investigated regular sequence.

Output. 0 or 1.

Work variable. i: cycle variable;
$r = (r_1, \dots, r_n)$: r_i the reserve belonging to s_i;
$w = (w_1, \dots, w_n)$: w_i the weight point belonging to s_i;
$H = (H_1, \dots, H_n)$: 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 $s_i \leq i$ 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. $\square$

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 $s = (s_1, \dots, s_n)$ 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 $(T_{\mathrm{EGL}}(n)/E_z(n))$, EGLJ $(T_{\mathrm{EGLJ}}(n)/E_z(n))$, and HHL $(T_{\mathrm{HHL}}(n)/E_z(n))$, when $n = 10, \dots, 19$. The values $n = 1, \dots, 9$ are omitted from the table since our program rounds the running time to zero.

n	$E_z(n)$	$\frac{T_{\mathrm{EGL}}(n)}{E_z(n)}$	$\frac{T_{\mathrm{EGLJ}}(n)}{E_z(n)}$	$\frac{T_{\mathrm{HHL}}(n)}{E_z(n)}$
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).

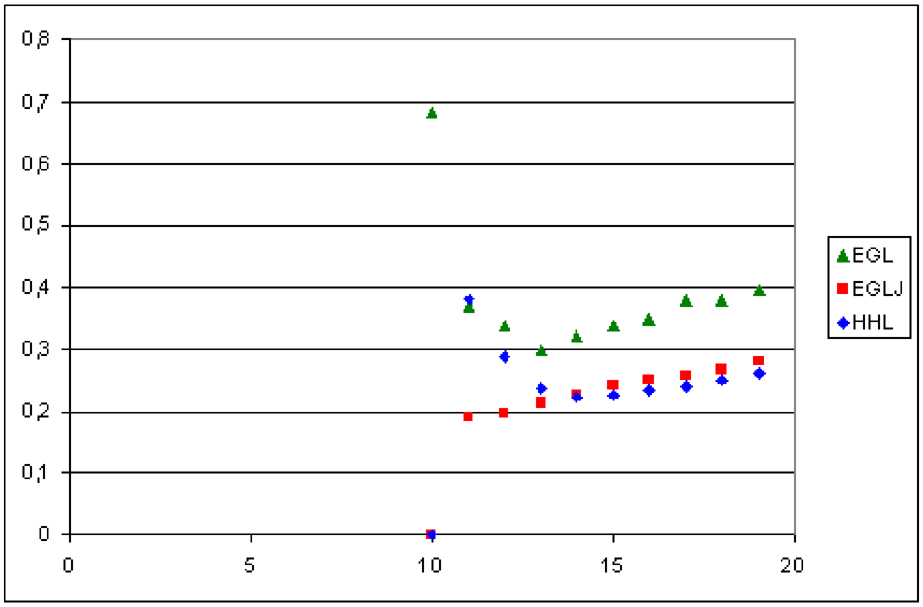


Figure 1: Average running time of EGL, EGLJ, and HHL.

Table 2 shows the average number of operations used to test one zerofree sequence in microseconds for EGL ($O_{\mathrm{EGL}}(n)/E_z(n)$), EGLJ ($O_{\mathrm{EGLJ}}(n)/E_z(n)$), and HHL ($O_{\mathrm{HHL}}(n)/E_z(n)$), 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	$\frac{O_{\mathrm{EGL}}(n)}{E_z(n)}$	$\frac{O_{\mathrm{EGLJ}}(n)}{E_z(n)}$	$\frac{O_{\mathrm{HHL}}(n)}{E_z(n)}$
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

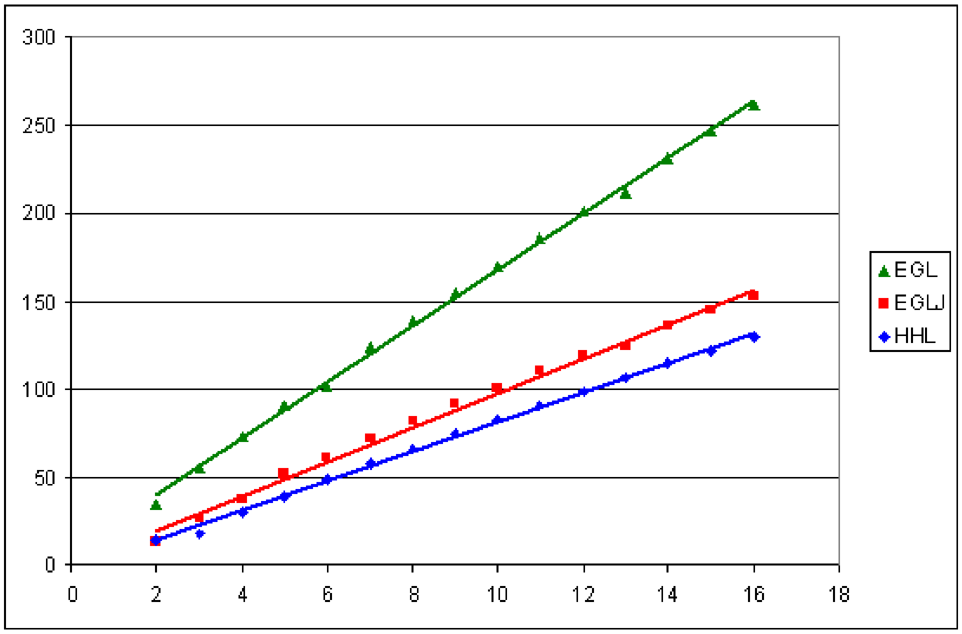


Figure 2: Amortized number of operations for EGL, EGLJ, and HHL.

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 ($o_{\text{EGL}}(n)/E_z(n)$), EGLJ ($o_{\text{EGLJ}}(n)/E_z(n)$), and HHL ($o_{\text{HHL}}(n)/E_z(n)$), 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	$G_z(n)$	$\frac{O_{\mathrm{EGL}}(n)}{E_z(n)}$	$\frac{O_{\mathrm{EGLJ}}(n)}{E_z(n)}$	$\frac{O_{\mathrm{HHL}}(n)}{E_z(n)}$
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.

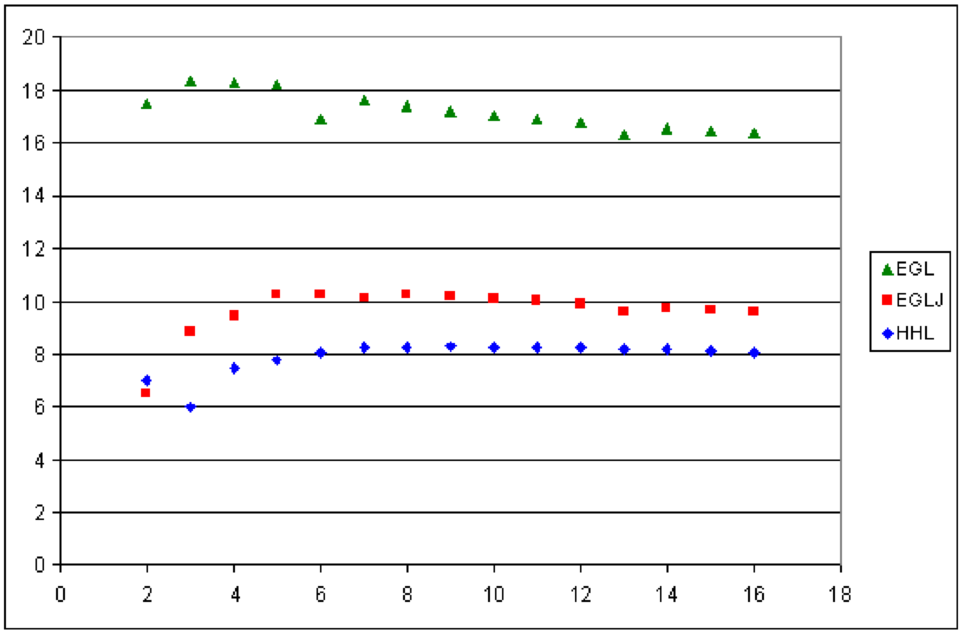


Figure 3: Average number of operations used for one element of zerofree sequences by EGL, EGLJ, and HHL.

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	26876802183368505747610
41	212392290424395860814420	106196145212266853671620
42	839455243105945545123660	419727621553107337030440
43	3318776542511877736535400	1659388271256207997204920
44	13124252690842425594480900	6562126345421738821981380
45	51913710643776705684835560	25956855321889404891899640
46	205397724721029574666088520	102698862360516845690726160
47	812850570172585125274307760	406425285086296679352517680
48	3217533506933149454210801550	1608766753466582789006321550
49	12738806129490428451365214300	6369403064745230349484448700
50	50445672272782096667406248628	25222836136391079936354733752
51	199804427433372226016001220056	99902213716686176213303828904
52	791532924062974587678774064068	395766462031487417819020269060
53	3136262529306125724764953838760	1568131264653063110341743393432
54	12428892245768720464809261509160	6214446122884360719139487166608
55	49263609265046928387789436527216	24631804632523465167364431087664
56	195295022443578894680165266232892	97647511221789449252255283306556
57	774327632846470705223111406467256	387163816423235356435901003613848
58	3070609578529107968988200404956360	1535304789264553992010916827363440
59	12178349853827309571919303301013360	6089174926913654800993284900277200
60	48307454420181661301946569760686328	24153727210090830680539430271558520

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 $b_2 \geq 2$ or $b_1 \geq 3$                // line 10: last sequence was?
11     if $b_n \geq 3$                   // 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

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

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             $M_{i,j} = \binom{i+j-2}{i-1}$
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 $S + M_{i,j} \le ms$              // line 03: if we can add more sequences
04             $S = S + M_{i,j}$                            // line 04: add more sequences
05             if $j \le j_m$       // lines 05–06: line: move to next column in matrix
06                $j = j + 1$
07          else if $S \ne 0$                            // 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 $j \le j_m$           // line 14: if decomposable
15                   GENERATE-NEW-SEQUENCE($M, n, i - 1, 1, j, ms, [J, j]$)
```

```
16              $j = j + 1$
17 if $S \neq 0$                          // 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. b: input sequence;
$H = (H_1, \ldots, H_n)$: sums of the elements of b;
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 $i \leq c$ 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 $b_{J_p} \leq p$
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($n, b, last\_index, last\_value$)
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 $b_n \neq n - 1$                // 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** $b_i \neq b_{i+1}$ **and** $b_i \neq b_n$
14         $c = c + 1$
15         $C_c = i$
16 $L = \textsc{Check}(b, H, c, C)$               // 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** $b_j \leq 1$
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** $(c \leq 2$ **or** $(c > 2$ **and** $C_{c-2} \neq j - 2))$ **and**
           $(c > 1$ **and** $C_{c-1} \neq j - 2)$
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 $C_c \neq j - 1)$ or $(c > 1$ and $C_{c-1} \neq j - 1)$
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 $C_c \neq n - 2)$
                  or $(c > 1$ and $C_{c-1} \neq n - 2$ and $C_c \neq n - 2)$
                                               // 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 $C_c \neq n - 1$
                                   // lines 82–84: update check points
83                $c = c + 1$
84                $C_c = n - 1$
85        $G_z^p = G_z^p + \text{Check}(b, H, c, C)$       // 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 $n = 1, \dots, 100$, 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

AN INTRODUCTION TO GRAPH THEORY

Shariefuddin Pirzada
Universities Press, Hyderabad (India), 2012
ISBN: 978 81 7371 760 4

The book is primarily intended for use as textbook at the graduate level, but the first eight chapters can be used as a one-semester course in the undergraduate level for students of mathematics and engineering.

The final sections of several chapters introduce advanced topics and unsolved problems that are the object of current research in graph theory. Thus, the book can also be used by students pursuing research work in PhD programs.

The book consists of preface, 12 chapters, references, and index, the total size is 6+396 pages. The first chapter (*Introduction*) contains the basic definitions and theorems of graph theory. The second chapter (*Degree Sequences*) deals with degree sequences. The chapter contains much more results as other textbooks of graph theory. The following 6 chapters (*Eulerian and Hamiltonian Graphs, Trees, Connectivity, Planarity, Colourings, Matchings and Factors*) contain the basic results of the given topic. The last four chapters (*Edge Graphs and Eccentricity Sequences, Graph Matrices, Digraphs, Score Structures in Digraphs*) deals with advanced topics of graph theory. Especially rich material is gathered on score structures including many recent results of the author of the book and his coauthors.

The book is closed by 266 references on papers and books which appeared between 1736 and 2010 and by index.

In the textbook is disturbing the inconsistent notation of mathematical symbols in the text and in the figures: instead of italic they are often written by normal letters.

We recommend author to follow the authors (as D. E. Knuth, T. H. Cormen, N. A. Lynch) of popular American textbooks: to assemble and maintain on his homepage a list of errors of the book and later to publish corrected editions. Another proposition is to extend the material with the performance analysis of the described algorithms to make the textbook more informative for the students of computer science.

Despite of the errors the book contains valuable material therefore we recommend it to undergraduate, graduate and PhD students.

Contents
Volume 4, 2012

Acta Universitatis Sapientiae

The scientific journal of Sapientia Hungarian University of Transylvania publishes original papers and surveys in several areas of sciences written in English. Information about each series can be found at `http://www.acta.sapientia.ro`.

Each volume contains two issues.

Sapientia University

Scientia Publishing House

ISSN 1844-6086
http://www.acta.sapientia.ro

Information for authors

Acta Universitatis Sapientiae, Informatica publishes original papers and surveys in various fields of Computer Science. All papers are peer-reviewed.

Papers published in current and previous volumes can be found in Portable Document Format (pdf) form at the address: `http://www.acta.sapientia.ro`.

The submitted papers should not be considered for publication by other journals. The corresponding author is responsible for obtaining the permission of coauthors and of the authorities of institutes, if needed, for publication, the Editorial Board is disclaiming any responsibility.

Submission must be made by email (`acta-inf@acta.sapientia.ro`) only, using the LaTeX style and sample file at the address `http://www.acta.sapientia.ro`. Beside the LaTeX source a pdf format of the paper is necessary too.

Prepare your paper carefully, including keywords, ACM Computing Classification System codes (`http://www.acm.org/about/class/1998`) and AMS Mathematics Subject Classification codes (`http://www.ams.org/msc/`).

References should be listed alphabetically based on the Intructions for Authors given at the address `http://www.acta.sapientia.ro`.

Illustrations should be given in Encapsulated Postscript (eps) format.

One issue is offered each author free of charge. No reprints will be available.

Contact address and subscription:
Acta Universitatis Sapientiae, Informatica
RO 400112 Cluj-Napoca
Str. Matei Corvin nr. 4.
Email: acta-inf@acta.sapientia.ro

Printed by Gloria Printing House
Director: Péter Nagy

ISSN 1844-6086
http://www.acta.sapientia.ro