Acta Universitatis Sapientiae

Informatica

Volume 3, Number 2, 2011

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 141–157

C++ Standard Template Library by template specialized containers

Norbert PATAKI
Dept. of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest,
Hungary
email: patakino@elte.hu

Abstract. The C++ Standard Template Library is the flagship example for libraries based on the generic programming paradigm. The usage of this library is intended to minimize the number of classical C/C++ errors, but does not warrant bug-free programs. Furthermore, many new kinds of errors may arise from the inaccurate use of the generic programming paradigm, like dereferencing invalid iterators or misunderstanding remove-like algorithms.

In this paper we present some typical scenarios that may cause runtime or portability problems. We emit warnings and errors while these risky constructs are used. We also present a general approach to emit "customized" warnings. We support the so-called "believe-me marks" to disable warnings. We present another typical usage of our technique, when classes become deprecated during the software lifecycle.

1 Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [2]. In this way containers are defined as class templates and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [23]. C++ STL is widely-used because it is a

Computing Classification System 1998: D.3.2
Mathematics Subject Classification 2010: 68N19
Key words and phrases: C++, STL, template, compilation

very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.) and a large number of algorithms (like sort, find, count, etc.) among other utilities [5].

The STL was designed to be extensible [14]. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with the existing containers. Iterators bridge the gap between containers and algorithms [4]. The expression problem [26] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [1].

However, the usage of C++ STL does not guarantee bugless or error-free code [7]. Contrarily, incorrect application of the library may introduce new kinds of problems [22].

One of the problems is that the error diagnostics are usually complex, and very hard to figure out the root cause of a program error [27, 28]. Violating requirement of special preconditions (e.g. sorted ranges) is not tested, but results in runtime bugs [20]. A different kind of stickler is that if we have an iterator object that pointed to an element in a container, but the element is erased or the container's memory allocation has been changed, then the iterator becomes *invalid* [17]. Further reference of using invalid iterators causes undefined behaviour [19].

Another common mistake is related to removing algorithms. The algorithms are container-independent, hence they do not know how to erase elements from a container, just relocate them to a specific part of the container, and we need to invoke a specific erase member function to remove the elements phisically. Since, for example the `remove` algorithm does not actually remove any element from a container [13].

Some of the properties are checked at compilation time [8]. For example, the code does not compile if one uses sort algorithm with the standard list container, because the list's iterators do not offer random accessibility [10]. Other properties are checked at runtime [21], like the standard vector container offers an `at` method which tests if the index is valid and it raises an exception otherwise [18].

Unfortunately, there are still a large number of properties that are tested neither at compilation-time nor at run-time. The observance of these properties is in the charge of the programmers [6]. On the other hand, type systems can provide a high degree of safety at low operational costs. As part of the compiler, they discover many semantic errors very efficiently.

Associative containers (e.g. `multiset`) use functors exclusively to keep their

elements sorted. Algorithms for sorting (e.g. `stable_sort`) and searching in ordered ranges (e.g. `lower_bound`) are typically used with functors because of efficiency. These containers and algorithms need *strict weak ordering*. Containers become inconsistent if used functors do not meet the requirement of strict weak ordering [15].

Certain containers have member functions with the same names as STL algorithms. This phenomenon has many different reasons, for instance efficiency, safety or avoidance of compilation errors. For example, as mentioned before list's iterators cannot be passed to `sort` algorithm, hence code with this mistake cannot be compiled [24]. To overcome this problem list has a member function called `sort`. In these cases, although the code compiles, the member function calls are preferable to the usage of generic algorithms.

Whereas C++ STL is pre-eminent in a sequential realm, it is not aware of multicore environment [3]. For example, the Cilk++ language aims at multicore programming. This language extends C++ with new keywords and one can write programs for multicore architectures easily. Although the language does not contain an efficient multicore library, just the C++ STL only which is an efficiency bottleneck in multicore environment. We develop a new STL implementation for Cilk++ to cope with the challenges of multicore architectures[25]. This new implementation can be safer solution, too. Hence, our safety extensions will be included in the new implementation. However, the advised techniques presented in this paper concern to the original C++ STL, too.

In this paper we argue for an approach that generates warnings or errors when a template container is instantiated with improper parameters. These instantiations mean erroneous, unportable code or other weird compilation effects. A general technique is presented to express custom warnings at compilation time. Our technique is able to indicate the usage of deprecated classes.

This paper is organized as follows. In Section 2 we present an approach to generate "customized" warnings at compilation time. After, in Section 3 we describe the specialized vector container which contains boolean values. We show why this container is problematic, and argue for warnings when it is in use. We explain the forbidden *containers of auto pointers* and present an approach to disable their usage by template specializations. In Section 5 the so-called *believe-me marks* are introduced. Finally, this paper concludes in Section 7.

2 Generation of warnings

Compilers cannot emit warnings based on the erroneous usage of the library. STLlint is the flagship example for external software that is able to emit warnings when the STL is used in an incorrect way [9]. We do not want to modify the compilers, so we have to enforce the compiler to indicate these kinds of potential problems. However, `static_assert` as a new keyword is introduced in C++0x to emit compilation errors based on conditions, but no similar construct is designed for warnings.

```
template <class T>
inline void warning( T t ) { }

struct VECTOR_BOOL_IS_IN_USE { };

// ...

warning( VECTOR_BOOL_IS_IN_USE() );
```

When the `warning` function is called, a dummy object is passed. This dummy object is not used inside the function template, hence this is an unused parameter. Compilers emit warning to indicate unused parameters. Compilation of `warning` function template results in warning messages, when it is referred and instantiated [16]. No warning message is shown if it is not referred. In the warning message the template argument is referred. New dummy type has to be written for every new kind of warning.

Different compilers emit this warning in different ways. For instance, Visual Studio emits the following message:

```
warning C4100: 't' : unreferenced formal parameter
...
see reference to function template instantiation 'void
warning<VECTOR_BOOL_IS_IN_USE>(T)'
being compiled

        with
        [
            T=VECTOR_BOOL_IS_IN_USE
        ]
```

And g++ emits the following message:

```
In instantiation of 'void warning(T)
      [with T = VECTOR_BOOL_IS_IN_USE]':
... instantiated from here
... warning: unused parameter 't'
```

Unfortunately, implementation details of warnings may differ, thus there is no universal solution to generate custom warnings.

This approach of warning generation has no runtime overhead inasmuch as the compiler optimizes the empty function body. On the other hand—as the previous examples show—the message refers to the warning of unused parameter, incidentally the identifier of the template argument type is appeared in the message.

3 The weirdest vector

In this section we present the basic idea behind the specialized `vector<bool>` container. We present the pros and cons of this weird type. We argue for generate warnings at compilation-time if a programmer uses `vector<bool>` because it is the embodiment of the weird container.

Many programmers think that the `vector<bool>` is the instantiation of STL's `vector` template, but it is not true. On many platforms `sizeof( int ) == sizeof( bool )` because of reverse compatibility. (In the C programming language `int` type has been used to represent Boolean values.) Hence, the `vector<bool>` is a template specialized container to develop a more advanced, denser implementation for boolean values. This representation is able to represent 32 boolean values on 4 bytes.

The following code sketch represents the connection between `vector<bool>` and `vector` template:

```
template <class T, class Alloc = std::alloc>
class vector
{
  T* p;
  size_t capacity;
  size_t size;
public:
  vector()
```

```
  {
    // ...
  }

  void push_back( const T& t )
  {
    // ...
  }

  // ...
};

template <class Alloc>
class vector<bool, class Alloc>
{
  // dense representation of vector bool
  // No bool* member
public:
  // public interface is similar to the previous one

  void push_back( const bool& t )
  {
    // ...
  }

  vector()
  {
    //...
  }
};
```

So, the `vector<bool>` has a special representation to handle dense boolean values. It is designed to be effective when someone stores boolean values. But it has weird behaviour compared to the `vector` template:

```
std::vector<int> a;
a.push_back( 3 );
int* p = &a[0];
```

```
std::vector<bool> b;
b.push_back( true );
bool* q = &b[0];
```

The previous code does not compile because of the `bool* q = &b[0];` assignment. However, when the template `vector` is in use, its counterpart does compile. It is a contradiction in terms, because this way the `vector<bool>` cannot meet the requirements of C++ Standard. Hence, it is not advised to use. Let us see the background of this compilation issue:

```
template <class T, class Alloc = std::alloc>
class vector
{
  T* p;
  //..
public:
  T& operator[]( int idx )
  {
    return p[idx];
  }

  const T& operator[]( int idx ) const
  {
    return p[idx];
  }
  // ...
};

template <class Alloc>
class vector<bool, class Alloc>
{
  // dense representation of vector bool
  // No bool* member
public:

  class bool_reference
  {
     // ...
  };
```

```
  bool_reference operator[]( int idx )
  {
     // ...
  }
};
```

Because the `vector<bool>` does not hold actual bool values it cannot return `bool&`. Hence, a proxy class is developed which actually simulates `bool&`. However, conversions cannot be defined between *pointer to a* `bool_reference` and a *pointer to a* `bool`. This behaviour can be much more appalling, when the programmer uses `vector` as a base class. Arcane error messages are emitted when the subtype is instantiated with `bool`.

Unfortunately, most of STL references hardly mention that `vector<bool>` is not the instantiation of template, but a completely different class. It would be useful if the compiler indicated if the programmer used `vector<bool>` container, even intentionally or inadvertently.

Now it is not difficult to emit warning with the presented function. Fortunately, `vector<bool>` is still a class template because the type of its allocator is a template parameter. So, the compilation warning is emitted only when this template class is instantiated, hence someone uses it:

```
template<class Allocator>
class vector<bool, Allocator>
{
  // ...
public:
  vector()
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

  template<class InputIterator>
  vector( InputIterator first, InputIterator last )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }
```

```
  vector( size_t n, const bool& value = bool() )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

  vector( const vector& rhs)
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
    // ...
  }

};
```

In Section 2 the emitted warning message can be seen.

4 Containers of auto pointers

In this section the containers of auto pointers are detailed. We present their motivation and reason why are they problematic. We present a solution to forbid the usage of these kinds of containers.

Usually, auto pointers (`std::auto_ptr` objects) make easier to manage objects in the heap memory. This class assists in memory management. The auto pointers deallocate the pointed memory when they are gone out of scope [23]. Hence, they prevent memory leaks:

```
void f()
{
  std::auto_ptr<int> p( new int( 5 ) );
  // no memory leak
}
```

Containers of STL are template classes, so technically they should be instantiated with auto pointers and store auto pointers that point to the heap:

```
std::vector<std::auto_ptr<int> > v;
v.push_back( new int( 7 ) );
// ...
```

The previous code snippet seems to be safe. On the other hand, the C++ Standardization Committee forbid the usage of *containers of auto pointers (COAPs)*. The motivation behind this idea is that the copy of auto pointers is strange:

```
std::auto_ptr<int> p( new int( 3 ) );
std::auto_ptr<int> q = p;
// At this point p is null pointer
```

The copied auto pointer becomes null pointer. Only one auto pointer is able to point to any object in the heap. This one is responsible for the deallocation.

So, if COAPs are not be forbidden, the following code snippet results in a very strange behaviour:

```
struct Auto_ptr_less
{
  bool operator()( const std::auto_ptr<int>& a,
                   const std::auto_ptr<int>& b )
  {
    return *a < *b;
  }
};

std::vector<std::auto_ptr<int> > v;
v.push_back( new int( 7 ) );
// ...
std::sort( v.begin(), v.end(), Auto_ptr_less() );
```

Some of the pointers may become null pointer because of the assignments during swapping vector's elements when it is necessary. This is the reason why COAPs are forbidden.

Unfortunately, some of the compilers and STL platforms are still permitting the usage of COAPs, some of them are not. This inhibits the writing of portable code [13].

We argue for an extension to emit compilation error if COAPs are in use. We have to create specializations for auto pointers. The trick that is we do not write the implementation for the auto pointer specializations. Thus, these specializations are declared, but are not defined types. For instance, the vector declaration can be the following:

```
template <class T, class Alloc>
class vector< std::auto_ptr<T>, Alloc>;
```

The instantiation of a COAP results in the hereinafter error message:

```
error: aggregate 'std::vector<std::auto_ptr<int>,
  std::allocator<std::auto_ptr<int> > > v'
  has incomplete type and cannot be defined
```

We have to develop these declarations for all standard containers. These declarations mean bugless and more portable code.

5 Believe-me marks

Generally, warnings should be eliminated. On the other hand, the usage of `vector<bool>` does not necessarily mean a problem. It can be used safely. However, we cannot disable the generated warning if it is in use.

Believe-me marks [12] are used to identify the points in the program text where the type system cannot obtain if the used construct is risky. For instance, in the hereinafter example, the user of the library asks the type system to "believe" that the programmer is conscious of the specialized vector container. This way we enforce the user to reason about the parameters of containers.

First, we create a new type which stands for the believe-me mark:

```
struct I_KNOW_VECTOR_BOOL { };
```

After, we extend the vector template container with one new template parameter. The new template parameter has default parameter value, so it is reverse compatible with the original container. This parameter has not been taken advantage of, and has no effect on the implementation:

```
template <class T, class Alloc = std::alloc, class Info = int>
class vector { };
```

Let us consider, that the original implementation of `vector<bool>` which does not generate warning has been removed to a new template class:

```
template <class Alloc>
class __VectorBool
{
  // original implementation of vector<bool>
};
```

The new template parameter has effect on the `vector<bool>` specialization:

```
template <class Alloc>
class vector<bool, I_KNOW_VECTOR_BOOL, Alloc>:
  public __VectorBool<Alloc>
{ };

template <class Alloc>
class vector<bool, Alloc, I_KNOW_VECTOR_BOOL>:
  public __VectorBool<Alloc> { };

template <class Alloc, class Info>
class vector<bool, Alloc, Info>:
  public __VectorBool<Alloc>
{
public:
  vector(): __VectorBool<Alloc>()
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  template<class InputIterator>
  vector( InputIterator first, InputIterator last ):
    __VectorBool<Alloc>( first, last )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  vector( size_t n, const bool& value = bool() ):
    __VectorBool<Alloc>( n, value )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }

  vector( const vector& rhs): __VectorBool<Alloc>( rhs )
  {
    warning( VECTOR_BOOL_IS_IN_USE() );
  }
};
```

In this case no compilation warning is emitted if the last added template parameter is `I_KNOW_VECTOR_BOOL`, otherwise the mentioned warning can be seen during compilation.

6 Deprecated classes

In section 3 we generated warnings when a template-specialized class was used. A similar idea can be mentioned. It would be useful to generate warnings when the usage of classes becomes unsupported.

A common idea during a software lifecycle is, that some of the classes are not deleted from the project, but their usage is not advised. These classes are called *deprecated*. Deprecated annotation can be added to classes in Java. Instantiation of deprecated classes results in compilation warnings [11]. However, no similar technique is used in C++.

First, we create some utility classes for warning generation:

```
struct DeprecatedClass { };

template <class DEPRECATED>
struct Deprecated
{
  Deprecated()
  {
    warning( DeprecatedClass() );
  }
};
```

The role of the template parameter in `Deprecated` struct is to pass the identifier of deprecated class to the emitted warning.

Now, let us consider that the following class becomes deprecated during software lifecycle:

```
class Foo
{
  // ...
public:
  Foo( int a, int b)
  {
    // ...
  }
};
```

The user has to add one more base class to the deprecated class. This does not mean limitation because the C++ programming language supports multiple inheritance. For example:

```
class Foo: public Deprecated<Foo>
{
  // very same...
};
```

The following warning is received from the compiler:

```
In instantiation of 'void warning(T)
  [with T = DeprecatedClass]':
...   instantiated from
  'Deprecated<DEPRECATED>::Deprecated()
  [with DEPRECATED = Foo]'
...   instantiated from here
... warning: unused parameter 't'
```

However, this message is received irrespectively of its usage. If the usage is important, the deprecated class or a called method or constructor must be a template. This transformation cannot be executed automatically with the respect of client code. Our future work is to overcome this situation.

We do not advise to make believe-me marks for the deprecated classes inasmuch as always exists a better approach to use.

7 Conclusions

C++ STL is the most widely-used library based on the generic programming paradigm. It is efficient and convenient, but the incorrect usage of the library results in weird or undefined behaviour.

In this paper we argue for some extensions to make the STL itself safer. Not supported or not advised instantiations result in compilation warnings and errors to prevent unportable or defective code.

We present an effective approach to generate custom warnings. Believe-me marks are also written to disable warning messages. With our technique classes can be marked deprecated, too.

Acknowledgements

The Project 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] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, Reading, MA, 2001. ⇒142

[2] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, Reading, MA, 1998. ⇒141

[3] M. H. Austern, R. A. Towle, A. A. Stepanov, Range partition adaptors: a mechanism for parallelizing STL, *ACM SIGAPP Applied Computing Review* **4,** 1 (1996) 5–6. ⇒143

[4] T. Becker, STL & generic programming: writing your own iterators, *C/C++ Users Journal* **19,** 8 (2001) 51–57. ⇒142

[5] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, Reading, MA, 2000. ⇒142

[6] G. Dévai, N. Pataki, Towards verified usage of the C++ Standard Template Library, *Proc. 10th Symposium on Programming Languages and Software Tools (SPLST) 2007*, Dobogókő, Hungary, pp. 360–371. ⇒142

[7] G. Dévai, N. Pataki, A tool for formally specifying the C++ Standard Template Library, *Ann. Univ. Sci. Budapest. Comput.* **31** (2009) 147–166. ⇒142

[8] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, *Proc. 21st Annual ACM SIGPLAN 2006 Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, Portland, Oregon, USA, pp. 291–310. ⇒142

[9] D. Gregor, S. Schupp, Stllint: lifting static checking from languages to libraries, *Software – Practice & Experience*, **36,** 3 (2006) 225-254. ⇒144

[10] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: challenges of constrained generics in C++, *Proc. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, Ottawa, Canada, pp. 272–282. ⇒142

[11] Z. Juhász, M. Juhás, L. Samuelis, Cs. Szabó, Teaching Java programming using case studies, *Teaching Mathematics and Computer Science* **6,** 2 (2008) 245–256. ⇒153

[12] T. Kozsik, Tutorial on subtype marks, in *Proc. Central European Functional Programming School (CEFP 2005), Lecture Notes in Comput. Sci.* **4164** (2006) 191–222. ⇒151

[13] S. Meyers, *Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, Reading, MA, 2001. ⇒142, 150

[14] D. R. Musser, A. A. Stepanov, Generic programming, *Proc. International Symposium ISSAC'88 on Symbolic and Algebraic Computation, Lecture Notes in Comput. Sci.* **358** (1989) 13–25. ⇒142

[15] N. Pataki: C++ Standard Template Library by safe functors, *Abstracts 8th Joint Conference on Mathematics and Computer Science (MaCS'10)*, Komárno, Slovakia, July 14-17, 2010, p. 44. ⇒143

[16] N. Pataki, Advanced functor framework for C++ Standard Template Library, *Stud. Univ. Babeş-Bolyai Inform.* **56,** 1 (2011) 99–113. ⇒144

[17] N. Pataki, C++ Standard Template Library by ranges, *Proc. 8th International Conference on Applied Informatics (ICAI 2010)* Vol. 2., pp. 367–374. ⇒142

[18] N. Pataki, Z. Porkoláb, Z. Istenes, Towards soundness examination of the C++ Standard Template Library, *Proc. Electronic Computers and Informatics (ECI 2006)*, pp. 186–191. ⇒142

[19] N. Pataki, Z. Szűgyi, G. Dévai, C++ Standard Template Library in a safer way, *Proc. Workshop on Generative Technologies 2010 (WGT 2010)*, Paphos, Cyprus, pp. 46–55. ⇒142

[20] N. Pataki, Z. Szűgyi, G. Dévai, Measuring the overhead of C++ Standard Template Library safe variants, *Electronic Notes in Theoret. Comput. Sci.* **264,** 5 (2011) 71–83. ⇒142

[21] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Runtime concepts for the C++ Standard Template Library, *Proc. ACM Symposium on Applied Computing 2008*, Fortaleza, Ceará, Brazil, pp. 171–177. ⇒142

[22] Z. Porkoláb, Á. Sipos, N. Pataki, Inconsistencies of metrics in C++ Standard Template Library, *Proc. 11th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, Berlin, Germany, pp. 2–6. ⇒142

[23] B. Stroustrup, *The C++ Programming Language (Special edition)*, Addison-Wesley, Reading, MA, 2000. ⇒141, 149

[24] Z. Szűgyi, Á. Sipos, Z. Porkoláb, Towards the modularization of C++ concept maps, *Proc. Workshop on Generative Programming (WGT 2008)*, Budapest, Hungary, pp. 33–43. ⇒143

[25] Z. Szűgyi, M. Török, N. Pataki, Towards a multicore C++ Standard Template Libary, *Proc. Workshop on Generative Programming (WGT 2011)*, Saarbrücken, Germany, pp. 38–48. ⇒143

[26] M. Torgersen, The expression problem revisited – Four new solutions using generics, *Proc. European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Comput. Sci.* **3086** (2004) 123–143. ⇒142

[27] L. Zolman: An STL message decryptor for visual C++, *C/C++ Users Journal*, **19(7)** (2001) 24–30. ⇒142

[28] I. Zólyomi, Z. Porkoláb: Towards a general template introspection library, *Proc. of Generative Programming and Component Engineering: Third International Conference (GPCE 2004), Lecture Notes in Comput. Sci.* **3286**, 266–282. ⇒142

Received: March 16, 2011 • Revised: July 20, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 158–171

Large primes in generalized Pascal triangles

Gábor FARKAS
Eötvös Loránd University
email: farkasg@compalg.inf.elte.hu

Gábor KALLÓS
Széchenyi István University, Győr
email: kallos@sze.hu

Gyöngyvér KISS
Eötvös Loránd University
email: kissgyongyver@gmail.com

Abstract. In this paper, after presenting the results of the generalization of Pascal triangle (using powers of base numbers), we examine some properties of the 112-based triangle, most of all regarding to prime numbers. Additionally, an effective implementation of ECPP method is presented which enables Magma computer algebra system to prove the primality of numbers with more than 1000 decimal digits.

1 Generalized Pascal triangles using the powers of base numbers

As it is a well-known fact, the classic Pascal triangle has served as a model for various generalizations. Among the broad variety of ideas of generalizations we can find e.g.: the generalized binomial coefficients of s^{th} order (leading to generalized Pascal triangles of s^{th} order), the multinomial coefficients (leading to Pascal pyramids and hyperpyramids), special arithmetical sequences (leading to resulting triangles which we might call as Lucas, Fibonacci, Gaussian, Catalan, ... triangle) (details in [3]).

One of the present authors has devised, and then worked out in detail and published such a type of generalization, which is based on the idea of using

Computing Classification System 1998: G.4
Mathematics Subject Classification 2010: 05A10, 11Y11
Key words and phrases: generalized Pascal triangles, elliptic curve primality proving

"the powers of the base number". Referring to our former results (presented in detail in [7] and [8]; here we don't repeat/echo the theorems and propositions) we show here the first few rows of the 112-based triangle (Figure 1), which will gain outstanding importance below in this paper.

						1						
					1	1	2					
				1	2	5	4	4				
			1	3	9	13	18	12	8			
		1	4	14	28	49	56	56	32	16		
	1	5	20	50	105	161	210	200	160	80	32	
1	6	27	80	195	366	581	732	780	640	432	192	64
						...						

Figure 1: The 112-based triangle

Let us use the notation $E_{k,n}^{a_0a_1\dots a_{m-1}}$ for the k^{th} element in the n^{th} row of $a_0a_1\dots a_{m-1}$-based triangle ($0 \le a_0, a_1, \dots, a_{m-1} \le 9$ are integers). Then we have the definition rule, as follows:

$$E_{k,n}^{a_0a_1\dots a_{m-1}} = a_{m-1}E_{k-m+1,n-1}^{a_0a_1\dots a_{m-1}} + a_{m-2}E_{k-m+2,n-1}^{a_0a_1\dots a_{m-1}} + \dots +$$

$$+a_1E_{k-1,n-1}^{a_0a_1\dots a_{m-1}} + a_0E_{k,n-1}^{a_0a_1\dots a_{m-1}}.$$

The indices in the rows and columns run from 0, elements with non-existing indices are considered to be zero. Applying this general form to the 112-based triangle (now: $m = 3$), we get the specific rule

$$E_{k,n}^{112} = 2E_{k-2,n-1}^{112} + E_{k-1,n-1}^{112} + E_{k,n-1}^{112}.$$

The historical overview of this special field is presented in [8]. In the last few years there were published several new results which are related to our topic (e.g. [2]). Moreover, besides that, up to about 2005, all generalized triangle sequences of the type $ax + by$ were added to the database On-line Encyclopedia of Integer sequences [11], since that time there have been several new applications, too, based on sequences appearing in our triangles. However, e.g. the sequences based on the general abc-based triangles are still not widely known.

Recalling the basic properties of generalized triangles—most of all in connection with powering the base number $a_0a_1\dots a_{m-1}$ and with the polynomial

$(a_0x_0 + a_1x_1 + \cdots + a_{m-1}x_{m-1})^n$—we can state that we have the "right" to call these types of triangles as generalized Pascal triangles (details in [8], summary in [6]).

2 Divisibility of elements and prime numbers

The classic divisibility investigations in Pascal triangle (for binomial coefficients) are very popular and even spectacular, if the traditional "strict" mathematical approach is moved toward coloring and fractals (details in [3]). For generalized binomial coefficients (with our notation: in triangles with bases $11\cdots1$) we have similar results, too, with a remark that in these cases general proofs are harder, and there are many conjectures, too.

We recall here the beautiful result of Richard C. Bollinger, who proved for generalized Pascal triangles of p^{th} order that for large n, "almost every" element in the n^{th} row is divisible by p (see [3], p. 24). For example, for the 111-based triangle this means divisibility by 3. (We mention that the p^{th} order Pascal triangle is a triangle with base $11\cdots1$, where we have p pieces of 1.)

Now we turn our attention specially to the 112-based triangle, and in the following we are interested mostly in prime numbers. It is obvious that the right part of the triangle contains only even numbers. Moreover, if we move to the right, the powers of 2 are usually (not always) growing as divisors. Analyzing connections with the multinomial theorem we can conclude that the left part of the triangle contains mostly (with possible exception of the first two places) composite numbers, too. Of course, this can be not true for the 0^{th} and 1^{st} numbers, which are the same as in the classic Pascal triangle. Moreover, using induction we can see that the center element in every row is always odd.

We can pose obviously two (not hard) questions in connection with prime numbers:

1. Can we find every prime number as an element in our triangle?

2. Can we find every prime number as an element in our triangle in non-trivial places?

The answer to question 1 is "yes", as we already saw above (the 1^{st} elements in every row, however, this is a trivial match). To question 2, we fix first that primes are worth looking for only in the middle position.

With a computer investigation (using e.g. the Maple program) we can find 6 small primes up to the 100^{th} row (Figure 2).

Extending the examination up to the 1900^{th} row, we get only one more

Position (row, column)	Prime
2, 2	5
3, 3	13
8, 8	7393
15, 15	65753693
21, 21	175669746209
24, 24	9232029156001

Figure 2: Small primes in the 112-based triangle

positive answer, in position $(156, 156)$, a 90-digit prime (candidate). So, the answer to our second question (considering only this triangle) is "no".

Our possibilities are extended rapidly, if we look up not only pure prime numbers, but even decompositions. So now we modify our question 2 as "can we find every prime number as a factor of any element in our triangle?" (Examining only non-trivial places, so, positions 0 and 1 are in every row excluded.)

We see immediately that every one-digit prime occurs as a factor at least once up to the 4^{th} row. Here 2 and 5 are triangle elements themselves; 3 is a factor of 9, 7 is a factor of 14.

Continuing with an easy computer examination for two-digit primes we find all but 4 up to the 12^{th} row. For the rest of the numbers we get the following first occurrences (in number–row form): 79–14, 71–15, 59–17 and, surprisingly 41–27.

Now, we turn our attention to 3-digit primes. Here we need a much larger triangle-part. Let's choose, say, a 100-row triangle in an easy-factorized form. With a small Maple program on a normal table-PC, we can generate the necessary data in a few minutes. (Easy factorization is very important here, otherwise, with full factorization the generation could take an extremely long time...) The output of the program in txt form will be approximately 1.15 MBytes.

From the 143 3-digit primes we find 105 up to 40^{th} row. For the remaining 38 numbers, 18 numbers are situated in rows $41-50$, 11 additional primes in rows $51-60$, and 2 (823 and 827) in rows $61-70$. The still missing "hardest" 3-digit primes finally give the following first occurrences (in number – row form): 479 – 74, 499 – 74, 677 – 76, 719 – 77, 859 – 72, 937 – 98 and 947 – 73. To the contrary, the "easiest" 3-digit primes are 103, 191 and 409 in the 7^{th} row.

With this we give up the claim "to find all of the primes as divisors".

Our next investigation focuses on very large prime factors (more accurately: prime candidates).

Computer investigations suggest that the largest prime factors in a given row occur very likely in the center position or very close to that place. Of course, this is not an absolute rule, but since our goal is "only" to find very large prime (candidate) factors, we can limit the investigation to the center element. (This has a significant importance to achieving: go as "deep" relatively quickly in the triangle as possible.)

Moreover, the center element carries special properties compared with other elements. Recalling Richard C. Bollinger's result above, we can set up a similar interesting conjecture:

For large n*, the center element in the* n^{th} *row "almost surely" will be divisible by* 5 *and* 7 *(but surely not by* 2 *and usually not by* 3*).*

So, with a relatively simple Maple program we set out to the easy-factorization of the center element up to the 1900^{th} row. On a normal table-PC, the execution time is approximately 11 hours, with an output file in txt form roughly 110 KBytes.

Analyzing the output we can deduce that prime divisors here follow the Knuth-observation [9], too: we usually find few small factors some of which are repetitive; composite (not decomposable with the 'easy' option) large factors are common, pure large prime factors are however rare or extremely rare.

Position (row, column)	Digits of the prime candidate
1726, 1726	1002
1793, 1793	1028
1794, 1794	1030

Figure 3: Large "pure" prime factors (candidates)—112-based triangle, center position

Considering only the primes (prime candidates) with digits more than 1000 we get 3 matches.

Here the second and third matches are especially interesting, since they can be considered as a special kind of "twin-primes" (candidates) in the triangle. In general, our chance to find "pure" large prime factors in consecutive rows is very little...

Here the factorization of element with position $1793, 1793$ is as follows:

$1793, 1793; "(5) * "(7)^2 * "(673) * "(65119) * "(1485703) * "(15578887875328$
$926423851777567602680378792003694589981499750631818308971422277975$
$902867850432471811687112334064063828539296067422531997963055491323$
$406425659317001574425151788919713654021679547897110675223861482309$
$644220358490739245691930715715021145166205571510978302005857149111$
$239471032734380710285002174983967604232152940389858538629493812650$
$108566716591594874813194189360195173091031608755605756723631900973$
$625032697091409833078265261680211635427069757196618031458397872466$
$034789488450265204214587550269112317436588892430166513888148357222$
$480962630168478230243146450158020142586939406221546644931686618139$
$068737541801842683626194613956159330873776421795220707554672321055$
$658602305273678940456712151943459348907356567358277310497505925970$
$210070347980231047308886323693790450859256057748541430119354204022$
$527748661261790305800487349106563678280226712828838174678186252307$
$070941149885645163684441661612796581751766644659424590726902531393$
$104098376100305217952214533052008783687240950373043230661705142861$
$901235736247002277563333)$

In [6] we proved the primality of the largest factor of $1726, 1726$ which has 1002 decimal digits. That time we used a freeware software developed by F. Morain. In the remaining part of this paper our goal is to present our selfmade program which is appropriate to prove the prime property of such large numbers. Let us denote the 1028 digits long factor of $1793, 1793$ by n_1 and the 1030 digits long factor of $1794, 1794$ by n_2. We investigated n_1 and n_2 with our program, and have found that both of them are really primes. Moreover, the process of the proof and shematic structure of the evidence will be presented, too.

3 Atkin's primality test

We described the theoretical foundations of the elliptic curve primality proving in [6]. Unfortunately, most computer algebra systems include just probability primality test, so we can not use them to reach our purposes. Although the Magma system (described below) is able to carry out primality proving with ECPP (Elliptic Curve Primality Proving), we did not get any result even after two days running for n_2. Thus we have developed an own primality proving program presented in the next section.

According to the notation of [6] let us denote an elliptic curve over $\mathbb{Z}/n\mathbb{Z}$ by E_n. The first step in the basic ECPP algorithm is choosing randomly an

E_n elliptic curve, the second one is counting $|E_n|$, the order of E_n. The latter action is very time-consuming, so we had to find an improved version of ECPP. Finally we have implemented an algorithm suggested by A. O. L. Atkin. A specification of this method can be found in [1]. Lenstra and Lenstra published a heuristic running time analysis of Atkin's elliptic curve primality proving algorithm in [10]. They conjectured that with fast arithmetic methods the running time of ECPP can be reduced to $O(\ln^{4+\epsilon}(n))$.

Atkin brilliant idea was founding an appropriate m order in advance and then constructing E_n for this m avoiding the order-counting. Moreover, we get simultaneously two elliptic curves increasing the chance of the successful running of the test. m order has to be chosen from the algebraic integer of an imaginary quadratic field $\mathbb{Q}(\sqrt{D})$. An appropriate D, so-called *fundamental discriminant*, has some properties: $D \equiv 0 \pmod 4$, or $D \equiv 1 \pmod 4$, for every $k(>1)$ D/k^2 is not a fundamental discriminant, $D \leq -7$ and $(D|n) = 1$, where $(D|n)$ is the Jacobi symbol.

The function NEXTD() gives a value D which meets the above mentioned requirements. A given D value is suitable if there exist such $x, y \in \mathbb{Z}$ for which

$$4n = (2x + yD)^2 - y^2D. \tag{1}$$

In that case we get two possible orders: $m = |\nu \pm 1|^2$, where

$$\nu = x + y\frac{D + \sqrt{D}}{2}.$$

If (1) is valid, then we can compute an x_0 root of the *Hilbert polynomial* (mod n). The function HILBERT(n, D) returns with a root of the appropriate Hilbert polynomial. Then we get two elliptic curves with order $m = |\nu \pm 1|^2$. The rest of the algorithm works as we described in [6].

PROOF(E_n, m, f)

1 $P \leftarrow$ RANDOMPOINT(E_n)
2 **if** $f \cdot P$ is not defined
3 **then return** COMPOSITE
4 **if** $f \cdot P = O$
5 **then goto** 1
6 **if** $mP \neq O$
7 **then return** NO
8 **return** YES

Here symbol O means the "point infinitely far" e.g. the unit of the Abelian group. The function PROOF() has three input values: E_n, m, f, where E_n is an elliptic curve with order m, $m = f \cdot s$, the factorization of f is known and s is probably prime. The output value COMPOSITE means that n is surely composite. If the output is NO, then n is composite or we have to choose the other elliptic curve. In case YES the next recursion step follows. In the following we present the pseudocode of the Atkin's test.

ATKIN-PRIMALITY-TEST(n)

1 $D \leftarrow$ NEXTD()
2 $\omega \leftarrow (D + \sqrt{D})/2$
3 **if** $\exists\, x, y \in \mathbb{Z} : 4n = (2x + yD)^2 - y^2D$
4 **then** $\nu \leftarrow x + y\omega$
5 **else** **goto** 1
6 $m \leftarrow |\nu + 1|^2$
7 **if** $m = f \cdot s$, where s "probably prime" and $s > \left(\sqrt[4]{n} + 1\right)^2$
8 **then goto** 12
9 $m \leftarrow |\nu - 1|^2$
10 **if** $m = f \cdot s$ can not be produced so that s is "probably prime" and $s > \left(\sqrt[4]{n} + 1\right)^2$
11 **then goto** 1
12 $x_0 \leftarrow$ HILBERT(n, D)
13 $c \leftarrow$ arbitrary integer for which $(c/n) = -1$
14 $k \leftarrow$ arbitrary integer for which $k \equiv x_0/(1728 - x_0) \pmod{n}$
15 $E_n \leftarrow \{(x, y) \mid y^2 = x^3 + 3kx + 2k\}$
16 **if** PROOF(E_n, m, f) = COMPOSITE
17 **then return** COMPOSITE
18 **else if** PROOF(E_n, m, f) = YES
19 **then goto** 23
20 $E_n \leftarrow \{(x, y) \mid y^2 = x^3 + 3kc^2x + 2kc^3\}$
21 **if** PROOF(E_n, m, f) = COMPOSITE or PROOF(E_n, m, f) = NO
22 **then return** COMPOSITE
23 **if** s surely prime
24 **then return** PRIME
25 **else** ATKIN-PRIMALITY-TEST(s)

4 Magma Computer Algebra System

Magma [5] is a large software system specialized in high-performance computations in number theory, group theory, geometry, combinatorics and other branches of algebra. It was launched at the First Magma Conference on Computational Algebra held at Queen Mary and Westfield College, London, August 1993. It contains a large body of intrinsic functions (implemented in C language), but also allows the user to implement functions on top of this, making use of the Pascal-like user language and the programming environment that is provided.

4.1 Primality tests in Magma

Magma has several built-in functions for primality testing purposes.

`IsProbablyPrime(n:` *parameter*`) : RngIntElt` $\mapsto$ `BoolElt`

The function returns TRUE if and only if `n` is a probable prime. This function uses the Miller-Rabin test; setting the optional integer parameter `Bases` to some value B, the Miller-Rabin test will use B bases while testing compositeness. The default value is 20. This function will never declare a prime number composite, but with very small probability (much smaller than 2^{-B}, and by default less than 10^{-6}) it may fail to find a witness for compositeness, and declare a composite number probably prime.

`IsPrime(n:` *parameter*`) : RngIntElt` $\mapsto$ `BoolElt`

This function proves primality using ECPP which is of course more time-consuming. It is possible though to set the optional Boolean parameter `Proof` to FALSE; in which case the function uses the probabilistic Miller-Rabin test, with the default number of bases.

`PrimalityCertificate(n:` *parameter*`) : RngIntElt` $\mapsto$ `List`

This function proves primality and provides a certificate for it using ECPP. If the number `n` is proven to be composite or the test fails, a runtime error occurs.

`IsPrimeCertificate(c:` *parameter*`) : List` $\mapsto$ BoolElt

To verify primality from a given certificate `c` this function is used. This returns the result of the verification by default, a more detailed outcome can be obtained by setting the optional Boolean parameter `ShowCertificate` to TRUE.

The numbers n_1 and n_2 were tested with Magma's own ECPP, using the intrinsic `IsPrime` function, and with our ECPP implementation written in Magma language. We refer to Magma's ECPP algorithm as Magma-ECPP and to our implementation as modified-ECPP. Both tests were running in Magma 2.16 on a machine with 7425 MB RAM and four 2400 MHz Dual-Core AMD Opteron (TM) Processors.

The Magma-ECPP provided a primality proof for n_1 in 32763.52 seconds, but seemed to stuck after the third iteration during the test of n_2; the modified-ECPP provided proof for n_1 in 5666.96 seconds and for n_2 in 5153.37 seconds. As the modified-ECPP is not finished yet, the running time can still be improved.

4.2 The implementation of ECPP algorithm

The ECPP algorithm consists of iteration steps, where the i^{th} iteration step outputs an s_i which will be the input of the next iteration step. In one iteration step an attempt is made to factor order m_i of the group of points on a curve E_i. Curve E_i is defined using the input s_{i-1} and a discriminant of an imaginary quadratic field, read in from a list.

If the attempt is successful, factor s_i is the output; if not, we need to backtrack. A different discriminant in an iteration step results in a different s_i. The possible iteration chains that occur this way, can be represented as paths in a directed graph $G(n)$. The nodes of $G(n)$ are the s_i's, the root represents n, the edges are the iteration steps. An edge leads from s_i to s_{i+1} if there is an iteration that produces s_{i+1} with input s_i. Consider a path successful if the corresponding iteration-chain starts with input n and ends with input s_l, where s_l is a small prime, which can be verified by easy inspection, or trial division. In the rest of the paper we refer to the s_i's also as nodes.

Magma-ECPP uses a small fixed set of discriminants during the process. Each iteration goes through this set until it finds a discriminant which produces a new node. Using a small set of discriminants makes the algorithm faster, but increases the probability of producing no new node. If no discriminant produces new node in the set it backtracks to the previous node and retries that with the same set of discriminants but possibly stronger factorization methods to factor the m_i's. If backtracking does not produce a new node, it will try to factor again with more effort; these hard factorizations may

consume a large amount of time, and the process appears to get stuck in a seemingly endless loop. This happened during the test of our number n_2 with Magma-ECPP.

4.2.1 Modifications

During the iteration steps certain limits are used; for example, the bound B on the primes found in factoring the m_i-s. Imposing a small B decreases the difference between the size of the s_i-s and thus may extend the path down to the small primes. On the other hand, setting a large B significantly increases the running time needed for factoring. Of course, choosing a more sophisticated factoring method smoothes the differences in running time, but the size of B still remains an important factor. Other important limits are the bound D on the discriminants and the limit S on the prime factors of the discriminants. Decreasing them leads to speed improvement but to a smaller set of discriminants, too.

The modified-ECPP uses a huge file which contains a list of fully factored discriminants up to 10^9. During the selection of discriminants useful for the current input we extract a modular square root of its prime divisors and build up the square root of the discriminant by multiplication. After using one prime, the square root is stored, and thus it will be computed only once in an iteration step. The speed that we gain this way makes it possible to increase limits D, S in the iterations, which are adjusted to the size of the current input.

The steps can be extended to result in a *series* of s_i-s at a time instead of just a single one: if the iteration step does not stop at the first good discriminant but will collect several good ones. This way, we can select the input of the next step from a set of new nodes.

The numbers have individual properties, which makes a difference from the point of usability. The modified-ECPP predicts the minimal value of D which is still enough to produce at least one new node for each s_i produced by earlier steps and, building upon this prediction, sets up a priority between them. It selects the one with the highest priority as input for the next iteration step. If the step does not provide output the limit D will be increased in order to use a new set of discriminants next time when the node is selected. The priority is reevaluated after each step because either there are new nodes or in case of no output D is increased. This way the possibility of getting stuck is lower (details can be found in [4]).

Table 1: The first rows of the proof of $\mathfrak{n}_1$

i	s_i	a_i	b_i	x_i	y_i	f_i
1	165490139	148629518369919	154064198784106	1248188509129	156779851067219	1047222
2	173304931274467	83727826741233492116021	26748895837956005585121	23717486180315890605060	11528948633455951893545	503211105
3	872089659685989674766 79	277074009572472999774111947197	384653752681961110410774819991	281425185963869506523604739355	206266449970544277927805954752	6877885
4	599813238900937331684100565579	44360713177559177572547682547718474	11117040242765996352 2375118328811693	19775111407066701253 6508097973496758	18428910703962901519 3733563259937266	408110
5	244789780927861535422029989890998131	18957568139328887813 3604105862968021099	10802087315828632193 95669478393483623141	69127839011266503431 9346422505776608550	35593578508679194394 5359772187301402486	4853
6	118796480684291203135905692901203343257 9	42083366473007987927 62804820140778821039 92386685393	2805557764867199195175203213427185880693 28257790262	18066828358351637326 30229471640748615572 09385839814	25827423550789508626 77731097240769328473 18909337972	486045540848
7	577404997050353029657199197365170395975202732115661	11380443233447151682 41036572010957871951 7294768226496	12206202132034258692 19783734272205524829 2902663774895	96784583503642168856 07646244567179724818 972920682916	59710222084606095711 75786070086466618975 533228021487	24
8	13857719929208472711772780587947008305844118454871693	41694582424591111764 43156453894775152544 45255867194500535484548	63775518208646414603 03125686908495336379 05480470784855197824527	34124747439640525997 72929688220778601290 76542652423518714847638	13433669109612418315 59617778582791580192 42180614563041511515824	53096907911
9	73580207893761171469 79557764345765335962 72772075969232836325203	43072369720090946264 13350749758127925795 34831485143675218313234009372	37871678106813894501 76380954770420227217 88363264525523890880507401845	70859759652434266069499533970811791751 13479474574680812743326336082	87503435487309350451056616952045997191182 08741123870099436602667558083	1271492
10	935566456952541794344733186090352580484140605445293133424460870453037	18992439425877599779 81729792883189473382 67763448710029974361 4358086171211009277	12661626283918399853 21153195255459648921 78508965806686649574 2905390780807339518	25459301193842798167 48588095939934589673 54516398347837548526 9028646284881593862	20234087314600234645 56960079052485667673 33324851231828927752 7747596369915016860	6762000366

4.3 The proof

On input n, a probable prime, the primality test results in a list, which provides sufficient data to prove the correctness of the sequence of the steps along the successful path. If we consider the length of the proof list as $\#L$, the i^{th} list element, as the proof runs in reverse order, starting from the smallest s_i, corresponds to the $\#L - i^{th}$ step in the sequence and consists of s_i, a_i, b_i, P_i, f_i, where $s_i f_i = m_i$ and s_i is a probable prime, the factorization of f_i is known, $y^2 = x^3 + a_i x + b_i$ is an elliptic curve of order m_i over $\mathbb{Z}/s_{i+1}\mathbb{Z}$, and P_i is a point on this curve that satisfies the condition $m_i P_i = 0$, $f_i P_i \neq 0$. P_i is given by its two coordinates x_i and y_i. The correctness proof guarantees recursively that all s_i are genuine primes, and eventually that the input n is prime.

Since the size of the above mentioned list is too large (approximately 809 KB in txt form), the exact details can not be presented in this paper. Instead of this, we give here only a small part of this file (see in Table 1). The full text can be downloaded from page: `http://compalg.inf.elte.hu/tanszek/farkasg/proof-tri.txt`

Acknowledgements

Prepared in the framework of application TÁMOP 4.1.1/A-10/1/KONV-2010-0005 with the support of Universitas-Győr Foundation (pages: 158–164); The Project 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) (pages: 165–171).

References

[1] A. O. L. Atkin, F. Morain, Elliptic curves and primality proving, *Math. Comp.* **61,** 203 (1993) 29–68. ⇒164

[2] H. Belbachir, S. Bouroubi, A. Khelladi, Connection between ordinary multinomials, Fibonacci numbers, Bell polynomials and discrete uniform distribution, *Ann. Math. Inform.* **35** (2008) 21–30. ⇒159

[3] B. A. Bondarenko, *Generalized Pascal Triangles and Pyramids, Their Fractals, Graphs and Applications*, The Fibonacci Association, Santa Clara, CA, USA, 1993. ⇒158, 160

[4] W. Bosma, A. Járai, Gy. Kiss, *Better paths for elliptic curve primality proofs*, http://www.math.ru.nl/~bosma/pubs/reportfinal.pdf, 2009. ⇒ 168

[5] W. Bosma, J. Cannon, C. Playoust, The Magma algebra system. I. The user language, *J. Symbolic Comput.*, **24**, 3-4 (1997) 235–265. ⇒166

[6] G. Farkas, G. Kallós, Prime numbers in generalized Pascal triangles, *Acta Tech. Jaur.* **1,** 1 (2008) 109-118. ⇒160, 163, 164

[7] G. Kallós, *A Pascal háromszög általánosításai* (in Hungarian), Master thesis, Eötvös Loránd University, Budapest, 1993. ⇒159

[8] G. Kallós, A generalization of Pascal's triangle using powers of base numbers, *Ann. Math. Blaise Pascal* **13,** 1 (2006) 1–15. ⇒159, 160

[9] D. E. Knuth, *The Art of Computer Programming, Vol. 2.,* Addison-Wesley, Reading, MA, USA, 1981. ⇒162

[10] A. K. Lenstra, H. W. Lenstra, Algorithms in number theory, in: *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity,* Ed. J. van Leeuwen, Elsevier Science Publisher, B.V., Amsterdam; MIT Press, Cambridge, MA, 1990, pp. 673–715. ⇒164

[11] *The On-line Encyclopedia of Integer Sequences*, Published electronically at http://oeis.org, 2011. ⇒159

Received: June 11, 2011 • Revised: September 14, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 172–191

Modular exponentiation of matrices on FPGA-s

Tamás HERENDI
University of Debrecen
email: herendi.tamas@inf.unideb.hu

Roland Sándor MAJOR
University of Debrecen
email: mroland@digikabel.hu

Abstract. We describe an efficient FPGA implementation for the exponentiation of large matrices. The research is related to an algorithm for constructing uniformly distributed linear recurring sequences. The design utilizes the special properties of both the FPGA and the used matrices to achieve a very significant speedup compared to traditional architectures.

1 Introduction

Field-programmable gate arrays (FPGA) offer a number of special options in computation. Utilizing the unique properties of an FPGA, some algorithms that are impractical to implement on a more traditional architecture can become both convenient to create and resource-efficient. The programmable array of look-up tables commonly found on an FPGA provide both flexibility in creating logic to suit specific needs and naturally lend themselves to great parallelism in computations.

Fast operations on matrices are of great practical interest. Ways to speed up certain matrix calculations still find their way into numerous applications.

Faster implementations of matrix algorithms can be achieved either from a "software" point of view, by improving upon the algorithm itself, or from a

Computing Classification System 1998: B.2.4
Mathematics Subject Classification 2010: 65F60 11Y55
Key words and phrases: pseudo random number generators, linear recurring sequences, uniform distribution, matrix exponentiation, parallel arithmetic, FPGA design, hardware acceleration of computations, hardware implementation of computations

"hardware" point of view, by using faster or differently structured architectures.

Theoretical improvements on matrix algorithms include Strassen's algorithm [12] and the Coppersmith-Winograd algorithm [2]. The naive algorithm for matrix multiplication is a well-known $\Theta(n^3)$ algorithm. Strassen's algorithm uses an idea similar to the Karatsuba-multiplication. It has a time complexity of $O(n^{\lg 7})$ by dividing the matrices into sub-matrices. Then by multiplying them in a different arrangement, it manages an overall lower multiplication count compared to the classical algorithm. Research implementing it on the Cell Broadband Engine can be found in [5]. Strassen's algorithm and its applicability to the project is briefly discussed in Section 7. The Coppersmith-Winograd algorithm further improves the complexity to $O(n^{2.376})$ by combining the idea of Strassen with the Salem-Spencer theorem. [9] discusses and compares the performance of implementations of these algorithms.

Numerous research has been done on creating efficient realizations of different matrix operations on different architectures. [8] and [10] both use FPGAs to perform matrix inversion.

The design presented here is an implementation of matrix multiplication on an FPGA. Works of similar nature can be found in [1] and [4], dealing with FPGA configurations used for floating point matrix multiplication. [11] uses an FPGA design for digital signal processing. [3] discusses another FPGA implementation for accelerating matrix multiplication.

The research in this paper is related to an algorithm for the construction of pseudo random number generators. It requires the exponentiation of large matrices to an extremely high power. This allows for numerous optimizations to be made on the FPGA implementation, resulting in an extremely fast design. A speedup factor of ~200 is achieved compared to a highly optimized program on a more traditional architecture.

We give the details of a design implemented on a Virtex-5 XC5VLX110T FPGA that multiplies two 896×896 sized matrices. The matrices are defined over the mod 4 residue class ring. Using this property and the fact that the hardware uses 6-LUTs (Lookup Tables), we describe first a module that computes the dot product of vectors taken from $\mathbb{Z}_4^{28}$ in a single clock cycle at 100MHz clock speed. With these modules we construct a matrix multiplier module that computes the $C \in \mathbb{Z}_4^{20\times 20}$ product matrix of $A \in \mathbb{Z}_4^{20\times 28d}$ and $B \in \mathbb{Z}_4^{28d\times 20}$ in d clock cycles at 100MHz. The significance of the value 28 in the implementation and its experimental determination is also discussed. Finally, we describe how to use these modules for multiplying matrices taken from $\mathbb{Z}_4^{896\times 896}$. The proposed algorithm deals with the management of stored

data in such a way that it can be accomplished completely in parallel with the computations. The resulting design completes the multiplication in 64800 clock cycles at 100MHz.

Future work for increasing the size of the used matrices, and further optimizing the design's performance using Strassen's algorithm is also described.

2 Mathematical background

The present work is initiated by a method for the construction of uniformly distributed pseudo random number generators. (See [7].) The generator uses recurring sequences modulo powers of 2 of the form

$$u_n \equiv a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \cdots + a_0u_{n-d} \mod 2^s,\ a_i \in \{0,1,2,3\}, s \in \mathbb{Z}^+$$

The theoretical background can be found in [6].

The construction assumes that the values $a_0, a_1, \ldots, a_{d-1}$ are such that

$$x^d - a_{d-1}x^{d-1} - \cdots - a_0 \equiv (x-1)^2 P(x) \mod 2$$

holds for some $P(x)$ irreducible polynomial. It is practical to choose $P(x)$ to have maximal order, since the order of P is closely related to the period length of the corresponding recurring sequence. The sequence u_n obtained this way does not necessarily have uniform distribution, however exactly one of the following four sequences does:

$$\begin{aligned}
u_n^{(0)} &\equiv a_{d-1}u_{n-1}^{(0)} + a_{d-2}u_{n-2}^{(0)} + \cdots + a_1u_{n-d+1}^{(0)} + a_0u_{n-d}^{(0)} \mod 2^s \\
u_n^{(1)} &\equiv a_{d-1}u_{n-1}^{(1)} + a_{d-2}u_{n-2}^{(1)} + \cdots + a_1u_{n-d+1}^{(1)} + (a_0+2)u_{n-d}^{(1)} \mod 2^s \\
u_n^{(2)} &\equiv a_{d-1}u_{n-1}^{(2)} + a_{d-2}u_{n-2}^{(2)} + \cdots + (a_1+2)u_{n-d+1}^{(2)} + a_0u_{n-d}^{(2)} \mod 2^s \\
u_n^{(3)} &\equiv a_{d-1}u_{n-1}^{(3)} + a_{d-2}u_{n-2}^{(3)} + \cdots + (a_1+2)u_{n-d+1}^{(3)} + (a_0+2)u_{n-d}^{(3)} \mod 2^s\ .
\end{aligned}$$

For the details see [7]. Finding the sequence with uniform distribution is of interest. Let

$$M(u) = \begin{pmatrix} 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \\ a_0 & a_1 & \ldots & a_{d-2} & a_{d-1} \end{pmatrix}$$

be the companion matrix of sequence u. To find which of the above sequences has a uniform distribution, we have to compute $M(u)^{2^{d+1}-2} \mod 4$. If $M(u)^{2^{d+1}-2} \mod 4$ equals the identity matrix, then the period length of u_n is $2^{d+1} - 2$, which means it is not the sequence we are searching for.

The exponentiation of matrices to high powers can quickly become time consuming on traditional computers. The aim of the project was to utilize the special properties of an FPGA to achieve a significant upgrade in speed compared to implementations on more traditional architectures.

3 Hardware used in the implementation

The project was implemented on a Xilinx XUPV505-LX110T development platform. The board features a variety of ports for communication with the device. As a first approach the RS-232 serial port was used to send data between the board and a PC. A high-speed PCI Express connection is also available if the amount of data transferred would necessitate its use.

The board's most prominent feature is the Virtex-5 XC5VLX110T FPGA. The FPGA's main tool for computation is the array of 6-input look-up tables, arranged into 17280 Slices, with four look-up tables found in each Slice, adding up to a total of 69120 LUTs. A single 6-input LUT can store 64 bits of data, where its six input bits are used as an address to identify the single bit of data that is to be outputted. By manipulating the 64 bit content of the look-up table, it can be configured to carry out arbitrary Boolean functions with at most six input bits. In our design they are used to create LUTs performing a multiply-accumulate function, which are hierarchically arranged into larger and more complex modules. One out of four LUTs on the device can also be used as a 32 bit deep shift register; these are the basis to implement containers storing the data, which is directly fed to the computational module.

Attached to the board, there is a 256MB DDR2 SODIMM module, which is used for storing data exceeding the amount that can be practically stored on the FPGA.

4 Structure of modules used in the computation

The basic elements of the design are the LUTs denoted by $L(a, b, s) = c$, where a, b, c and s are two-digit binary numbers. The function carried out by L is a multiply-accumulate (for short: MA) function, i.e.:

$$c \equiv (a \cdot b) + s \mod 4 \,.$$

Let $a = 2\alpha_1 + \alpha_0$, $b = 2\beta_1 + \beta_0$, $s = 2\sigma_1 + \sigma_0$, $c = 2\gamma_1 + \gamma_0$, where $\alpha_0, \alpha_1, \beta_0, \beta_1, \sigma_0, \sigma_1, \gamma_0, \gamma_1 \in \{0, 1\}$, and $L = (l_1, l_0)$ where l_1 and l_0 are two single bit LUTs, according to the following:

- $l_0(\alpha_0, \beta_0, \sigma_0) = \gamma_0$
- $l_1(a, b, s) = \gamma_1$

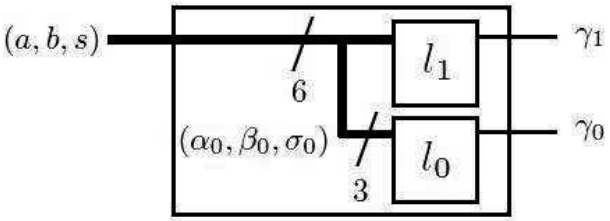

Figure 1: The structure of $L(a, b, s)$

We remark that while l_0 needs only three input bits to accomplish its function, l_1 requires all six bits of input.

The LUTs l_0 and l_1 were configured to the values shown in Table 1 and Table 2 to perform the multiply-accumulate function.

(α_0, β_0) \ σ_0	0	1
(0,0)	0	1
(0,1)	0	1
(1,0)	0	1
(1,1)	1	0

Table 1: Contents of l_0

With the help of these basic units one can compute the dot product w of two vectors $u = (u_0, u_1, \ldots, u_{n-1})$ and $v = (v_0, v_1, \ldots, v_{n-1})$. Let us define a module $m = (L[0], L[1], \ldots, L[n-1])$ by cascading n MA units denoted by $L[i]$. In this module m we use the output of a given MA unit as the sum input of the next unit, i.e. $s_{i+1} = c_i$ for $i = 0, 1, \ldots, n-2$, where s_i and c_i are the s input and c output of $L[i]$.

Therefor m is a function that accepts a pair of vectors u, v of two-digit numbers of length n and outputs on c_{n-1} the two-digit dot-product of the two vectors, i.e. $m(u, v) = w$.

(a, b) \ s	0	1	2	3
(0,0)	0	0	1	1
(0,1)	0	0	1	1
(0,2)	0	0	1	1
(0,3)	0	0	1	1
(1,0)	0	0	1	1
(1,1)	0	1	1	0
(1,2)	1	1	0	0
(1,3)	1	0	0	1
(2,0)	0	0	1	1
(2,1)	1	1	0	0
(2,2)	0	0	1	1
(2,3)	1	1	0	0
(3,0)	0	0	1	1
(3,1)	1	0	0	1
(3,2)	1	1	0	0
(3,3)	0	1	1	0

Table 2: Contents of l_1

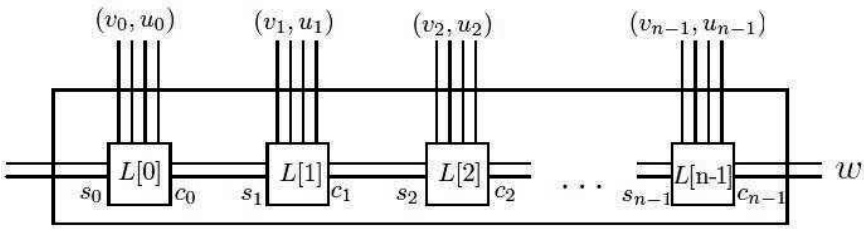

Figure 2: The structure of $m(u, v)$

In total, the number of LUTs used in m is $2n$. Note that vectors of arbitrary length can be used in the computation if we connect the output of module m to the sum input of L[0] ($c_{n-1} = s_0$), and then iteratively shift u and v onto the module's input by n elements at a time:

Function $\mathtt{iterated\_m}(u, v)$ // $k = \mathtt{length}(u) = \mathtt{length}(v)$
1. Define $\kappa = \lceil \frac{k}{n} \rceil$, $v', u' \in \mathbb{Z}_4^{\kappa \cdot n}$
2. for $i = 0$ to $\kappa \cdot n - 1$ do // fill v and u with 0's
3. if $i < k$ then $v'_i = v_i$ else $v'_i = 0$
4. if $i < k$ then $u'_i = u_i$ else $u'_i = 0$
5. end for
6. Define $v_{\text{temp}}, u_{\text{temp}}, w$, let $w = 0$
7. for $i = 0$ to $\kappa - 1$ do // shift v' and u' to v_{temp} and u_{temp}
8. $v_{\text{temp}} = (v'_{i \cdot n}, v'_{1+(i \cdot n)}, \ldots, v'_{n-1+(i \cdot n)})$
9. $u_{\text{temp}} = (u'_{i \cdot n}, u'_{1+(i \cdot n)}, \ldots, u'_{n-1+(i \cdot n)})$
10. $w = w + m(v_{\text{temp}}, u_{\text{temp}})$
11. end for
12. return w

end Function

Here u' and v' are the extensions of u and v by 0's.

We shall see that the number chosen for n is critical in setting many characteristics of the entire project. The experiment used for determining n will be discussed in the following chapter.

Our aim is to obtain a module that performs the matrix multiplication of $A, B \in \mathbb{Z}_4^{k \times k}$, where $\mathbb{Z}_4$ is the mod 4 residue class ring. In the following, let $C \in \mathbb{Z}_4^{k \times k}$ be the output matrix, such that $C = A \times B$. Furthermore, let a_i be the ith row of matrix A and let b_j be the jth column of matrix B.

The multiplier units denoted by m are used to create more complex modules in a hierarchical manner. First, by taking ten m multiplier blocks we create a row of multipliers $R = (m_0, m_1, \ldots, m_9)$. This is used to compute ten consecutive elements of a single row of the output matrix:

$$R(a_i, b_j, b_{j+1}, \ldots, b_{j+9}) = (c_{i,j}, c_{i,j+1}, \ldots, c_{i,j+9}) \ ,$$

where $c_{i,j} = a_i \cdot b_j$. The input vector a_i is used by all ten multiplier units of R. The length of these vectors, as mentioned above, can be arbitrary, but vectors of length greater than n will need to be iteratively shifted to the input of R.

By taking ten row multipliers we can create a unit $M_{10 \times 10} = (R_0, R_1, \ldots, R_9)$ which outputs a 10×10 sub-matrix of C:

$$M_{10\times10}(a_i, a_{i+1}, \ldots, a_{i+9}, b_j, b_{j+1}, \ldots, b_{j+9}) =$$

$$\begin{pmatrix} c_{i,j} & c_{i,j+1} & \cdots & c_{i,j+9} \\ c_{i+1,j} & c_{i+1,j+1} & \cdots & c_{i+1,j+9} \\ \vdots & \ddots & & \\ c_{i+9,j} & c_{i+9,j+1} & \cdots & c_{i+9,j+9} \end{pmatrix}.$$

Finally, four such units are arranged so that a 20×20 sub-matrix of C could be obtained as output:

$$M_{20\times20}(a_i, a_{i+1}, \ldots, a_{i+19}, b_j, b_{j+1}, \ldots, b_{j+19}) =$$

$$\begin{pmatrix} c_{i,j} & c_{i,j+1} & \cdots & c_{i,j+19} \\ c_{i+1,j} & c_{i+1,j+1} & \cdots & c_{i+1,j+19} \\ \vdots & \ddots & & \\ c_{i+19,j} & c_{i+19,j+1} & \cdots & c_{i+19,j+19} \end{pmatrix}.$$

The $M_{20\times20}$'s inputs are twenty vectors from both matrices A and B. Because of hardware constraints — in particular the number of LUTs on the used device — a larger arrangement of multipliers would be impractical to implement. The module $M_{20\times20}$ is comprised of 400 m multiplier units. Figure 3 shows the hierarchy of units used to build $M_{20\times20}$.

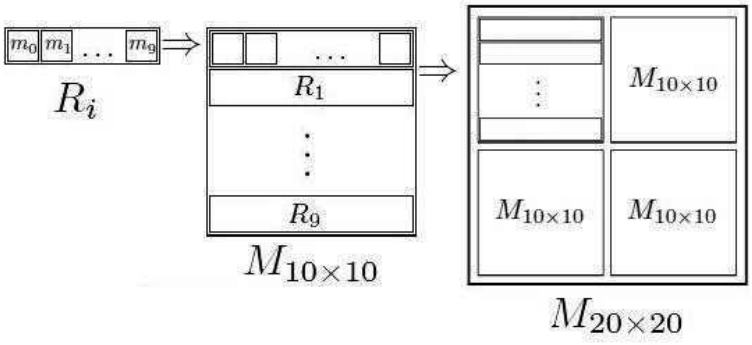

Figure 3: The structure of $M_{20\times20}$

The $M_{20\times20}$ unit can be used iteratively to multiply matrices of arbitrary size, producing 20×20 sub-matrices of the output matrix C with each iteration. After inputting twenty rows from matrix A and twenty columns from matrix B and obtaining the desired output, we can simply repeat the process for a

set of rows and columns of A and B respectively, until we obtain the entire output matrix C:

Function $\mathtt{large\_matrix\_mult}(A, B)$
1. Define $\kappa = \lceil \frac{k}{20} \rceil$, $A', B', C' \in \mathbb{Z}_4^{\kappa\cdot n \times \kappa\cdot n}$
2. for $i = 0$ to $20\kappa - 1$ do
3. for $j = 0$ to $20\kappa - 1$ do
4. if $i < k$ and $j < k$ $a'_{ij} = a_{ij}$ else $a'_{ij} = 0$
5. if $i < k$ and $j < k$ $b'_{ij} = b_{ij}$ else $b'_{ij} = 0$
6. end for end for
7. for $i = 0$ to $\kappa - 1$ do
8. for $j = 0$ to $\kappa - 1$ do
9. $C'^{[i,j}_{[i+19,j+19]} = M_{20\times 20}(a_i, a_{i+1}, \ldots, a_{i+19}, b_j, b_{j+1}, \ldots, b_{j+19})$
10. end for end for
11. return $C'^{[0,0}_{[k-1,k-1]}$

end Function

Here

$$C'^{[i,j}_{[k,l]} = \begin{pmatrix} c'_{i,j} & c'_{i,j+1} & \cdots & c'_{i,l} \\ c'_{i+1,j} & c'_{i+1,j+1} & \cdots & c'_{i+1,l} \\ \vdots & \ddots & & \\ c'_{k,j} & c'_{k,j+1} & \cdots & c'_{k,l} \end{pmatrix}.$$

Note that in the naive algorithm $\mathtt{large\_matrix\_mult}(A, B)$, during the main loop (lines 7-10), for each twenty rows read from A, the entire matrix B is read. During the whole procedure, matrix A will be read entirely exactly once, while matrix B will be read κ times. Methods improving on this number are described in section 6.

Since for almost all practical cases the size k of matrices $A, B \in \mathbb{Z}_4^{k\times k}$ will be greater than the parameter n, the vectors taken from these matrices will need to be iteratively shifted onto the input of the multiplier $M_{20\times 20}$, n elements at a time. Therefore, an efficient way to both store and then use the vectors taken from the matrices is the creation of FIFO type containers made of shift registers.

Let t_n^d be a shift register of width n and depth d. It means that t_n^d can store at most d vectors of length n, or equivalently a single vector of length at most nd. We choose d such that $nd \geq k$, thus it can store one row or column from the input matrices A or B. Let the vector filling t_n^d be $f = (f_0, f_1, \ldots, f_{d-1})$,

where $f_i \in \mathbb{Z}_4^n$, $i = 0, 1, \ldots, d-1$. In practice, t_n^d is a queue data structure. In a single step, t_n^d outputs a vector of length n and shifts its content by n places. For the i^{th} activation, the container will output f_i. After d activations, the container becomes empty.

One container t_n^d is used to store a single row or column of matrices A or B respectively. Connecting twenty of them in parallel, denoted by $T_{20n}^d = (t_n^d[0], t_n^d[1], \ldots, t_n^d[19])$, we obtain a container that stores twenty rows or columns. This is exactly the amount of data the $M_{20\times 20}$ multiplier structure requires as input in d iteration steps. After d activations T_{20n}^d has shifted all its stored data to $M_{20\times 20}$, broken up into pieces of length n for each activation. Two such T_{20n}^d containers are connected to $M_{20\times 20}$, one for the rows taken from matrix A and one for the columns taken from matrix B.

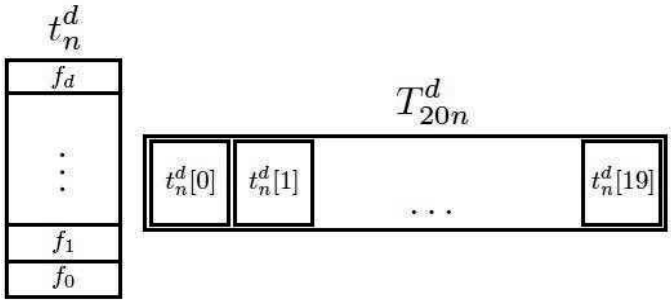

Figure 4: The structure of T_{20n}^d

Using $M_{20\times 20}$ and T_{20n}^d in a proper structure, we can execute one iteration cycle of the computation. After filling one T_{20n}^d container with the desired twenty rows from matrix A and one T_{20n}^d container with the desired twenty columns from matrix B, we simply send d activation signals to the containers. This will shift the data onto $M_{20\times 20}$, which computes the 20×20 product matrix in the way described in function $\mathtt{iterated\_m}(u, v)$. The number of steps in one iteration cycle is d.

5 Experimental determination of parameters

Now, we turn to the determination of n (how many MA modules should be connected into a single multiplier m). This sets the length of the vectors that we use in the computation in a single step and thus has an effect on many other technical parameters of the design. The goal was to find the greatest number

such that the multiplier would still reliably produce the correct dot product in a single clock cycle. Clearly, this number dependents on the used hardware and the clock frequency. For the device used, the chosen clock frequency was 100 MHz, the default frequency provided by the board.

The following experiment was devised to determine the value of n:

Let S be a multiplier m, called the "Subject", and let $E_0, E_1, \ldots, E_9$ be ten more m multipliers, called the "Examiners". Informally, the Examiners' duty was to verify the answers given by the Subject to questions they already knew the answer to. The "questions" here are test data: two vectors v, u of length n generated by the following sequence to obtain suitable pseudo-random values:

$$D_i = D_{i-1} + D_{i-2} + 2D_{i-4} + D_{i-5},$$

where $D_0 = D_1 = D_2 = D_3 = 0,\ D_4 = 1$.

More formally, let $p \in \mathbb{Z}_{10}$ be a counter that cycles between values $0, 1, \ldots, 9$, incrementing its value by one with each clock cycle, and returning to value 0 after 9. For each clock cycle during the experiment, the following happens depending on the value of p:

- The output of S is checked for equality with the output of E_p. If inequality is detected, then an error is noted.
- The test data E_{p+1} is currently working on is given to S.
- New test data is given to E_{p-1}.

Procedure testing
1. Let $S, E_0, E_1, \ldots, E_9$ be m multipliers
2. Let D be the test data generator
3. Let $i \in \mathbb{N}, p \in \mathbb{Z}_{10}$
4. forever do
5. $i = i + 1$
6. $p \equiv i \mod 10$
7. if $S_{out} \neq E_{p\_out}$ then return ERROR
8. $E_{p-1\_in} \leftarrow D(i)$
9. $S_{in} \leftarrow E_{p+1\_in}$
10. end forever

end Procedure

Note that the output of S is checked every clock cycle, which yields that S has only a single cycle to calculate its answer to the question it was given

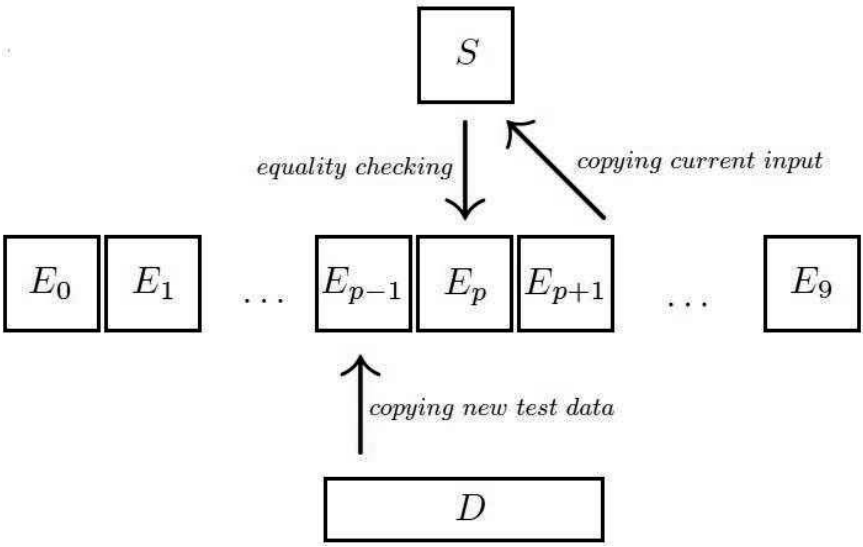

Figure 5: Activity of testing module when counter's value is $\mathtt{p}$

in the preceding clock cycle. A given Examiner, however, has ten times more time to work on its test data. Once in every ten clock cycles, new data is given to the Examiner to work on, and its output is only checked nine clock cycles later, just before it is given new input again. This way the Examiners have enough time to compute the correct answer to the question by the time it is needed.

As the initial value for $\mathtt{n}$, we have chosen 16, a number small enough to be reasonably expected to pass the criteria set for $\mathtt{n}$, but large enough to be of interest. If the experiment reported no error, meaning the Subject was flawlessly able to calculate the dot product for a sufficiently long time, then the value of $\mathtt{n}$ was increased and the experiment repeated. After the first error was encountered, meaning the Subject was not able to keep up with the calculations, the largest value was chosen for $\mathtt{n}$ for which there were no errors.

On the used device, the largest such value was found to be 28 at a clock speed of 100 MHz and setting the length of $\mathtt{m}$ multipliers to $\mathtt{n} = 28$ were able to work error-free for days without interruption.

6 Computation of large matrices

In the Section 4 we gave an algorithm for using the described modules for computing the product of large matrices. Following the description, the implemented design would make use of the parallelism offered by the FPGA only in the computation of dot products. Making further use of parallel operations, the design's performance can be significantly improved. In this section we describe the implementation choices made to raise the overall performance.

The biggest factor to consider is the management of data. When computing the product of large matrices, the amount of data to store and to move between the computation modules can easily exceed the size which can be practically stored on the FPGA. Fortunately, as mentioned before, a 256MB DDR2 SODIMM is connected to the board as the main data storage device. A module is generated using the Memory Interface Generator v3.5 intellectual property core provided by Xilinx to implement the logic needed to communicate with the DDR2 RAM. The module is structured hierarchically, connecting the memory device to a user interface. All communication with the device is done through two FIFO queues: one queue to send the command and address signals, while the other queue is used for write data and write data mask (when masking is allowed).

A naive utilization of the memory would be to simply read the required data before each iteration of the computation, and writing the output back after it is finished. An undesirable effect of this approach would be that the design would spend significantly more time with memory management than with the actual computation. The desired result would be that memory management (and all other auxiliary operations) were done during the time interval of the computation. Note that since both the size of the matrices and the multiplier module is fixed, the time the multiplication consumes is a fixed constant, which cannot be lowered. Optimally, the time of the computation should be an upper bound for the running time of the entire design. The difficulty of reaching this optimum lies in the high speed of the multiplier modules compared to the memory module.

One way to resolve the problem caused by slow transmission speed is to increase the amount of data stored on the FPGA. Informally, the main idea is to keep enough data in a prepared state, i.e. by the time the multiplier module finishes all of its computations, we have enough new data to continue working. More formally, let us define the following quantities:

- Let d be the time necessary to complete one iteration of the computation.

As described in the previous sections, this is equal to the depth of the containers T^d_{20n}.

- Let $\kappa = \lceil \frac{k}{20} \rceil$, where k is the size of the matrices. ($A, B \in \mathbb{Z}_4^{k \times k}$) This quantity is already used in algorithm `large_matrix_mult`. For the rest of the section, it is practical to think of A and B as $\kappa \times \kappa$ sized block matrices, where each element is a 20×20 matrix.

- Let $f(A, B)$ be an arbitrary algorithm executing matrix multiplication on A and B, including the memory management needed for the computation. Let $K(f)$ be the number of times the algorithm needs to fill a T^d_{20n} container, i.e. the number of times it has to read twenty rows or columns from the matrices. Note that completely reading either input matrices once means filling T^d_{20n} containers κ times, since one T^d_{20n} can store twenty rows or columns at a time. Algorithm `large_matrix_mult`'s main loop (starting at line 7) reads twenty rows from matrix A (filling a T^d_{20n} once) and reads matrix B entirely for each step. Since the loop has κ steps, it follows that $K(\texttt{large\_matrix\_mult}) = \kappa^2 + \kappa$.

- Let δ be the time it takes to fill a T^d_{20n} container. This quantity depends on both the width and depth of the container. The total time $f(A, B)$ spends on reading from memory to fill the containers is $K(f)\delta$.

- Let $\Phi(f)$ be the total time the design has to spend with memory management. This is the sum of the time it spends on reading matrices A and B from the memory and the time it spends on writing the product matrix C into the memory. The number of times f has to read A and B from the memory depends on f. Note that since the size of the total output matrix C is the same as the size of A and B, writing C into the memory takes time equal to reading either matrices once from the memory. In other words, it takes $\kappa\delta$ time. The total time the design has to spend with memory management is $\Phi(f) = K(f)\delta + \kappa\delta$.

- Let $\Gamma(f)$ be the time $f(A, B)$ spends on the computation itself. From the definition of d and κ it follows that $C(\texttt{large\_matrix\_mult}) = d\kappa^2$.

The goal here is to reduce $K(f)$ in such a way that the data required for the next iteration of the computation is always ready by the time the previous iteration ends. If this arrangement is achieved then $C(f)$ becomes the upper bound for the running time of the design.

Storing more data on the FPGA can be done by adding more T^d_{20n} containers to the design. During an iteration only two such containers are used directly. The rest can be used to load data necessary for the forthcoming iteration steps.

Suppose the design has $z+2$ pieces of T^d_{20n} containers. We assign z of the containers to store rows from matrix A, called "row-stores", and two of them to store columns from matrix B, called "column-stores". With this arrangement, we can carry out $z-1$ iterations of the computation, using up the data stored in $z-1$ row-stores and one column-store. This leaves one row-store and one column-store to load new data into during the computation. Using the above definitions, the allover computation takes $(z-1)d$ time.

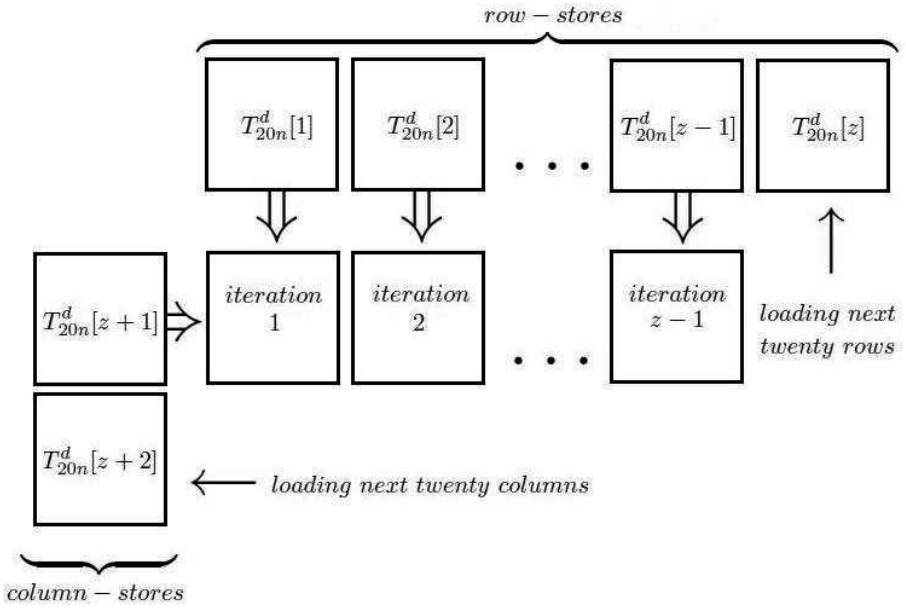

Figure 6: Configuration of data stored on the FPGA

If we use all z row-stores and one column-store for the computation while the remaining column-store is devoted to loading new columns into, then we would have to load all z row-stores with new rows once we read all the columns before we can continue the computation. This would take $z\delta$ time for each case where we read all the columns but haven't read all the rows yet, which happens $\lfloor \frac{\kappa}{z} \rfloor$ times. In total, it would add $\lfloor \frac{\kappa}{z} \rfloor z\delta$ to the running time.

Instead, the computation of the output matrix moves slightly diagonally. See Figure 7. The $z-1$ row-stores used in the computations store a total of $(z-1)\cdot 20$ rows. Initially, the row-stores are filled with rows $a_0 \to a_{(z-1)\cdot 20-1}$.

New rows are loaded in at a slower pace than columns are. By the time all columns are read once, the contents of the row-stores have shifted exactly to the next segment of data needed, the next $(z-1)\cdot 20$ rows. After matrix B is completely read once, the row-stores are filled with rows $a_{(z-1)\dot{2}0} \rightarrow a_{2(z-1)\cdot 20-1}$. Reading rows and columns proceeds in this manner until we've completely read matrix A once. For this reason, it is practical to choose z such that $(z-1) \mid \kappa$. All together we read matrix B $\frac{\kappa}{z-1}$ times and matrix A once. During each $z-1$ iterations shown in Figure 6, twenty new columns and $\frac{(z-1)\cdot 20}{\kappa}$ new rows are loaded into the column-store and row-store currently unused by the computation. When the unused row-store is filled with twenty new rows, it becomes active, to be used in the following iterations. The row-store containing the rows with the least index becomes inactive in the computation and starts accepting the new rows read.

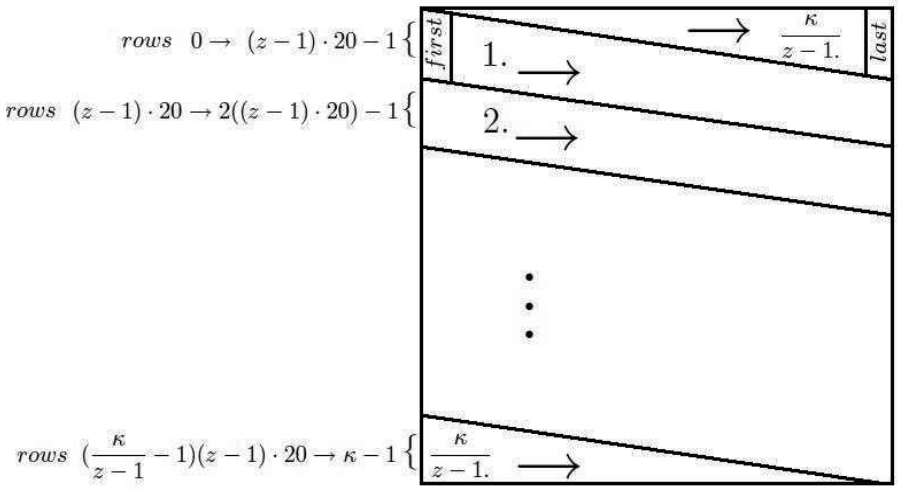

Figure 7: Progression of computations through matrix C

Function improved_matrix_mult(A, B)

1. Define $z, \kappa = \lceil \frac{k}{20} \rceil$, $A', B', C' \in \mathbb{Z}_4^{\kappa\cdot n \times \kappa\cdot n}$
2. for $i = 0$ to $20\kappa - 1$ do
3. for $j = 0$ to $20\kappa - 1$ do
    4. if $i < k$ and $j < k$ then $a'_{ij} = a_{ij}$ else $a'_{ij} = 0$
    5. if $i < k$ and $j < k$ then $b'_{ij} = b_{ij}$ else $b'_{ij} = 0$
6. end for end for

7. Fill the row-stores with rows $a_0 \rightarrow a_{(z-1)\cdot 20-1}$
8. Fill the column-stores with columns $b_0 - b_{19}$
9. For $i = 1$ to $\frac{\kappa^2}{z-1}$
Do in parallel: |perform $z-1$ iterations of the computation
|READ the next 20 columns mod $\kappa \cdot 20$
|READ the next $\frac{(z-1)\cdot 20}{\kappa}$ rows mod $\kappa \cdot 20$
|WRITE the result of the previous $z-1$ iterations
11. return $C'^{[0,0}_{k-1,k-1]}$

end Function

The possible values for the parameters used in this section depend on the used hardware.

The size of the matrices used in the implementation are determined by parameters $n = 28$ and $d = 32$. The LUTs on the device that comprise the T^d_{20n} containers can be configured as $d = 32$ bit deep shift registers. For this reason the matrices are of size 896×896. Rows with length $k = 896$ are the largest that can be stored in containers that are one LUT deep, making them any larger would double the number of LUTs needed for creating a T^d_{20n}. Because of the limited number of LUTs which can be used for storage purposes, $z = 10$ was chosen. This yields that twelve T^d_{20n} containers are defined in the design. Dealing with matrices larger than $k = 896$ is part of future work.

For convenience, time quantities are measured in clock cycles at 100MHz, the clock speed of the $M_{20\times 20}$ multiplier.

The value of δ depends on the DDR2 RAM used. The device was used at 200MHz, and has a 64 bit wide physical data bus.

From these values we determine the following parameters:

- $\kappa = \lceil \frac{896}{20} \rceil = 45$,
- $K(\mathrm{improved\_matrix\_mult}) = \frac{\kappa}{z-1}\kappa + \kappa = \frac{45}{9} \cdot 45 + 45 = 270$,
- $\delta = 140$ clock cycles at 100MHz,
- $\Phi(\mathrm{improved\_matrix\_mult}) = K(\mathrm{improved\_matrix\_mult})\delta + \kappa\delta = 270 \cdot 140 + 45 \cdot 140 = 44100$ clock cycles at 100MHz,
- $\Gamma(\mathrm{improved\_matrix\_mult}) = d\kappa^2 = 32 \cdot 45^2 = 64800$ clock cycles at 100MHz.

The goal of $\Gamma(\texttt{improved\_matrix\_mult}) > \Phi(\texttt{improved\_matrix\_mult})$ is achieved, meaning that the running time of the design is equal to the time used by the computation.

The speedup provided by the configuration can be shown by comparing its performance to a similar implementation created on a more traditional architecture. A highly optimized C++ program was created for a machine using an Intel E8400 3GHz Dual Core processor with 2GB RAM. The algorithm is strongly specialized for the task, making use of all available options for increasing performance. It uses 64 bit long variables to perform multiplication on 16 pairs of two-digit elements at once in parallel on both processor cores.

The running time of the multiplication of matrices of the same size is over 100 ms. The FPGA implementation, as mentioned above, achieves a runtime of ~0.6 ms. On average, a speedup factor of 200 is reached using the described FPGA design.

7 Future work

The future course of research will focus on increasing the size of the used matrices.

As mentioned in the previous section, simply increasing the depth d of the T^d_{20n} containers would be impractical. Since a single LUT on the device can only be configured as a 32 bit deep shift register, setting $d > 32$ would double the number of LUTs needed for a T^d_{20n}, and the design is already using well over half of the device's LUTs that can be configured this way (13440 out of 17280, to be exact). Increasing the size of the matrices this way would require the restructuring of both the multiplier module and the algorithm used for memory management.

Instead, the currently implemented module can be used as a basic unit for the multiplication of larger matrices. Then the entries of the large matrices are 896×896 blocks.

This also allows for further optimization using Strassen's algorithm. Suppose we double the matrix sizes, interpreting them as matrices with four blocks. Using the classical algorithm, multiplying two 1792×1792 sized matrices would take eight multiplication of the blocks. Using a divide-and-conquer strategy, we can exchange one multiplication for a few extra additions.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} -D_2 + D_4 + D_5 + D_6 & D_1 + D_2 \\ D_3 + D_4 & D_1 - D_3 + D_5 - D_7 \end{bmatrix},$$

where

$$
\begin{aligned}
D_1 &= A_{11}(B_{12} - B_{22}) \\
D_2 &= (A_{11} + A_{12})B_{22} \\
D_3 &= (A_{21} + A_{22})B_{11} \\
D_4 &= A_{22}(B_{21} - B_{11}) \\
D_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
D_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
D_7 &= (A_{11} - A_{21})(B_{11} + B_{12}).
\end{aligned}
$$

This algorithm, with its $O(n^{\lg 7})$ time complexity, could speed up the design on large matrices. We should note however, that the speed of the extra additions have to be carefully considered. Since the multiplication is already extremely fast, a similar improvement may also be necessary for additions if the overall performance upgrade is to remain significant.

Acknowledgements

Research supported by the TÁMOP 4.2.1/B-09/1/KONV-2010-0007 project and TARIPAR3 project grant Nr. TECH 08-A2/2-2008-0086.

References

[1] F. Bensaali, A. Amira, R. Sotudeh, Floating-point matrix product on FPGA, *IEEE/ACS International Conference on Computer Systems and Applications*, Amman, Jordan, 2007, pp. 466–473. ⇒173

[2] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, *J. Symbolic Comput.* **9,** 3 (1990) 251–280. ⇒173

[3] N. Dave, K. Fleming, M. King, M. Pellauer, M. Vijayaraghavan, Hardware acceleration of matrix multiplication on a Xilinx FPGA, *MEMOCODE '07 Proc. 5th IEEE/ACM International Conference on Formal Methods and Models for Co-Design,* Nice, France, 2007, pp. 97–100. ⇒173

[4] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev, 64-bit floating-point FPGA matrix multiplication, *Proc. 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, Monterey, CA, USA, 2005, pp. 86–95. ⇒173

[5] T. J. Earnest, Strassen's Algorithm on the Cell Broadband Engine, 2008, http://mc2.umbc.edu/docs/earnest.pdf ⇒173

[6] T. Herendi, Uniform distribution of linear recurrences modulo prime powers, *J. Finite Fields Appl.* **10,** 1 (2004) 1–23. ⇒174

[7] T. Herendi, Construction of uniformly distributed linear recurring sequences modulo powers of 2 (to appear). ⇒174

[8] A. Irturk, S. Mirzaei, R. Kastner, An Efficient FPGA Implementation of Scalable Matrix Inversion Core using QR Decomposition, *UCSD Technical Report*, CS2009-0938, 2009. ⇒173

[9] B. Kakaradov, Ultra-fast matrix multiplication, An empirical analysis of highly optimized vector algorithms, *Stanford Undergraduate Research Journal* **3** (2004) 33–36. ⇒173

[10] M. Karkooti, J. R. Cavallaro, C. Dick, FPGA implementation of matrix inversion using QRD-RLS algorithm, *Proc. 39th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, 2005, pp. 1625–1629. ⇒173

[11] S. M. Qasim, A. A. Telba, A. Y. AlMazroo, FPGA design and implementation of matrix multiplier architectures for image and signal processing applications, *IJCSNS International Journal of Computer Science and Network Security* **10,** 2 (2010) 168–176. ⇒173

[12] V. Strassen, Gaussian elimination is not optimal, *Numer. Math.* **13** (1969) 354–356. ⇒173

Received: May 17, 2011 • Revised: October 11, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 192–204

The debts' clearing problem: a new approach

Csaba PĂTCAŞ
Babeş-Bolyai University
Cluj-Napoca
email: patcas@cs.ubbcluj.ro

Abstract. The debts' clearing problem is about clearing all the debts in a group of n entities (e.g. persons, companies) using a minimal number of money transaction operations. In our previous works we studied the problem, gave a dynamic programming solution solving it and proved that it is NP-hard. In this paper we adapt the problem to dynamic graphs and give a data structure to solve it. Based on this data structure we develop a new algorithm, that improves our previous one for the static version of the problem.

1 Introduction

In [2] we studied the debts' clearing problem, and gave a dynamic programming solution using $\Theta(2^n)$ memory and running in time proportional to 3^n. The problem statement is the following:

Let us consider a number of n entities (e.g. persons, companies), and a list of m borrowings among these entities. A borrowing can be described by three parameters: the index of the borrower entity, the index of the lender entity and the amount of money that was lent. The task is to find a minimal list of money transactions that clears the debts formed among these n entities as a result of the m borrowings made.

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 05C85
Key words and phrases: debt clearing, dynamic graph

Example 1

Borrower	*Lender*	*Amount of money*
1	*2*	*10*
2	*3*	*5*
3	*1*	*5*
1	*4*	*5*
4	*5*	*10*

Solution:

Sender	*Reciever*	*Amount of money*
1	*5*	*10*
4	*2*	*5*

In [2] we modeled this problem using graph theory:

Definition 2 *Let* $\mathsf{G}(\mathsf{V}, \mathsf{A}, W)$ *be a directed, weighted multigraph without loops,* $|\mathsf{V}| = \mathfrak{n}$, $|\mathsf{A}| = \mathfrak{m}$, $W : \mathsf{A} \to \mathbb{Z}$, *where* V *is the set of vertices,* A *is the set of arcs and* W *is the weight function.* G *represents the borrowings made, so we will call it the* ***borrowing graph****.*

Example 3 *The borrowing graph corresponding to Example 1 is shown in Figure 1.*

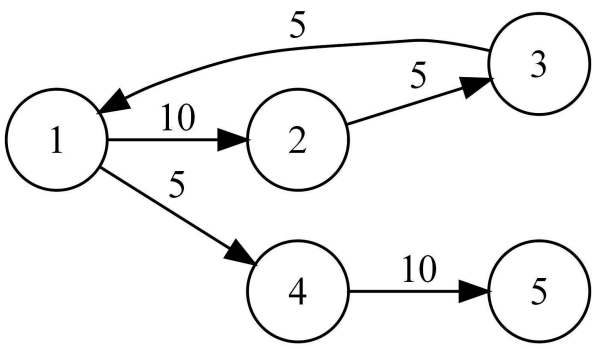

Figure 1: The borrowing graph associated with the given example. An arc from node $\mathfrak{i}$ to node $\mathfrak{j}$ with weight w means, that entity $\mathfrak{i}$ must pay w amount of money to entity $\mathfrak{j}$.

Definition 4 *Let us define for each vertex* $v \in \mathsf{V}$ *the* ***absolute amount of debt*** *over the graph* G*:* $\mathsf{D}_{\mathsf{G}}(v) = \sum_{\substack{v' \in \mathsf{V} \\ (v,v') \in \mathsf{A}}} W(v, v') - \sum_{\substack{v'' \in \mathsf{V} \\ (v'',v) \in \mathsf{A}}} W(v'', v)$

Definition 5 *Let* $G'(V, A', W')$ *be a directed, weighted multigraph without loops, with each arc* (i, j) *representing a transaction of* $W'(i, j)$ *amount of money from entity* i *to entity* j. *We will call this graph a* ***transaction graph****. These transactions clear the debts formed by the borrowings modeled by graph* $G(V, A, W)$ *if and only if:*

$D_G(v_i) = D_{G'}(v_i), \forall i = \overline{1, n}$, *where* $V = \{v_1, v_2, \ldots, v_n\}$.

We will note this by: $G \sim G'$.

Example 6 *See Figure 2 for a transaction graph with minimal number of arcs corresponding to Example 1.*

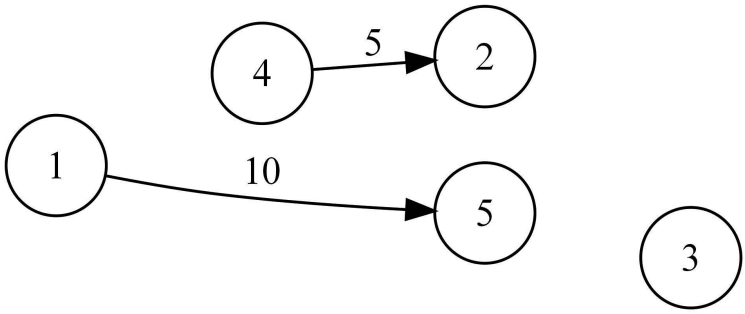

Figure 2: The respective minimum transaction graph. An arc from node i to node j with weight w means, that entity i pays w amount of money to entity j.

Using the terms defined above, the debt's clearing problem can be reformulated as follows:

Given a borrowing graph $G(V, A, W)$ *we are looking for a minimal transaction graph* $G_{min}(V, A_{min}, W_{min})$, *so that* $G \sim G_{min}$ *and* $\forall G'(V, A', W') : G \sim G', |A_{min}| \leq |A'|$ *holds.*

2 The debts' clearing problem in dynamic graphs

Definition 7 *A* ***dynamic graph*** *is a graph, that changes in time, by undergoing a sequence of updates. An update is an operation, that inserts or deletes edges or nodes of the graph, or changes attributes associated to edges or nodes.*

In a typical dynamic graph problem one would like to answer queries regarding the state of the graph in the current time moment. A good dynamic graph algorithm will update the solution efficiently, instead of recomputing it from scratch after each update, using the corresponding static algorithm [1].

In the dynamic debts' clearing problem we want to support the following operations:

- InsertNode(u) – adds a new node u to the borrowing graph.
- RemoveNode(u) – removes node u from the borrowing graph. In order for a node to be removed, all of its debts must be cleared first. In order to affect the other nodes as little as possible, the debts of u will be cleared in a way that affects the least number of nodes, without compromising the optimal solution for the whole graph.
- InsertArc(u, v, x) – insert an arc in the borrowing graph. That is, u must pay x amount of money to v.
- RemoveArc(u, v) – removes the debt between u and v.
- Query() – returns a minimal transaction graph.

Example 8 *For instance calling the* Query *operation after adding the third arc in the borrowing graph corresponding to Example 1 would result in the minimal transaction graph from Figure 3.*

These operations could be useful in the implementation of an application that facilitates borrowing operations among entities, such as BillMonk [5] or Expensure [6]. When a new user registers to the system, it is equivalent with an InsertNode operation, and when a user wants to leave the system it is the same as a RemoveNode. When a borrowing is made, it can be implemented by a simple call of InsertArc. Two persons may decide, that they no longer owe each other anything. In this case RemoveArc can be useful. If the whole group decides, that it is time to settle all the debts, the Query operation will be used.

3 A data structure for solving dynamic debts' clearing

As the static version of the problem is NP-hard [3], it is not possible to support all these operations in polynomial time (unless P = NP). Otherwise we could

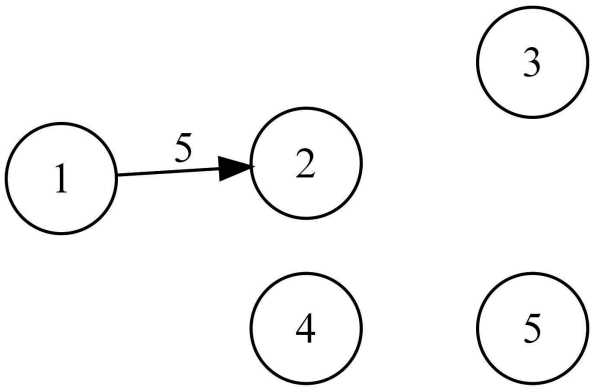

Figure 3: Result of the QUERY operation called after the third arc was added

just build up the whole graph one arc at a time, by m calls of INSERTARC, then construct a minimal transaction graph by a call of QUERY, which would lead to a polynomial algorithm for the static problem.

Our data structure used to support these operations is based on maintaining the subset of nodes, that have non-zero absolute amount of debt $V^* = \{u|D(u) \neq 0\}$. The sum of D values for all the $2^{|V^*|}$ subsets of V^* is also stored in a hash table called sums.

3.1 InsertNode

As for our data structure only nodes having non-zero D values are important, and a new node will always start with no debts, it means that nothing has to be done when calling INSERTNODE.

3.2 InsertArc

When INSERTARC is called, the D values of the two nodes change, so V^* can also change. When a node leaves V^*, we do not care about the updating the sum of the subsets it is contained in, because when a new node enters V^* we will have to calculate the sum of all of the subsets it is contained in anyway.

If both u and v were in V^* and remained in it after changing the D values, then we simply add x to the sum of all subsets containing u, but not v, and subtract x from those containing v but not u. The sum of the subsets containing both nodes does not change.

If one of the nodes was just added to V^* ($D[u] = x$, or $D[v] = -x$), then all the sums of the subsets containing it must be recalculated. This recalculation can be done in $O(1)$ for each subset, taking advantage of sums already calculated for smaller subsets.

Procedure 1: UpdateSums(u, v, x)

```
   // Updates the sum of all subsets containing u but not v
1  foreach S ⊂ V*, such that u ∈ S and v ∉ S do
2  |  if D[u] = x then sums[S] := sums[S \ {u}] + x;
3  |  else sums[S] := sums[S] + x;
```

Algorithm 2: InsertArc(u, v, x)

```
1  if D[u] = 0 then V* := V* ∪ {u};
2  if D[v] = 0 then V* := V* ∪ {v};
3  D[u] := D[u] + x; D[v] := D[v] − x;
4  if D[u] = 0 then V* := V* \ {u};
5  if D[v] = 0 then V* := V* \ {v};
6  if D[u] ≠ 0 then UpdateSums (u, v, x);
7  if D[v] ≠ 0 then UpdateSums (v, u, −x);
8  if D[u] = x or D[v] = −x then
9  |  foreach S ⊂ V*, such that u, v ∈ S do
10 |  |  sums[S] := sums[S \ {u, v}] + D[u] + D[v];
```

One call of UpdateSums iterates over $2^{|V^*|-2}$ subsets, thus lines 6 and 7 of InsertArc together take $2^{|V^*|-1}$ steps. Additionally line 9 takes $2^{|V^*|-2}$ more steps.

3.3 Query

To carry out Query we observe, that finding a minimal transaction graph is equivalent to partitioning V^* in a maximal number of disjoint zero-sum subsets, more formally $V^* = P_1 \cup \ldots \cup P_{max}, sums[P_i] = 0, \forall i = \overline{1, max}$ and $P_i \cap P_j = \emptyset, \forall i, j = \overline{1, max}, i \neq j$. The reason for this is, that all the debts in a zero-sum subset P_i can be cleared by $|P_i| - 1$ transactions (see [2, 3, 4]), thus to clear all the debts, $|V^*| - max$ transactions are necessary. Let S^0 be the set of all subsets of V^*, having zero sum: $S^0 = \{S | S \subset V^*, sums[S] = 0\}$. Then, to find the maximal partition, we will use dynamic programming.

Let $dp[S]$ be the maximal number of zero-sum sets, $S \subset V^*$ can be partitioned in.

$$dp[S] = \begin{cases} \text{not defined}, & \text{if } sums[S] \neq 0 \\ 0, & \text{if } S = \emptyset \\ \max\{dp[S \setminus S'] + 1 | S' \subset S, S' \in S^0\}, & \text{otherwise.} \end{cases}$$

Building dp takes at most $2^{|V^*|} \cdot |S^0|$ steps.

As the speed at which QUERY can be carried out depends greatly on the size of S^0, we can use two heuristics to reduces its size, without compromising the optimal solution. To facilitate the running time of these heuristics, S^0 can be implemented as a linked list.

Clear pairs Choosing sets containing exactly two elements in the partition will never lead to a suboptimal solution, if the remaining elements are partitioned correctly [4]. Thus, before building dp, sets having two elements can be removed from S^0, along with all the sets, that contain those two elements (because we already added them to the solution, so there is no need to consider sets that contain them in the dynamic programming): $S^0 := S^0 \setminus (\{u, v\} \cup \{S' | u \in S' \text{ or } v \in S'\})$.

Procedure 3: ClearPairs

```
1  max := 0;
2  inPair := ∅;
3  foreach S ∈ S^0 do
4  |  if |S| = 2 then
5  |  |  if S ∩ inPair = ∅ then
6  |  |  |  max := max + 1;
7  |  |  |  P_max := S;
8  |  |  inPair := inPair ∪ S ;
9  foreach S ∈ S^0 do
10 |  if |S| ∩ inPair ≠ ∅ then  S^0 := S^0 \ S;
```

The running time of this heuristic is $\Theta(|S^0|)$.

Clear non-atomic sets If a set $S_i \in S^0$ is contained in another set $S_j \in S^0$, then S_j can be safely discarded, because $S_j \setminus S_i$ will also be part of S^0, and combining S_i with $S_j \setminus S_i$ always leads to a better solution, than using S_j alone: $S^0 := S^0 \setminus \{S_j | \exists S_i \in S^0 : S_i \subset S_j\}$.

This heuristic can be carried out in $\Theta(|S^0|^2)$.

Procedure 4: ClearNonAtomic

```
1 foreach S_i ∈ S^0 do
2 |   foreach S_j ∈ S^0, S_i ≠ S_j do
3 |   |   if S_i ⊂ S_j then S^0 := S^0 \ S_j;
```

3.4 RemoveNode

To delete a node u with the conditions listed in the introduction is equivalent to finding a set P of minimal cardinality containing u, that can still be part of an optimal partition, that is $dp[V^*] = dp[V^* \setminus P] + 1$. This algorithm can not be used together with the **Clear pairs** heuristic, because clearing pairs may compromise the optimal removal of u. The running time is the same as for QUERY, because dp must be built.

3.5 RemoveArc

Because clearing an arc between two nodes is the same as adding an arc in the opposite direction, this can be easily implemented using INSERTARC. If the D values of the two nodes have the same sign, it means, that no arc could appear in a minimal transaction between the two nodes, so nothing has to be done.

Algorithm 5: REMOVEARC(u, v)

```
1 if D[u] < 0 and D[v] > 0 then INSERTARC(u, v, min{−D[u], D[v]});
2 else
3 |   if D[u] > 0 and D[v] < 0 then INSERTARC(v, u, min{D[u], −D[v]});
```

3.6 Implementation details

In our implementation we used 32-bit integers to represent subsets. A subset of at most 32 nodes can be codified by a 32-bit integer by looking at its binary representation: node i is in the subset if and only if the i^{th} bit is one. This idea allows using bit operations to improve the running time of the program.

Because we did not use test cases having more than 20 nodes, the hash table $sums$ was implemented as a simple array having 2^n elements. V^* was stored as an ordered array, but other representations are also possible, because the running time of the operations on V^* is dominated by other calculations in our algorithm.

Before using the methods of the data structure for the first time, the memory for its data fields containing V^* and $\mathtt{sums}$ should be allocated and their values initialized, both being empty at the beginning. In a destructor type method these memory fields can be deallocated.

4 A new algorithm for the static problem

We can observe, that the QUERY operation needs only the set S^0 to be built, and in order to build S^0 the sum of all subsets of V^* needs to be calculated. Thus, after processing all the arcs in $\Theta(m)$ time and finding the D values, we build V^* in $\Theta(n)$ time along with the $\mathtt{sums}$ hash table, that can be built in $\Theta(2^{|V^*|})$ by dynamic programming:

$$\mathtt{sums}[S = \{s_1, \dots s_k\}] = \begin{cases} 0, & \text{if } S = \emptyset \\ D[s_1], & \text{if } |S| = 1 \\ \mathtt{sums}[\{s_2, \dots s_k\}] + D[s_1], & \text{otherwise.} \end{cases}$$

After $\mathtt{sums}$ is built, we can construct S^0 by simply iterating once again over all the subsets of V^* and adding zero-sum subsets to S^0. Then we clear pairs and non-atomic sets, call QUERY and we are done. This yields to a total complexity of $\Theta(m + n + 2^{|V^*|} + |S^0|^2 + 2^{|V^*|} \cdot |S^0|)$.

5 Practical behavior

As it can be seen from the time complexities of the operations, the behavior of the presented algorithms depends on the cardinalities of V^* and S^0 and their running times may vary from case to case.

We have made some experiments to compare our new algorithms and the static algorithm presented in [2]. We used the same 15 test cases which were used, when the problem was proposed in 2008 at the qualification contest of the Romanian national team. Figure 4 contains the structure of the graphs used for each test case.

In our first experiment we compared three algorithms: the old static algorithm based on dynamic programming from [2], our new static algorithm described in Section 4 and the dynamic graph algorithm based on the data structure presented in Section 3. For the third algorithm we called INSERTARC for each arc, then QUERY once in the end, after all arcs were added.

We executed each algorithm three times for each test case, and computed the average of the running times. The new static algorithm was the fastest in nine test cases, while the old static algorithm was the fastest in the remaining six

Test	n	m	$\lvert A_{\min}\rvert$	Short description
1	20	19	1	A path with the same weight on each arc
2	20	20	0	A cycle with the same weight on each arc
3	8	7	7	Minimal transaction graph equals to borrowing graph
4	20	19	19	Two connected stars
5	20	15	15	Yields to $D[i] = 2, \forall i = \overline{1,10}$, $D[i] = -1, \forall i = \overline{11,19}$ and $D[20] = -11$, maximizing the number of triples (zero-sets with cardinality three)
6	20	10	10	Yields to $D[i] = 99, \forall i = \overline{1,10}$, $D[i] = -99, \forall i = \overline{11,20}$, maximizing the number of pairs
7	20	19	12	A path with random weights having close values (50 ± 10)
8	20	20	10	A cycle with random weights having close values (50 ± 10)
9	10	100	7	Random graph with weights ≤ 10
10	12	100	9	Random graph with weights ≤ 10
11	15	100	11	Random graph with weights ≤ 10
12	20	100	14	Random graph with weights ≤ 10
13	20	19	15	A path with consecutive weights
14	20	30	15	Ten pairs, a path, a star and triples put together
15	20	100	15	Dense graph with weights ≤ 3

Figure 4: The structure of the test cases

test cases. Looking at the average running time over all the test cases, the new static algorithm was clearly the fastest with an average of 0.08 seconds. The old static algorithm came second with 0.64 seconds, and the dynamic algorithm third with 1.22 seconds. The difference between the last two is surprisingly small, taking into account that the dynamic algorithm may perform 2^n steps after each arc insertion. Running times are shown in Figure 5.

In the second experiment we used the same methodology to compare our new dynamic algorithm and the static algorithm presented in [2]. For the first algorithm the solution was recomputed from scratch each time an arc was read from the input file, and for the second after each InsertArc a Query was also executed. The dynamic algorithm was faster for eight test cases, recalculating from scratch was faster in the other seven cases. The average running time over all test cases is 23.6 seconds for the first algorithm and 41.9 seconds for

Test	Old static algorithm	New static algorithm	Dynamic algorithm	Improvement
1	0.018	**0.017**	0.018	3.7%
2	0.019	**0.007**	0.010	64.4%
3	0.013	**0.007**	0.007	43.9%
4	**0.036**	0.050	0.441	-38.1%
5	6.383	**0.551**	0.932	91.3%
6	**0.013**	0.138	0.488	-909.7%
7	**0.014**	0.034	0.169	-145.2%
8	**0.015**	0.019	0.084	-26.6%
9	0.014	**0.008**	0.010	45.5%
10	0.014	**0.007**	0.022	47.6%
11	0.015	**0.008**	0.106	44.4%
12	**0.016**	0.056	5.526	-242.8%
13	0.765	**0.079**	0.465	89.6%
14	**0.013**	0.048	1.527	-271.7%
15	2.274	**0.218**	8.553	90.3%

Figure 5: Average running times for the first experiment, all given in seconds. The best running times are bolded for each test. The last column shows the improvement of the new static algorithm over the old one in percentage. A negative value means, that no improvement was done.

the dynamic algorithm, mostly due to the last test case which runs for a long time compared to the others. Without taking into account the last test case the average running times are 1.05 and 1.28 seconds respectively.

By comparing the last two columns of the table depicted in Figure 6, one can see how powerful our heuristics are, reducing the cardinality of S^0 by several magnitudes in many cases. We can observe, that the dynamic algorithm usually performs slower than recomputing from scratch, when the size of S^0 before applying the heuristics is quite large, at least several hundreds. The reason behind this is probably the quadratic complexity of clearing non-atomic sets.

6 Conclusions

In this paper we introduced a new data structure capable of supporting arc insertions and deletions, node insertions and deletions in a dynamic borrow-

Test	Old static algorithm	Dynamic algorithm	Improvement	$\overline{\lvert V^* \rvert}$	$\overline{\lvert S^0 \rvert}$	$\overline{\lvert S^0 \rvert}$ after heuristics
1	0.079	**0.019**	76.0%	2	1	0
2	0.066	**0.013**	79.8%	1.9	0.95	0
3	0.029	**0.009**	69.6%	5	1	0.85
4	**0.109**	0.588	-437.8%	11	1	0.94
5	10.541	**1.515**	85.6%	11.66	7257	407.26
6	**0.040**	0.469	-1072.5%	11	25094.2	0
7	**0.063**	0.217	-243.6%	10.15	1035.63	3.57
8	**0.066**	0.142	-113.5%	10.75	801.3	7.7
9	0.294	**0.017**	94.1%	9.35	10.4	4.2
10	0.300	**0.046**	84.4%	11.28	40.28	13.17
11	0.329	**0.258**	21.6%	13.59	283.9	33.19
12	**1.575**	11.346	-620.0%	17.72	6002.9	189.38
13	1.101	**0.588**	46.5%	11	969.947	80.94
14	**0.105**	2.801	-2551.4%	16.46	1428.6	6.36
15	**340.719**	611.282	-79.4%	18.26	11790.8	9501.37

Figure 6: Average running times for the second experiment, all given in seconds. The best running times are bolded for each test. The third column shows the improvement of the dynamic algorithm over recomputing from scratch with the old static one in percentage. A negative value means, that no improvement was done. The last three columns show the average cardinality of V^*, S^0 and S^0 after applying both heuristics respectively.

ing graph, along with finding the minimal transaction graph. Using this data structure we developed a new static algorithm, which is faster than the one described in [2] in many cases and in average.

We find the running times of the dynamic algorithm and recomputing from scratch with the old static algorithm to be comparable on average. With a good heuristic, that runs in reasonable time, but still reduces the size of S^0 significantly, a better performance could be possible for the dynamic algorithm. Finding such a heuristic remains an open problem.

Our experiments are not meant to be an exact comparison among the algorithms, as the running time can greatly depend on the details of the implementation. Their purpose was just to get a general overview on the behavior of the various algorithms for different kind of graphs.

References

[1] C. Demetrescu, I. Finocchi, G. F. Italiano, Dynamic graph algorithms, in: *Handbook of Graph Theory* (ed. J. L. Gross, J. Yellen), CRC Press, Boca Raton, London, Washington, DC, 2004, pp. 985–1014. ⇒195

[2] C. Pătcaş, On the debts' clearing problem, *Studia Universitatis Babeş-Bolyai, Informatica*, **54,** 2 (2009) 109–120. ⇒ 192, 193, 197, 200, 201, 203

[3] C. Pătcaş, The debts' clearing problem's relation with complexity classes, *to appear* ⇒195, 197

[4] T. Verhoeff, Settling multiple debts efficiently: an invitation to computing science, *Informatics in Education*, **3,** 1 (2003), 105–126 ⇒197, 198

[5] *** BillMonk, http://www.billmonk.com ⇒195

[6] *** Expensure, http://expensure.com ⇒195

Received: May 16, 2011 • Revised: October 10, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 205–223

Cache optimized linear sieve

Antal JÁRAI
Eötvös Loránd University
email: ajarai@moon.inf.elte.hu

Emil VATAI
Eötvös Lorńd University
email: emil.vatai@gmail.com

Abstract. Sieving is essential in different number theoretical algorithms. Sieving with large primes violates locality of memory access, thus degrading performance. Our suggestion on how to tackle this problem is to use cyclic data structures in combination with in-place bucket-sort.

We present our results on the implementation of the sieve of Eratosthenes, using these ideas, which show that this approach is more robust and less affected by slow memory.

1 Introduction

In this paper we present the results obtained by implementing the sieve of Eratosthenes [1] using the methods described at the 8th Joint Conference on Mathematics and Computer Science in Komárno [3]. In the first section, the problem which is to be solved by the algorithm and some basic ideas about implementation and representation are presented. In Section 2, the methods to speed up the execution are discussed. In Section 3 the numerical data of the measurement of runtimes is provided on two different platforms in comparison with the data found at [4].

Given an array (of certain size) and a set P of (p, q) pairs, where p is (usually) a prime and $0 \leq q < p$ is the offset (integer) associated with p. A sieving algorithm for each $(p, q) \in P$ pair performs an action on every element of the array with a valid index $i = q + mp$ (for $m \geq 0$ integers).

Sieving with small primes can be considered a simple and efficient algorithm: start at q and perform the action for sieving, then increase q by p and repeat.

Computing Classification System 1998: F.2.1, E.1
Mathematics Subject Classification 2010: 11-04, 11Y11, 68W99
Key words and phrases: sieve, cache memory, number theory, primality

But when sieving with large primes, larger than the cache, memory hierarchy comes into play. With these large primes, sieving is not sequential (i. e. q skips great portions of memory), thus access to the sieve array is not sequential and this causes the program to spend most of its time waiting to access memory, because of cache misses.

1.1 Sieve of Eratosthenes

The sieve of Eratosthenes is the oldest algorithm for generating consecutive primes. It can be used for generating primes "by hand" but it is also the simplest and most efficient way to generate consecutive primes of high vicinity using computers. The algorithm is quite simple and well-known. Starting with the number 2, declare it as prime and mark every even number as a composite number. The first number, which is not marked is 3. It is declared as the next prime, so all the numbers divisible by 3 (that is every third number) is sieved out (marked), and so on.

1.2 Basic ideas about implementation

The program finds all primes in an interval $[u, v] \subset \mathbb{N}$ represented in a bit table. The program addresses the issue of memory locality by sieving the $[u, v]$ in subintervals of predefined size, called segments, which can fit in the cache, thus every segment of the sieve table needs to pass through the cache only once. For simplicity and efficiency the size of segment is presumed to be the power of two.

Because every even number, except 2, is a composite, a trivial improvement is to represent only the odd numbers, and cutting the size of the task at hand in half.

1.2.1 Input and output

The program takes three input parameters: the base 2 logarithm of the segment size denoted by l,[1] and instead of the explicit interval boundaries u and v, the "index" of the first segment denoted by f, and the number of segments to be sieved denoted by n, is given. This means, the numbers between $u = f2^{l+1} + 1$ and $v = (f + n)2^{l+1}$ are sieved. For simpler comparison with results from [4],

[1] cache size $\approx$ segment size $= 2^l$ bits $= 2^{l-3}$ bytes $= 2^{l+1}$ numbers represented, because only odd numbers are represented in the bit table.

the exponent of the approximate midpoint of the interval can also be given as an input parameter instead of f[2].

As for the output, the program stores the finished bit table in a file in the `/tmp` directory, with the parameters written in the filename.

1.3 Sieve table and segments

Definition 1 (Sieve table, Segments) *The sieve table* S *(for the above given parameters) is an array of* $n2^l$ *bits. For* $0 \le j < n2^l$*, the bit* S_j *represents the odd number* $2(f2^l + j) + 1 \in [u, v] = [f2^{l+1}, (f+n)2^{l+1}]$*.* S_j *is initialized to* 0 *(which indicates that* $2(f2^l + j) + 1$ *is prime);* S_j *is set to* 1*, if* $2(f2^l + j) + 1$ *is sieved out i.e. it is composite.*

The t*-th segment, denoted by* $S^{(t)}$ *is subtable of the* t*-th* 2^l *bits of the sieve table* S*, i.e.* $S_q^{(t)} = S_{t2^l+q}$ *for* $0 \le q < 2^l$ *and* $0 \le t < n$*.*

After p sieves at S_j, marking $2(f2^l + j) + 1$ as a composite, the next odd composite divisible by p is the $2(f2^l + j) + 1 + 2p = 2(f2^l + j + p) + 1$, so the index j has to be incremented only by p. That is, not representing the even numbers doesn't change the sieving algorithm, except the calculation of the offsets (described in Lemma 3).

1.4 Initialization phase

For every prime p, the first composite number not marked by smaller primes, will be p^2, i.e. sieving with p can start from p^2. To sieve out the primes in the $[u, v]$ interval, only the primes $p \le \sqrt{v}$ are needed. Finding these primes and calculating the q offsets, so that $q \ge 0$ is the smallest integer satisfying $p \mid 2(fs^s + q) + 1$ is the initialization phase. Presumably $\sqrt{v}$ is small ($\sqrt{v} < u$), and finding primes $p < \sqrt{v}$ (and calculating offsets) can be done quickly.

Definition 2 *The set of primes, found during the initialization of the sieve is called the base.* $P = \{p \text{ prime} : 2 < p \le \sqrt{v}\}$

2 Addressing memory locality

Because the larger the prime, the more it violates locality of memory access when sieving, the basic idea is to treat primes of different sizes in a different

[2]Of course, this just relieves the user from the tedious task of calculating f by hand for the given exponent of the midpoint of the interval, but internally the flow of the program was same as if f is given.

way, and process the sieve table by segments in a linear fashion, loading each segment in the cache only once and sieving out all the composites in it.

2.1 Medium primes

The primes $p < 2^l$ are *medium primes*. Segment-wise sieving with medium primes is simple: (p, q) prime-offset pairs with $p \leq 2^l$ and $0 \leq q < p$ are stored. Each prime marks at least one bit in each segment. For each prime p, starting from q, every p-th bit has to be set, by sieving at the offset q and then incrementing it to $q \leftarrow q + p$ while $q < 2^l$. Now q would sieve in the next segment, so the offset is replaced with $q - 2^l$. The first offset for a prime p and the given parameters f and l can be found using the following Lemma.

Lemma 3 *For each odd prime* p *and positive integers* l *and* f*, there is a unique offset* $0 \leq q < p$ *satisfying:*

$$p \mid 2(f2^l + q) + 1 \tag{1}$$

Proof. Rearranging (1) gives $f2^{l+1} + 2q + 1 = mp$ for some m. The integer m has to be odd, because the left hand side and p are odd. The equation can further be rearranged to a form, which yields a coefficient and something similar to a reminder:

$$f2^{l+1} = (m-1)p + (p - (2q+1)).$$

The last term is even, so if the remainder $r = f2^{l+1} \mod p$ is even, then $q = (p-r-1)/2$ satisfies $0 \leq q < p$ and (1). If r is odd, then $q = (2p-r-1)/2$ satisfies $0 \leq q < p$ and (1). Because r is unique, q is also unique. □

2.2 Large primes

The primes $p > 2^l$ are *large primes*. These primes "skip" segments i.e. if a bit is marked in one segment by the large prime p, (usually) no bit is marked by the prime p in the adjacent segment. The efficient administration of large primes is based on the following observation:

Lemma 4 *If the prime* p*, which satisfies the condition* $k2^l \leq p < (k+1)2^l$ *(for some integer* $k \geq 0$*), marks a bit in* $S^{(t)}$*, then the next segment where the sieve marks a bit (with* p*) is the segment* $S^{(t')}$ *for* $t' = t + k$ *or* $t' = t + k + 1$.

Proof. If the prime p marks a bit in $S^{(t)}$ then it is the $S_q^{(t)}$ bit, for some offset $0 \le q < 2^l$. The S_{t2^l+q} bit is marked first, then the S_{t2^l+q+p} bit, so the index of the next segment is $t' = \lfloor (t2^l + q + p)/2^l \rfloor$, thus

$$t + k = \frac{t2^l + 0 + k2^l}{2^l} \le \underbrace{\left\lfloor \frac{t2^l + q + p}{2^l} \right\rfloor}_{=t'} < \frac{t2^l + 2^l + (k+1)2^l}{2^l} = t + k + 2.$$

□

2.3 Circles and buckets

The goal is, always to have the right primes available for sieving at the right time. This is done by grouping primes of the same magnitude together in so called circles, and within these circles grouping them together by magnitude of their offsets in so called buckets.

Definition 5 (Circles and Buckets) *A circle (of order k, in the t-th state) denoted by $C^{k,t}$ is sequence of $k+1$ buckets $B_d^{k,t}$ (where $0 \le d \le k$). Each bucket contains exactly those (p, q) prime-offset pairs, which have the following properties:*

$$k2^l < p < (k+1)2^l \tag{2}$$

$$0 \le q < \max\{p, 2^l\} \tag{3}$$

$$p \mid 2\big((f + t + d - b + k + 1)2^l + q\big) + 1 \qquad \text{if } 0 \le d < b \tag{4}$$

$$p \mid 2\big((f + t + d - b)2^l + q\big) + 1 \qquad \text{if } b \le d \le k \tag{5}$$

where $b = t \mod (k+1)$ is the index of the current bucket.

$(p, q) \in C^{k,t}$ means that there is an index $0 \le d \le k$ for which $(p, q) \in B_d^{k,t}$ and $p \in C^{k,t}$ means that there is an offset $0 \le q < 2^l$ for which $(p, q) \in C^{k,t}$.

As the state t is incremented b changes from 0 to k cyclically. This can be imagined as a circle turning through $k+1$ positions, justifying its name. Also, each bucket contains all the right primes with all the right offsets, that is when it becomes the current bucket, it will contain exactly those prime-offsets which are needed for sieving the current segment.

Circles and buckets can be defined for arbitrary $k, t \in \mathbb{N}$, but only $0 \le k \le \lfloor \max P/2^l \rfloor = K$ and $0 \le t < n$ are needed.

Theorem 6 *For each* $p \in P$ *there is a unique* $0 \le k \le K$ *and for each state* t*, a unique* $0 \le d \le k$ *and offset* q *such that* $(p, q) \in B_d^{k,t}$.

Proof. For every p prime, dividing (2) with 2^l gives $k = \lfloor p/2^l \rfloor$, and if $p \in P$, then $p \le \max P$, so $\lfloor p/2^l \rfloor \le \lfloor \max P/2^l \rfloor = K$, therefore $0 \le k \le K$. For each p, (2) is true independent of the state t.

For each t, a unique offset q satisfying (3) and a unique index $0 \le d \le k$ satisfying (4) and (5) has to be found . It should be noted that the precondition of (4) and (5) are mutually exclusive, that is an index d satisfies only one of the two preconditions, and only that one has to be proven.

Medium primes, that is $p < 2^l$ is the special case of $k = 0$. $C^{0,t}$ has only one bucket with the index $d = b = 0$, so the precondition of (4) is always false. (3) is equivalent to $0 \le q < p$ since $p < 2^l$. (5) is equivalent to $p \mid 2\big((f+t)2^l+q\big)+1$ because $0 \le b \le d$, that is $b = 0 = d$. Lemma 3 for the prime p and integers l and $t + f$ shows that there is an integer q which satisfies (3) and (5).

For $p > 2^l$, the proof is by induction. If $t = 0$ is fixed, then $b = 0$ is the current bucket's index. The precondition of (4) is false and (5) is equivalent to $p \mid 2\big((f+d)2^l+q\big)+1$. Lemma 3 for the prime p and integers l and f gives a q' which satisfies $p \mid 2(f2^l+q')+1$ and $0 \le q' < p$. Dividing q' by 2^l gives $q' = d2^l + q$, where d and q are unique and satisfy (3) and (5).

If the statement holds for $t \ge 0$, then there is an index $0 \le d \le k$ and an offset $0 \le q < 2^l$ such that $(p, q) \in B_d^{k,t}$. The current bucket is $b = t \mod (k+1)$, and the statement will be proven for the next state $t' = t + 1$ with $b' = (b+1) \mod (k+1)$ as the index of the "next" current bucket.

The first case is $d \ne b$. It can be shown that incrementing the state, the prime remains in the same bucket with the same offset, i.e. $(p, q) \in B_d^{k,t'}$. If $b < k$, then $b' = b + 1 \le k$ holds, that is $t' - b' = (t+1) - (b+1) = t - b$ so (4) and (5) remain the same, except for the preconditions. But since $d \ne b$, $0 \le d < b < b'$ or $b < b' \le d$ will still remain true. If $b = k$, then $0 \le d < b = k$ and $b' = 0 = b - k$, so the precondition of (4) is false, and (5) becomes:

$$p \mid \big((f + (t+1) + d - (-k))2^l + q\big) + 1 = \big((f + t + d + k + 1)2^l + q\big) + 1$$

for $0 = b' \le d$. Since (4) was true for d and t, now (5) is true for d and $t+1$.

The second case is when $d = b$, and it can be shown that incrementing the state the prime remains in the same bucket or goes into the previous one (modulo $(k+1)$) with a different offset. If $d = b$, d satisfies the precondition of (5), that is $p \mid 2\big((f+t)2^l+q\big)+1$ is true. The next odd number divisible by p can be obtained by incrementing the offset by p. As seen in Lemma 4 $q + p$

can be written as $q + p = k'2^l + q'$, where $k' = k$ or $k + 1$ and $0 \leq q' < 2^l$. With incrementing the offset by p for t (5) gives:

$$p \mid 2\big((f + t + k')2^l + q'\big) + 1. \tag{6}$$

Let d' be $d + (k' - 1) \mod (k+1)$, that is $d' \equiv d \pmod{k+1}$ or $d' \equiv d - 1 \pmod{k+1}$. The precondition of (5) for the next state t' is true if $b = k$ (then $b' = 0$ and $d' = k$ or $k - 1$) or if $b = 0$ and $k' = k$ (then $b' = 1$ and $d' = k$). If these values are plugged in (5) for t', i.e. $p \mid \big((f + t' + d' - b')2^l + q'\big) + 1$, equation (6) is obtained, which is true. The preconditions of (4) are satisfied for every other case, that is, when $0 < d = b < k$ (then $0 \leq d' < b' \leq k$) or $b = 0$ and $k' = k + 1$ (then $b' = 1$ and $d' = 0$). Again, by plugging these values in (4) for t', that is $p \mid \big((f + t' + d' - b' + k + 1)2^l + q'\big) + 1$ equation (6) is obtained, which is also true. □

As a consequence, if it doesn't cause any confusion, the state may be omitted from the notation, because each prime with its offset is maintained only for the current state. As the program iterates through states, the primes may "move" between buckets, and offsets usually change.

The following Corollary shows, that sieving with circles and buckets sieves out all composites marked by large primes (sieving with medium primes is more or less trivial).

Corollary 7 *For each* $p > 2^l$ *and odd* $i' \in [u, v]$ *satisfying* $p \mid i'$ *there exists a unique state* t *and an offset* q*, so that* (p, q) *is in the current bucket of the circle to which* p *belongs to.*

Proof. Let i' be represented by S_i for $0 \leq i < n2^l$, that is $i' = 2(f2^l + i) + 1$. The statement is true for $t = \lfloor i/2^l \rfloor$, because then $i = t2^l + q$, $b = t \mod (k+1)$, and substituting d with b in (5), the equation $p \mid 2\big((f + t)2^l + q\big) + 1 = 2(f2^l + i) + 1$ is obtained. □

The proof of Theorem 6 could have been simpler, but the proof by induction gives some insight on how the circles and buckets work and behave, giving some idea about how to implement them. This behavior is explicitly stated in the following Corollary.

Corollary 8 *For each state* t*, each order* k*,* $b = t \mod (k + 1)$ *and* $b' = (t + k) \mod (k + 1)$*, if* $(p, q) \in B_b^{k,t+1}$ *then* $(p, q') \in B_b^{k,t}$ *for some offset* q'*, and* $B_{b'}^{k,t} \subset B_{b'}^{k,t+1}$ *and for every* $b \neq d \neq b'$ *and* $d \neq d'$ $B_d^{k,t} = B_d^{k,t+1}$.

The first statement says that with respect *only* to primes $B_b^{k,t+1} \subset B_b^{k,t}$.

Proof. The index b' refers to the current bucket in the previous state and as seen in the remarks in the proof of Theorem 6, iterating from state t to $t+1$ leaves the buckets with indexes $d \neq b$ and $d \neq b'$ untouched, and some primes with new offsets are left in the current bucket while others are put in the previous one. □

2.4 Modus operandi

The goal is to perform a segment-wise sieve:

Medium primes belonging to C^0 are a special case, and they sieve at least once in a segment. The t-th state of C^0 contains all medium primes with the smallest offsets for sieving in the $S^{(t)}$ segment. For a prime in C^0, after sieving with it the offset is replaced with the smallest offset for sieving the next segment $S^{(t+1)}$. After sieving with all medium primes, C^0 is in the $t+1$-th state. This is implemented in a single loop, iterating through all medium primes.

The circle C^k ($k > 0$), in the t-the state, for primes between $k2^l$ and $(k+1)2^l$, has the prime-offset pairs, needed for sieving $S^{(t)}$ in the current bucket. After sieving with all these primes, the circle is in it's next state, with offsets replaced, and some primes moved to the previous bucket, ready for sieving $S^{(t+1)}$. Sieving large primes is implemented via two embedded loops, the outer iterating through circles by their order, covering all primes, and the inner loop iterating through the primes of the current bucket of the current circle.

The above two procedures are called in a loop for segment $S^{(t)}$, iterating from $t = 0$ to $n - 1$. Corollary 7 shows, that the primes for sieving the t-th segment are in the current buckets of circles in t-th state, so this procedure performs the sieve correctly.

2.5 Implementation

For sequential access, all prime-offset pairs, buckets and circles are stored as linear arrays: the array of prime-offset pairs is denoted by $(\hat{p}_i, \hat{q}_i)$, the array of buckets denoted by $\hat{b}_i$ and the array of circles denoted by $\hat{c}_i$ ($i \in \mathbb{N}$).

2.5.1 Array of circles

$\hat{c}_k$ is the data structure (C `struct`) implementing the circle C^k. It is responsible for most of the administration of the associated primes and buckets. Of course memory to store $K+1$ circles is allocated.

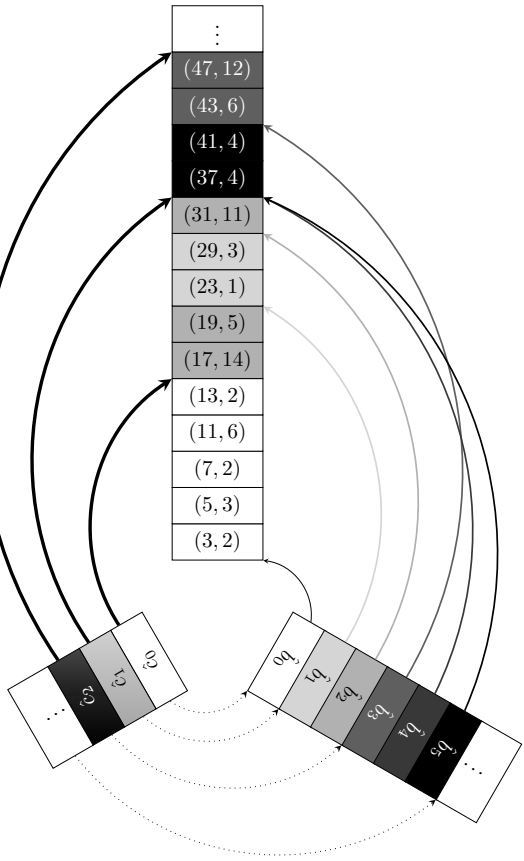

Figure 1: Array of circles, buckets and primes

In the implementation, primes of one circle are a continuous part of the array of primes. It was convenient to store the end-pointers of circles, i.e. a pointer to the prime after the last prime in the circle. So the medium primes are all primes before the up to but not including the prime at the end-pointer of $\hat{c}_0$, and all primes in C^k are the primes from the end-pointer of $\hat{c}_{k-1}$ up to but not including the prime pointed to by the end-pointer of $\hat{c}_k$. Since the primes are generated in an ascending order, these end-pointers can be determined easily, and they don't change during the execution of the program.

The circle $\hat{c}_k$ maintains the index of the current bucket. It is incremented by one (modulo $k+1$), that is $b \leftarrow b+1$, if $b < k$ or $b \leftarrow 0$ if $b = k$ assignment is performed after sieving with primes from the circle. Circle need to maintain

the index of the broken bucket explained in section 2.5.4.

The circle $\hat{c}_k$ could also maintain a pointer, to the first bucket B_0^k represented by $\hat{b}_{k(k+1)/2}$ in the buckets array, but it is not necessary because it can be calculated from k. There is a more efficient solution if the circles are processed with ascending orders: The starting bucket of $\hat{c}_1$ is $\hat{b}_1$ and this is stored as a temporary pointer. For every circle $\hat{c}_{k+1}$ the starting bucket is at $k+1$ buckets after the first bucket of the previous circle $\hat{c}_k$, so when finished with circle $\hat{c}_k$, this pointer has to be increased by $k+1$.

In Figure 1 the value l is 4 so the cache size is 16. C^0 has a white background as well as the bucket and primes associated with it. C^1 is light gray with black text, with two different shades for the two buckets and primes in them, and in a similar way C^2 is black with white text and slightly lighter shades of gray for the buckets and primes. The end-pointers are drawn as thick arrows, indicating that they don't move during the execution. The dotted lines are the calculable pointers to the first buckets.

2.5.2 Array of buckets

All buckets are stored consecutively in one array, i.e. the bucket B_d^k of circle C^k for $0 \le d \le k$ is represented by $\hat{b}_{k(k+1)/2+d}$. There are $K+1$ circles, with $k+1$ buckets for each $0 \le k \le K$, so memory for storing $(K+1)(K+2)/2$ buckets needs to be allocated.

The value of each $\hat{b}_d$ is the index (`uint32_t`) of the first prime-offset pair which belongs to the bucket $\hat{b}_d$. Primes that belong to one bucket are also in a continuous part of the primes array, so the bucket B_d^k contains the primes $\hat{p}_i$ (and the associated offsets $\hat{q}_i$) for $\hat{b}_{d'} \le i < \hat{b}_{d''}$ where $d' = k(k+1)/2+d$ and $d'' = k(k+1)/2+(d+1) \mod (k+1)$. The broken bucket is an exception to this. Empty buckets are represented with entries in the buckets array having the same value, i.e. B_d^k is empty if $\hat{b}_{d'} = \hat{b}_{d''}$ for $d' = k(k+1)/2+d$ and $d'' = k(k+1)/2+(d+1) \mod (k+1)$, e.g. $\hat{b}_4$ is empty in Figure 1.

Buckets are set during the initialization, but change constantly during sieving. First, using Lemma 3 an offset $0 \le q' < p$ is found for each p prime. All prime-offset pairs of the circle C^k are sorted in ascending offsets. Similarly, as primes are collected in circles, within one circles the offsets are collected in buckets. B_d^k contains all prime-offset pairs, so that $d2^l \le q' < (d+1)2^l$, but instead q', $0 \le q = q' - d2^l < 2^l$ is stored.

For each circle C^k, for each pair $(p,q) \in B_b^k$, the bit with index q in the current segment is set. After that, the new offset $q' = q+p$ is calculated, which is, because of Lemma 4, either $k2^l \le q' < (k+1)2^l$ or $(k+1)2^l \le q' < (k+2)2^l$.

In the former case $q' - k2^l$ is stored in the previous bucket (modulo $k+1$), or in the latter case $q' - (k+1)2^l$ is stored in the current bucket, as described in Corollary 8.

This can be implemented efficiently by keeping copies of a prime from the beginning and another prime from the end of the bucket. That way, in the first case (k segments were skipped), the prime-offset pair in the beginning of the bucket is overwritten and the value in the buckets array indicating the beginning of the bucket is incremented, thus putting the replaced, new prime-offset pair, in the previous bucket. In the second case ($k+1$ segments were skipped), the prime-offset pair in the end of the bucket is overwritten with the new prime-offset pair and it stays in the current bucket. After replacing a pair in one of the ends (the beginning or the end) of the bucket, the next prime is read from that end of the bucket, that is from the next entry, closer to the center of the bucket.

2.5.3 Array of primes

The array of primes contains all primes (medium and large), with the appropriate offsets, needed for sieving. The prime-offset pairs are stored as two 32 bit unsigned integers (two `uint32_t`s in a `struct`). Enough memory to store about $\frac{\sqrt{v}}{\log\sqrt{v}}$ pairs is allocated.

The array of primes is filled during the initialization phase. The values of the offsets q change after finishing a segment. Sometimes pairs from the end and the beginning of a bucket are swapped (as explained earlier), but this is all done in-place, that is, the array itself does not need to be modified or copied, just the values stored. All primes that belong to one circle as well as those that belong to one bucket (except the broken bucket) are stored in a coherent and continuous region of memory.

2.5.4 Broken bucket

For the circle C^k, the index of the broken bucket is $r = \max\{d : \hat{b}_{k(k+1)/2+d} = M\}$, where $M = \max\{\hat{b}_d : k(k+1)/2 \le d < (k+1)(k+2)/2\}$. The primes which belong to this bucket, are the ones from the index $\hat{b}_{k(k+1)/2+r}$ and up to but not including the prime at the end-pointer of $\hat{c}_k$ and the primes from the end-pointer of $\hat{c}_{k-1}$ up to but not including the prime with the index $\hat{b}_{k(k+1)/2+r'}$ where $r' = (r+1) \mod (k+1)$, e.g. $\hat{b}_2$ in Figure 1. This idea also justifies the name circles, because logically the next prime after the end-pointer of a circle is the first prime of the circle. When the broken bucket is

not actually broken, the value of $\hat{b}_{k(k+1)/2+r'}$ is set to the index of the prime at the end-pointer of $\hat{c}_{k-1}$, e.g. $\hat{b}_3$ in Figure 1.

Every circle has a broken bucket and this has to be stored as a variable for each circle. The fact, that this can not be omitted is not trivial, but if all primes of a circle are one bucket, then all other buckets in that circle are empty. Because empty buckets are represented by having the same value as the following bucket, all buckets in that circle, that is all entries of $\hat{b}_d$ which represent the buckets of that circle, will have the same value. In this situation the program can't decide which buckets are empty and which one contains all the primes.

The broken bucket also moves around. When sieving with a bucket, its lower boundary is incremented. If sieving with the broken bucket, when the beginning of the bucket moves past the end of the circle (and jumps to the beginning), the previous bucket (modulo $k+1$) becomes the new broken bucket.

3 Speeding up the algorithm

The roughly described implementation of sieving can be further refined to gain valuable performance boosts.

3.1 Small primes

Sieving with primes $p < 64$ can be sped up by not marking individual bits, but rather applying bit masks. The subset of medium primes below 64 are called *small* primes.

The *AMD64*[3] architecture processors with *SSE2* extension, have sixteen 64-bit general purpose R registers, and sixteen 128-bit XMM registers. For sieving with small primes, the generated 64bit wide bit masks are loaded in these registers and `or`-ed together, to form the sieve table with small primes applied to it. The masks are then `shift`-ed, to be applied to the next 64 bits for R registers and 128 bits for XMM registers.

This is of course done in parallel, sieving by 128 bits at a time. The XMM registers first 64 bits are loaded from the memory at the beginning of sieving of a segment, and the last 64 bits are `shift`-ed (just like the R registers are shifted "mod 64"). There is two times as much sieving with the R registers than with the XMM registers.

With the first four primes "merged" into two, all the small primes can fit

[3] AMD™ is a trademark of Advanced Micro Devices, Inc.

in the R and XMM registers, so the only memory access is sequential and done once when starting and once when finished sieving. The primes 3 and 11 are merged into 33, that is, the masks of 3 and 11 are combined at the initialization, and the shifting needs to be done as if 33 was the prime for sieving, because the pattern repeats after 33 bits. 5 and 7 are merged into 35 and treated similarly.

3.2 Medium primes

As described earlier, for (p, q) pairs with medium primes, sieving starts from q by increasing it by p after sieving, until $q \geq 2^l$. Then the sieving is finished for that segment, and the sieving of the next segment starts from $q' = q - 2^l$. There are two methods in which this algorithm can be sped up.

3.2.1 Wheel sieve

In the special case of the sieve of Eratosthenes, the "wheel" algorithm (described in [5]) can be used to speed up the program. In some sense, it is an extension of the idea of not sieving with number 2.

Let W be the set of the first few primes and $w = \prod_{p \in W} p$. Sieving with the primes from W, sieves out a major part of the sieve table, and these bits can be skipped. Basically, when sieving with a prime $p \notin W$, the number i needs to be sieved (marked) by p, only if it is relative prime to w, that is, if i is in the reduced residue system modulo w denoted by W' (if $i \notin W'$ some prime from W will mark it).

Let $w_0 < \cdots < w_{\varphi(w)-1}$ be the elements of W', and Δ_s the number of bits that should be skipped, after sieving the bit with index congruent to w_s, that is $\Delta_i = (w + w_{(i+1) \bmod \varphi(w)} - w_i) \bmod w$. When i is sieved out by $p \notin W$, instead of sieving $i+p$ next, the program can skip to $i+\Delta_s p$ if $i \equiv w_s \pmod{w}$.

In the implementation, $W = \{2, 3, 5\}$, but 2 is "built in" the representation and this complicates thing a little bit: $w = 15$, $\varphi(w) = 8$ and $w_0 = 0$, $w_1 = 3$, $w_2 = 5$, $w_3 = 6$, $w_4 = 8$, $w_5 = 9$, $w_6 = 11$, $w_7 = 14$ are used (instead of $w = 30$ and 1, 7, 11, 13, 17, 19, 23, 29 for w_s). For each prime the offset q is initialized to the value $q' + mp$, where q' the offset found using Lemma 3 and m is the smallest non-negative integer, so that $f2^l + q \equiv w_s \pmod{w}$ for some $0 \leq s \leq 7$.

Let p^{-1} be the inverse of p modulo 15, and x a non-negative integer so that:

$$f2^l + q + xp \equiv 7 \pmod{15}. \tag{7}$$

Note that the residue class represented by 7 is 15, and that is the class divisible both by 3 and 5, and it is in a sense the "beginning" of the pattern generated by the primes in W when sieving. (7) states that after x times sieving (regularly) with p, the offset is at the "beginning" of the pattern, that is, in the residue class represented by 7, so $y = 7 - x$ is the residue class in which q actually is. $x \equiv (7 - (f2^l + q))p^{-1} \pmod{15}$ can be obtained from (7) by multiplying it with p^{-1}.

There is an index $0 \leq s \leq 7$, so that $w_s = y$. The index s, indicating where in the pattern is the offset q, is stored beside each (p, q) pair. Before sieving with p, the array $\Delta_0 p, \ldots, \Delta_7 p$ is generated in memory, and a pointer is set to $\Delta_s p$. After marking a bit, the offset is incremented by the values found at that pointer, and the pointer is incremented modulo 8, which can be implemented very efficiently with a logical `and` operation and a bit mask. Also all prime-offset pairs are stored on 64 bits and medium primes are $p < 2^l$ (where l is never more than 30), so at least 4 bits are not used where the index $0 \leq s \leq 7$ can fit.

3.2.2 Branch misses

Another speed boost can be obtained by treating the *larger medium primes* (near to 2^l) differently. This idea is somewhat similar to the one used with circles, because it is based on the observation that, if $\frac{2^l}{(k+1)} < p < \frac{2^l}{k}$, then p sieves k or $k+1$ times in one segment ($0 < k \in \mathbb{N}$). There is a different procedure g_k, for each of the first few values of k (e.g. $0 < k < 16$). g_k iterates the offset $k+1$ time, with the last iteration implemented using conditional move (`cmov`) operations. So, for each k, primes $\frac{2^l}{(k+1)} < p < \frac{2^l}{k}$ are collected in a different array, and the procedure g_k is invoked for each prime in that array. Having fixed number of iterations with a conditional move is faster then a branch miss, because the CPUs instruction stream is not interrupted.

3.3 Large primes

The sieving with large primes is roughly described above. Sieving with one prime, putting it back, with the new offset, and modifying the bucket boundary can be accomplished with only about 15 assembly instructions using conditional moves (`cmov`). This is very efficient, but other techniques can also be applied to reduce execution time.

3.3.1 Interleaved processing

Because the order in which the primes are processed doesn't matter, the memory latency can be hidden by processing primes from both ends of the bucket. As described earlier, for each bucket, two prime-offset pairs are loaded from the beginning and end of the current bucket and one of them is processed. To hide memory latency, the next prime is loaded into place of the processed prime *while* the other one is being processed. "Processing a prime" covers the following steps: marking the bit at the offset q; determining if $q+p$ skips k or $k+1$ segments; calculating the new offset $q' \leftarrow q+p-k2^l$, replacing the pair at the beginning of the bucket and incrementing the bucket's lower boundary, for the former case; or in the latter case, decrementing the pointer indicating the finished primes at the top of the bucket, after replacing the pair at the end of the bucket with the offset $q' \leftarrow q+p-(k+1)2^l$. Processing of one prime is about 15-18 assembly instructions, which is approximately 5-6 clock cycles on today's processors, about the same time needed for the other prime to be loaded in the registers.

3.3.2 Broken bucket and loop unrolling

The well-known technique of loop unrolling can efficiently be used for processing primes-offset pairs. The core of the loop described above, which processes two primes terminates when the difference between the pointer from the beginning and end of the bucket becomes zero. With right `shift` and a logical `and` instructions, the quotient a and remainder r of this difference when divided by 2^h can be obtained (e.g. $h = 4$ or 5). Then the loop core can be executed a times in batches of 2^h runs, and afterward r times, thus reducing the time spent on checking if the difference is zero.

The loop unrolling of the broken bucket is a bit trickier, but manageable. Let δ_1 denote the difference between the beginning of the bucket and the end of the circle, and δ_2 the difference between the beginning of the circle and the end of the bucket. The difference used for unrolling, as described above, would be $\delta_1 + \delta_2$ but the when modifying the pointers after processing a prime, it would have to be checked, if it moves past the beginning or end of the circle (to jump to the other side). Instead, the unrolling is applied to $\min\{\delta_1, \delta_2\}$. Since it can't be predicted if the beginning or end pointer is going to be modified, the values of δ_1 and δ_2, the maximum, quotient a and reminder r have to be reevaluated after each batch, until one of the pointers "jump" to the other side. Then the bucket will no longer be broken, so the simpler unrolling described above can be applied.

4 Results

The program was run on (a single core of) two computers referred to by their names *lime* and *complab07*. The goal was to supersede the implementation found in the speed comparison chart of [4], but the results can not be compared directly, because of the differences in hardware. Our implementation, running on *lime* would come in 7th and *complab07* the 16th in the speed comparison chart, but with significantly slower memory.

e	lime	cl07	[a0F80]	[a0FF0]	[i06E8]	[a0662]
1e12	1.45	2.09	0.57	0.68	1.28	1.07
2e12	1.45	2.10	0.64	0.75	1.37	1.17
5e12	1.45	2.27	0.74	0.85	1.48	1.29
1e13	1.45	2.28	0.80	0.92	1.57	1.38
2e13	1.46	2.38	0.86	0.99	1.66	1.47
5e13	1.45	2.46	0.95	1.08	1.76	1.59
1e14	1.46	2.5	1.01	1.14	1.85	1.67
2e14	1.45	2.58	1.08	1.21	1.94	1.76
5e14	1.68	2.70	1.16	1.29	2.04	1.87
1e15	1.63	2.80	1.22	1.36	2.11	1.96
2e15	1.71	2.86	1.28	1.42	2.19	2.06
5e15	1.83	2.95	1.37	1.50	2.29	2.20
1e16	1.89	3.04	1.42	1.56	2.37	2.32
2e16	1.95	3.13	1.49	1.63	2.45	2.47
5e16	2.03	3.25	1.58	1.75	2.57	2.72
1e17	2.08	3.34	1.64	1.86	2.67	2.93
2e17	2.15	3.45	1.72	2.02	2.79	3.23
5e17	2.22	3.60	1.84	2.21	2.96	3.66
1e18	2.29	3.75	1.99	2.39	3.13	4.03
2e18	2.33	0	2.26	2.61	3.31	4.52

Table 1: Execution times in seconds for intervals of $10^9 \approx 2^{30}$ with the midpoint at 10^e

Compared to the 666MHz DDR2 memory of *lime*, the first five or so computers from the speed comparison chart have memory speeds of 800MHz and above, and with a better memory our implementation could probably compete

lime	2000MHz Intel Core2 Duo (E8200) model 23, stepping 6, DDR2 666MHz
cl07	1595MHz AMD Athlon64 3500+, model 47, stepping 2, DDR 200MHz
a0F80	2600MHz 6-Core AMD Opteron (Istanbul), model 8, stepping 0, DDR2
a0FF0	2210MHz Athlon64 (Winchester), model 15, stepping 0, DDR 333
i06E8	1830MHz T2400 (Core Duo), model 14, stepping 8, DDR2 533
a0662	1669MHz Athlon (Palomino), model 6, stepping 2, DDR 333

Table 2: The CPU and memory configurations of the computers used for measurements

better. But the real improvement can be seen, when running on older hardware, like *complab07*. With the slow memory of 200MHz, the plot in Figure 2, is much flatter and closer to the theoretical speed of $n \log\log n$ than for example the similar *i0662* with a faster 333MHz RAM.

It should also be noted, that the major part of execution is spent on sieving with medium primes and more optimization is desired out of that part of the algorithm. We also had some unexpected difficulties optimizing assembly code for the Intel processors, due to confusing documentation and slow execution of the `bts` (bit test set) instruction.

For *lime*, *complab07* and some computers from [4], Table 1 shows the time needed to sieve out an interval (represented by $2^{30} \approx 10^9$ bits, with its midpoint at 10^e). This data is plotted out in Figure 2: values of e are represented on the horizontal, execution times in seconds on the vertical axis.

5 Future work

The program was originally written for verifying the Goldbach conjecture, but only the sieve for generating the table of primes was finished and measured because that takes up the majority of the work for the verification. The completion of the verification application would be desirable. Also the current implementation supports sieving with primes only up to 32 bits, on current architectures, the implementation of sieving with primes up to 64 bits would not be a problem.

Most of the techniques described here (except the wheel algorithm), especially the use of circles and buckets can be applied for a wider range of sieving algorithms. For example, in [2] a similar attempt is made to exploit

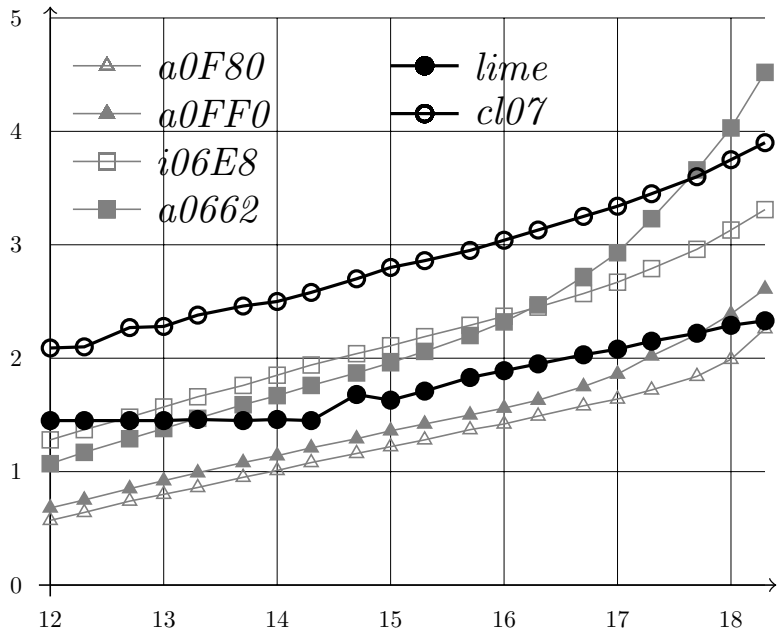

Figure 2: Speed comparison chart

the cache hierarchy, but the behavior of the large primes is more predictable with our method, and even an implementation for processors not designed for sieving algorithms is possible. Therefore the multiple polynomial quadratic sieve, on the Cell Broadband Engine Architecture[4], with 128K byte cache (i.e. Local Store) controlled by the user via DMA, can be implemented efficiently. Further performance can be gained by combining buckets and circles with parallel processing: sieving with different polynomials on different processors for MPQS-like algorithms, and a segment-wise pipeline-like processing for algorithms similar to the sieve of Eratosthenes.

Acknowledgements

The Project 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).

[4]Cell Broadband Engine™ is a trademark of Sony Computer Entertainment Incorporated

References

[1] D. M. Bressoud, *Factorization and Primality Testing*, Springer-Verlag, New York, 1989. ⇒205

[2] J. Franke, T. Keinjung, *Continued fractions and lattice sieving*, Proceeding SHARCS 2005, `http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf`. ⇒221

[3] A. Járai, E. Vatai, Cache optimized sieve, *8th Joint Conf. on Math and Comput. Sci. MaCS 2010, Selected papers,* Komárno, Slovakia, July 14–17, 2010, Novadat, 2011, pp. 249–256. ⇒205

[4] T. Oliveira e Silva, Goldbach conjecture verification, 2011, `http://www.ieeta.pt/~tos/goldbach.html`. ⇒205, 206, 220, 221

[5] P. Pritchard, Explaining the wheel sieve, *Acta Inform.*, **17,** 4 (1982) 477–485. ⇒217

Received: October 2, 2011 • Revised: November 8, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 224–229

Lower bounds for finding the maximum and minimum elements with k lies

Dömötör PÁLVÖLGYI
Loránd Eötvös University Budapest
Institute of Mathematics
email: dom@cs.elte.hu

Abstract. In this paper we deal with the problem of finding the smallest and the largest elements of a totally ordered set of size n using pairwise comparisons if k of the comparisons might be erroneous where k is a fixed constant. We prove that at least $(k+1.5)n+\Theta(k)$ comparisons are needed in the worst case thus disproving the conjecture that $(k+1+\epsilon)n$ comparisons are enough.

1 Introduction

Search problems with lies have been studied in many different settings (see surveys Deppe [2] and Pelc [5]). In this paper we deal with the model when a fixed number, k, of the answers may be false, which we call lies. There are also several models depending on what kind of questions are allowed as well, the most famous being the Rényi-Ulam game. In this paper we deal with the case when we are given n different elements and we can use pairwise comparisons to decide which element is bigger from the two.

The problem of finding the maximum (or the minimum) element with k lies was first solved by Ravikumar et al. [7]. They have shown that $(k+1)n-1$ comparisons are necessary and sufficient. The topic of this paper is finding the maximum and the minimum. If all answers have to be correct then the minimum number of comparisons needed is $\lceil \frac{3n}{2} \rceil - 2$ (see [6]). Aigner in [1]

Computing Classification System 1998: F.2.2
Mathematics Subject Classification 2010: 68P10
Key words and phrases: search, lies

proved that $(k + \Theta(\sqrt{k}))n + \Theta(k)$ comparisons are always sufficient*. It was proved by Gerbner et al. [3] that if $k = 1$, then $\frac{87n}{32} + \Theta(1)$ comparisons are necessary and sufficient. We also made the conjecture that for general k, there is an algorithm using only $(k + 1 + c_k)n$ comparisons where c_k tends to 0 as k tends to infinity. Hoffmann et al. [4] showed that $(k + 1 + C)n + O(k^3)$ comparisons are sufficient for some absolute constant C (whose value is less than 10 but no attempts to optimize it were made yet). Until now the best lower bound on c_k was $\Omega((1 + \sqrt{2})^{-k})$ by Aigner [1]. The main result of this paper is the following theorem.

Theorem 1 *At least* $\lceil (k + 1.5)(n - 1) - 0.5 \rceil = (k+1.5)n+\Theta(k)$ *comparisons are needed in the worst case to find the largest and the smallest element if there might be* k *erroneous answers.*

This bound is tight for $k = 0$ (see Theorem 4) but not for $k = 1$ as shown in [3] and using a slightly more involved argument than the one presented here it is easy to see that the bound can be simply improved for any $k \geq 1$. The reason why the theorem is presented in this "weak" form is that it already disproves the conjecture and the argument is simply, yet gives a perfectly matching bound for $k = 0$. To find a stronger version would involve a thorough case analysis, similar to the one in [3] and improving the constant a bit is uninteresting at the moment. It would be more interesting to study the behavior of c_k in future works. Now we know that $1.5 \leq c_k \leq C \sim 10$. But is c_k monotonously increasing as k grows? This would imply, of course, the existence of a limit, which is likely to exist.

The rest of the paper is organized as follows. In Section 2 we develop a method to increase the lower bound by k for many search problems and give proofs using it for some known results. In Section 3 we prove our main result, Theorem 1.

2 k more questions

In this section, as a warm-up, we prove a very general result that holds for all search problems and generally gives an additional constant to the lower bounds that are proved using a consistent adversary.

Claim 2 *Suppose we have a search problem where we want to determine the value of some function* f *using (not necessarily yes-no) questions from a family*

*He also obtained asymptotically tight results in another model.

of allowed questions. The answers are given by an adversary who can lie at most k *times. Suppose that we have already asked some questions and the answers we got are consistent, i.e. it is possible that none of them is a lie. If we do not yet know the value of* f*, then we need at least* $k+1$ *further questions to determine it.*

This claim has an immediate, quite weak corollary.

Corollary 3 *If there is a search problem as in Claim 2 with a non-trivial* f*, then we need at least* $2k+1$ *questions to determine* f*.*

Although it is not too standard, we first give a proof of the Corollary, as it is a simplified version of the proof of the Claim.

Proof. Take two possible elements of the universe, x and y, for which $f(x) \neq f(y)$. The adversary can answer the first k questions according to x and the next k questions according to y, thus after $2k$ questions both are still possible. □

Proof of Claim 2. Suppose we have already asked some consistent questions, i.e. there is an x such that they are all true for x. However, if we do not yet know f, there is a y for which at most k of these questions would be false, such that $f(x) \neq f(y)$. We can answer the next k questions according to y. □

To show the power of this simple claim, let us prove the following theorem.

Theorem 4 (Ravikumar et al. [7]) *To find the maximum among* n *elements using comparisons of which* k *might be incorrect, we need* $(k+1)n-1$ *comparisons in the worst case.*

Proof. The upper bound follows from using any tournament scheme and comparing any two elements until one of them is bigger than the other $k+1$ times. This is $(k+1)(n-1)$ plus the possible k lies that might prolong our search.

To prove the lower bound, answer the first $(k+1)(n-1)-1$ questions consistently. Now we have an element that was always bigger, and another that was the smaller one at most k times, thus the conditions of Claim 2 are satisfied, so we need $k+1$ more questions. □

3 Proof of Theorem 1

We start with defining some standard terminology. Define the actual *comparison graph* as a directed graph whose vertices are the elements and it has an edge for every comparison between the compared elements, directed from the bigger towards the smaller. We say that the comparison graph is *consistent* if there is no directed cycle in the comparison graph. In this case any vertex with in-degree at most k can still be the maximum element and any vertex with out-degree at most k can still be the minimum element. We also denote the comparison graph after the first t questions by G_t. So if there are no lies among the first t answers, then they are necessarily consistent and there is no directed cycle in G_t.

Now we prove Theorem 1, which states that $\lceil (k+1.5)(n-1)-0.5 \rceil$ comparisons are needed to find the largest and the smallest element if there might be k erroneous answers.

Proof of Theorem 1. We have to give an adversary argument, i.e., for every possible comparing algorithm, we have to give answers such that it is not possible to determine with less than $(k+1.5)(n-2)+1$ questions the maximum and the minimum. Our answers will be always consistent, i.e., that there will be no directed cycle in the comparison graph.

First, we suppose that n is even and the (undirected) edges of $G_{n/2}$ (the graph of the first $n/2$ questions) form a perfect matching, i.e., every element is compared exactly once during the first $n/2$ comparisons. Denote the set of elements that were bigger in their first comparison by TOP and the ones that were smaller by BOTTOM. Whenever in the future an element from TOP is compared to an element from BOTTOM, we always answer that the one from TOP is bigger. This way the problem reduces to finding the maximum from $n/2$ elements and the minimum from $n/2$ other elements. Every vertex but one from TOP must have in-degree at least $k+1$ at the end and, similarly, every vertex but one from BOTTOM must have out-degree at least $k+1$ at the end. Therefore after $n/2+2(k+1)(n/2-1)-1$ comparisons we still cannot know both the maximum and the minimum, and the answers we got are all consistent, thus we need $k+1$ more questions because of Claim 2. This implies that at least $(k+1.5)(n-1)-0.5$ comparisons are needed in the worst case.

In general, define the sets TOP and BOTTOM to be empty at the beginning and whenever an element is first compared, put it to TOP if it is bigger and to BOTTOM if it is smaller than the element it is compared to. Whenever we compare and element from TOP with an element from BOTTOM, always the

TOP one will be bigger, so the maximum will be in TOP and the minimum in BOTTOM. At the end of the algorithm every element must be assigned to TOP or BOTTOM. Denote the number of elements that are put to TOP by n_1 and the number of the ones that are put to BOTTOM by n_2 (so we have $n_1 + n_2 = n$). It is clear that there are at least $\lceil n/2 \rceil$ questions that compare at least one element that was not compared before. Also note, that if we compare two elements one of which is not in TOP, then the in-degree of the vertices in TOP will not increase. Therefore we need at least $(k+1)(n_1-1)$ comparisons inside TOP. We similarly need at least $(k+1)(n_2-1)$ comparisons inside BOTTOM. Therefore after $\lceil n/2 \rceil + (k+1)(n-2) - 1$ comparisons we still cannot know both the maximum and the minimum, and the answers we got are all consistent, thus we can apply Claim 2. This implies that at least $\lceil (k+1.5)(n-1) - 0.5 \rceil$ comparisons are needed in the worst case. Note that this equals $\lceil (k+1.5)n \rceil - k - 2$, which for $k = 0$ is $\lceil 3n/2 \rceil$, matching the best algorithm and the result of [6]. □

Acknowledgement

I would like to thank the members of Gyula's search seminar, especially Dani and Keszegh, to listen to my attempts to prove the conjecture that I eventually ended up disproving.

The European Union and the European Social Fund have provided financial support to the project under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

[1] M. Aigner, Finding the maximum and the minimum, *Discrete Appl. Math.* **74,** 1 (1997) 1–12. ⇒224, 225

[2] C. Deppe, Coding with feedback and searching with lies, in: *Entropy, Search, Complexity*, Bolyai Society Mathematical Studies, **16**(2007) 27–70. ⇒224

[3] D. Gerbner, D. Pálvölgyi, B. Patkós, G. Wiener, Finding the biggest and smallest element with one lie, *Discrete Appl. Math.*, **158,** 9 (2010) 988–995. ⇒225

[4] M. Hoffmann, J. Matoušek, Y. Okamoto, Ph. Zumstein, Minimum and maximum against k lies, http://arxiv.org/abs/1002.0562. ⇒225

[5] A. Pelc, Searching games with errors – Fifty years of coping with liars, *Theor. Comp. Sci.* **270,** 1-2 (2002) 71–109. ⇒224

[6] I. Pohl, A sorting problem and its complexity, *Comm. ACM* **15,** 6 (1972) 462–464. ⇒224, 228

[7] B. Ravikumar, K. Ganesan, K. B. Lakshmanan, On selecting the largest element in spite of erroneous information, *4th Annual Symposium on Theoretical Aspects of Computer Science*, Passau, Germany, Febr. 19–21, 1987. *Lecture Notes in Comp. Sci.* **247** (1987) pp. 88–99. ⇒224, 226

Received: October 15, 2011 • Revised: November 10, 2011

Acta Univ. Sapientiae, Informatica, **3,** 2 (2011) 230–268

On Erdős-Gallai and Havel-Hakimi algorithms

Antal Iványi
Department of Computer Algebra,
Eötvös Loránd University, Hungary
email: tony@inf.elte.hu

Loránd Lucz
Department of Computer Algebra,
Eötvös Loránd University, Hungary
email: lorand.lucz@gmail.com

Tamás F. Móri
Department of Probability Theory and Statistics, Eötvös Loránd University, Hungary
email: moritamas@ludens.elte.hu

Péter Sótér
Department of Computer Algebra,
Eötvös Loránd University, Hungary
email: mapoleon@freemail.hu

Abstract. Havel in 1955 [28], Erdős and Gallai in 1960 [21], Hakimi in 1962 [26], Ruskey, Cohen, Eades and Scott in 1994 [69], Barnes and Savage in 1997 [6], Kohnert in 2004 [49], Tripathi, Venugopalan and West in 2010 [83] 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 is $\Omega(n^2)$ in worst case. In this paper we propose a new algorithm called EGL (Erdős-Gallai Linear algorithm), whose worst running time is $\Theta(n)$. As an application of this quick algorithm we computed the number of the different degree sequences of simple graphs for 24, ..., 29 vertices (see [74]).

1 Introduction

In the practice an often appearing problem is the ranking of different objects as hardware or software products, cars, economical decisions, persons etc. A

Computing Classification System 1998: G.2.2. [Graph Theory]: Subtopic - Network problems.
Mathematics Subject Classification 2010: 05C85, 68R10
Key words and phrases: simple graphs, prescribed degree sequences, Erdős-Gallai theorem, Havel-Hakimi theorem, graphical sequences

typical method of the ranking is the pairwise comparison of the objects, assignment of points to the objects and sorting the objects according to the sums of the numbers of the received points.

For example Landau [51] references to biological, Hakimi [26] to chemical, Kim et al. [45], Newman and Barabási [61] to net-centric, Bozóki, Fülöp, Poesz, Kéri, Rónyai and Temesi to economical [1, 10, 11, 42, 80], Liljeros et al. [52] to human applications, while Iványi, Khan, Lucz, Pirzada, Sótér and Zhou [30, 31, 38, 65, 67] write on applications in sports.

From several popular possibilities we follow the terminology and notations used by Paul Erdős and Tibor Gallai [21].

Depending from the rules of the allocation of the points there are many problems. In this paper we deal only with the case when the comparisons have two possible results: either both objects get one point, or both objects get zero points. In this case the results of the comparisons can be represented using simple graphs and the number of points gathered by the given objects are the degrees of the corresponding vertices. The decreasing sequence of the degrees is denoted by $b = (b_1, \ldots, b_n)$.

From the popular problems we investigate first of all the question, how can we quickly decide, whether for given b does exist there a simple graph G whose degree sequence is b. In connection with this problem we remark that the main motivation for studying of this problem is the question: what is the complexity of deciding whether a sequence is the score sequence of some football tournament [24, 32, 35, 36, 43, 44, 54].

As a side effect we extended the popular data base *On-line Encyclopedia of Integer Sequences* [72] with the continuation of contained sequences.

In connection with the similar problems we remark, that in the last years a lot of papers and chapters were published on the undirected graphs (for example [8, 9, 12, 16, 19, 29, 37, 41, 55, 68, 81, 83, 84, 85]) and also on directed graphs (for example [7, 11, 14, 23, 24, 30, 31, 33, 38, 45, 48, 50, 57, 58, 63, 65, 64, 66]).

The majority of the investigated algorithms is sequential, but there are parallel results too [2, 18, 20, 36, 60, 62, 77].

Let l, u and m integers ($m \geq 1$ and $u \geq l$). A sequence of integer numbers $b = (b_1, \ldots, b_m)$ is called (l, u, m)-*bounded,* if $l \leq b_i \leq u$ for $i = 1, \ldots, m$. A (l, u, m)-*bounded* sequence b is called (l, u, m)-*regular,* if $b_m \geq b_{m-1} \geq \cdots \geq b_1$. An (l, u, m)-regular sequence is called (l, u, m)-*even*, if the sum of its elements is even. A $(0, n-1, n)$-*regular* sequence b is called n-*graphical,* if there exists a simple graph G whose degree sequence is b. If $l = 0$, $u = n-1$ and $m = n$, then we use the terms n-bounded, n-regular, n-even, and n-

graphical (or simply bounded, regular, even, graphical).

In the following we deal first of all with regular sequences. In our definitions the bounds appear to save the testing algorithms from the checking of such sequences, which are obviously not graphical, therefore these bounds do not mean the restriction of the generality.

The paper consists of nine parts. After the introductory Section 1 in Section 2 we describe the classical algorithms of the testing and reconstruction of degree sequences of simple graphs. Section 3 introduces several linear testing algorithms, then Section 4 summarizes some properties of the approximate algorithms. Section 5 contains the description of new precise algorithms and in Section 6 the running times of the classical testing algorithms are presented. Section 7 contains enumerative results, in Section 8 we report on the application of the new algorithms for the computation of the number of score sequences of simple graphs. Finally Section 9 contains the summary of the results.

Our paper [37] written in Hungarian contains further algorithms and simulation results. [35] contains a short summary on the linear Erdős-Gallai algorithm while in [36] the details of the parallel implementation of enumerating Erdős-Gallai algorithm are presented.

2 Classical precise algorithms

For a given n-regular sequence $b = (b_1, \ldots, b_n)$ the first i elements of the sequence we call *the head* of the sequence belonging to the index i, while the last $n - i$ elements of the sequence we call *the tail* of the sequence belonging to the index i.

2.1 Havel-Hakimi algorithm

The first algorithm for the solution of the testing problem was proposed by Vaclav Havel Czech mathematician [28, 53]. In 1962 Louis Hakimi [26] published independently the same result, therefore the theorem is called today usually as *Havel-Hakimi theorem*, and the method of reconstruction is called *Havel-Hakimi algorithm*.

Theorem 1 (Hakimi [26], Havel [28]). *If $n \geq 3$, then the n-regular sequence $b = (b_1, \ldots, b_n)$ is n-graphical if and only if the sequence $b' = (b_2 - 1, b_3 - 1, \ldots, b_{b_1} - 1, b_{b_1+1} - 1, b_{b_1+2}, \ldots, b_n)$ is $(n-1)$-graphical.*

Proof. See [9, 26, 28, 37]. □

If we write a recursive program based on this theorem, then according to the RAM model of computation its running time will be in worst case $\Omega(n^2)$, since the algorithm decreases the degrees by one, and e.g. if $b = ((n-1)^n)$, then the sum of the elements of b equals to $\Theta(n^2)$. It is worth to remark that the proof of the theorem is constructive, and the algorithm based on the proof not only tests the input in quadratic time, but also construct a corresponding simple graph (of course, only if it there exists).

It is worth to remark that the algorithm was extended to directed graphs in which any pair of the vertices is connected with at least $a \geq 0$ and at most $b \geq a$ edges [30, 31]. The special case $a = b = 1$ was reproved in [23].

In 1965 Hakimi [27] gave a necessary and sufficient condition for *two sequences* $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ to be the in-degree sequences and out-degree sequence of a directed multigraph without loops.

2.2 Erdős-Gallai algorithm

In chronological order the next result is the necessary and sufficient theorem published by Paul Erdős and Tibor Gallai [21].

For an n-regular sequence $b = (b_1, \ldots, b_n)$ let $H_i = b_1 + \cdots + b_i$. For given i the elements $b_1, \ldots, b_i$ are called *the head* of b, belonging to i, while the elements $b_{i+1}, \ldots, b_n$ are called *the tail* of b belonging to i.

When we investigate the realizability of a sequence, a natural observation is that the degree requirement H_i of a head is covered partially with inner and partially with outer degrees (with edges among the vertices of the head, resp. with edges, connecting a vertex of the head and a vertex of the tail). This observation is formalized by the following Erdős-Gallai theorem.

Theorem 2 (Erdős, Gallai [21]) *Let* $n \geq 3$. *The* n*-regular sequence* $b = (b_1, \ldots, b_n)$ *is* n*-graphical if and only if*

$$\sum_{i=1}^{n} b_i \quad \textit{even} \tag{1}$$

and

$$\sum_{i=1}^{j} b_i \leq j(j-1) + \sum_{k=j+1}^{n} \min(j, b_k) \quad (j = 1, \ldots, n-1). \tag{2}$$

Proof. See [9, 15, 21, 70, 83]. □

n	$R(n)$	$E(n)$	$E(n)/R(n)$
1	1	1	1.0000000000000
2	3	2	0.6666666666667
3	10	6	0.6000000000000
4	35	19	0.5428571428571
5	126	66	0.5238095238095
6	462	236	0.5108225108225
7	1716	868	0.5058275058275
8	6435	3235	0.5027195027195
9	24310	12190	0.5014397367339
10	92378	46252	0.5006819805581
11	352716	176484	0.5003572279114
12	1352078	676270	0.5001708481315
13	5200300	2600612	0.5000888410284
14	20058300	10030008	0.5000427753100
15	77558760	38781096	0.5000221251603
16	300540195	150273315	0.5000107057227
17	1166803110	583407990	0.5000055150693
18	4537567650	2268795980	0.5000026787479
19	17672631900	8836340260	0.5000013755733
20	68923264410	34461678394	0.5000006701511
21	269128937220	134564560988	0.5000003432481
22	1052049481860	526024917288	0.5000001676328
23	4116715363800	2058358034616	0.5000000856790
24	16123801841550	8061901596814	0.5000000419280
25	63205303218876	31602652961516	0.5000000213918
26	247959266474052	123979635837176	0.5000000104862
27	973469712824056	486734861612328	0.5000000053420
28	3824345300380220	1912172660219260	0.5000000026224
29	15033633249770520	7516816644943560	0.5000000013342
30	59132290782430712	29566145429994736	0.5000000006558
31	232714176627630544	116357088391374032	0.5000000003333
32	916312070471295267	458156035385917731	0.5000000001640
33	3609714217008132870	1804857108804606630	0.5000000000833
34	14226520737620288370	7113260369393545740	0.5000000000410
35	56093138908331422716	28046569455332514468	0.5000000000208
36	221256270138418389602	110628135071477978626	0.5000000000103
37	873065282167813104916	436532641088444120108	0.5000000000052
38	3446310324346630677300	1723155162182151654600	0.5000000000026

Figure 1: Number of regular and even sequences, and the ratio of these numbers

Although this theorem does not solve the problem of reconstruction of graphical sequences, the systematic application of (2) requires in worst case

(for example when the input sequence is graphical) $\Theta(n^2)$ time.

Recently Tripathi and Vijay [83] published a constructive proof of Erdős-Gallai theorem and proved that their construction requires $O(n^3)$ time.

Figure 1 shows the number of n-regular ($R(n)$ and n-even ($E(n)$ sequences and their ratio ($E(n)/R(n)$ for $n = 1, \ldots, 38$. According to (34) the sequence of these ratios tends to $\frac{1}{2}$ as n tends to ∞. According to Figure 1 the convergence is quick: e.g. $E(20)/R(20) = 0.5000006701511$.

The pseudocode of ERDŐS-GALLAI see in [37].

3 Testing algorithms

We are interested in the investigation of football sequences, where often appears the necessity of the testing of degree sequences of simple graphs.

A possible way to decrease the expected testing time is to use quick (linear) filtering algorithms which can state with a high probability, that the given input is not graphical, and so we need the slow precise algorithms only in the remaining cases.

Now we describe a *parity checking*, then a *binomial,* and finally a *headsplitting* filtering algorithm.

3.1 Parity test

Our first test is based on the first necessary condition of Erdős-Gallai theorem. This test is very effective, since according to Figure 1 and Corollary 14 about the half of the regular sequences is odd, and our test establishes in linear time, that these sequences are not graphical.

The following simple algorithm is based on (1).

Input. n: number of the vertices ($n \geq 1$);
$b = (b_1, \ldots, b_n)$: an n-regular sequence.

Output. L: logical variable ($L =$ FALSE shows, that b is not graphical, while the meaning of the value $L =$ TRUE is, that the test *could not decide,* whether b is graphical or not).

Working variable. i: cycle variable;
$H = (H_1, \ldots, H_n)$: H_i is the sum of the first i elements of b.

PARITY-TEST(n, b, L)
01 $H_1 = b_1$
02 **for** $i = 2$ **to** n
03 $\quad H_i = H_{i-1} + b_i$

```
04 if H_n odd
05     L = False
06     return L
07 L = True
08 return L
```

The running time of this algorithm is $\Theta(n)$ in all cases. Figure 1 characterizes the efficiency of Parity-test.

(1) is only a necessary condition, therefore Parity-Test is only an approximate (filtering) algorithm.

3.2 Binomial test

Our second test is based on the second necessary condition of Erdős-Gallai theorem. It received the given name since we estimate the number of the inner edges of the head of b using a binomial coefficient. Let $T_i = b_{i+1} + \cdots + b_n$ $(i = 1, \ldots, n)$.

Lemma 3 *If $n \geq 1$ and b is an n-graphical sequence, then*

$$H_i \leq i(i-1) + T_i \quad (i = 1, \ldots, n-1). \tag{3}$$

Proof. The left side of (3) represents the degree requirement of the head of b. On the right side of (3) $i(i-1)$ is an upper bound of the inner degree capacity of the head, while T_i is an upper bound of the degree capacity of the tail, belonging to the index i. □

The following program is based on Lemma 3.

Input. n: number of the vertices $(n \geq 1)$;
$b = (b_1, \ldots, b_n)$: an n-regular even sequence;
$H = (H_1, \ldots, H_n)$: H_i the sum of the first i elements of b;
H_0: auxiliary variable, helping to compute the elements of H.

Output. L: logical variable (L = False signals, that b is surely not graphical, while L = True shows, that the test *could not decide,* whether b is graphical).

Working variables. i: cycle variable;
$T = (T_1, \ldots, T_n)$: T_i the sum of the last $n - i$ elements of b;
T_0: auxiliary variable, helping to compute the elements of T.

```
Binomial-Test(n, b, H, L)
01 T_0 = 0
02 for i = 1 to n − 1
```

```
03      T_i = H_n − H_i
04      if H_i > i(i − 1) + T_i
05          L = FALSE
06          return L
07  L = TRUE
08  return L
```

The running time of this algorithm is $\Theta(n)$ in worst case, while in best case is only $\Theta(1)$.

According to our simulation experiments BINOMIAL-TEST is an effective filtering test (see Figure 2 and Figure 3).

3.3 Splitting of the head

We can get a better estimation of the inner capacity of the head, than the binomial coefficient gives in (3), if we split the head into two parts. Let $\lfloor i/2 \rfloor = h_i$, p the number of positive elements of b. Then the sequence $(b_1, \ldots, b_{h_i})$ is called *the beginning of the head* belonging to index i and the sequence $(b_{h_i+1}, \ldots, b_i)$ *the end of the head* belonging to index i.

Lemma 4 *If $n \geq 1$ and b is an n-graphical sequence, then*

$$\begin{aligned} H_i \leq \min(&\min(H_{h_i}, T_n - T_i, h_i(n-i)) \\ &+ \min(H_i - H_{h_i}, T_n - T_i, (i-h_i)(n-i)), T_i) \\ &+ \min(h_i(i-h_i) + \binom{h_i}{2} + \binom{i-h_i}{2} \quad (i = 1, \ldots, n), \end{aligned} \tag{4}$$

further

$$\min(H_{h_i}, T_n - T_i, h_i(n-i)) + \min(H_i - H_{h_i}, T_n - T_i, (i-h_i)(n-i)) \leq T_i. \tag{5}$$

Proof. Let G be a simple graph whose degree sequence is b. Then we divide the set of the edges of the head belonging to index i into five subsets: $(S_{i,1})$ contains the edges between the beginning of the head and the tail, $(S_{i,2})$ the edges between the end of the head and the tail, $S_{i,3}$ the edges between the parts of the head, $S_{i,4}$ the edges in the beginning of the head and $S_{i,5}$ the edges in the end of the head. Let us denote the number of edges in these subsets by $X_{i,1}, \ldots, X_{i,5}$.

$X_{i,1}$ is at most the sum H_{h_i} of the elements of the head, at most the sum $T_n - T_i$ of the elements of the tail, and at most the product $h_i(n-i)$ of the

elements of the pairs formed from the tail and from the beginning of the head, that is

$$X_{i,1} \le \min(H_{h_i}, T_n - T_i, h_i(n-i)). \tag{6}$$

A similar train of thought results

$$X_{i,2} \le \min(H_i - H_{h_i}, T_n - T_i, (i-h_i)(n-i)). \tag{7}$$

$X_{i,3}$ is at most $h_i(i-h_i)$ and at most H_i, implying

$$X_{i,3} \le \min(h_i(i-h_i), H_i). \tag{8}$$

$X_{i,4}$ is at most $\binom{h_i}{2}$ and at most H_{h_i}, implying

$$X_{i,4} \le \min(\binom{h_i}{2}, H_{h_i}), \tag{9}$$

while $X_{i,5}$ is at most $\binom{i-h_i}{2}$ and at most $H_i - H_{h_i}$, implying

$$X_{i,5} \le \binom{i-h_i}{2}. \tag{10}$$

A requirement is also, that the tail can overrun its capacity, that is

$$X_{i,1} + X_{i,2} \le T_i. \tag{11}$$

Summing of (6), (7), (8), (9), and (10) results

$$H_i \le X_{i,1} + X_{i,2} + X_{i,3} + 2X_{i,4} + 2X_{i,5}. \tag{12}$$

Substituting of (6), (7), (8), (9), and (10) into (12) results (4), while (11) is equivalent to (5). □

The following algorithm executes the test based on Lemma 4.

Input. n: the number of vertices ($n \ge 1$);
$d = (d_1, \ldots, d_n)$: an n-even sequence, accepted by BINOMIAL-TEST;
$H = (H_1, \ldots, H_n)$: H_i the sum of the first i elements of d;
$T = (T_1, \ldots, T_n)$: T_i the sum of the last $n-i$ elements of d.

Output. L: logical variable (L = FALSE signals,that d is not graphical, while L = TRUE shows, that the test *could nor decide*, whether d is graphical).

Working variables. i: cycle variable;
h: the actual value of h_i;
$X = (X_1, X_2, X_3, X_4, X_5)$: X_j is the value of the actual $X_{i,j}$.

HEADSPLITTER-TEST(n, b, H, T, L)

01 **for** $i = 2$ **to** $n - 1$
02 $\quad h = \lfloor i/2 \rfloor$
03 $\quad X_1 = \min(H_h, T_n - T_i, h(n - i))$
04 $\quad X_2 = \min(H_i - H_h, T_n - T_i, (i - h)(n - i))$
05 $\quad X_3 = \min(h(i - h)$
06 $\quad X_4 = \binom{h}{2}$
07 $\quad X_5 = \binom{i-h}{2}$
06 $\quad$ **if** $H_i > X_1 + X_2 + X_3 + 2X_4 + 2X_5$ or $X_1 + X_2 > T_i$
07 $\quad\quad L =$ FALSE
08 $\quad\quad$ **return** L
09 $L =$ TRUE
10 **return** L

The running time of the algorithm is $\Theta(1)$ in best, and $\Theta(n)$ in worst case.

It is a substantial circumstance that the use of Lemma 3 and Lemma 4 requires only *linear* time (while the earlier two theorems require quadratic time). But these improvements of Erdős-Gallai theorem decrease only the coefficient of the quadratic member in the formula of the running time, the order of growth remains unchanged.

Figure 2 contains the results of the running of BINOMIAL-TEST and HEADSPLITTER-TEST, further the values $G(n)$ and $\frac{G(n)}{G(n+1)}$ (the computation of the values of the function $G(n)$ will be explained in Section 8).

Figure 3 shows the relative frequency of the zerofree regular, binomial, headsplitted and graphical sequences compared to the number of regular sequences.

3.4 Composite test

COMPOSITE-TEST uses approximate algorithms in the following order: PARITY-TEST, BINOMIAL-TEST, POSITIVE-TEST, HEADSPLITTER-TEST.

COMPOSITE-TEST(n, b, L)

01 PARITY-TEST(n, b, L)
02 **if** $L ==$ FALSE
03 $\quad$ **return** L
04 BINOMIAL-TEST(n, b, H, L)
05 **if** $L ==$ FALSE
06 $\quad$ **return** L
07 HEADSPLITTER-TEST(n, b, H, T, L)

```
08 if L == False
09     return L
10 L = True
11 return L
```

The running time of this composite algorithm is in all cases $\Theta(n)$.

4 Properties of the approximate testing algorithms

We investigate the efficiency of the approximate algorithms testing the regular algorithms. Figure 1 contains the number $R(n)$ of regular, the number $E(n)$ of even, and the number $G(n)$ of graphical sequences for $n = 1, \ldots, 38$.

The relative efficiency of arbitrary testing algorithm A for sequences of given length n we define with the ratio of the number of accepted by A sequences of length n and the number of graphical sequences $G(n)$. This ratio as a function of n will be noted by $X_A(n)$ and called the *error function* of A [34].

We investigate the following approximate algorithms, which are the components of Composite-Test:

1) Parity-Test;
2) Binomial-Test;
3) Headsplitter-Test.

According to (23) there are $R(2) = 3$ 2-regular sequences: $(1,1)$, $(1,0)$ and $(0,0)$. According to (25) among these sequences there are $E(2) = 2$ even sequences. Binomial-Test accepts both even ones, therefore $B(2) = 2$. Both sequences are 2-graphical, therefore $G(2) = 2$ and so the efficiency of Parity-Test (PT) and Binomial-Test (BT) is $X_{PT}(2) = X_{BT}(2) = 2/2 = 1$, in this case both algorithms are optimal.

The number of 3-regular sequences is $R(3) = 10$. From these sequences $(2,2,2)$, $(2,2,0)$, $(2,1,1)$, $(2,0,0)$ $(1,1,0)$ and $(0,0,0)$ are even, so $E(3) = 6$. Binomial-Test excludes the sequences $(2,2,0)$ and $(2,0,0)$, so remains $B(3) = 4$. Since these sequences are 3-graphical, $G(3) = 4$ implies $X_{PT}(3) = \frac{3}{2}$ and $X_{BT}(3) = 1$.

The number of 4-regular sequences equals to $R(4) = 35$. From these sequences 16 is even, and the following 11 are 4-graphical: $(3,3,3,3)$, $(3,3,2,2)$, $(3,2,2,1)$, $(3,1,1,1,)$, $(2,2,2,2)$, $(2,2,2,0)$, $(2,2,1,1)$, $(2,1,1,0)$, $(1,1,1,1)$, $(1,1,0,0)$ and $(0,0,0,0)$. From the 16 even sequences Binomial-Test also excludes the 5 sequences, so $B(4) = G(4) = 11$ and $X_{BT}(4) = 1$.

According to these data in the case of $n \leq 4$ Binomial-Test recognizes all nongraphical sequences. Figure 2 shows, that for $n \leq 5$ we have $B(n) = G(n)$,

n	$B_z(n)$	$F_z(n)$	$G(n)$	$G(n+1)/G(n)$
1	**1**	**0**	**1**	2.000000
2	**2**	**2**	**2**	2.000000
3	**4**	**4**	**4**	2.750000
4	**11**	**11**	**11**	2.818182
5	**31**	**31**	**31**	3.290323
6	103	**102**	**102**	3.352941
7	349	344	**342**	3.546784
8	1256	1230	1213	3.595218
9	4577	4468	4361	3.672552
10	17040	16582	16016	3.705544
11	63944	62070	59348	3.742620
12	242218	234596	222117	3.765200
13	922369	891852	836315	3.786674
14	3530534	3409109	3166852	3.802710
15	13563764	13082900	12042620	3.817067
16	52283429	50380684	45967479	3.828918
17	202075949	194550002	176005709	3.839418
18	782879161	753107537	675759564	3.848517
19	3039168331	2921395019	2600672458	3.856630
20	11819351967	11353359464	10029832754	3.863844
21			38753710486	3.870343
22			149990133774	3.876212
23			581393603996	3.881553
24			2256710139346	3.886431
25			8770547818956	3.890907
26			34125389919850	3.895031
27			132919443189544	3.897978
28			518232001761434	3.898843
29			2022337118015338	

Figure 2: Number of zerofree binomial, zerofree headsplitted and graphical sequences, further the ratio of the numbers of graphical sequences for neighbouring values of n

that is Binomial-Test accepts the same number of sequences as the precise algorithms. If $n > 5$, then the error function of Binomial-Test is increasing: while $X_{BT}(6) = \frac{103}{102}$ (BT accepts one nongraphical sequence), $X_{BT}(7) = \frac{349}{342}$ (BT accepts 7 nongraphical sequences) etc.

Figure 4 presents the average running time of the testing algorithms BT

n	$E_z(n)$	$E_z(n)/R(n)$	$B_z(n)/R(n)$	$F_z(n)/R(n)$	$G(n)/R(n)$
1	0	0.000000	1.000000	1.000000	1.000000
2	1	0.333333	0.666667	0.666667	0.666667
3	2	0.300000	0.400000	0.400000	0.400000
4	9	0.257143	0.314286	0.314286	0.314286
5	28	0.230159	0.246032	0.246031	0.246032
6	110	0.238095	0.222943	0.220779	0.220779
7	396	0.231352	0.203380	0.200466	0.199301
8	1519	0.236053	0.195183	0.191142	0.188500
9	5720	0.235335	0.188276	0.183793	0.179391
10	21942	0.237524	0.184460	0.179502	0.173375
11	83980	0.238098	0.181290	0.175977	0.168260
12	323554	0.239301	0.179145	0.173508	0.164278
13	1248072	0.240000	0.177368	0.171500	0.160821
14	4829708	0.240784	0.176014	0.169960	0.157882
15	18721080	0.241379	0.174884	0.168684	0.155271
16	72714555	0.241946	0.173965	0.167634	0.152950
17	282861360	0.242424	0.173188	0.166738	0.150844
18	1101992870	0.242860	0.172533	0.165972	0.148926
19	4298748300	0.243243	0.171970	0.165306	0.147158
20	16789046494	0.243590	0.171486	0.164725	0.145521
21					0.143997
22					0.142569
23					0.141228
24					0.139961
25					0.138762
26					0.137625
27					0.136542
28					0.135509
29					0.134521

Figure 3: The number of zerofree even sequences, further the ratio of the numbers binomial/regular, headsplitted/regular and graphical/regular sequences

and HT in secundum and in number of operations. The data contain the time and operations necessary for the generation of the sequences too.

n	BT, s	BT, operation	HT, s	HT, operation
1	0	14	0	15
2	0	41	0	43
3	0	180	0	200
4	0	716	0	815
5	0	2 918	0	3 321
6	0	11 918	0	13 675
7	0	48 952	0	56 299
8	0	201 734	0	233 182
9	0	831 374	0	964 121
10	0	3 426 742	0	3 988 542
11	0	14 107 824	0	16 469 036
12	0	58 028 152	0	67 929 342
13	0	238 379 872	0	279 722 127
14	0	978 194 400	1	1 150 355 240
15	2	4 009 507 932	3	4 724 364 716
16	6	16 417 793 698	13	19 379 236 737
17	26	67 160 771 570	51	79 402 358 497
18	106	274 490 902 862	196	324 997 910 595
19	423	1 120 923 466 932	798	1 328 948 863 507
20	1 627	4 573 895 421 484	3 201	5 429 385 115 097

Figure 4: Running time of Binomial-Test (BT) and Headsplitter-Test (HT) in secundum and as the number of operations for $n = 1, \ldots, 20$

5 New precise algorithms

In this section the zerofree algorithms, the shifting Havel-Hakimi, the parity checking Havel-Hakimi, the shortened Erdős-Gallai, the jumping Erdős-Gallai, the linear Erdős-Gallai and the quick Erdős-Gallai algorithms are presented.

5.1 Zerofree algorithms

Since the zeros at the and of the input sequences correspond to isolated vertices, so they have no influence on the quality of the sequence. This observation is exploited in the following assertion, in which p means the number of the positive elements of the input sequence.

Corollary 5 *If* $n \geq 1$*, the* $(b_1, \ldots, b_n)$ n*-regular sequence is* n*-graphical if and only if* $(b_1, \ldots, b_p)$ *is* p*-graphical.*

Proof. If all elements of b are positive (that is $p = n$), then the assertion is equivalent with Erdős-Gallai theorem. If b contains zero element (that is

$p < n$), then the assertion is the consequence of Havel-Hakimi and Erdős-Gallai algorithms, since the zero elements do not help in the pairing of the positive elements, but from the other side they have no own requirement. □

The algorithms based on this corollary are called HAVEL-HAKIMI-ZEROFREE (HHZ), resp. ERDŐS-GALLAI-ZEROFREE (EGZ).

5.2 Shifting Havel-Hakimi algorithm

The natural algorithmic equivalent of the original Havel-Hakimi theorem is called HAVEL-HAKIMI SORTING (HHSo), since it requires the sorting of the reduced input sequence in every round.

But it is possible to design such implementation, in which the reduction of the degrees is executed saving the monotonity of the sequence. Then we get HAVEL-HAKIMI-SHIFTING (HHSh) algorithm.

For the pseudocode of this algorithms see [37].

5.3 Parity checking Havel-Hakimi algorithm

It is an interesting idea the join the application of the conditions of Erdős-Gallai and Havel-Hakimi theorems in such a manner, that we start with the parity checking of the input sequence, and only then use the recursive Havel-Hakimi method.

For the pseudocode of the algorithm HAVEL-HAKIMI-PARITY (HHP) see [37].

5.4 Shortened Erdős-Gallai algorithm (EGSh)

In the case of a regular sequence the maximal value of H_i is $n(n-1)$, therefore the inequality (2) certainly holds for $i = n$, therefore it is unnecessary to check.

Even more useful observation is contained in the following assertion due to Tripathi and Vijai.

Lemma 6 (Tripathi, Vijay [82]) *If* $n \geq 1$, *then an* n*-regular sequence* $b = (b_1 \ldots, b_n)$ *is* n*-graphical if and only if*

$$H_n \quad \textit{even} \tag{13}$$

and

$$H_i \leq \min(H_i, i(i-1)) + \sum_{k=i+1}^{n} \min(i, b_k) \quad (i = 1, 2, \ldots, r), \tag{14}$$

where

$$r = \max_{1 \le s \le n} (s \mid s(s-1) < H_s) \tag{15}$$

Proof. If $i(i-1) \ge H_i$, then the left side of (2) is nonpositive, therefore the inequality holds, so the checking of the inequality is nonnecessary. □

The algorithm based on this assertion is called ERDŐS-GALLAI-SHORTENED. For example if the input sequence is $b = (5^{100})$, then ERDŐS-GALLAI computes the right side of (2) 99 times, while ERDŐS-GALLAI-SHORTENED only 6 times.

5.5 Jumping Erdős-Gallai algorithm

Contracting the repeated elements a regular sequence $(b_1, \dots, b_n)$ can be written in the form $(b_{i_1}^{e_1}, \dots, b_{i_q}^{e_q})$, where $b_{i_1} < \dots < b_{i_q}$, $e_1, \ \dots, \ e_q \ge 1$ and $e_1 + \dots + e_q = n$. Let $g_j = e_1 + \dots + e_j$ $(j = 1, \ \dots, \ q)$.

The element b_i is called the *checking points* of the sequence b, if $i = n$ or $1 \le i \le n-1$ és $b_i > b_{i+1}$. Then the checking points are $b_{g_1}, \dots, b_{g_q}$.

Theorem 7 (Tripathi, Vijay [82]) *An n-regular sequence $b = (b_1, \dots, b_n)$ is n-graphical if and only if*

$$H_n \quad \textit{even} \tag{16}$$

and

$$H_{g_i} - g_i(g_i - 1) \le \sum_{k=g_i+1}^{n} \min(i, b_k) \quad (i = 1, \dots, q). \tag{17}$$

Proof. See [82]. □

Later in algorithm ERDŐS-GALLAI-ENUMERATING we will exploit, that in the inequality (17) g_q is always n, therefore it is enough to check the inequality only up to $i = q - 1$.

The next program implements a quick version of Erdős-Gallai algorithm, exploiting Corollary 5, Lemma 6 and Lemma 7. In this paper we use the pseudocode style proposed in [17].

Input. n: number of vertices $(n \ge 1)$;
$b = (b_1, \dots, b_n)$: an n-even sequence.

Output. L: logical variable (L = FALSE signalizes, that, b is not graphical, while L = TRUE shows, that b is graphical).

Working variables. i and j: cycle variables;
$H = (H_0, H_1, \dots, H_n)$: H_i is the sum of the first i elements of b;

C: the degree capacity of the actual tail;
b_{n+1}: auxiliary variable helping to decide, whether b_n is a jumping element.

ERDŐS-GALLAI-JUMPING(n, b, H, L)

```
01 H_1 = b_1                                   // lines 01–06: test of parity
02 for i = 2 to n
03     H_i = H_{i-1} + b_i
04 if H_n odd
05   L = FALSE
06   return L
07 b_{n+1} = -1                    // lines 07–20: test of the request of the head
08 i = 1
09 while i ≤ n and i(i-1) < H_i
10       while b_i == b_{i+1}
11             i = i + 1
12       C = 0
13       for j = i + 1 to n
14           C = C + min(j, b_j)
15       if H_i > i(i-1) + C
16          L = FALSE
17          return L
18       i = i + 1
19 L = TRUE
20 return L
```

The running time of EGJ varies between the best $\Theta(1)$ and the worst $\Theta(n^2)$.

5.6 Linear Erdős-Gallai algorithm

Recently we could improve ERDŐS-GALLAI algorithm [35, 37]. The new algorithm ERDŐS-GALLAI-LINEAR exploits, that b is monotone. It determines the capacities C_i in constant time. The base of the quick computation is the sequence $w(b)$ containing the *weight points* w_i of the elements of the input sequence b.

For given sequence b let $w(b) = (w_1, \ldots, w_{n-1})$, where w_i gives the index of b_k having the maximal index among such elements of b which are greater or equal to i.

Theorem 8 (Iványi, Lucz [35], Iványi, Lucz, Móri, Sótér [37]) *If* $n \geq 1$, *then*

the n*-regular sequence* $(b_1, \dots, b_n)$ *is* n*-graphical if and only if*

$$H_n \quad \textit{is even} \tag{18}$$

and if $i > w_i$, *then*

$$H_i \le i(i-1) + H_n - H_i,$$

further if $i \le w_i$, *then*

$$H_i \le i(i-1) + i(w_i - i) + H_n - H_{w_i}.$$

Proof. (18) is the same as (1).

During the testing of the elements of b by ERDŐS-GALLAI-LINEAR there are two cases:

- if $i > w_i$, then the contribution $C_i = \sum_{k=i+1}^{n} \min(i, b_k)$ of the tail of b equals to $H_n - H_i$, since the contribution c_k of the element b_k is only b_k.
- if $i \le w_1$, then the contribution of the tail of b consists of contributions of two types: $c_{i+1}, \dots, c_{w_i}$ are equal to i, while $c_j = b_j$ for $j = w_i + 1, \dots, n$.

Therefore in the case $n - 1 \ge i > w_i$ we have

$$C_i = H_n - H_i, \tag{19}$$

and in the case $1 \le i \le w_i$

$$C_i = i(w_i - i) + H_n - H_{w_i}. \tag{20}$$

□

The following program is based on Theorem 8. It decides on arbitrary n-regular sequence whether it is n-graphical or not.

Input. n: number of vertices ($n \ge 1$);
$b = (b_1, \dots, b_n)$: n-regular sequence.

Output. L: logical variable, whose value is TRUE, if the input is graphical, and it is FALSE, if the input is not graphical.

Work variables. i and j: cycle variables;
$H = (H_1, \dots, H_n)$: H_i is the sum of the first i elements of the tested b;
b_0: auxiliary element of the vector b
$w = (w_1, \dots, w_{n-1})$: w_i is the weight point of b_i, that is the maximum of the indices of such elements of b, which are not smaller than i;
$H_0 = 0$: help variable to compute the other elements of the sequence H;
$b_0 = n - 1$: help variable to compute the elements of the sequence w.

```
ERDŐS-GALLAI-LINEAR(n, b, L)
01 H_0 = 0                                              // line 01: initialization
02 for i = 1 to n                  // lines 02–03: computation of the elements of H
03     H_i = H_{i-1} + b_i
04 if H_n  odd                                  // lines 04–06: test of the parity
05    L = FALSE
06    return L
07 b_0 = n - 1                    // line 07: initialization of a working variable
08 for i = 1 to n                    // lines 08–12: computation of the weights
09     if b_i < b_{i-1}
10        for j = b_{i-1} downto b_i + 1
11            w_j = i - 1
12        w_{b_i} = i
13 for j = b_n downto 1                                 // lines 13–14: large weights
14     w_j = n
15 for i = 1 to n                          // lines 15–23: test of the elements of b
16     if i ≤ w_i                      // lines 16–19: test of indices for large w_i's
17        if H_i > i(i - 1) + i(w_i - i) + H_n - H_{w_i}
18           L = FALSE
19           return L
20     if i > w_i                      // lines 20–23: test of indices for small w_i's
21        if H_i > i(i - 1) + H_n - H_i
22           L = FALSE
23           return L
24 L = TRUE              // lines 24–25: the program ends with the value TRUE
25 return L
```

Theorem 9 (Iványi, Lucz [35], Iványi, Lucz, Móri, Sótér [37]) *Algorithm* ERDŐS-GALLAI-LINEAR *decides in* $\Theta(n)$ *time, whether an* n*-regular sequence* $b = (b_1, \ldots, b_n)$ *is graphical or not.*

Proof. Line 1 requires $O(1)$ time, lines 2–3 $\Theta(n)$ time, lines 4–6 $O(1)$ time, line 07 $O(1)$ time, lines 08–12 $O(1)$ time, lines 13–14 $O(n)$ time, lines 15–23 $O(n)$ time and lines 24–25 $O(1)$ time, therefore the total time requirement of the algorithm is $\Theta(n)$. □

Since in the case of a graphical sequence all elements of the investigated sequence are to be tested, in the case of RAM model of computations [17] ERDŐS-GALLAI-LINEAR is asymptotically optimal.

6 Running time of the precise testing algorithms

We tested the precise algorithms determining their total running time for all the even sequences. The set of the even sequences is the smallest such set of sequences, whose the cardinality we know exact and explicite formula. The number of n-bounded sequences $K(n)$ is also known, but this function grows too quickly when n grows.

If we would know the average running time of the bounded sequences we would take into account that is is sufficient to weight the running times of the regular sequences with the corresponding frequencies. For example a homogeneous sequence consisting of identical elements would get a unit weight since it corresponds to only one bounded sequence, while a rainbow sequence consisting is n different elements as e.g. the sequence $n, n-1, \ldots, 1$ corresponds to $n!$ different bounded sequences and therefore would get a corresponding weight equal to $n!$.

We follow two ways of the decreasing of the running time of the precise algorithms. The first way is the decreasing of the number of the executable operations. The second way is, that we try to use quick (linear time) preprocessing algorithms for the filtering of the sequences in order to decrease of the part of sequences requiring the relative slow precise algorithms.

For the first type of decrease of the expected running time is the shortening of the sequences and the application of the checking points, while for the the second type are examples the completion of HH algorithm with the parity checking or the completion of the EG algorithm with the binomial and headsplitted algorithms.

In this section we investigate the following precise algorithms:

1) Havel-Hakimi-Shorting (HHSo).
2) Havel-Hakimi-Shifting (HHSh).
3) Erdős-Gallai algorithm (EG).
4) Erdős-Gallai-Jumping algorithm (EGJ).
5) Erdős-Gallai-Linear algorithm (EGL).

Figure 5 contains the total number of operations of the algorithms HHSo, HHSh, EG, and EGL required for the testing of all even sequences of length $n = 1, \ldots, 15$. The operations necessary to generate the sequences are included.

Comparison of the first two columns shows that algorithm HHSh is much quicker than HHSo, especially if n increases. Comparison of the third and fourth columns shows that we get substantial decrease of the running time if we have to test the input sequence only in the check points. Finally the comparison of the third and fifth columns demonstrates the advantages of a

n	HHSo	HHSh	EG	EGJ	EGL
1	10	15	87	-	-
2	40	61	119	12	37
3	231	236	267	116	148
4	1 170	1 052	946	551	585
5	5 969	4 477	4 000	2 677	2 339
6	31 121	20 153	18 206	12 068	9 539
7	157 345	88 548	82 154	54 184	38 984
8	784 341	393 361	372 363	238 813	160 126
9	3 628 914	1 726 484	1 666 167	1 666 167	656 575
10	17 345 700	7 564 112	7 418 447	4 552 276	2 692 240
11	80 815 538	32 895 244	32 737 155	19 680 986	11 018 710
12	385 546 527	142 460 352	143 621 072	84 608 529	45 049 862
13	1 740 003 588	613 739 913	626 050 861	362 141 061	183 917 288
14	8 066 861 973	2 633 446 908	2 715 026 827	1 543 745 902	750 029 671
15	36 630 285 216	11 254 655 388	11 717 017 238	6 557 902 712	3 055 289 271

Figure 5: Total number of operations as the function of n for precise algorithms HHSo, HHSh, EG, EGJ, and EGL.

n	$E(n)$	$T(n)$, s	$Op(n)$	$T(n)/E(n)/n$, s	$Op(n)/E(n)/n$
2	2	0	37	0	9.25000000000
3	6	0	148	0	8.22222222222
4	19	0	585	0	7.69736842105
5	66	0	2 339	0	7.08787878788
6	236	0	9 539	0	6.73658192090
7	868	0	38 984	0	6.41606319947
8	3 235	0	160 126	0	6.18724884080
9	12 190	0	656 575	0	5.98464132714
10	46 252	0	2 692 240	0	5.82080774885
11	176 484	0	11 018 710	0	5.67587378511
12	676 270	0	45 049 862	0	5.55126675243
13	2 600 612	0	183 917 288	0	5.44005937537
14	10 030 008	1	750 029 671	0.000000007121487	5.34132654018
15	38 781 096	5	3 055 289 271	0.000000008595253	5.25219687963
16	150 273 315	23	12 434 367 770	0.000000009565903	5.17156346504
17	583 407 990	79	50 561 399 261	0.000000007965367	5.09797604337
18	2 268 795 980	297	205 439 740 365	0.00000000727258	5.03056202928

Figure 6: Total and amortized running time of Erdős-Gallai-Linear in secundum, resp. in the number of executed operations

$E(n)-G(n)$	n/i	f_1	f_2	f_3	f_4	f_5	f_6	f_7
2	3	2	0	0	0	0	0	0
8	4	6	2	0	0	0	0	0
35	5	33	2	0	0	0	0	0
134	6	122	12	0	0	0	0	0
526	7	459	65	2	2	0	0	0
2022	8	1709	289	24	0	0	0	0
7829	9	6421	1228	176	4	0	0	0
30236	10	24205	4951	1013	67	0	0	0
115136	11	91786	19603	5126	610	11	0	0
454153	12	349502	76414	23755	4274	208	0	0
1764297	13	1336491	296036	104171	25293	2277	29	0
6863156	14	5128246	1142470	439155	133946	18673	666	0
26738476	15	19739076	4404813	1803496	655291	127116	8603	81

Figure 7: Distribution of the even nongraphical sequences according to the number of tests made by Erdős-Gallai-Jumping to exclude the given sequence for $n = 3, \ldots, 15$

linear algorithm over a quadratic one.

Figure 6 shows the running time of Erdős-Gallai-Linear in secundum and operation, and also the amortized number of operation/even sequence.

The most interesting data of Figure 6 are in the last column: they show that the number of operations/investigated sequence/length of the investigated sequence is monotone decreasing (see [69]).

Figure 7 shows the distribution of the $E(n) - G(n)$ even nongraphical sequences according to the number of tests made by Erdős-Gallai-Jumping to exclude the given sequence for $n = 3, \ldots, 15$ vertices. $f_i(n) = f_i$ gives the frequency of even nongraphical sequences of length n, which requeired exactly i round of the test.

These data show, that the maximal number of tests is about $\frac{n}{2}$ in all lines.

Figure 8 shows the average number of required rounds for the nongraphical, graphical and all even sequences. The data of the column belonging to $G(n)$ are computed using Lemma 17. It is remarkable that the sequences of the coefficients are monotone decreasing in the last three columns.

Figure 9 presents the distribution of the graphical sequences according to their first element. These data help at the design of the algorithm Erdős-Gallai-Enumerating which computes the new values of $G(n)$ (in the slicing of the computations belonging to a given value of n).

n	$E(n)$	$G(n)$	$E(n) - G(n)$	average of $E(n) - G(n)$	average of $G(n)$	average of $E(n)$
3	6	4	2	$0.3333n$	$0.8000n$	$0.6444n$
4	19	11	8	$0.3125n$	$0.5714n$	$0.4661n$
5	66	31	35	$0.2114n$	$0.5555n$	$0.3730n$
6	236	102	134	$0.1967n$	$0.5455n$	$0.3730n$
7	868	342	526	$0.1649n$	$0.5385n$	$0.3475n$
8	3233	1213	2020	$0.1458n$	$0.5333n$	$0.2911n$
9	12190	4363	7829	$0.1337n$	$0.5294n$	$0.2753n$
10	46232	16016	30216	$0.1249n$	$0.5263n$	$0.2700n$
11	174484	59348	115136	$0.1175n$	$0.5238n$	$0.2557n$
12	676270	222117	454153	$0.1085n$	$0.5217n$	$0.2444n$
13	2603612	836313	1767299	$0.1035n$	$0.5200n$	$0.2373n$
14	10030008	3166852	6863156	$0.0960n$	$0.5185n$	$0.2294n$
15	38761096	12042620	26718476	$0.0934n$	$0.5172n$	$0.2251n$

Figure 8: Weighted average number of tests made by ERDŐS-GALLAI-JUMPING while investigating the even sequences for $n = 3, \ldots, 15$

n/b_1	0	1	2	3	4	5	6	7	8	9	10	11
1	1											
2	1	1										
3	1	1	2									
4	1	1	4	4								
5	1	2	7	10	11							
6	1	3	10	22	35	31						
7	1	3	14	34	78	110	102					
8	1	4	18	54	138	267	389	342				
9	1	4	23	74	223	503	968	1352	1213			
10	1	5	28	104	333	866	1927	3496	4895	4361		
11	1	5	34	134	479	1356	3471	7221	12892	17793	16016	
12	1	6	40	176	661	2049	5591	13270	27449	47757	65769	59348

Figure 9: The distribution of the graphical sequences according to b_1 for $n = 1, \ldots, 12$

We see in Figure 9 that from $n = 6$ the multiplicities increase up to $n - 2$, and the last positive value is smaller then the last but one element.

7 Enumerative results

Until now for example Avis and Fukuda [4], Barnes and Savage [5, 6], Burns [13], Erdős and Moser [59], Erdős and Richmond [22], Frank, Savage and Selers [25], Kleitman and Winston [47], Kleitman and Wang [46], Metropolis and Stein [56], Rødseth et al. [68], Ruskey et al. [69], Stanley [78], Simion [71] and Winston and Kleitman [86] published results connected with the enumeration of degree sequences. Results connected with the number of sequences investigated by us can be found in the books of Sloane és Ploffe [76], further Stanley [79] and in the free online database *On-line Encyclopedia of Integer Sequences* [73, 74, 75]

It is easy to show, that if l, u and m are integers, further $u \geq l$, $m \geq 1$, and $l \leq b_i \leq u$ for $i = 1, \ldots, m$, then the number of (l, u, m)-bounded sequences $a = (a_1, \ldots, a_m)$ of integer numbers $K(l, u, m)$ is

$$K(l, u, m) = (u - l + 1)^m. \tag{21}$$

It is known (e.g. see [39, page 65]), that if l, u and m are integers, further $u \geq l$ and $m \geq 1$, and $u \geq b_1 \geq \cdots \geq b_n \geq l$, then the number of (l, u, m)-regular sequences of integer numbers $R(l, u, m)$ is

$$R(l, u, m) = \binom{u - l + m}{m}. \tag{22}$$

The following two special cases of (22) are useful in the design of the algorithm Erdős-Gallai-Enumerating.

If $n \geq 1$ is an integer, then the number of $R(0, n-1, n)$-regular sequences is

$$R(0, n-1, n) = R(n) = \binom{2n-1}{n}. \tag{23}$$

If $n \geq 1$ is an integer, then the number of $R(1, n-1, n)$-regular sequences is

$$R(1, n-1, n) = R_z(n) = \binom{2n-2}{n}. \tag{24}$$

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

Lemma 10 (Ascher [3], Sloane, Pfoffe [76]) *If*

Lemma 11 *lemma-En* $n \geq 1$, *then the number of* n*-even sequences* $E(n)$ *is*

$$E(n) = \frac{1}{2}\left(\binom{2n-1}{n} + \binom{n-1}{\lfloor n \rfloor}\right). \tag{25}$$

Proof. See [3, 76]. □

At the designing and analysis of the results of the simulation experiments is useful, if we know some features of the functions $R(n)$ and $E(n)$.

Lemma 12 *If* $n \geq 1$, *then*

$$\frac{R(n+2)}{R(n+1)} > \frac{R(n+1)}{R(n)}, \tag{26}$$

$$\lim_{n\to\infty} \frac{R(n+1)}{R(n)} = 4, \tag{27}$$

further

$$\frac{4^n}{\sqrt{4\pi n}}\left(1 - \frac{1}{2n}\right) < R(n) < \frac{4^n}{\sqrt{4\pi n}}\left(1 - \frac{1}{8n+8}\right). \tag{28}$$

Proof. On the base of (23) we have

$$\frac{R(n+2)}{R(n+1)} = \frac{(2n+3)!(n+1)n!}{(n+2)!(n+1)!(2n+1)!} = \frac{4n+6}{n+2} = 4 - \frac{2}{n+2}, \tag{29}$$

from where we get directly (26) and (27). □

Using Lemma 13 we can give the precise asymptotic order of growth of $E(n)$.

Lemma 13 *If* $n \geq 1$, *then*

$$\frac{E(n+2)}{E(n+1)} > \frac{E(n+1)}{E(n)}, \tag{30}$$

$$\lim_{n\to\infty} \frac{E(n+1)}{E(n)} = 4, \tag{31}$$

further

$$\frac{4^n}{\sqrt{\pi n}}(1 - D_3(n)) < E(n) < \frac{4^n}{\sqrt{\pi n}}(1 - D_4(n)), \tag{32}$$

where $D_3(n)$ *and* $D_4(n)$ *are monotone decreasing functions tending to zero.*

Proof. The proof is similar to the proof of Lemma 12. □

Comparison of (23) and Lemma 13 shows, that the order of growth of numbers of even and odd sequences is the same, but there are more even sequences than odd. Figure 1 contains the values of $R(n)$, $E(n)$ and $E(n)/R(n)$ for $n = 1, \ldots, 37$.

As the next assertion and Figure 1 show, the sequence of the ratios $E(n)/R(n)$ is monotone decreasing and tends to $\frac{1}{2}$.

Corollary 14 *If* $n \geq 1$*, then*

$$\frac{E(n+1)}{R(n+1)} < \frac{E(n)}{R(n)} \tag{33}$$

and

$$\lim_{n\to\infty} \frac{E(n)}{R(n)} = \frac{1}{2}. \tag{34}$$

Proof. This assertion is a direct consequence of (23) and (25).

□

The expected value of the number of jumping elements has a substantial influence on the running time of algorithms using the jumping elements. Therefore the following two assertions are useful.

The number of different elements in an n-bounded sequence b is called *the rainbow number* of the sequence, and it will be denoted by $r_n(b)$.

Lemma 15 *Let* b *be a random* n*-bounded sequence. Then the expectation and variance of its rainbow number are as follows.*

$$E[r_n(b)] = n\left[1 - \left(1 - \frac{1}{n}\right)^n\right] = n\left(1 - \frac{1}{e}\right) + O(1), \tag{35}$$

$$\begin{aligned} Var[r_n(b)] &= n\left(1 - \frac{1}{n}\right)^n \left[1 - \left(1 - \frac{1}{n}\right)^n\right] \\ &\quad + n(n-1)\left[\left(1 - \frac{2}{n}\right)^n - \left(1 - \frac{1}{n}\right)^{2n}\right] \\ &= \frac{n}{e}\left(1 - \frac{2}{e}\right) + O(1). \end{aligned} \tag{36}$$

Proof. Let ξ_i denote the indicator of the event that number i is not contained in a random n-bounded sequence. Then the rainbow number of a random

sequence is $n-\sum_{i=0}^{n-1}\xi_i$, hence its expectation equals $n-\sum_{i=0}^{n-1}E[\xi_i]$. Clearly,

$$E[\xi_i]=\left(1-\frac{1}{n}\right)^n \tag{37}$$

holds independently of i, thus

$$E[r_n(b)]=n\left[1-\left(1-\frac{1}{n}\right)^n\right]. \tag{38}$$

On the other hand,

$$Var[r_n(b)]=Var\left[\sum_{i=0}^{n-1}\xi_i\right]=\sum_{i=0}^{n-1}Var[\xi_i]+2\sum_{0\le i<j\le n-1}cov[\xi_i,\xi_j]. \tag{39}$$

Here

$$Var[\xi_i]=\left(1-\frac{1}{n}\right)^n\left[1-\left(1-\frac{1}{n}\right)^n\right], \tag{40}$$

and

$$cov[\xi_i,\xi_j]=E[\xi_i\xi_j]-E[\xi_i]E[\xi_j]=\left(1-\frac{2}{n}\right)^n-\left(1-\frac{1}{n}\right)^{2n}, \tag{41}$$

implying (36). □

We remark that this lemma answers a question of Imre Kátai [40] posed in connection with the speed of computers having interleaved memory and with checking algorithms of some puzzles (e.g sudoku).

Lemma 16 *The number of* $(0,n-1,m)$*-regular sequences composed from* k *distinct numbers is*

$$\binom{n}{k}\binom{m-1}{k},\ k=1,\dots,n. \tag{42}$$

In other words, the distribution of the rainbow number $r_n(b)$ *of a random* $(0,n-1,m)$*-regular sequence* b *is hypergeometric with parameters* $n+m-1$*,* n*, and* m*.*

Proof. The k-set of distinct elements of the sequence can be selected from $\{0,1,\dots,n-1\}$ in $\binom{n}{k}$ ways. Having this values selected we can tell their multiplicities in $\binom{m-1}{k-1}$ ways. Let us consider the k blocks of identical elements. The first one starts with b_1, and the starting position of the other $k-1$ blocks can be selected in $\binom{m-1}{k-1}$ ways. □

From this the expectation and the variance of a random n-regular sequence follow immediately.

Corollary 17 *Let* b *be a random* n*-regular sequence. Then the expectation and the variance of its rainbow number* $r_n(b)$ *are as follows:*

$$E[r_n(b)] = \frac{n^2}{2n-1} = \frac{n}{2} + \frac{n}{4n-2} = \frac{n}{2} + O(1), \tag{43}$$

$$Var[r_n(b)] = \frac{n^2(n-1)}{2(2n-1)^2} = \frac{n}{8} + \frac{n}{128n^2-128n+32} = \frac{n}{8} + O(1). \tag{44}$$

Lemma 18 *Let* b *be a random* n*-regular sequence. Let us write it in the form* $b = (b_1^{e_1}, \dots, b_r^{e_r})$*. Then the expected value of the exponents* e_j *is*

$$E[e_j \mid r(b) \geq j] = 4 + o(1). \tag{45}$$

Proof. Let $c(n,j)$ denote the number of n-regular sequences with rainbow number not less than j. By Lemma 16,

$$c(n,j) = \sum_{k=j}^{n} \binom{n}{k}\binom{n-1}{k-1}. \tag{46}$$

Let us turn to the number of n-regular sequences with rainbow number not less than j and $e_j = \ell$. This is equal to the number of $(0, n-1, n-\ell+1)$-regular sequences containing at least j different numbers, that is,

$$\sum_{k=j}^{n} \binom{n}{k}\binom{n-\ell}{k-1}. \tag{47}$$

From this the sum of e_j over all n-regular sequences with $e_j > 0$ is equal to

$$\begin{aligned}\sum_{\ell=1}^{n-j+1} \ell \sum_{k=j}^{n} \binom{n}{k}\binom{n-\ell}{k-1} &= \sum_{k=j}^{n} n\binom{n}{k} \sum_{\ell=1}^{n-j+1} \binom{\ell}{1}\binom{n-\ell}{k-1} \\ &= \sum_{k=j}^{n} \binom{n}{k}\binom{n+1}{k+1} = c(n+1, j+1). \end{aligned} \tag{48}$$

This can also be seen in a more direct way. Consider an arbitrary n-regular sequence with at least $j+1$ blocks, then substitute the elements of the $j+1$st block with the number in the jth block (that is, concatenate this two adjacent blocks) and delete one element from the united block; finally, decrease by 1 all elements in the subsequent blocks. In this way one obtains an n-regular

sequence with at least j blocks, and it easy to see that every such sequence is obtained exactly e_j times.

Thus the expectation to be computed is just

$$\frac{c(n+1,j+1)}{c(n,j)}. \tag{49}$$

Clearly, $c(n,1) = R(0,n-1,n) = \binom{2n-1}{n}$, hence

$$c(n,j) = \binom{2n-1}{n} - \sum_{k=1}^{j-1} \binom{n}{k}\binom{n-1}{k-1} = \binom{2n-1}{n} + O\left(n^{2j-3}\right), \tag{50}$$

as $n \to \infty$. This is asymptotically equal to

$$\frac{\binom{2n+1}{n+1}}{\binom{2n-1}{n}} = \frac{4n+2}{n+1} = 4 - \frac{2}{n+1} = 4 + o(1). \tag{51}$$

□

It is interesting to observe that by (43) the average block length in a random n-regular sequence is

$$\frac{1}{r}\sum_{j=1}^{r} e_j = \frac{n}{r(b)} \approx 2 \tag{52}$$

approximately, as $n \to \infty$. This fact could be interpreted as if blocks in the beginning of the sequence were significantly longer. However, fixing $r_n(b) = r$ we find that the lengths of the r blocks are exchangeable random variables with equal expectation n/r. At first sight this two facts seem to be in contradiction. The explanation is that exchangability only holds conditionally. Blocks in the beginning do exist even for smaller rainbow numbers, when the average block length is big, while blocks with large index can only appear when there are many short blocks in the sequence.

The following assertion gives the number of zerofree sequences and the ratio of the numbers of zerofree and regular sequences.

Corollary 19 *The number of the zerofree n-regular sequences $R_z(n)$ is*

$$R_z(n) = \binom{2n-2}{n-1} \tag{53}$$

and

$$\lim_{n\to\infty} \frac{R_z(n)}{R(n)} = \frac{1}{2}. \tag{54}$$

Proof. (53) identical with (22), (54) is a direct consequence of (22) and (23). □

As the experimental data in Figure 3 show, $\frac{E_z(n)}{R(n)} \approx \frac{1}{4}$.

The following lemma allows that the algorithm ERDŐS-GALLAI-ENUMERATING tests only the zerofree even sequences instead of the even sequences.

Lemma 20 *If* $n \geq 2$, *then the number of the* n*-graphical sequences* $G(n)$ *is*

$$G(n) = G(n-1) + G_z(n). \tag{55}$$

Proof. If an n-graphical sequence b contains at least one zero, that is $b_n = 0$, then $b' = (b_1, \ldots, b_{n-1})$ is $(n-1)$-graphical or not. If $a = (a_1, \ldots, a_{n-1})$ is an $(n-1)$-graphical sequence, then $a' = (a_1, \ldots, a_{n-1}, 0)$ is n-graphical.

The set of the n-graphical sequences S consists of two subsets: set of zerofree sequences S_1 and the set of sequences S_2 containing at least one zero. There is a bijection between the set of the $(n-1)$-graphical sequences and such n-graphical sequences, which contain at least one zero. Therefore $|S| = |S_1| + |S_2| = G_z(n) + G(n-1)$. □

Corollary 21 *If* $n \geq 1$, *then*

$$G(n) = 1 + \sum_{i=2}^{n} G_z(n). \tag{56}$$

Proof. Concrete calculation gives $G(1) = 1$. Then using (55) and induction we get (56). □

A promising direction of researches connected with the characterization of the function $G(n)$ is the decomposition of the even integers into members and the investigation, which decompositions represent a graphical sequence [5, 6, 13]. Using this approach Burns proved the following asymptotic bounds in 2007.

Theorem 22 (Burns [13]) *There exist such positive constants* c *and* C, *that the following bounds of the function* $G(n)$ *is true:*

$$\frac{4^n}{cn} < G(n) < \frac{4^n}{(\log n)^C \sqrt{n}}. \tag{57}$$

Proof. See [13]. □

This result implies that the asymptotic density of the graphical sequences is zero among the even sequences.

Corollary 23 *If* $n \geq 1$*, then there exists a positive constant* C *such that*

$$\frac{G(n)}{E(n)} < \frac{1}{(\log_2 n)^C} \tag{58}$$

and

$$\lim_{n\to\infty} \frac{G(n)}{E(n)} = 0. \tag{59}$$

Proof. (58) is a direct consequence of (25) and (58), and (58) implies (59). □

As Figure 1 and Figure 3 show, the convergence of the ratio $G(n)/E(n)$ is relative slow.

8 Number of graphical sequences

Erdős-Gallai-Enumerating algorithm (EGE) [37] generates and tests for given n every zerofree even sequence. Its input is n and output is the number of corresponding zerofree graphical sequences $G_z(n)$.

The algorithm is based on Erdős-Gallai-Linear algorithm. It generates and tests only the zerofree even sequences, that is according to Corollary 5 and Figure 3 about the 25 percent of the n-regular sequences.

EGE tests the input sequences only in the checking points. Corollary 17 shows that about the half of the elements of the input sequences are check points.

Figure 3 contains data showing that EGE investigates even less than the half of the elements of the input sequences.

Important property of EGE is that it solves in $O(1)$ expected time

1. the generation of one input sequence;
2. the updating of the vector H;
3. the updating of the vector of checking points;
4. the updating of the vector of the weight points.

We implemented the parallel version of EGE (EGEP). It was run on about 200 PC's containing about 700 cores. The total running time of EGEP is contained in Figure 10

n	running time (in days)	number of slices
24	7	415
25	26	415
26	70	435
27	316	435
28	1130	2001
29	6733	15119

Figure 10: The runnng time of EGEP for $n = 24, \ldots, 29$

The pseudocode of the algorithm see in [37]. The amortized running time of this algorithm for a sequence is $\Theta(1)$, so the total running time of the whole program is $O(E(n))$.

9 Summary

In Figure 1 the values of $R(n)$ up to $n = 24$ are the elements of the sequence A001700 of OEIS [73], the values of $E(n)$ up to $n = 23$ are the elements of the sequence A005654 [75] of the OEIS, and in Figure 2 the values $G(n)$ are up to $n = 23$ are the elements of sequence A0004251-es [74] of OEIS. The remaining values are new [37, 36].

Figure 2 contains the number of graphical sequences $G(n)$ for $n = 1, \ldots, 29$, and also $G(n+1)/G(n)$ for $n = 1, \ldots, 28$.

The referenced manuscripts, programs and further simulation results can be found at the homepage of the authors, among others at `http://compalg.inf.elte.hu/~tony/Kutatas/EGHH/`

Acknowledgements

The authors thank Zoltán Király (Eötvös Loránd University, Faculty of Science, Dept. of Computer Science) for his advice concerning the weight points, Antal Sándor and his colleagues (Eötvös Loránd University, Faculty of Informatics), further Ádám Mányoki (TFM World Kereskedelmi és Szolgáltató Kft.) for their help in running of our timeconsuming programs and the unknown referee for the useful corrections. The European Union and the European Social Fund have provided financial support to the project under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

[1] M. Anholcer, V. Babiy, S. Bozóki, W. W. Koczkodaj, A simplified implementation of the least squares solution for pairwise comparisons matrices. *CEJOR Cent. Eur. J. Oper. Res.* **19,** 4 (2011) 439–444. ⇒231

[2] S. R. Arikati, A. Maheshwari, Realizing degree sequences in parallel. *SIAM J. Discrete Math.* **9,** 2 (1996) 317–338. ⇒231

[3] M. Ascher, Mu torere: an analysis of a Maori game, *Math. Mag.* **60,** 2 (1987) 90–100. ⇒253, 254

[4] D. Avis, K. Fukuda, Reverse search for enumeration, *Discrete Appl. Math.* **2,** 1-3 (1996) 21–46. ⇒253

[5] T. M. Barnes, C. D. Savage, A recurrence for counting graphical partitions, *Electron. J. Combin.* **2** (1995), Research Paper 11, 10 pages (electronic). ⇒253, 259

[6] T. M. Barnes, C. D. Savage, Efficient generation of graphical partitions, *Discrete Appl. Math.* **78,** 1-3 (1997) 17–26. ⇒230, 253, 259

[7] L. B. Beasley, D. E. Brown, K. B. Reid, Extending partial tournaments, *Math. Comput. Modelling* **50,** 1 (2009) 287–291. ⇒231

[8] S. Bereg, H. Ito, Transforming graphs with the same degree sequence, *The Kyoto Int. Conf. on Computational Geometry and Graph Theory* (ed. by H. Ito et al.), LNCS **4535,** Springer-Verlag, Berlin, Heidelberg, 2008. pp. 25–32. ⇒231

[9] N. Bödei, *Degree sequences of graphs* (Hungarian), Mathematical master thesis (supervisor A. Frank), Dept. of Operation Research of Eötvös Loránd University, Budapest, 2010, 43 pages. ⇒231, 232, 233

[10] S. Bozóki S., J. Fülöp, A. Poesz, On pairwise comparison matrices that can be made consistent by the modification of a few elements. *CEJOR Cent. Eur. J. Oper. Res.* **19** (2011) 157–175. ⇒231

[11] Bozóki S., J. Fülöp, L. Rónyai: On optimal completion of incomplete pairwise comparison matrices, *Math. Comput. Modelling* **52** (2010) 318–333. ⇒231

[12] A. R. Brualdi, K. Kiernan, Landau's and Rado's theorems and partial tournaments, *Electron. J. Combin.* **16,** #N2 (2009) 6 pages. ⇒231

[13] J. M. Burns, *The Number of Degree Sequences of Graphs*, PhD Dissertation, MIT, 2007. ⇒253, 259, 260

[14] A. N. Busch, G. Chen, M. S. Jacobson, Transitive partitions in realizations of tournament score sequences, *J. Graph Theory* **64,** 1 (2010), 52–62. ⇒231

[15] S. A. Choudum, A simple proof of the Erdős-Gallai theorem on graph sequences, *Bull. Austral. Math. Soc.* **33** (1986) 67–70. ⇒233

[16] J. Cooper, L. Lu, Graphs with asymptotically invariant degree sequences under restriction, *Internet Mathematics* **7,** 1 67–80. ⇒231

[17] 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. ⇒245, 248

[18] S. De Agostino, R. Petreschi, Parallel recognition algorithms for graphs with restricted neighbourhoods. *Internat. J. Found. Comput. Sci.* **1,** 2 (1990) 123–130. ⇒231

[19] C. I. Del Genio, H. Kim, Z. Toroczkai, K. E. Bassler, Efficient and exact sampling of simple graphs with given arbitrary degree sequence, *PLoS ONE* **5,** 4 (2010) e10012. ⇒231

[20] A. Dessmark, A. Lingas, O. Garrido, On parallel complexity of maximum f-matching and the degree sequence problem. *Mathematical Foundations of Computer Science 1994* (Košice, 1994), LNCS **841,** Springer, Berlin, 1994, 316–325. ⇒231

[21] P. Erdős, T. Gallai, Graphs with prescribed degrees of vertices (Hungarian), *Mat. Lapok* **11** (1960) 264–274. ⇒230, 231, 233

[22] P. Erdős, L. B. Richmond, On graphical partitions, *Combinatorica* **13,** 1 (1993) 57–63. ⇒253

[23] P. L. Erdős, I. Miklós, Z. Toroczkai, A simple Havel-Hakimi type algorithm to realize graphical degree sequences of directed graphs, *Electron. J. Combin.* **17,** 1 (2010) R66 (10 pages). ⇒231, 233

[24] A. Frank, *Connections in Combinatorial Optimization,* Oxford University Press, Oxford, 2011. ⇒231

[25] D. A. Frank, C. D. Savage, J. A. Sellers, On the number of graphical forest partitions, *Ars Combin.* **65** (2002) 33–37. ⇒253

[26] 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. ⇒ 230, 231, 232

[27] S. L. Hakimi, On the degrees of the vertices of a graph, *F. Franklin Institute,* **279,** 4 (1965) 290–308. ⇒233

[28] V. Havel, A remark on the existence of finite graphs (Czech), *Časopis Pěst. Mat.* **80** (1955), 477–480. ⇒230, 232

[29] P. Hell, D. Kirkpatrick, Linear-time certifying algorithms for near-graphical sequences, *Discrete Math.* **309,** 18 (2009) 5703–5713. ⇒231

[30] A. Iványi, Reconstruction of complete interval tournaments, *Acta Univ. Sapientiae, Inform.* **1,** 1 (2009) 71–88. ⇒231, 233

[31] A. Iványi, Reconstruction of complete interval tournaments. II, *Acta Univ. Sapientiae, Math.* **2,** 1 (2010) 47–71. ⇒231, 233

[32] A. Iványi, Deciding the validity of the score sequence of a soccer tournament, in: *Open problems of the Egerváry Research Group,* ed. by A. Frank, Budapest, 2011. ⇒231

[33] A. Iványi, Directed graphs with prescribed score sequences, *The 7th Hungarian-Japanese Symposium on Discrete Mathematics and Applications* (Kyoto, May 31–June 3, 2011, ed by S. Iwata), 114–123. ⇒231

[34] A. Iványi, *Memory management,* in: *Algorithms of Informatics* (ed by. A. Iványi), AnTonCom, Budapest, 2011, 797–847. ⇒240

[35] A. Iványi, L. Lucz, Erdős-Gallai test in linear time, *Combinatorica* (submitted). ⇒231, 232, 246, 248

[36] A. Iványi, L. Lucz, Parallel Erdős-Gallai algorithm, *CEJOR Cent. Eur. J. Oper. Res.* (submitted). ⇒231, 232, 261

[37] A. Iványi, L. Lucz, T. F. Móri, P. Sótér, Linear Erdős-Gallai test (Hungarian), *Alk. Mat. Lapok* (submitted). ⇒ 231, 232, 235, 244, 246, 248, 260, 261

[38] A. Iványi, S. Pirzada, Comparison based ranking, in: *Algorithms of Informatics, Vol. 3,* ed. A. Iványi, AnTonCom, Budapest 2011, 1209–1258. ⇒ 231

[39] A. Járai, *Introduction to Mathematics* (Hungarian). ELTE Eötvös Kiadó, Budapest, 2005. ⇒ 253

[40] I. Kátai, Personal communication, Budapest, 2010. ⇒ 256

[41] K. K. Kayibi, M. A. Khan, S. Pirzada, A. Iványi, Random sampling of minimally cyclic digraphs with given imbalance sequence, *Acta Univ. Sapientiae, Math.* (submitted). ⇒ 231

[42] G. Kéri, On qualitatively consistent, transitive and contradictory judgment matrices emerging from multiattribute decision procedures, *CEJOR Cent. Eur. J. Oper. Res.* **19,** 2 (2011) 215–224. ⇒ 231

[43] K. Kern, D. Paulusma, The new FIFA rules are hard: complexity aspects of sport competitions, *Discrete Appl. Math.* **108,** 3 (2001) 317–323. ⇒ 231

[44] K. Kern, D. Paulusma, The computational complexity of the elimination problem in generalized sports competitions, *Discrete Optimization* **1**, 2 (2004) 205–214. ⇒ 231

[45] H. Kim, Z. Toroczkai, I. Miklós, P. L. Erdős, l. A. Székely, Degree-based graph construction, *J. Physics: Math. Theor. A* **42,** 39 (2009) 392–401. ⇒ 231

[46] D. J. Kleitman, D. L. Wang, Algorithms for constructing graphs and digraphs with given valencies and factors, *Discrete Math.* **6** (1973) 79–88. ⇒ 253

[47] D. J. Kleitman, K. J. Winston, Forests and score vectors, *Combinatorica* **1,** 1 (1981) 49–54. ⇒ 253

[48] D. E. Knuth, *The Art of Computer Programming. Volume 4A, Combinatorial Algorithms,* Addison–Wesley, Upper Saddle River, 2011. ⇒ 231

[49] A. Kohnert, Dominance order and graphical partitions, *Elec. J. Comb.* **11,** 1 (2004) No. 4. 17 pp. ⇒230

[50] M. D. LaMar, Algorithms for realizing degree sequences of directed graphs. arXiv-0906:0343ve [math.CO], 7 June 2010. ⇒231

[51] H. G. Landau, On dominance relations and the structure of animal societies. III. The condition for a score sequence, *Bull. Math. Biophys.* **15** (1953) 143–148. ⇒231

[52] F. Liljeros, C. R. Edling, L. Amaral, H. Stanley, Y. Áberg, The web of human sexual contacts, *Nature* **411**, 6840 (2001) 907–908. ⇒231

[53] L. Lovász, *Combinatorial Problems and Exercises* (corrected version of the second edition), AMS Chelsea Publishing, Boston, 2007. ⇒232

[54] L. Lucz, A. Iványi, P. Sótér, S. Pirzada, Testing and enumeration of football sequences, *Abstracts of MaCS 2012*, ed. by Z. Csörnyei (Siófok, February 9–12, 2012). ⇒231

[55] D. Meierling, L. Volkmann, A remark on degree sequences of multigraphs, *Math. Methods Oper. Res.* **69,** 2 (2009) 369–374. ⇒231

[56] N. Metropolis, P. R. Stein, The enumeration of graphical partitions, *European J. Comb.* **1,** 2 (1980) 139–153. ⇒253

[57] I. Miklós, Graphs with prescribed degree sequences (Hungarian), Lecture in Alfréd Rényi Institute of Mathematics, 16 November 2009. ⇒231

[58] I. Miklós, P. L. Erdős, L. Soukup, A remark on degree sequences of multigraphs (submitted). ⇒231

[59] J. W. Moon, *Topics on Tournaments,* Holt, Rinehart, and Winston, New York, 1968. ⇒253

[60] T. V. Narayana, D. H. Bent, Computation of the number of score sequences in round-robin tournaments, *Canad. Math. Bull.* **7,** 1 (1964) 133–136. ⇒231

[61] M. E. J. Newman, A. L. Barabási, *The Structure and Dynamics of Networks*, Princeton University Press, Princeton, NJ. 2006. ⇒231

[62] G. Pécsy, L. Szűcs, Parallel verification and enumeration of tournaments, *Stud. Univ. Babeş-Bolyai*, *Inform.* **45,** 2 (2000) 11–26. ⇒231

[63] S. Pirzada, *Graph Theory,* Orient Blackswan (to appear). ⇒231

[64] S. Pirzada, A. Iványi, Imbalances in digraphs, *Abstracts of MaCS 2012*, ed. by Z. Csörnyei (Siófok, February 9–12, 2012). ⇒231

[65] S. Pirzada, A. Iványi, M. A. Khan, Score sets and kings, in: *Algorithms of Informatics, Vol. 3,* ed. by A. Iványi. AnTonCom, Budapest 2011, 1410–1450. ⇒231

[66] S. Pirzada, T. A. Naikoo, U. T. Samee, A. Iványi, Imbalances in directed multigraphs, *Acta Univ. Sapientiae, Inform.* **2,** 1 (2010) 47–71. ⇒231

[67] S. Pirzada, G. Zhou, A. Iványi, On k-hypertournament losing scores, *Acta Univ. Sapientiae, Inform.* **2,** 2 (2010) 184–193. ⇒231

[68] Ø J. Rødseth, J. A. Sellers, H. Tverberg, Enumeration of the degree sequences of non-separable graphs and connected graphs, *European J. Comb.* **30,** 5 (2009) 1309–1319. ⇒231, 253

[69] F. Ruskey, R. Cohen, P. Eades, A. Scott, Alley CAT's in search of good homes, *Congr. Numer.* **102** (1994) 97–110. ⇒230, 251, 253

[70] G. Sierksma, H. Hoogeveen, Seven criteria for integer sequences being graphic, *J. Graph Theory* **15,** 2 (1991) 223–231. ⇒233

[71] R. Simion, Convex polytopes and enumeration, *Advances in Applied Math.* **18,** 2 (1996) 149–180. ⇒253

[72] N. J. A. Sloane (Ed.), *Encyclopedia of Integer Sequences,* 2011. ⇒231

[73] N. J. A. Sloane, The number of ways to put $n+1$ indistinguishable balls into $n+1$ distinguishable boxes, in: *The On-line Encyclopedia of the Integer Sequences* (ed. by N. J. A. Sloane), 2011. ⇒253, 261

[74] N. J. A. Sloane, The number of degree-vectors for simple graphs, in: *The On-line Encyclopedia of the Integer Sequences* (ed. by N. J. A. Sloane), 2011. ⇒230, 253, 261

[75] N. J. A. Sloane, The number of bracelets with n red, 1 pink and $n-1$ blue beads, in: *The On-line Encyclopedia of the Integer Sequences* (ed. by N. J. A. Sloane), 2011. ⇒253, 261

[76] N. J. A. Sloane, S. Plouffe, *The Encyclopedia of Integer Sequences,* Academic Press, Waltham, MA, 1995. ⇒253, 254

[77] D. Soroker, *Optimal parallel construction of prescribed tournaments, Discrete Appl. Math.* **29,** 1 (1990) 113–125. ⇒231

[78] R. P. Stanley, A zonotope associated with graphical degree sequence, in: *Applied geometry and discrete mathematics, Festschr. 65th Birthday Victor Klee.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science. **4** (1991) 555-570. ⇒253

[79] R. P. Stanley, *Enumerative Combinatorics*, Cambridge University Press, Cambridge, 1997. ⇒253

[80] J. Temesi, Pairwise comparison matrices and the error-free property of the decision maker, *CEJOR Cent. Eur. J. Oper. Res.* **19,** 2 (2011) 239–249. ⇒231

[81] A. Tripathi, H. Tyagi, A simple criterion on degree sequences of graphs, *Discrete Appl. Math.* **156,** 18 (2008) 3513–3517. ⇒231

[82] A. Tripathi, S. Vijay, A note on a theorem of Erdős & Gallai, *Discrete Math.* **265,** 1-3 (2003) 417–420. ⇒244, 245

[83] A. Tripathi, S. Venugopalanb, D. B. West, A short constructive proof of the Erdös-Gallai characterization of graphic lists, *Discrete Math.* **310,** 4 (2010) 833–834. ⇒230, 231, 233, 235

[84] E. W. Weisstein, *Degree Sequence,* From MathWorld—Wolfram Web Resource, 2011. ⇒231

[85] E. W. Weisstein, *Graphic Sequence,* From MathWorld—Wolfram Web Resource, 2011. ⇒231

[86] K. J. Winston, D. J. Kleitman, On the asymptotic number of tournament score sequences, *J. Combin. Theory Ser. A.* **35** (1983) 208–230. ⇒253

Received: September 30, 2011 • Revised: November 10, 2011

Contents
Volume 3, 2011

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.

Acta Universitatis Sapientiae, Informatica

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 needed 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