Acta Universitatis Sapientiae

Informatica

Volume 1, Number 1, 2009

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

ALGORITHMS OF INFORMATICS
vol. 1. Foundations, vol. 2. Applications

Editor: Antal Iványi
monAt Kiadó (http://www.mondat.hu), Budapest, 2007.
Distributor: AnTonCom (http://www.antoncom.hu/)
ISBN 978-963-87596-1-0, ISBN 978-963-87596-2-7

The chapters of the first volume are divided into three parts. The chapters of Part 1 are connected with automata: *Automata and Formal Languages* (written by Zoltán Kása, Babeş-Bolyai University of Cluj-Napoca), *Compilers* (Zoltán Csörnyei, Eötvös Loránd University), *Compression and Decompression* (Ulrich Tamm, Chemnitz University of Technology Commitment), *Reliable Computations* (Péter Gács, Boston University).

The chapters of Part 2 have algebraic character: here are the chapters *Algebra* (written by Gábor Ivanyos, and Lajos Rónyai, Budapest University of Technology and Economics), *Computer Algebra* (Antal Járai, Attila Kovács, Eötvös Loránd University), further *Cryptology* and *Complexity Theory* (Jörg Rothe, Heinrich Heine University).

The chapters of the third part contain *Competitive Analysis* (Csanád Imreh, University of Szeged), *Game Theory* (Ferenc Szidarovszky, The University of Arizona), *Recurrences* (Zoltán Kása, Babeş-Bolyai University) and *Scientific Computations* (Aurél Galántai, András Jeney, University of Miskolc).

The second volume is also divided into three parts. The chapters of Part 4 are *Distributed Algorithms* (Burkhard Englert, California State University; Dariusz Kowalski, University of Liverpool; Grzegorz Malewicz, University of Alabama; Alexander Allister Shvartsman, University of Connecticut), *Parallel Computation* (Antal Iványi, Eötvös Loránd University; Claudia Leopold, University of Kassel), *Network Simulation* (Tibor Gyires, Illinois State University) and *Systolic Systems* (Eberhard Zehendner, Friedrich Schiller University).

The chapters of Part 5 are *Relational Databases* and *Query in Relational Databases* (János Demetrovics, Eötvös Loránd University; Attila Sali, Alfréd Rényi Institute of Mathematics), *Semistructured Data Bases* (Attila Kiss, Eötvös Loránd University) and *Memory Management* (Ádám Balog, Antal Iványi, Eötvös Loránd University).

The chapters of the third part of the second volume have close connections with biology: *Bioinformatics* (István Miklós, Eötvös Loránd University), *Human-Computer Interactions* (Ingo Althöfer, Stefan Schwarz, Friedrich Schiller University), and *Computer Graphics* (László Szirmay-Kalos, Budapest University of Technology and Economics).

The book contains verbal description, pseudocode and analysis of over 200 algorithms, and over 350 figures and 120 examples illustrating how the algorithms work. Each section ends with exercises and each chapter ends with problems. The book contains over 330 exercises and 70 problems.

The book has a web site: http://elek.inf.elte.hu/EnglishBooks, which can be used to obtain a list of known errors, report errors, or make suggestions. The website contains the maintaned PDF version of the bibliography in which the names of the authors, journals and publishers are usually active links to the corresponding web sites.

INSIGHT INTO COMPUTER SCIENCE WITH MAPLE

Zoltán Benyó, Béla Paláncz, László Szilágyi
Scientia Publishing House (http://www.scientiakiado.ro), Cluj-Napoca, 2005.
ISBN: 978-973-7953-56-8

The purpose of the book is to provide a systematic overview of the different areas of computer science with the help of integrated systems like Maple, Mathematica and Mathcad. These are popular representatives of integrated systems providing a new philosophy for using computers in teaching, research and industry. Definitions, principles, methods, techniques and applications illustrated by interactive examples are presented.

Contents: Problems, algorithms and programs, Data structures, Numerical algorithms, Simulation (of the dynamical performance of linear and non-linear syustems), Control (for linear and non-linear systems), Graphs and their applications, Neural networks, Computer graphics, Computer animation, Image processing.

Most of the chapters conclude with exercises.

The book is written basically as a second year text for students in science and engineering, who already have some knowledge of programming and in working with Maple, Mathematica or Mathcad. However, it could be a first year text for mathematics students or students in computer science, who generally acquire skills in these type of packages (early computing/electronics courses that traditionally use C, C++, Java) or some other high level languages in their first year, and could easily acquire the required level of skill in one of these systems in their second year.

This book can also be a helpful source of reference for postgraduate students.

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 5–13

Shepard interpolation with stationary points

László Pál
Sapientia University, Faculty of Business and Humanities, Department of Mathematics and Informatics, Miercurea-Ciuc, Romania
email: pallaszlo@sapientia.siculorum.ro

Róbert Oláh-Gál
Babeş-Bolyai University, Faculty of Mathematics and Informatics, Extension of Miercurea-Ciuc, Romania
email: olah_gal@topnet.ro

Zoltán Makó
Sapientia University, Faculty of Business and Humanities, Department of Mathematics and Informatics, Miercurea-Ciuc, Romania
email: makozoltan@sapientia.siculorum.ro

Abstract. In the paper [10] we analyzed how it is possible to approximate spatial points which mark real methane probes. The origin of the discussed problem is the modelling of real geological reserve calculating. In most of the cases the specialists consider that the contact points of borings to the envelope surface are stationary points. In this paper we will study an interesting feature of the Shepard's interpolation method in m dimensional space, where the control points are stationary and the interpolation function is continuous derivable. Using this interpolation in the real three dimensional space, we will show that the envelope surface may be approximated with this interpolation function.

AMS 2000 subject classifications: 65D05, 65D17
CR Categories and Descriptors: G.1.1. [Interpolation]: Subtopic - Interpolation formulas.
Key words and phrases: interpolation methods, geological reserve calculating

1 Introduction

In the paper [10] we analyzed how it is possible to approximate spatial points which mark real methane probes. The origin of the discussed problem is the modelling of real geological reserve calculating ([5], [8]).

There are given planar, spatial or m dimensional points and we are looking for a smooth hypersurface which could interpolate so that these points are stationary.

The classical methods are the Lagrange interpolation method and Gauss' least square method. Both of them have applicable features but there are also disadvantages. The Lagrange interpolation method may fluctuate between two points while with the Gauss method we must choose the desired hipersurface in advance. None of them is natural because the points provide the only piece of information that we have. There is no rule which could describe the mathematical instruments that we must use with the interpolation and approximation methods. Therefore in most of the cases people choose polynomials of $1, 2, 3, ..., n$ degrees because they can solve them more easily. Thus the modern graphical computing evolved and it has the most flexible methods such as the Bezier curves and the B-splines.

Our task comes from a practical problem: how can we reconstruct the methane reserve samples provided by the probes? We partially solved this problem and the results ware published in our paper [10]. Therefore we analyzed the raised question and mathematically rephrased the problem. Thus we came to the conclusion that we must search for a hypersurface to which the given points are stationary.

During the the examination of the literature we found out that in the papers [1], [9] approximation-interpolation methods were presented to terrain models. This is called the *calculating of the arithmetic mean weighted by the inverse of distance interpolation function* (4).

We have tested and compared the Shepard's interpolation method ([2], [3], [6], [7], [4], [11]) given by the functions (5), (7) in two and three dimensions: *arithmetic mean weighted by the distance approximation* (1), with the *arithmetic mean weighted by the square of the distance approximation* (2), with *Lagrange interpolation* (3), with *the arithmetic mean weighted by the inverse of distance interpolation* (4). These comparisons we summarized in Figure 1 and Figure 2. These figures show with the enumerated methods the evolution of the discrete approximation-interpolation in the two respectively three dimensional space.

We can observe that the first two methods (1), (2) – first two rows from

Figure 1 and Figure 2 – only approximate the control points and the curve or surface is determined by these points. The third method (3) – third row from Figure 1 and Figure 2 – interpolate the control points but these points are not stationary. The fourth method (4)-fourth row from Figure 1 and Figure 2- also interpolates, the control points are extremely, but the curve and surface are not smooth in the control points. In the second section, we will show that the Shepard's method (5), (7) – fifth row from Figure 1 and Figure 2 – also interpolates, the control points are stationary, and the curve and surface are smooth.

In the m dimensional space different $A_i, i = \overline{1,n}$ points are given. We denote by $\mathbf{r}_i$ the positional vector of A_i. For every point A_i we assign a z_i scalar value. Let us define the interpolation functions of the enumerated methods in the following manner:

$$E_1(\mathbf{r}) = \frac{\sum\limits_{i=1}^{n} d_i z_i}{\sum\limits_{i=1}^{n} d_i}; \tag{1}$$

$$E_2(\mathbf{r}) = \frac{\sum\limits_{i=1}^{n} d_i^2 z_i}{\sum\limits_{i=1}^{n} d_i^2}; \tag{2}$$

$$L(\mathbf{r}) = \sum_{i=1}^{n} z_i \frac{\prod\limits_{\substack{k=1 \\ k\neq i}}^{n} (\mathbf{r} - \mathbf{r}_k)}{\prod\limits_{\substack{k=1 \\ k\neq i}}^{n} d_{ki}}; \tag{3}$$

$$E_3(\mathbf{r}) = \begin{cases} \dfrac{\sum\limits_{i=1}^{n} \frac{z_i}{d_i}}{\sum\limits_{i=1}^{n} \frac{1}{d_i}}, & \text{if} \quad \mathbf{r} \neq \mathbf{r}_i, \\ z_i, & \text{if} \quad \mathbf{r} = \mathbf{r}_i, \end{cases} \tag{4}$$

where d_i is the length of the vector $\mathbf{r} - \mathbf{r}_i$ and d_{ki} is the distance between points A_k and A_i.

2 The stationary points of the Shepard's interpolation

2.1 Curve interpolation in the plane

Theorem 1 *Given* (x_i, y_i), $i = \overline{1,n}$ *points in the real plane where* $x_i \neq x_j$ *if* $i \neq j$. *Let us define the* $G : \mathbf{R} \to \mathbf{R}$ *function where*

$$G(x) = \begin{cases} \dfrac{\sum\limits_{i=1}^{n} \frac{y_i}{(x-x_i)^2}}{\sum\limits_{i=1}^{n} \frac{1}{(x-x_i)^2}} & \text{if} \quad x \neq x_i, \\ y_i & \text{if} \quad x = x_i. \end{cases} \tag{5}$$

The G *is continuous derivable and* $G'(x_i) = 0$ *for all* $i = \overline{1,n}$.

The theorem is a particular case of the Theorem 2 in one dimensional space.

2.2 Hypersurface interpolation in the m dimensional space

We can generalize the G function in the m dimensional space.

Theorem 2 *Given* $A_i, i = \overline{1,n}$ *different points in the* m *dimensional space. We denote by* $\mathbf{r}_i$ *the positional vector of* A_i. *For every point* A_i *we assign a* z_i *scalar value. Let us define the function* $F : \mathbf{R}^m \to \mathbf{R}$, *where*

$$F(\mathbf{r}) = \begin{cases} \dfrac{\sum\limits_{i=1}^{n} \frac{z_i}{d_i^2}}{\sum\limits_{i=1}^{n} \frac{1}{d_i^2}} & \text{if} \quad \mathbf{r} \neq \mathbf{r}_i, \\ z_i & \text{if} \quad \mathbf{r} = \mathbf{r}_i, \end{cases} \tag{6}$$

where d_i *is the length of the vector* $\mathbf{r} - \mathbf{r}_i$. *The* F *function is continuous derivable and* $F'(\mathbf{r}_i) = 0$ *for all* $i = \overline{1,n}$.

Proof. From the definition we get:

$$F(\mathbf{r}) = \frac{\sum\limits_{i=1}^{n} \frac{z_i}{d_i^2}}{\sum\limits_{i=1}^{n} \frac{1}{d_i^2}} = \frac{\frac{z_1}{d_1^2} + \sum\limits_{i=2}^{n} \frac{z_i}{d_i^2}}{\frac{1}{d_1^2} + \sum\limits_{i=2}^{n} \frac{1}{d_i^2}} = \frac{\frac{1}{d_1^2}\left(z_1 + \sum\limits_{i=2}^{n} \left(\frac{d_1}{d_i}\right)^2 z_i\right)}{\frac{1}{d_1^2}\left(1 + \sum\limits_{i=2}^{n} \left(\frac{d_1}{d_i}\right)^2\right)}$$

and

$$\lim_{\mathbf{r} \to \mathbf{r}_1} F(\mathbf{r}) = \lim_{d_1 \to 0} F(\mathbf{r}) = z_1.$$

Consequently F is continuous. Furthermore if x is one of the coordinates of $\mathbf{r}$ then

$$\begin{aligned}\frac{\partial F(\mathbf{r})}{\partial x} &= \frac{\partial}{\partial x}\left(\frac{\sum\limits_{j=1}^{n}\frac{z_j}{d_j^2}}{\sum\limits_{i=1}^{n}\frac{1}{d_i^2}}\right)\\ &= \frac{-\sum\limits_{i=1}^{n}\frac{1}{d_i^2}\cdot\sum\limits_{j=1}^{n}\frac{2z_j\cdot(x-x_j)}{d_j^4}+\sum\limits_{j=1}^{n}\frac{z_j}{d_j^2}\cdot\sum\limits_{i=1}^{n}\frac{2\cdot(x-x_i)}{d_i^4}}{\left(\sum\limits_{i=1}^{n}\frac{1}{d_i^2}\right)^2}\\ &= \frac{2\sum\limits_{i=1}^{n}\sum\limits_{j=1}^{n}\frac{(x-x_j)(z_i-z_j)}{d_j^4 d_i^2}}{\left(\sum\limits_{i=1}^{n}\frac{1}{d_i^2}\right)^2}\end{aligned}$$

If $\mathbf{r}\to\mathbf{r}_1$ then $x\to x_1$, $d_1\to 0$ and

$$\begin{aligned}\lim_{\mathbf{r}\to\mathbf{r}_1}\frac{\partial F(\mathbf{r})}{\partial x} &= 2\cdot\lim_{\mathbf{r}\to\mathbf{r}_1}\frac{\sum\limits_{i=1}^{n}\left[\frac{(x-x_j)}{d_j^4}\left(\sum\limits_{i=1}^{n}\frac{z_i-z_j}{d_i^2}\right)\right]}{\left(\sum\limits_{i=1}^{n}\frac{1}{d_i^2}\right)^2}\\ &= 2\cdot\lim_{\mathbf{r}\to\mathbf{r}_1}\frac{\frac{(x-x_1)}{d_1^4}\cdot\sum\limits_{i=2}^{n}\frac{z_i-z_1}{d_i^2}+\sum\limits_{j=2}^{n}\left[\frac{(x-x_j)}{d_j^4}\left(\sum\limits_{i=1}^{n}\frac{z_i-z_j}{d_i^2}\right)\right]}{\left(\sum\limits_{i=1}^{n}\frac{1}{d_i^2}\right)^2}\\ &= 2\cdot\lim_{\mathbf{r}\to\mathbf{r}_1}\frac{\frac{(x-x_1)}{d_1^4}\cdot\sum\limits_{i=2}^{n}\frac{z_i-z_1}{d_i^2}}{\left(\frac{1}{d_1}\right)^4\cdot\left(\sum\limits_{i=1}^{n}\left(\frac{d_1}{d_i}\right)^2\right)^2}\\ &\quad+2\cdot\lim_{\mathbf{r}\to\mathbf{r}_1}\frac{\sum\limits_{j=2}^{n}\left[\frac{(x-x_j)}{d_j^4}\left(\frac{z_1-z_j}{d_1^2}+\sum\limits_{i=2}^{n}\frac{z_i-z_j}{d_i^2}\right)\right]}{\left(\frac{1}{d_1}\right)^4\cdot\left(\sum\limits_{i=1}^{n}\left(\frac{d_1}{d_i}\right)^2\right)^2}\\ &= 0.\end{aligned}$$

Consequently F is derivable and

$$F'(\mathbf{r}_i) = 0, \text{ for all } i = \overline{1,n}.$$

■

2.3 Surface interpolation in the space

There is a special case. If $m = 2$ we will get the H function in the real space:

$$H(x,y) = \begin{cases} \frac{\sum_{i=1}^{n} \frac{z_i}{d_i^2}}{\sum_{i=1}^{n} \frac{1}{d_i^2}} & \text{if} \quad x \neq x_i \text{ or } y \neq y_i, \\ z_i & \text{if} \quad x = x_i \text{ and } y = y_i, \end{cases} \tag{7}$$

where $d_i = \sqrt{(x-x_i)^2 + (y-y_i)^2}$ is the euclidean distance and $(x_i, y_i, z_i)\,, i = \overline{1,n}$ are the points we want to interpolate.

3 Conclusions

The F function it is like the (4) but here the weights are the inverse of the distance's square. This function has the properties of Lagrange's interpolation method and those of the arithmetic mean weighted by the inverse of distance method, because it interpolates the control points. The F function also has an important property. It is continuous derivable and the control points are stationary. We illustrate these features in the fifth row of the Figure 1 and Figure 2.

Consequently, with the Shepard's interpolation function we can derive smooth curves, surfaces and it allows the making of beautiful and aesthetic drawings in computer graphics.

Furthermore it is an important geological requirement and an empirical fact that methane and petrol have the shape of a mushroom. They cannot have a polyhedron like, plicate surface. In most of the cases when people make borings, first they find the maximum points of the methane. Therefore if we want to appreciate the volume of the methane we need a surface which crosses the maximum point and it has the form of a mushroom. On Figure 2 is visible that the Shepard's function has approximately this form in the maximum points, but the other functions do not have this feature. Naturally we have tested this function with higher powers of distance but we didn't get better interpolations.

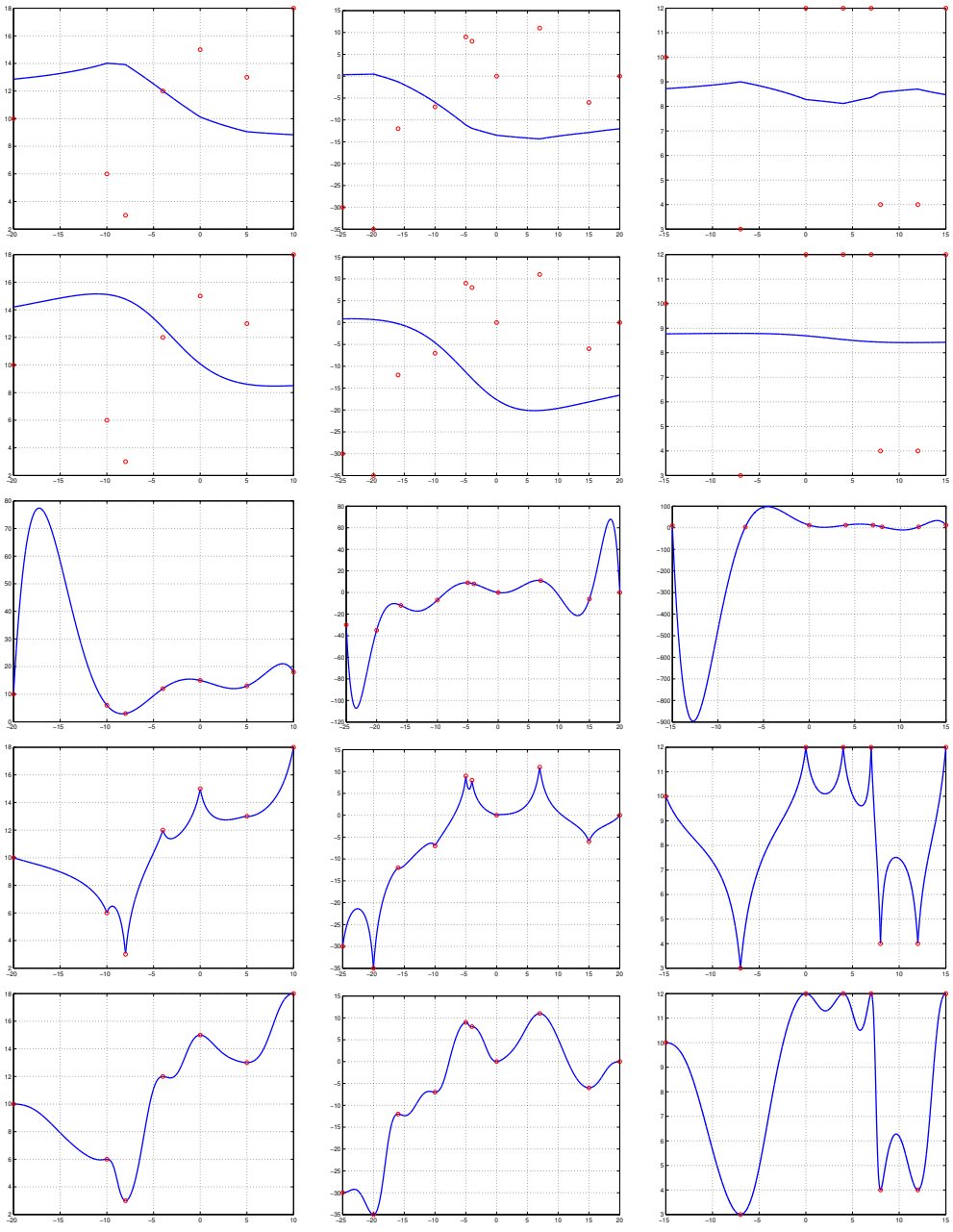


Figure 1: Approximation-interpolation in two dimension

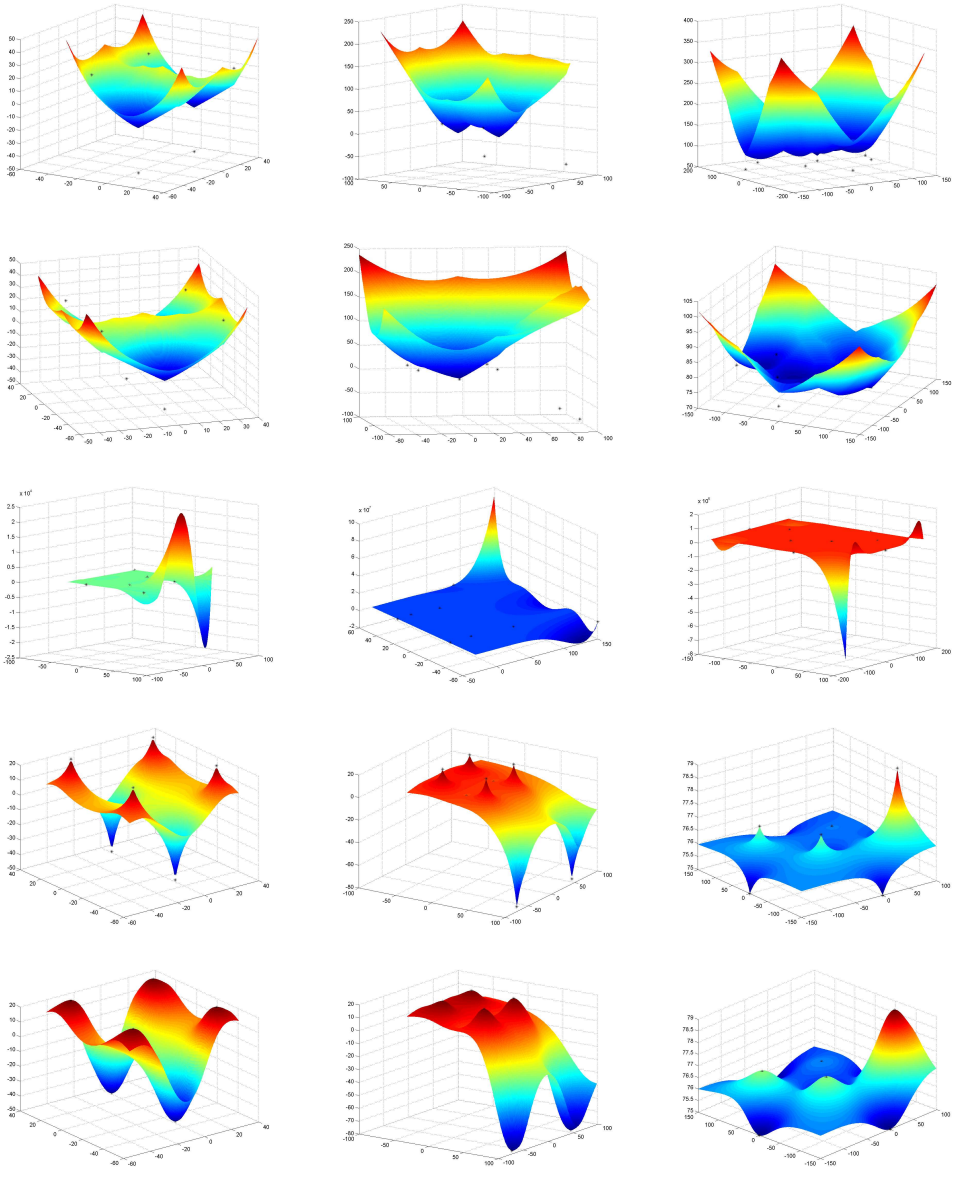


Figure 2: Approximation-interpolation in three dimension

References

[1] K. Brand, Multigrid bibliography, in: *Multigrid Methods*, Springer-Verlag, **960** (1982) 631–650.

[2] R.E. Barnhill, A survey of the representation and design of surfaces, *IEEE Computer Graphics and Applications*, **3**, 7 (1983) 9–16.

[3] R.E. Barnhill, R.P. Dube, F.F. Little, Properties of Shepard's Surface, *Rocky Mountain J. Math*, **13** (1983) 365–382.

[4] I. Cozac, Shepard method – from approximation to interpolation, *Studia Univ. Babeş-Bolyai Math.*, **48**, 2 (2003) 49–52

[5] J.R. Eastman, *Technical Reference, The Clark Labs for Cartographic Technology and Geographic Analysis*, IDRISI Version 4.0., 1992

[6] R. Franke, Scattered data interpolation: Tests of some methods, *Math. of Comput.*, **38**, 157 (1982) 181–200.

[7] W.J. Gordon, J.A. Wixom, Shepard's method of "Metric interpolation" to bivariate and multivariate interpolation, *Math. Comp.*, **2** (1978) 253–264.

[8] J.W. Harbaugh, D.F. Merriam, *Computer applications in stratigraphic analysis*, John Wiley & Sons, New York, 1968.

[9] E. Katona, Constructing digital terrain models (in Hungarian), *Polygon*, **11**, 2 (2002) 35–55.

[10] R. Oláh-Gál, L. Pál, Discrete approximation, *Proceedings of the 6th International Conference on Applied Informatics*, Eger, Hungary, (2004) 409–415

[11] D. Shepard, A two-dimensional function for irregularlly-spaced data, *Proceedings ACM National Conference*, (1968) 517–524.

Received: September 23, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 15–34

Meta programming on the proof level

Gergely Dévai
Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and
Compilers
email: deva@elte.hu

Abstract. Computer aided proof generation is used for many reasons from formalization of mathematics to formal computer program development. Our research concentrates on completely declarative style proofs used to develop imperative programs in a refinement-based model (i.e. deriving the algorithm from the specification).

In this paper we investigate why and how to use meta programming techniques for proof development. We examine techniques already used in programming languages if they are applicable for proof construction and point out the specialities caused by the different application area. It is also shown that while meta programming techniques are often dangerous when used to develop programs, they are safe tools for proof development.

1 Introduction

1.1 Human-readable proofs

There is a wide range of theorem provers from completely automatic ones to proof checkers. Examining the history of formal program development and automatic theorem proving, it seems hopeless to create a system that proves the correctness of industrial sized programs without considerable human effort. It is crucial that users of a proof system can easily understand the given proof situations if the machine is not able to complete the proof automatically. For

AMS 2000 subject classifications: 68Q60
CR Categories and Descriptors: F.3.1. [Specifying and Verifying and Reasoning about Programs]
Key words and phrases: formal program verification, meta programming, proof generation

```
true() => abs(x) >= 0
select
{
  x >= 0 => abs(x) >= 0
  {
    x >= 0 => x = abs(x);
    x = abs(x) & x >= 0 => abs(x) >= 0;
  }
  !(x >= 0) => abs(x) >= 0
  {
    !(x >= 0) => 0-x = abs(x);
    !(x >= 0) => 0-x >= 0;
    0-x = abs(x) & 0-x >= 0 => abs(x) >= 0;
  }
}
```

Figure 1: The non-negativity of the absolute value function

that reason, there is a growing interest in human-readable proof languages. The proof language of the *Mizar* proof assistant [1] was designed to be similar to textbook proofs. Similar style is used in the *Isar* [2] language for *Isabelle* [3] and in a declarative style proof language for *Coq* [4].

In our case proofs are used mainly to specify the behaviour of imperative programs and to develop the algorithm by refining the original specification. When refining a specification statement, one gives a set of more detailed statements. The refinement is sound, if every program that corresponds to the detailed specification also fulfils the requirements of the original one. We can say that the original specification is a theorem about the behaviour of the resulting program and the refinement steps are the proof. In this proof style, instead of indicating the proof actions, one breaks up the original theorem into several smaller theorems.

As an example we show a toy proof about the non-negativity of the absolute value function on figure 1. The first line states the theorem to prove. The select keyword indicates case-distinction, the cases $\mathtt{x} \geq 0$ and $\neg(\mathtt{x} \geq 0)$ are inside the pair of curly braces. Each case is refined further: In the case where $\mathtt{x} \geq 0$ holds we first conclude that $\mathtt{x} = \mathtt{abs}(\mathtt{x})$ and from this (and the previous knowledge $\mathtt{x} \geq 0$) we get that $\mathtt{abs}(\mathtt{x}) \geq 0$. The second case is similar. The unrefined statements are accepted by the proof checker based on previously

defined axioms and tactics.

In this system there are two ways to refine a statement (both shown in the previous example): sequence and case-distinction. Sequence introduces intermediate steps in the reasoning, while case-distinction splits up the proof into several cases. For more information on the refinements and the techniques to check them the reader is referred to [5].

1.2 Programming by proof

According to the programming paradigm we use, one first writes the specification of the program, then refines the specification in several steps. Unlike traditional refinement systems [6], during the refinement process we do not introduce program fragments. The proof tree (consisting of the specification as the theorem and the refinements as the proof steps) is complete when one reaches specifications of primitive instructions in the leaves.

In the following example on figure 2, the specification states that the program swaps the values of the two variables `x` and `y`. Instead of the operator `=>` that we used for classical logic statements in the previous example, here we use the `>>` temporal operator to express that the program proceeds from the first condition to the second one. We use the parameters `xVal` and `yVal` to denote the values of `x` and `y` respectively in the pre-state. The variable `ip` (instruction pointer or program counter) is used explicitly in specification statements. (This makes the specification of control statements, like jumps and procedure calls, much easier.)

The specification is refined by a sequence of three statements that describes how do the values of variables change during the execution of the program.

The compiler of this proof language has two tasks: it first checks the soundness of the proof, then it collects the primitive instructions whose specifications are used in the proof, and generates the program in the target language. In case of the current example the following instructions are extracted from the proof.

```
A: t = x;
B: x = y;
C: y = t;
D:
```

The resulting program is guaranteed to be correct with respect to the specification, provided that the specifications of the primitive instructions were sound.

```
variable(x,Integer);
variable(y,Integer);
parameter(xVal,Integer);
parameter(yVal,Integer);

ip = A & x=xVal & y=yVal >> ip = D & x=yVal & y=xVal
{
  variable(t,Integer);
  ip = A & x = xVal >> ip = B & t = xVal;
  ip = B & y = yVal >> ip = C & x = yVal;
  ip = C & t = xVal >> ip = D & y = xVal;
}
```

Figure 2: Swapping the values of two variables

The system is independent of the target language. One can add support for a new programming language by specifying (part of) its instruction set in the system and writing a code generator module for that language.

This paradigm differs from the classical *program extraction from proofs* [7]. In that case by developing a constructive proof for the existence of a mathematical object M, one can extract a program that evaluates M. In our case the program is extracted from its own correctness proof and there is no restriction on the logic used.

2 Meta programming

2.1 Minimal trusted base

Is the output program really sound? It depends on the correctness of the proof checking algorithm used by the compiler and on the correctness of the specifications of primitive instructions. This is called the *trusted base* of the system. To reduce the risk of errors in it, the trusted base should be minimal.

Our currently supported target language is *C++*, a language that is extremly rich in high level language construts like different kinds of loops, variable scopes, argument passing modes, classes, inheritance and a lot more. If we wanted a formal system supporting all these features using built-in rules, the programming model would be very complex, harly exendable and target language dependent.

In order to keep the trusted base of our system minimal and general, we elected to reduce the programming model as much as possible. We consider a program as a set of simple state transitions. Instructions that perform a simple state transition are easy to specify.

Although this programming model is very simple, it is expressive enough. We were able to specify for example pointer instructions [8] and vector operations [9] of *C++* in it.

2.2 Motivation for meta programming

Formal program development in the model presented above is like programming in assembly languages: No high-level language constructs are available, only a set of elementary instructions. We have discussed above that hardwiring the verification conditions for high level constructs in the system is not desirable. The questions is, how to enable the user to extend the system with these high level constructs without affecting the minimality of the trusted base?

A possible answer is meta programming, which was already used in case of assembly languages in the form of macros. For assembly programmers macros are useful to emulate instructions that are not part of the instruction set, generalize often-used program fragments using arguments and emulate high-level language constructs (loops, conditional branching, etc.). We use meta programming techniques for the same reasons *on the proof level*: We generalize often used proof parts. We call these proof fragments *proof templates* and they may be both classical logic proofs (like the schema of indirect proofs) or temporal logic proofs (like the schema of proving the correctness of a loop).

While meta programming techniques in assembly languages are quite low level features (simple text-based replacement of arguments, for example), the techniques we use are more sophisticated. We apply type checking for arguments and perform their substitutions in the syntax tree instead of the error-prone text-based replacement.

The user is also allowed to define own proof templates and it is possible to build libraries of them to help the work of other users. In traditional program development, a library consisting of a great number of functions can help the developer to make the code shorter and more understandable while it also rises the efficiency of the development process. The same is true for proof development: It is a general observation that proof systems with a huge set of tactics are more efficient.

If the user defines a proof template, it does not become part of the trusted

base. The compiler checks the proofs inside templates. This check occurs either when the template is defined or when it is instantiated depending on the type of the template. If the template was wrong or was used in an inappropriate situation, an error report is generated during proof checking.

2.3 Naming conventions

Meta programming is a general notion. In this paper we use it to name techniques to define meta language entities that are transformed to object level entities during a preprocessing phase after which some kind of compilation of the object language entities takes place. In case of traditional programming, assembly macros or *C++* templates (meta language) are first transformed to pure assembly or *C++* (object language) and compiled further by an assembler or a *C++* compiler. In our case, the meta language consists of proof templates. Template calls are transformed to pure proofs which are then checked and compiled to a traditional programming language.

In the literature the terms *proof schema*, *proof sketch*, *proof template* are used in various senses. In [10] *formal proof sketches* are defined to be shortened formal proofs which have gaps from the point of view of mechanical proof checking but are easier to understand. In formal program verification it happens quite often that the proof attempt fails because the program is wrong. In that case, after correcting the program, one can reuse parts of the previous proof attempt. In [11] these reused proofs are called *proof templates* and in [12] generalized proofs to replay are called *proof schemas*. Our proof templates are similar to these techniques in the sense that they are generalized and reusable proof fragments with the goal making proofs shorter and more understandable, but are completely different in the way of their definition and application.

2.4 Overview of techniques

In the following we examine the meta programming techniques that we have found useful for the purposes of proof-development.

2.4.1 Arguments

We can generalize a proof fragment by giving it a name and replacing parts of it by arguments. This way we obtain a meta-proof that we call a *template*. It turned out to be useful to syntactically distinguish formal template arguments

from program variables and parameter variables. (We start them by a sharp symbol: `#`.)

To define a template called `example` having two arguments of types `Integer` and `Character` we write:

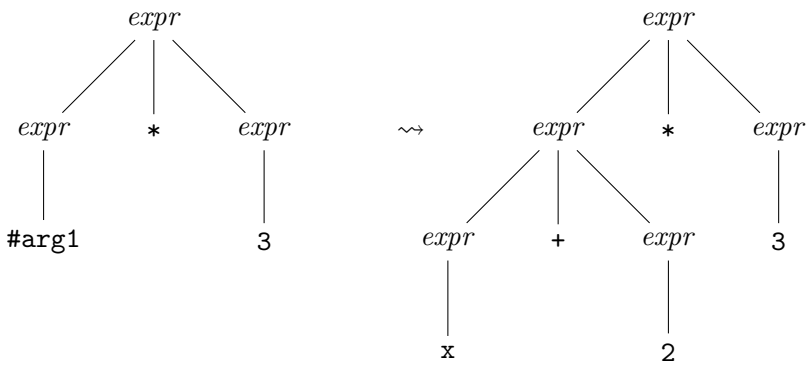
```
template example( Integer #arg1, Character #arg2 )
{
  // template body
}
```

In the body of the template we can write a proof fragment containing the formal template arguments. We instantiate the template using actual arguments in the following way:

```
example( x+2, 'a' );
```

The template call is type checked. The compiler instantiates the template by replacing the formal arguments in the body by the actual ones.

Simple text-based replacement of arguments would lead to surprising results in some cases. A well-known example of low level macros is the expression `#arg1*3` which becomes `x+2*3` in the previous template call. By the precedence rules of operators this means `x+(2*3)` instead of `(x+2)*3` which, supposedly, was the intention of the programmer. To avoid this pitfall in our system, instantiation of templates takes place after parsing. As shown in the figure below, the replacement is done in the syntax tree of the expression and produces the correct result.



An other common feature of low level macros is that symbols used inside the macro may not be declared at the point of the macro definition. The

meaning of the symbols depend on the declarations visible at the point of the instantiation. This can induce many errors that are hard to recover. In case of our templates any symbol inside the template body must be declared and they are bound to the declaration visible at the point of the template definition. Even if an other declaration hides the original declaration of the symbol at the point where one calls the template, the symbol in the instantiated proof will belong to the original declaration.

It is quite important that the proofs obtained by instantiating a template may not be sound in general. Templates should be thought of as proof attempts or proof schemas instead of theorems with their proofs. That is, when a template is defined, the soundness of its body is not checked. But, each time the template is called, the resulting proof is verified and the errors are reported. In section 2.4.5 we introduce a special kind of template that is verified as soon as it is defined, so that there is no need for further checks when one instantiates it.

2.4.2 Compile time conditions

Meta programming makes it possible to perform computations in compile time. The result of the meta-level computations can influence the generated object-level code. For example, conditions expressed in the meta language can decide whether a piece of code is included or omitted. These techniques turned out to be useful also in case of proof generation.

A common situation is that different proofs are needed depending on the form of a template argument. In the following example the argument of the template is examined by *compile time conditions*. These conditions decide which variant of the proof should be used.

```
template variants( Integer #arg )
{
  constant( #arg ) :
    // proof in case of a constant argument
  variable( #arg ) :
    // proof in case of a variable argument
}
```

In case the template call is `variants( 2 )`, the first proof is the result of the instantiation, while the call `variants( x )` results in the second one.

We can increase the expressive power of these conditions if we enable pattern matching for their arguments. In the following example, the condition checks

whether the argument is a compound expression with addition as the top level operator and bounds the two sub-expressions to the arguments `#left` and `#right`. These can be used in the proof.

```
template patternMatcher( Integer #arg )
{
  equals( #arg, #left + #right ) :
    // proof that can use #left and #right
}
```

In case of the template call `patternMatcher( x+2 )`, the compiler replaces `#arg`, `#left` and `#right` by `x+2`, `x` and `2` respectively in the proof.

In the current implementation of our system there is a fixed set of such conditions. We have noticed that it would be useful to give the user the ability to define new conditions based on the old ones. We plan to carry out a *Prolog*-style implementation of this feature.

2.4.3 Axioms and instruction specifications

It seems natural to use the template features of the language to define the axioms that are used to close the branches of proof trees. We mark these templates with special keywords so that the compiler accepts the statements obtained by instantiating these templates without any further refinement.

There are two kinds of axioms in the system: One states properties of mathematical functions used in specifications and the other describes temporal properties of instructions in the target language. We call the former ones *axioms* and the latter ones *atoms*.

For example we show the temporal axiom of the instruction incrementing a variable. The arguments of the template will be the variable to increment (`#var`), an expression describing the value of the variable before the instruction (`#val`), and the labels before and after the instruction (`#before` and `#after`). The temporal axiom is basically the following:

```
ip = #before & #var = #val & #val < maxInt()
>> ip = #after & #var = #val + 1;
```

But this statement is not sound for all the possible combinations of the template arguments. For example, if we instantiate the template using the variable `x` both for the arguments `#var` and `#val`, the statement becomes invalid. The following conditions should be checked: the labels `#before` and `#after` have

```
atom increment( Integer #var, spec Integer #val,
                Label #before, Label #after )
{
  constant(#before) & constant(#after) & variable(#var)
  & independent(#var,#val) & independent(ip,#val) :
    ip = #before & #var = #val & #val < maxInt()
    >> ip = #after & #var = #val + 1;
}
```

Figure 3: Specification of the increment operation

to be label constants (this is needed for code generation), `#var` must be a variable and `#val` must not depend on `ip` and `#var`. These checks can be easily implemented using compile time conditions introduced in the previous section. The definition of the temporal axiom[1] is on figure 3.

There is one more thing to mention about this template: The `spec` keyword before the template argument `#val`. This marking informs the compiler that `#val` is not used for code generation, it is needed for specification purposes only. This means that the generated instruction does not depend on the actual expression provided for the argument `#val` in the template call. In section 2.4.5 we will see that it is sometimes important for the compiler to know which arguments does and which does not affect the generated instructions.

2.4.4 Proof fragments as arguments

Arguments of macros or templates are usually expressions or types in most systems using meta programming. In this section we show that allowing complete chunks of object level code (proofs in our case) as arguments rises the flexibility of the meta language.

To demonstrate this feature we construct a template that generates indirect proofs. To prove $P \Rightarrow Q$ by induction, we have to show $P \wedge \neg Q \Rightarrow \mathtt{false}()$. In our system, indirect proof is not a refinement possibility, we have sequence and case distinction only. We have to implement the indirect proof with these tools.

[1] Here we give only a progress property of this instruction. To make the axiom more useful, we could add a safety property describing which variables are affected by this operation. As the formal programming model behind this specification is not the main topic of this paper, we elected to simplify the example by omitting the safety property.

```
P => Q
select
{
  Q & P => Q
  {}
  !(Q) & P => Q
  {
    !(Q) & P => false()
    {
      // proof of contradiction
    }
  }
}
```

Figure 4: An indirect proof

The trick, shown on figure 4, is to perform a case distinction on Q. The case where Q holds is trivial, so an empty refinement is enough to complete it. In the second case we can use the proof of contradiction. As soon as `false`() is proved, the compiler also accepts Q.

As any indirect proof can be transformed to the same form, it is worth creating a template that does this transformation. Such a template must get the proof of contradiction as an argument. We use the `block` keyword in the argument list of a template to denote that a proof fragment has to be passed for that argument. Using this feature we can write our template shown on figure 5.

The proof inside the template is organised as we have discussed above. The argument marked by the `block` keyword can appear in any position where a proof is needed.

We also need a bit of special syntax to pass these special arguments when the template is called. It is done by writing the proof fragment to pass between curly braces after the template call (which is not terminated by a semicolon). The indirect proof calling the template we have just constructed is on figure 6. This template call results in the same proof shown on figure 4, but this variant is shorter and easier to understand.

In section 2.1 we have mentioned the advantages of the minimalistic programming model without built-in rules for programming constructs like conditional branching or loops. As a result, a simple *if-then-else* construct consist

```
template indirect( Boolean #hypothesis, Boolean #goal,
                   block #proof )
{
  #hypothesis => #goal
  select
  {
    #goal & #hypothesis => #goal
    {}
    !(#goal) & #hypothesis => #goal
    {
      !(#goal) & #hypothesis => false()
      {
        #proof;
      }
    }
}
```

Figure 5: Template for indirect proofs

of several instructions: First the condition is to be evaluated, then a conditional jump instruction follows to jump to the label of the *then-branch* if the condition was true or to continue at the label of the *else-branch* otherwise. At the end of both branches an instruction is needed to jump to the instruction that follows the branching.

It is possible to prove the correctness of such a low-level branching algorithm in the system, however, it is not desirable to force the programmer to develop a rather complex proof each time when using an *if-then-else* construct. Fortunately it is possible to generalise the proof and hide the details (that are same for every conditional branch) using a template. The condition of the branch and the proofs for the branches will be the arguments of the template, which can be called in the following way.

```
if( condition )
{
  // proof of the then-branch
}
{
  // proof of the else-branch
}
```

```
P => Q
{
  indirect( P, Q )
  {
    // proof of contradiction
  }
}
```

Figure 6: Indirect proof calling the `indirect` template

This means that there is no need to hard-wire techniques like indirect proofs or verification conditions for programming constructs into the system core. It is possible to define templates that provide the same convenience, without making the kernel of the proof system unnecessarily complex.

2.4.5 'Check once, use many times'

The templates we have seen so far could generate completely different proofs for different actual arguments and there was no guarantee that these proofs were sound. That is why the compiler had to instantiate and check the templates every time. If a template can be checked independently of the actual arguments, it is possible to validate it when it is defined. When such a template is called, the compiler can accept its top-level statement without instantiating and re-validating the whole proof inside it. Practically, a template of this kind contains a theorem and its proof.

As these templates are not dynamically instantiated and checked at every call, we call them *static templates*, and use the `static` keyword to introduce them. Arguments of static templates are not allowed to appear in compile time conditions, because that would make it impossible to validate the proof regardless of the actual arguments.

No matter how many times we call a static template, it is validated only once, and this is not only an efficiency issue. We can use it to implement induction: If a static template calls itself recursively, the recursive call is not expanded and checked, but its specification is used as an induction hypothesis. The well-foundness of the induction is ensured by the first argument of the recursive call: It must be an integer expression proved to be non-negative and strictly less then the first argument of the template containing the recursive call.

```
axiom p0( Integer #n )
{
  #n = 0 => p(#n);
}

axiom pNext( Integer #n )
{
  #n > 0 & p(#n-1) => p(#n);
}
```

Figure 7: Axioms for an inductive proof

For showing this recursive schema, let us have a logical function `p` on integers with the axioms on figure 7.

The static template on figure 8 proves `p` for all non-negative integers by induction. As in a usual induction proof, we have a base case (`#n=0`, solved by the first axiom) and an inductive case (`#n>0`). In the inductive case we first make it explicit that the induction is well-founded, then use the induction hypothesis by calling the template recursively with the argument `#n-1`. Then, by the second axiom we complete the proof.

Static templates are meaningful also for temporal logic proofs. In this case, not only the soundness of the refinements inside the template has to be independent of the actual template arguments, but also the instructions extracted from the proof. This additional condition practically means, that arguments of static templates must not be passed as arguments to atoms, if that argument influences the generated instruction. In section 2.4.3 we have introduced the `spec` keyword to denote that an argument of an atom is not used for code generation. That is, static template arguments are allowed to be passed to atoms only in `spec` arguments.

When should one place a piece of temporal proof into a static template? Every time the specification of that code is used more that once. For example, the proof of a procedure should be implemented in a static template. Each time the procedure is called, one can call the static template to use the specified properties of the procedure. An other example is the proof of a loop, as we usually need induction for that. The refinements describing the loop body form the static template, and when the program jumps back to the beginning of the loop, we call the template recursively.

```
static pAll( Integer #n )
{
  #n >= 0 => p(#n)
  select
  {
    #n = 0 => p(#n)
    {
      p0( #n );
    }
    #n > 0 => p(#n)
    {
      #n > 0 => #n-1 >= 0 & #n-1 < #n;
      pAll( #n-1 );
      p(#n-1) & #n-1 >= 0 => p(#n)
      {
        pNext( #n );
      }
    }
  }
}
```

Figure 8: A template implementing an inductive proof

2.4.6 Templates defined in templates

In section 2.4.4 we have seen a template that generates indirect proofs. We want to create a similar one for inductive proofs by generalising the example in the previous section. That is, we will pass the function `p`, the proof of the base case as well as the proof of the inductive case as arguments to that template, and it will generate the inductive proof seen in the previous section.

Our template will generate a name for the static template to define, using the compile time condition `templatename(#name)`. When evaluating this condition, the compiler will bound a fresh templatename to `#name`. The static template defined inside our template is the same as in the previous section, except the proofs of the base and the inductive cases, because these are arguments.

```
block( Integer #x )
{
  p0( #x );
}
```

Figure 9: An unnamed proof accepting an argument

Let us have a look on these proofs, that should now be passed as arguments. In the previous section we used

```
p0( #n );
```

as the base case and

```
pNext( #n );
```

as the inductive case. Notice that we use the argument `#n` of the static template in these proofs. That is, we can not pass these proofs 'as is', because the argument `#n` is not usable outside the static template. A solution is to pass these proofs accepting an argument. The syntax for this[2] is shown on figure 9. We will pass this block as an argument to our template and that will use it inside the static template and pass `#n` to it as an argument.

When the static template is defined, our template should also immediately call it to use the theorem just proved in the static template. This *template-defining-template*, called `induction`, can be seen on figure 10.

Having the `induction` template defined, we can write our inductive proof in a much more elegant way. For arbitrary parameter value `k`, we can prove `p(k)` as on figure 11.

3 Safety considerations

There are several programming errors that make the careless application of meta programming techniques dangerous, when the object-level code is a traditional programming language. This is especially true for low-level meta programming features.

A common example is, when one repeats a piece of code many times in the program using meta programming techniques instead of writing a loop. This

[2]The ability to write proofs accepting arguments (i.e. unnamed templates) is currently under development.

```
template induction( Integer --> Boolean #fun, Integer #arg
                    block(Integer #p) #base,
                    block(Integer #p) #induct )
{
  templatename( #name ) :
  block
  {
    static #name( Integer #n )
    {
      #n >= 0 => #fun(#n)
      select
      {
        #n = 0 => #fun(#n)
        {
          #base( #n );
        }
        #n > 0 => #fun(#n)
        {
          #n > 0 => #n-1 >= 0 & #n-1 < #n;
          #name( #n-1 );
          #fun(#n-1) & #n-1 >= 0 => #fun(#n)
          {
            #induct( #n );
          }
        }
      }
    }
    #name( #arg );
  }
}
```

Figure 10: Template for induction

```
k >= 0 => p(k)
{
  induction( p )
    block( Integer #x )
    {
      p0( #x );
    }
    block( Integer #x )
    {
      pNext( #x );
    }
}
```

Figure 11: An inductive proof using the `induction` template

can result in an extremely large program. The same happens when the programmer inserts the same instructions at several points in the code instead of defining a procedure and calling that each time it is needed. Confusing procedures with syntactically similar meta programming features can also mess up the code so that it produces erroneous behaviour. Compilers of traditional programming languages does not complain on these errors or inefficient solutions, because the program is syntactically correct.

After all this, is it safe to use meta programming techniques for proof generation? Using the techniques presented in this paper carelessly can lead to erroneous proofs. But all the errors in the proof are reported by the proof checker, and in that case, there is no program generated that could be compiled further to an executable. That is, each time the checker accepts the proof, the generated program conforms to its specification.

The difference between traditional program development and programming by proofs is that in the latter case the compiler can check also the behavioural semantics of our program, not just the syntax and static semantics. This additional check makes meta programming a safe tool for proof development.

In addition to this, our meta programming features are high level language constructs. As we have already discussed in section 2.4.1, instantiation of templates is not text-based, but uses the syntax tree provided by the parser. Visibility of arguments, variables and parameter variables are also correctly handled in conformation with the block structure of the proof. Compile time conditions use techniques usually applied in high-level declarative languages.

These features help avoiding the common pitfalls of low level meta programming and makes construction of sound proofs easier.

4 Conclusions and future work

We can conclude, that meta programming techniques are applicable for construction of declarative style proofs. They make the proofs considerably shorter and easier to understand and to maintain. A great advantage of this solution is that there is no need to make the proof system complex in order to provide rules for common proof patterns. Techniques like indirect proving, induction, or temporal logic patterns like proofs for loops or conditional branches can be implemented using templates instead of hard-wiring them into the system core. This improves the reliability of the proof system.

We have also shown that the dangers of meta programming are not a real risk in case of proof development, as proofs are checked anyway and the errors are discovered already in compile time.

We have implemented the features described in this paper, except the possibility of writing unnamed templates (see the footnote in section 2.4.6). The current implementation is done in *C++*. We use the system to specify instructions of imperative programming languages (including more complex ones, like vector or pointer operations) and to develop verified program code using the refinement techniques of the system. The library of tactics is not yet comparable with that of leading theorem provers available [3, 13], but the templates of our language turned out to be useful techniques in building such libraries.

Our plans include further development of the language and to experiment with refinement techniques not only for imperative programs, but also for functional ones. We also intend to use our system to specify and to formally develop DFA-s[3] or Petri-nets.

References

[1] M. Muzalewski, An Outline of PC Mizar, *Fondation Philippe le Hodey*, Brussels, 1993.

[2] M. Wenzel, F. Wiedijk, A Comparison of Mizar and Isar, *Journal of Automated Reasoning*, **29**, 3–4 (2002) 389–411.

[3] Deterministic Finite Automaton

[3] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL – A Proof Assistant for Higher-Order Logic, *LNCS 2283, Springer*, 2002.

[4] P. Corbineau, A declarative proof language for the Coq proof assistant, *Types for Proofs and Programs, LNCS 4941., Springer*, 2008.

[5] G. Dévai, Programming language elements for correctness proofs, *Acta Cybernetica* **18**, (2008) 403–425.

[6] J.-R. Abrial, The B-book: assigning programs to meanings, *Cambridge University Press*, 1996.

[7] U. Berger, H. Schwichtenberg, Program extraction from classical proofs, *LNCS 960, Springer*, (1995) 77–97.

[8] G. Dévai, Z. Csörnyei, Separation logic style reasoning in a refinement based language, *Proceedings of the 7th International Conference on Applied Informatics*, 2007.

[9] G. Dévai, N. Pataki, Towards verified usage of the C++ Standard Template Library, *Proceedings of the 10th Symposium on Programming Languages and Software Tools*, (2007) 360–371.

[10] Freek Wiedijk, Formal proof sketches, *LNCS 3085, Springer*, (2004) 378–393.

[11] B. Beckert, V. Klebanov, Proof reuse for deductive program verification, *Proceedings of Second International Conference on Software Engineering and Formal Methods*, (2004) 77–86.

[12] C. Hunter, P. Robinson, P. Strooper, Flexible Proof Reuse for Software Verification, *LNCS 3116, Springer*, (2004) 211–225.

[13] Y. Bertot, P. Castéran, Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions, *Texts in Theoretical Computer Science, Springer*, 2004.

Received: October 1, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 35–44

Automated subtree construction in a contracted abstract syntax tree

Róbert Kitlei
Faculty of Informatics,
Eötvös Loránd University,
Pázmány Péter sétány 1/c,
H–1117 Budapest, Hungary
email: kitlei@elte.hu

Abstract. Syntax trees are commonly used by compilers to represent the structure of the source code of a program, but they are not convenient enough for other tasks. One such task is refactoring, a technique to improve program code by changing its structure [7].

In this paper, we shortly describe a representation of the abstract syntax tree (AST), which is better suited for the needs of refactoring. This is achieved by contracting nodes and edges in the tree. The representation serves as the basis of the back-end of a prototype Erlang refactoring tool [8], however, it is adaptable to languages different than Erlang [2].

We introduce a method that helps us automatically generate syntactically correct subtrees. Since refactorings often have to create new parts of the tree, it is essential to make this task as convenient as possible.

1 Introduction

Syntax trees are usually created by parsers, which operate on tokens that are produced by a scanner directly from the source code, with possibly a preprocessing phase inserted. Most of the time, these syntax trees are used once – possibly the syntax tree is never constructed in its entirety, as is the

AMS 2000 subject classifications: 68P05 subtree construction
CR Categories and Descriptors: D.2.10. [**Software**]: Software engineering – *Representation*;
Key words and phrases: subtree construction

case with top-down and bottom-up parsers. However, in some cases the full syntax tree is needed. One such example is refactoring.

Refactoring is the systematic changing of source files while retaining the semantics of the code. Some refactorings, e.g. renaming a variable or a function, do not change the shape of the syntax tree, only update the information in the nodes, while others, e.g. extracting a function, do delete, move or insert new nodes or subtrees in the syntax tree. Since deletion does not pose a problem, and moving a subtree is equivalent to its removal and reinsertion, the most intriguing question of the above is the creation and insertion of new subtrees.

In addition to the above, refactorings have to gather additional information about semantic aspects of the source code as well. Since these bits of information can only be collected by visiting diverse parts of a syntax tree, syntax trees make inappropriate and inefficient representations for refactorings. A graph representation is proposed by the Erlang refactoring group at the university ELTE (Budapest, Hungary). The ELTE group proposed this representation after previous experience with refactoring [5, 8]. Details about the representation and the refactoring tool are found in [4].

The structure of the paper is as follows. In section 2, the graph representation is described to such depth as is necessary for understanding the rest of the paper. Section 3 describes a method that facilitates the creation and insertion of new subtrees. This method is the main contribution of the paper. Section 4 lists related work, and section 5 gives acknowledgements.

2 Representation structure

2.1 Node and edge contractions

ASTs built on top of source codes are typically created by compilers in compilation time. Such syntax trees are discarded after they have been used, and their construction does not involve complex traversals: they follow the construction of the tree. There are, however, applications in which the role of ASTs are augmented. In refactoring, for example, tree traversals are extensively used, because a lot of information is required that can be acquired from different locations.

In order to facilitate these traversals, a new representation of the AST was introduced, which is described in detail in [4]. Here we give an overview of the relevant parts of the representation.

ASTs inherently involve parts that are unnecessary for information collection, or are structured so that they make it more tedious. One obvious case

```
if
    X == 1 -> Y = 2;
    true   -> Y = 3
end
```

Figure 1: If clauses in Erlang.

```
to_list(Text) when is_atom(Text)    -> atom_to_list(Text);
to_list(Text) when is_integer(Text) -> integer_to_list(Text);
to_list(Text) when is_float(Text)   -> float_to_list(Text);
to_list(Text) when is_list(Text)    -> Text.
```

Figure 2: Function clauses with guards.

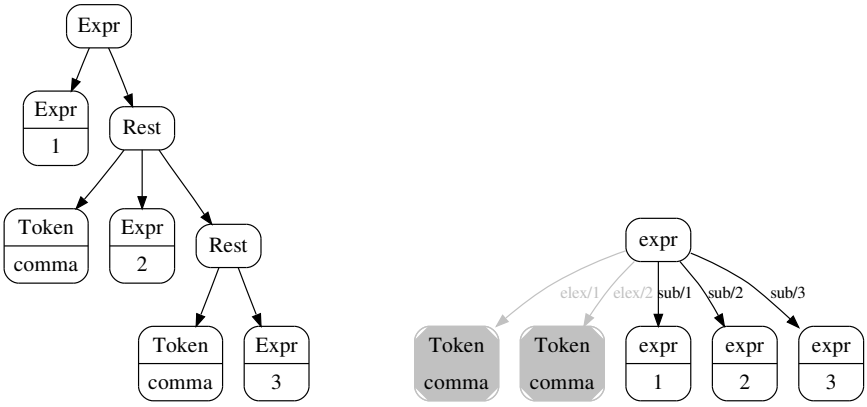
is that of chain rules: the information contained in them could be expressed as a single node, yet the traversing code has to be different for each node that occurs on the way.

Another case can be described by their functionality: the edges of the nodes can be grouped so that one traversal should follow exactly those that are in one group. To give a concrete example, clauses in Erlang have parameters, guard expressions and a body, and there are associated tokens: parentheses and an arrow. Yet the actual appearance of the clauses can be vastly different, see Figures 1 and 2. When collecting information, often either all parameters or all guard expressions are required at a time during a traversal pass, but seldom both at the same time of the traversal. Therefore, it is natural to partition the edges into groups along their uses. Since the partitions depend on the traversals used, the programmer has to decide by hand how groups should be made. This way, only as few groups have to be introduced as needed in a given application.

Another way to make the representation more compact is to contract repetitions. Repetitions are common constructs in programming languages: they are repeated uses of a rule with intercalated tokens as separators. Instead of having a slanted tree as constructed by an AST, it is more convenient for traversal purposes to represent them by a parent node with all of the repeated nodes and the intermediate tokens as its children. As a matter of fact, in the example in the above paragraph the parameters and guard expressions are

already a result of such a contraction.

Having done the above contractions has two main advantages. One is that much fewer cases have to be considered. In the case of Erlang, the grammar contained 69 nonterminals, which was reduced to three contracted node groups: forms, clauses and expressions.



(a) AST. (b) Contracted AST.

Figure 3: Repetition in the expression `1+2+3`.

A further advantage of contractions is that they enable introducing queries, which makes traversals even more effective. Queries can be further optimised by automatically adding semantic nodes and edges to the contracted AST, which make it a graph. In addition, the prototype tool also supports preprocessor constructs. Queries, semantic nodes and edges and preprocessor handling are described in detail in [4].

Since the contraction groups are different for each language (and may even differ in each application, depending on the needed level of detail), it is important that the approach should be adaptable to a wide range of grammars. For this reason, an XML representation was chosen for describing the grammar rules, the contraction groups and the edge labels. The scanner and parser are automatically generated from this file. The contracted structure is immediately constructed during parse time, and not converted from an AST.

2.2 Representation of the contracted AST

The inner nodes of the contracted AST are the contracted nodes, which also contain the originating nonterminal as information. The leaf nodes of the contracted AST are the tokens, which contain the token text and the whitespace before and after the token. The nodes are connected by labelled edges; the labels determine the contraction classes they can connect.

Contractions do not fully preserve edge ordering: order is preserved only between the edges with the same label, not between different labels. This is why the original AST cannot be restored easily: in Figure 4 it is not possible to determine whether the tokens of the clause come before, after or in between the expressions. To make it possible, more information about the structure of the contracted nodes is needed.

The lack of order between label groups is the result of using a database for storage, which is required for fast queries. However, it is expected to be a good trade-off, since the exact AST order of the nodes is seldom needed (most importantly, when reprinting the contents of the graph into a file), while it provides queries in linear time of their length. The order of the links with the same label, which is important during queries, is retained.

3 Construction of new AST subtrees

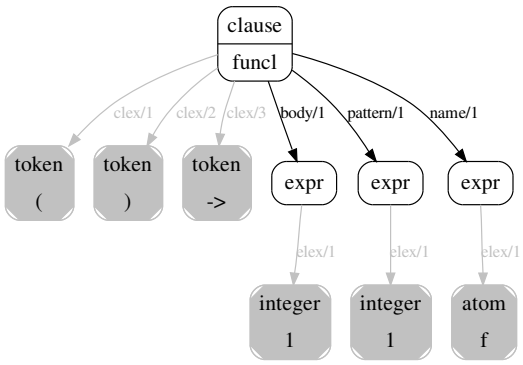
One possible solution for constructing AST subtrees would be to use the parser itself by providing the source code of the desired subtree – effectively, its front. This approach would require the user to manually fill in all the punctuation, and would require separate grammars for each nonterminal to be generated.

In this section, a different method is presented that makes constructing syntactically correct AST subtrees comfortable for the user. In section 3.1, structures are defined that describe the expected structure of a node (the node skeleton). The method itself is described in section 3.2.

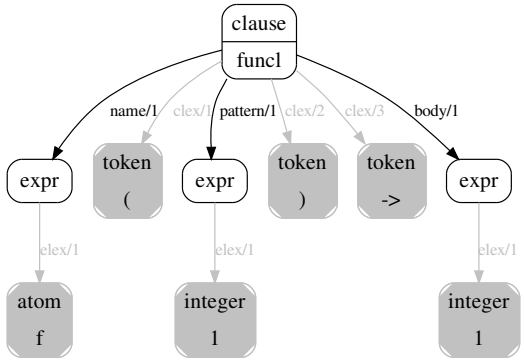
3.1 Node structure skeletons

The grammar description chosen is close to a BNF description. In it, the grammar rules are grouped by what contraction group their head belongs to. Rules, of course, may have more alternatives. The right hand sides of rules consist of a sequence of the following:

- **tokens** that contain the token node label,



(a) Part of an automatically printed contracted AST. The order of the edges between groups in unknown.



(b) The nodes rearranged in the right order. The order within the groups is retained. The tokens read: `f(1) -> 1`.

Figure 4:

- **symbols** that contain the child symbol's nonterminal and the edge label,
- **optional constructs**, sequences that either appear or not in a concrete instance and
- **repeat constructs** that contain a symbol and a token; its instances are

several (at least one) symbols with tokens intercalated.

We will such sequences node structure skeletons. Since optionals and repeats may contain one another, we shall refer to the number of contained nestings as the depth of the construct.

As an example that contains both constructs described above, let us examine the structure of lists. The structure of lists is described as follows. Lists start with an opening bracket token and end with a closing bracket token. Between them is an optional construct. The optional part consists of a repeat construct. The repeat construct uses comma tokens to separate symbols that are linked using "sub" edges from the parent node. The portion of the actual Erlang code that shows the above structure is shown in figure 5 in order to have a more concise overview.

```
[{token,"op_bracket"},
 {optional,[{repeat,"comma","sub"}]},
 {token,"cl_bracket"}];
```

Figure 5: The structure of lists as an Erlang structure used in the actual implementation. Slightly abridged.

Lists can be empty lists, or lists containing expression symbols separated by comment tokens. In the first case, the optional part is not present. In the second case, the optional is present. If there is one element in the repeat construct, there is exactly one symbol element present which denotes the expression.

$$\underbrace{[}_{\text{token}}\ \underbrace{\ }_{\text{empty optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\ \underbrace{1, 2, 3, 4, 5, 6, 7, 8 + 9, f()}_{\text{repeat in optional}}\ \underbrace{]}_{\text{token}}$$

3.2 Automated subtree construction

From the XML grammar description, node structure skeletons are automatically generated for each node type.

The user has to supply two pieces of information for the node creation algorithm. One is the contents of the newly created node, which also contains its type. By supplying the type of the node, the relevant node structure skeleton can be determined. The other piece of information to be supplied is a description of the desired actual content of the new node. This parallels the structure of the skeleton in the following way.

1. Almost all tokens are automatically created; these tokens are not included in the description. Information about tokens that cannot be automatically created, e.g. names of functions, have to be present.

2. Symbol, optional and repeat descriptions are at corresponding positions with the skeleton.

3. Symbol descriptions contain the actual node to be incorporated as a child. As stated before, they are either created before or moved from a different position in the tree.

4. Repeat descriptions contain the actual symbol nodes. The tokens of repeat constructs are always autocreated, therefore they need not be specified in the description.

5. Optional descriptions are either `no_optional`, or they contain sequential content that parallels that of the optional in the skeleton.

The algorithm used to construct a contracted node processes the sequence in the skeleton along with the one in the description.

1. If the next construct is an autocreated token, it does not appear in the description, because it can be automatically created. The created token link from the parent to the depends only on the contracted type of the parent node. Note that all other constructs have to be present both in the skeleton and the description.

2. If the next construct is a (not autocreated) token or a symbol, the corresponding description item contains the node itself to be linked from the parent.

3. If the next construct is a repeat, the token nodes are autocreated and linked as above, and the symbols to be linked are listed in the repeat description. Since the representation is contracted, the insertions of the symbols and tokens below the parent do not have to be intercalated, as their connecting edges are in different label groups.

4. If the next construct is an optional, but the description contains `no_optional`, it is skipped.

5. If the next construct is an optional, and the description has actual content, the contentsof the optional skeleton and description are processed.

Subtrees can be created by repeated use of the above algorithm. When constructing a new node, previously created nodes can be used as well as nodes that were already present in the graph. Practically, for each common subtree type, a function has to be created in order to facilitate the use of the algorithm. The function itself invokes the algorithm with the contents of the created node and the appropriate parameters.

4 Related work

The design of the representation was shaped through years of experimentation and experience with refactoring functional programs. The first refactoring tools produced at ELTE [5, 8] used standard ASTs for representing the syntax. It became evident that such a representation is not convenient enough for refactoring purposes, and a new design was needed. The resulting design [4] already used the contracted graph described in section 2 as representation of the syntax tree.

The Java language tools srcML [9], JavaML [3] and JaML [1] use XML to model Java source code. XML documents can be equipped with schema information against which they can be checked. If the schema is formulated in XML itself, subtree construction algorithms similar to the one presented in this paper can be devised.

5 Acknowledgements

The refactoring group at ELTE has helped the author conceive, shape, test and improve the algorithm described in the paper.

This work was supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK and Ericsson Hungary.

References

[1] G. Fischer, J. Lusiardi, and J. Wolff v. Gudenberg, Abstract syntax trees and their role in model driven software development. In *ICSEA online proceedings*. IEEE, 2007.

[2] J. Barklund and R. Virding, Erlang Reference Manual, 1999, Available from http://www.erlang.org/download/erl_spec47.ps.gz.

[3] Greg J. Badros, Javaml: a markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks: the international journal of computer and telecommunications networking*, pages 159–177. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 2000.

[4] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, Tamás Kozsik, Preprocessor and whitespace-aware toolset for Erlang source code manipulation. Abstract submitted to the *20th International Symposium on the Implementation and Application of Functional Languages*, Hatfield UK.

[5] R. Szabó-Nacsa, P. Divinszky, and Z. Horváth, Prototype environment for refactoring Clean programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004)*, Szeged, Hungary, July 1–4, 2004.

[6] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy, Refactoring Erlang Programs. In *Proceedings of the 12th International Erlang/OTP User Conference*, November 2006.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[8] L. Lövei, Z. Horváth, T. Kozsik, R. Király, A. Víg, and T. Nagy, Refactoring in Erlang, a Dynamic Functional Language. In *Proceedings of the 1st Workshop on Refactoring Tools*, Berlin, Germany, July 2007, pp. 45-46.

[9] Jonathan I. Maletic, Michael L. Collard, and Adrian Marcus, Source code files as structured documents. In *Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02)*, pp. 289–292. IEEE Computer Society Washington, DC, USA, 2002.

Received: October 1, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 45–52

The use of development models for improvement of software maintenance

Ján Kunštár
Dept. of computer and informatics
Technical university of Košice,
Letná 9, 042 00 Košice, Slovakia
email: jan.kunstar@tuke.sk

Iveta Adamuščínová
Dept. of computer and informatics
Technical university of Košice,
Letná 9, 042 00 Košice, Slovakia
email: iveta.adamuscinova@tuke.sk

Zdeněk Havlice
Dept. of computer and informatics
Technical university of Košice,
Letná 9, 042 00 Košice, Slovakia
email: zdenek.havlice@tuke.sk

Abstract. Nowadays, the cost of software system is one of the most important factors for choice of certain system by customer. Recent trends in software and system development have revealed the asset of usage of the abstract models through the software life cycle's phases. Abstract models streamline and speed up not only development but suitable models can also improve maintenance process to be more effective and safe. Presented paper briefly analyses SysML, which supports development process of complex systems. Main part is oriented to new approach to model driven system development supporting SysML concept named System Development Unified Process (SDUP) extended by concept of Model-Driven Maintenance.

1 Introduction

These days, models present one of the most important considerations of system and software development. Model-based design supports exploratory design

AMS 2000 subject classifications: 68N99
CR Categories and Descriptors: K.6.3 [Software Management]: Software maintenance
Key words and phrases: software maintenance, software system life cycle, system modeling language (SysML)

and analysis by allowing designers to effectively represent and investigate their knowledge about the system during the decomposition and definition process. Additionally, experiments can be performed on models to eliminate poor design alternatives and to ensure that a preferred alternative meets stakeholder objectives. This modeling concept stays at the core of Model Driven Architecture (MDA). However, one of the most important factors of modeling, in order to support the MDA development process, is the choice of modeling language. To support model-based design and to overcome some limitations related to Unified Modeling Language (UML) strict software focus, the Object Management Group (OMG) has developed the Systems Modeling Language (SysML) [1].
In this paper, a methodology for model-based system development using the SysML is presented with emphasis on Model-Driven Maintenance (MDM) which utilizes development models for improving software maintenance.

2 Model-driven system development and modeling languages

Model-driven architecture [2], defined and supported by the OMG, defines an approach to IT system specifications that separates the system functionalities from the implementation details on a particular technological platform. The MDA [1] is a framework for model driven software development defined by the OMG which has elevated the software development to the next step. Using MDA, it is possible to have an architecture that will be language, vendor and middleware neutral.
One of the key standards that make up the MDA is the UML [1]. UML has proved immensely popular with software engineers, but its software focus has discouraged many system engineers from adopting it earnest. The OMG customization of UML for systems engineering in form of new modeling language called SysML is intended to support modeling of a broad range of systems, which may include hardware, software, data, personnel, procedures, and facilities.

2.1 SysML

OMG SysML is a visual modeling language for systems engineering that extends UML 2 in order to analyze, specify, design and verify complex systems, intended to enhance systems quality, improve the ability to exchange systems

engineering information amongst tools and help bridge the semantic gap between systems, software and other engineering disciplines [1]. OMG SysML reuses a subset of UML 2 concepts and diagrams and augments them with some new diagrams and constructs appropriate for systems modeling. The benefits of using SysML in system development process are following [1], [3]:

- SysML semantics are better suited for systems engineering. SysML reduces UML software-centric restrictions and adds two new diagram types for requirements engineering and performance analysis.

- SysML allocation tables support various kinds of allocations. These tables support requirement, functional and structural allocation, thereby facilitating automated verification and validation and gap analysis.

- SysML's requirement modeling support provides the ability to assess the impact of changing requirements to a system's architecture.

- SysML is a precise language, including support for constraints and parametric analysis which allows models to be analyzed and simulated.

- SysML is an open standard and supports XMI and ISO 10303-303 (AP233) allowing for information interchange to other systems engineering tools.

OMG SysML includes diagrams that can be used to specify system requirements, behavior, structure and parametric relationships. These are known as the four pillars of OMG SysML [1]:
I. Structure. The block is the basic unit of structure in SysML and can be used to represent hardware, software, facilities, personnel, or any other system element. The system structure is represented by block definition and internal block diagrams.
II. Behavior. The behavior diagrams include the use case diagram, activity diagram, sequence diagram, and state machine diagram. The extensions made to standard UML activity diagrams support the compatibility with widely used EFFBD notation that will facilitate and improve interaction between SysML and traditional software engineering tools and facilitate the migration to SysML [3].
III. Requirements. The requirement diagram is a new SysML diagram type that captures requirements hierarchies and the derivation, satisfaction, verification and refinement relationships. This diagram provides a bridge between typical requirements management tools and the system models [3]. Hence requirements become an integral part of the product architecture [1].

IV. Parametrics. The parametric diagram is a new SysML diagram type that describes the constraints among the system's properties associated with blocks. This diagram is used to integrate behavior and structure models with engineering analysis models such as performance, reliability, and mass property models.

3 System development unified process

Presented model of system development life cycle includes all phases typical for the most of common life cycle models. However, within this model, the modifications regarding the MDA development approach using SysML were required. The model emphasizes the maintenance phase and its impact on the whole system development process (Section 4).

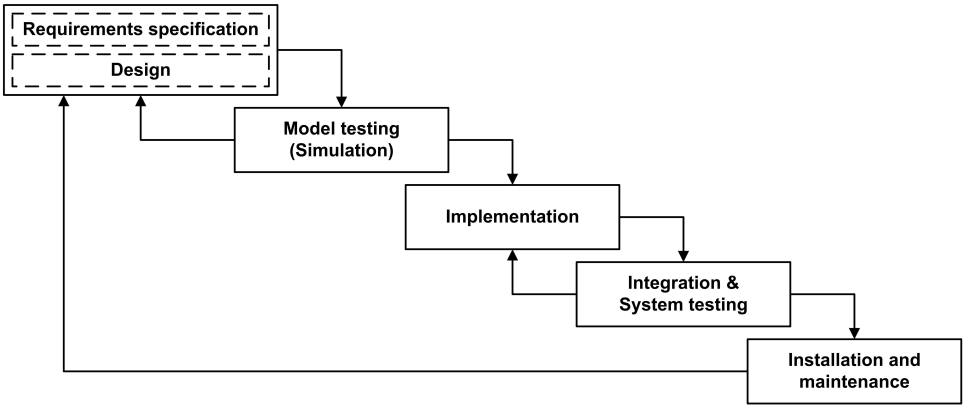


Figure 1: System development unified process

In general, the presented model (Figure 1) may be considered as having five distinct phases, described below:
1. Integrated phase that includes phases of requirements specification and design of the system. By means of using SysML as modeling language, it is possible to integrate these two previously distinct phases into one using the parametric, requirement and design models [1], [3]. This step involves gathering and defining the system's requirements that are directly related to design models with a high level of abstraction that is independent of any implementation technology (platform independent models).
2. Model testing. This phase consists of using the models created in pre-

vious step to be methodically verified to ensure that they are error-free and fully meet the specified requirements. This testing can be processed in form of simulation using the properties of SysML parametrics.
3. Implementation. In this step, the platform independent models are transformed into system's platform specific models that are linked to specific technological platforms (e.g. programming language, operating system or database) [2], [4]. These models are afterwards transformed into implementation artifacts as executable code and database schemas.
4. Integration and system testing. In this stage, both individual system components and the integrated whole are methodically tested and evaluated regarding to technological platforms and quality and reliability of system's performance.
5. Installation and Maintenance. This step involves preparing the system for installation and use at the customer site. A maintenance part involves making modifications to the system or individual component to alter attributes or improve performance. These modifications arise either due to change of requirements, or defects uncovered during system's testing. The main difference compared to standard system maintenance is that no change in system can be processed without accordant modification in design/requirement models (Section 4).

4 Model-driven maintenance

Program comprehension, impact analysis and regression testing are the most challenging problems of software maintenance in the present [5]. An inconsistent state of the software artifacts markedly contributes to all three mentioned problems. Each software system consists of artifacts (e.g. source code, documentation, makefile, models of system) which describe only a limited part of the software and the actual system is their composite. If all system artifacts are't in consistent state, they can't be used together as the source of knowledge about the system. This rapidly decreases the ease with which a software system or its component can be modified during its operation – the maintainability of software system.

4.1 Model-driven maintenance process

Model-driven maintenance process is one useful aspect of knowledge-based software life cycle oriented to better usability of all analysis, design and implementation models in maintenance of systems [6]. MDM is based on uti-

lization of knowledge from the system models and dependences among them for improving maintenance process. Inspiration for MDM is the MDA. MDA concentrates on development of software system using UML as programming language. The direction of progress is from models to application's code. If the change of the system needs to be done according the consistency rules of SDUP (Section 3), it is important to come back to system models, so reverse engineering needs to be used.
In MDM, models of system are the basis for whole maintenance process and therefore there is a requirement to preserve essential models together with the code of application. These essentials models are taken from project database and joined to conjunctive preservation during the installation phase.
Knowledge from essential models, which is element of application, allows us to go cyclically through the phases of life cycle during the maintenance process without the need of browsing project database.

4.2 Model-driven maintenance life cycle

The main difference between the life cycle of normal software maintenance and MDM is in the phase of software system life cycle where the maintenance starts. Normal maintenance life cycle starts with the operation of software system. As system is used, requirements for error correction or requirements (user defined or as a consequence of environment change) for change of the system are detected. The last phase of maintenance life cycle is modification of the system itself. After modification, the system returns to the operation again.
The view that maintenance is strictly a post-delivery activity is one of the reasons that make maintenance hard. According to Pigoski's definition of software maintenance [7], it is very important to prepare software system for its modifications still during the development of the system and not only after delivery. Therefore MDM starts as early as during the system development by conjunction of essential models to application's code. MDM life cycle has the same phases like mentioned normal maintenance life cycle. The difference is in the way how the changes are performed at the basis of the user's requirements. On Figure 2 is displayed the modification process in accord with MDM.
As a first step, the requirements need to be well specified because it is very important to avoid the misunderstandings between users of the system and maintenance programmer. In here, active user participation is very important. Unfortunately most users don't understand the complex diagrams preferred by many traditional modelers. Solution presents the adoption of inclusive models

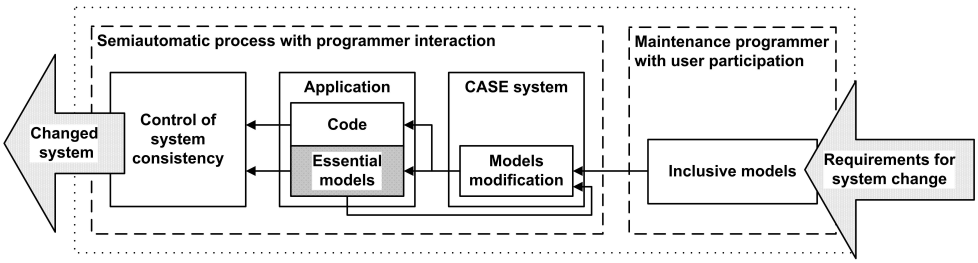


Figure 2: Modification of software system during MDM

which are used to help capture and analyze requirements for certain system [8]. The maintainer can build the requirements diagram after all requirements were exactly specified by users.
After requirements are well specified, maintainer can modify essential models which are part of application. The use of extern CASE system for visualization and applying of changes is useful in this phase. The changes of models can be done without modification of working software system. In this way, the maintenance programmer is able to discover impacts of required changes before they are really implemented to the code of system. When a programmer knows about all required changes he can implement them all in one step, without impacts to unchanged part of the system.
After all, when required modifications are implemented to the code and to the models of application, the consistency control needs to be done. If all changes done in models were processed also in the code, they both describe the same system - they are in consistent state.

5 Conclusions and future work

This paper presents the SDUP, which support the concepts of MDM. This approach based on the conjunctive preservation of program code and models, supports consistency between the essential models and code, as no change in code can be processed without accordant modification in system's models. MDM utilizes knowledge acquired from system's abstract models for uncovering the unwanted side effects of required changes before they are really performed to the system's code. Utilization of system's models streamlines maintenance process and also helps to system comprehension.
In our next research we want to complete realization of proposed MDM. We will work on the proper format of knowledges acquired from essential models.

We will also perform an experimental confirmation of contribution of proposed approach to software maintenance.

Acknowledgement

This work was supported by VEGA Grant No. 1/0350/08 Knowledge-Based Software Life Cycle and Architectures.

References

[1] OMG Systems Modeling Language (OMG SysML) v 1.0 (07-09-01), *OMG Available Specification*, http://www.omgsysml.org/.

[2] A. Kleppe, J. Warmer, W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, 2003, 192 pp, Addison Wesley, ISBN 0-321-19442-X (2003).

[3] T. Weilkiens, *Systems engineering with SysML/UML: modeling, analysis, design*, 322 pp, Morgan Kaufmann Publishers, ISBN: 978-0-12-374274-2 (2007).

[4] A. Benczúr, Z. Hernáth, Z. Porkoláb, LORD: Lay-Out Relationship and Domain Definition Language, *10th Advances in Databases and Information Systems*, Ed: Yannis Manolopoulos et al., Thessaloniki, pp. 215-230 (2006).

[5] G. Canfora, A. Cimitile, Software Maintenance, *Handbook of Software Engineering and Knowledge Engineering*, volume 1. World Scientific, 2001, ISBN: 981-02-4973-X (2001).

[6] Z. Havlice et al., Knowledge-based software life cycle and architectures, *Computer Science and Technology Research Survey*, Košice, ISBN 978-80-8086-071-4 (2007).

[7] T. M. Pigoski, *Practical Software Maintenance Best Practices for Managing Your Software Investment*, John Wiley & Sons, New York, (1997).

[8] S. W. Ambler, R. Jeffries, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, John Wiley & Sons, New York (2002).

Received: October 13, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 53–61

On the role of the reusability concept in automatic programming research

Ladislav Samuelis
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: Ladislav.Samuelis@tuke.sk

Abstract. The central question of this paper is: "How the concept of reusability influences the research in automatic programming?" After discussing briefly the historical background of the question we analyze the research of automatic programming from the software reuse point of view. We try to show the presence of the reusability concept in the automatic programming research throughout its relatively short history. Based on observations, we argue that the concept of software reuse is an inherent idea of the automatic programming research. Finally, we stress the necessity to introduce the historical view into the curricula of teaching informatics at universities.

1 Introduction

The motif of automatic programming is spreading over the history of software engineering in various dimensions. It has been a moving target, which is constantly shifting to reflect increasing expectations. We observe recently the incredible increase in power of the hardware. This increase is itself the reason for the incredible growth of the software complexity. The Wirth's [1] "law": *Software is getting slower, faster than hardware is getting faster*, allegorically points to this fact.

The big challenges for automatic program construction are presented (e.g.) by Laurianne McLaughlin [2] in the following manner:

AMS 2000 subject classifications: 68N19
CR Categories and Descriptors: I.2.2 [Automatic programming]: Subtopic - Program synthesis; D.2.13 [Reusable Software]: Subtopic - Reuse models.
Key words and phrases: software reusability, automatic programming

- To produce good runtime performance;
- To produce code that someone can look at, deal with, and understand;
- To ensure that the code is provably correct.

The aim of this contribution is highlighting the role of the reusability concept within the automatic program construction efforts in the emergent science of software engineering. In particular, to show how the concept of reuse is interwoven into the automatic programming research. We demonstrate selected research ideas in automatic programming in chronological order.

In order to address the question: "How the concept of reusability influences the research in automatic programming?", we have collected several leading ideas, which are grouped around the term of *automatic program construction or program synthesis*. This selection is in no way a complete and representative survey of the field. We hope that the reflections and discussions to the selected concepts will attract minds and in this way enable more deep comprehension of the notion of automatic program construction.

We believe that students studying informatics have to study the history of software engineering very seriously. In particular they have to study deeply the intellectual movements and to read the original works of informaticians whose ideas mostly influenced the field. Students may observe the blind alleys in the history, steps backward, and to follow the evolution of ideas. Pondering and discussions about the ideas could encourage students to create their own ideas, standpoints, and in this way to join the community of educated informaticians. This historical view should be also an essential part of teaching informatics at the universities. This is considered as nutriment that is a necessity for being educated and later being expert in informatics.

This paper begins with a brief history of the idea of automatic programming in Chapter 2. It continues by discussing the reusability principle in the era before object-oriented programming, in Chapter 3. Obstacles to software reuse are referred in Chapter 4 and finally we make conclusions in Chapter 5.

2 Where does the idea of automatic program construction come from?

Looking backward into the history, we observe that the first mentioning and the first discussions around the *industrial production of software* appeared formally in 1968 and were presented at a NATO conference by McIlroy [3].

Many expert-like papers and books concerning automatic software construction have been already presented. However, only few of them tried to discover "laws" which are beyond the software construction or program synthesis along the history of computing [4].

This is an exciting story to follow the interpretations of the notion of automatic program construction and to observe the interleaving of the more-less well-known ideas. This endeavor involves also an attempt to untangle ideas and concepts, which appear suddenly in various contexts.

The term "automated programming" (automatic program construction or program synthesis) is used to refer to the study and implementation of methods for automating a significant part of the process of creating and enhancing software. Its meaning somewhat varies, but often includes several aspects of the programming processes.

A broader goal of this field is to make computer programs much easier by means of automation of the software creation process. More particular goals include increasing software productivity, lowering costs, increasing reliability, making more complex systems tractable, as mentioned in the Introduction.

The *theoretical* question addressed in research in program synthesis is the discovery and articulation of the principles (or appropriate programming languages) underlying the creation of software. The important *practical* question addressed is how to implement systems that embodies a particular knowledge and applying it to assist the programmer or end user.

The history shows us that there is freedom toward the use of more declarative and less procedural specification where appropriate. Another way of description is that the specification is closer to *what* than to the *how* end of the spectrum. The implementation is closer to the how end.

Methods for automated program synthesis vary. One approach is to view synthesis as a set of transformations that perform the steps of successive refinement of the specification into the implementation, performing data structure refinement and control structure refinement of the program into the final implementation. Other methods of program synthesis focus more on theorem proving, where an inference engine derives the steps of the program and proves the necessary facts along the way, e.g. the PROLOG program is such a set of declarations. In particular, a prover proves that an implementation exists, and the proof is made constructive, that is, it constructs the implementation that exists. We note, that both of the above mentioned formulations describe the same goal: to allow users focusing more on solving problems than on the details of implementation.

We have to be aware that programming cannot be fully automated since the

computer must at least be told what to do. There is no way for a computer "to invent an idea". The automation may refer only to the way how "to execute an idea".

3 Automatic program construction before the object-oriented programming

Nowadays it is almost forgotten that before the object oriented approach, within the frame of the classical procedural programming, the automatic program construction was associated (e.g.) with the "construction of programs by examples" or with the "construction of loops".

In the 1950s, the term "automatic computing" referred to almost anything related to computing with a computer. The biggest problem of automatic coding systems was efficiency, or the lack of it. Human-computer interaction was very inefficient. This resulted in an atmosphere in which the idea of automatic coding was conceived as fundamentally wrong: "efficient programming was something that could not be automated" was an often-heard statement.

In the period of "structured programming tide" (1967-1977) the optimizers mattered because they free programmers from the need to deal with specific details to focus on larger issues. In this way developers are able to do more important things. One research stream was dedicated to the loop optimizers. Typical representative of the loop optimizer is the programming by example approach. This technology is based on the induction principle.

In the next section we outline the idea, which represents the automatic program construction efforts in the 1980th. In other words, how the automatic programming by example was perceived before the object orientation [5], [6], [7]. The research before the object-orientation was influenced in great extent by the results gained in the artificial intelligence research. Algorithm described below was discussed in detail by the author elsewhere [8].

The principle of programming by examples

There exist many areas when the demonstration is a suitable tool for automating tasks. For example, paths of robots represent linear plans and the task is to construct program; or the sequence of learning objects represent the progress of the student in the learning material and the task is to construct the navigation plan (learning by watching). The structures of the systems devoted to synthesis of programs by examples are similar also to the structure

of linguistic pattern recognition systems.

In the next paragraphs we describe formally the idea of synthesis the loops from examples [9]. The aim of the synthesis is to construct a minimal final deterministic automaton with branches and loops, which are expressed as:

1. $I_1 \overset{c_1}{\rightarrow} I_2$
2. $I_2 \overset{c_1}{\rightarrow} I_3$

where I_1, I_2 and I_3 are the instructions of the trace and c_1 and c_2 represent the conditions for the execution of the respective instructions. In this model the program equals to the regular grammar:

- (V_n, V_t, D, I_0)

where

- V_n - is the set of program instructions (non terminal symbols),
- V_t - is set of conditions, which belong to the appropriate transitions between the program instructions (set of terminal symbols),
- D - is set of rules, which does not contain 2 or more rules with the same left side,
- I_0 - is the start non terminal symbol.

The algorithm for building the model is summarized as follows. Let the symbol P be a set of available instructions, which are necessary for constructing the example.

$$P = (I_1, I_2, \ldots, I_K) \tag{1}$$

We introduce notation $[I_j]$, for the set of equal instructions $I_j | 1 \leq j \leq K$. We introduce, that

$$[I_j] = 1I_j, 2I_j, \ldots, X_jI_j \tag{2}$$

where the integers in front of I_j are called labels. Let the overall number of I_js in the model equal to $|[I_j]|$ and the $[I_j]^*$ is the actual number of $[I_j]$. The number of the total instructions c_2 in the model is:

$$L = \sum_{j=1}^{K} |[I_j]| \tag{3}$$

where the number of various types of instructions is K. Because the value of the L is varying during the synthesis, we introduce the L^* for the actual value of L. Then

$$L^* = \sum_{j=1}^{K} |[I_j]^*| \tag{4}$$

"Step"in the example is defined as a pair of (c_p, I_q). Different steps (l) may contain the same pairs, i.e. the same condition c_p and same instruction I_q. That is why we introduce the notion of N_l for the condition and O_l for the instruction in certain step l. The $u(l)$ will denote the label of the instruction $O(l)$. The principle of the program synthesis is in searching the value of $u(l)$, which will fulfill the following conditions:

1. The number of instructions L in the program is minimal and it is true that $K \leq L \leq M$, where M is the maximum number of instructions of the example.

2. If the M is the maximum number of instructions of the example, then during the synthesis it is necessary to assign a label $u(l)$ to every instruction $O(l)$ of the example and at the same time to achieve deterministic flow of control. I.e. for every step i where $i \leq l$, and $O(i-1) = O(l-1), u(i-1) = u(l-1)$ and $N(i) = N(l)$, then either

   - $O(l) = O(i)$ and in this case it is possible to provide merging, i.e. $u(l) = u(i)$ or
   - the above-mentioned conditions are not true and $O(l) = O(i)$, then new node has to be created, i.e. $u(l) \neq u(i)$ for the respective instructions in $O(i)$ and $O(l)$. This creation of the new node has to be done in order to secure the deterministic control of flow.

It is evident that when there does not exist a node in the model, which is merge-able with the given instruction in the example, then new node has to be created.

4 Obstacles to software reuse

The book H. Mili et. al. [12] defines the reusability as follows: *Software reuse is the process whereby an organization defines a set of systematic operating*

procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities.

Software reusability is an attribute of software that facilitates its incorporation into new application programs. Reusable software shares many attributes in common with "good software" (i.e. transportability, maintainability, flexibility, understandability, usability and reliability).

The status and the future of the software reuse research is described exhaustly in the paper of W.B.Frakes [11]. The work of Z.Porkoláb [13] points to the relation between metaprogramming and reusability. Interesting fact (or friction) is that "reusability" is not usually a distinguished attribute of artifacts in other engineering disciplines. This induces the following question: "Why do we emphasize so intensively reusability in software engineering?"

If we follow the idea of W. Wang [10] *"The key reason is that software is a tangible form of mathematics that lends itself to being engineered.... This tangibility is both software's strength and Achilles heel."*, then it is clear that this "executability" feature is the driving force behind the "software engineering" activities (e.g. software testing).

It is also often argued that the reusable software assets are "information rich". What does it mean that "information rich"? In fact it means that: software assets represent written ideas and the "customization" of these representations within other context requires excessive mental effort. This idea is valid also for other kinds of representations, as software patterns and models.

The history shows that all contemporary techniques always contained some mechanism of reusability. For example: data encapsulation, information hiding, polymorphism, abstract data types, classes and methods, pipes and filters, inheritance, parametrization and generality, etc. All these techniques could help in certain implementations and domains. More detailed analysis on these issues is out of scope of this contribution.

5 Conclusion

We have discussed some selected ideas around the notion of automatic programming research. We tried focusing on the importance of raising students' awareness to sustainable ideas, which are beyond the "fashionable" ones. This paper could be a basis for formulating further questions in this direction. We expect that the formulation of adequate questions is the first step toward discovery of relevant knowledge in the emergent science of software engineering. We think that this approach could support more thorough understanding of

other software engineering principles too.

Acknowledgments

This work was supported by project VEGA No. 1/0350/08 “Knowledge-Based Software Life Cycle and Architectures” and project VEGA No. 1/0175/08 “Behavioral categorical models for complex programme systems.”

References

[1] N. Wirth, Plea for Lean Software. *Computer*, **28**, 2 (1995) 64–68.

[2] L. McLaughlin, The Next Wave of Developer Power Tools, *IEEE Software*, **23**, 3 (2006) 91–93.

[3] D. McIlroy, Mass-Produced Software Components *Proceedings of the 1st International Conference on Software Engineering, Garmisch Partenkirchen, Germany*, pp. 88–98, 1968.

[4] D. King, C. Kimble, Uncovering the epistemological and ontological assumptions of software designers. *Paper presented at the conference Proceedings 9e Colloque de I”AIM, Evry, France*, 2004.

[5] J. S. Poulin, Technical opinion: reuse: been there, done that. *Comm. ACM* **42**, 5 (1999) 98–100.

[6] Z. Manna, R. J. Waldinger, Toward automatic program synthesis, *Comm. ACM* **14**, 3 (1971) 151–165.

[7] C. H. Smith, The Power of Pluralism for Automatic Program Synthesis. *Journal ACM* **29**, 4 (1982) 1144–1165.

[8] L. Samuelis, Programming by examples, *Technical University of Budapest,1990*, PhD thesis.

[9] A. W. Biermann, Automatic programming, In *Stuart C. Shapiro, editor, Encyclopedia of Artificial Intelligence*. John Wiley and Sons, January 1992.

[10] Wei-Lung Wang, Beware the Engineering Metaphor, *Comm. ACM*, **45**, 5 (2002) p. 29.

[11] W .B. Frakes, K.Kang, Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, **31**, 7 (2005) 529–535.

[12] H. Mili, A. Mili, S. Yacoub, E. Addy, *Reuse based software engineering (techniques, organization, and measurement)*, John Wiley & Sons, 2002.

[13] Z. Porkoláb, Debugging C++ Template Metaprograms, *Generative Programming and Component Engineering*, The ACM Digital Library pp. 255–264.

Received: October 13, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 63–70

Observations on incrementality principle within the test preparation process

Csaba Szabó
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: Csaba.Szabo@tuke.sk

Ladislav Samuelis
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: Ladislav.Samuelis@tuke.sk

Abstract. This paper deals with the abilities of consolidating the system design and test planning phases of the software life cycle (SWLC). We present our observations on the presence of evolution-like features inside test plans during design and development of the application. We focus on the role of incrementality principle within the test preparation process. The presence of this principle is not evoked by the planning process itself, but is inherent in the development stream that incrementally interferes the software design. We analyze this feature and give explanation. Finally, we discuss the impacts of the incrementality principle on management and improvement of software processes.

1 Motivation

Nowadays, a lot of software development methods is available for use in SWLC implementations to achieve the best fit to users' requirements [1]. The main task of each one of them is to deliver faultless software.

When software reliability, safety, stability and overall applicability are questioned, software engineers use testing methods to satisfy and prove these features.

AMS 2000 subject classifications: 68N19
CR Categories and Descriptors: D.2.5. [Testing and Debugging]: Subtopic - Tracing.
Key words and phrases: incrementality principle, test preparation, software evolution, test evolution

Test method selection and execution (i.e. the realization of proofs) strictly depend on the characteristics of the used development method and on the overall SWLC management strategy. Even if no strategy is present, there is no strictly meant freedom in testing process selection. No classical testing is needed in case of mathematically proven algorithm implementation [2] if there is used an also proven implementation technique. (S-type software according to Lehman's laws [3].)

The connection between development and testing does not end by strategy selection and SWLC principles definition. The final verification method of the software depends on the software itself, on its architecture and aim. The most of SWLC models propose test preparation as a separate process that might be executed parallel to or immediately after the design/implementation processes.

On other hand, any interconnection between two models indicates dependencies between them and this is the basis of change propagation across these models [4, 5, 6]. This propagation preserves the actual state of each one according to the development stage and requirements. The mentioned interconnection might be tight or loose depending on the SWLC, but will be present, due to the main principles of software development and testing.

In our paper, we present three observations that highlight change propagation across the whole SWLC, especially on test preparation. Section 2 deals with so-called old school SWLCs based on top-down or bottom-up development strategies. Section 3 is denoted to agile processes of SWLC. In Section 4, we show an example development method called cowboy coding, where might be neither SWLC nor development practices considered. Section 5 concludes our observations and points out more issues to take into consideration in the future.

2 Observation one: old school techniques

Old school methodologies include the waterfall, spiral, staged, iterative, incremental strategies [1]. We classify classical model-driven development (MDD) [7], component based development and the Unified Process (UP) [8] as belonging to this group as well.

Within these methodologies, test preparation is a separate process running parallel divided into smaller activities or sequentially after implementation of software is done.

Having a parallel process implementation, development and test prepara-

tion processes operate the same requirements, functions and interfaces. These techniques principle is to prepare test right after a component interface and/or implementation is finished. Any incremental change of those components having tests already prepared is propagated across these tests as well as across all other affected interfaces/implementations within the same design model. Figure 1 shows a typical example of such a SWLC, where the software reached a specific development level (i.e. can be tested), but there are still parts of it that are not implemented yet.

```
Running make test
Prepending /root/.cpan/build/ExtUtils-MakeMaker-6.46/blib/arch
/root/.cpan/build/ExtUtils-MakeMaker-6.46/blib/lib to PERL5LIB.
PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM"
"-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/00-load.............1/1 # Testing Test::Pod 1.26, Perl 5.008
008, /usr/local/bin/perl
t/00-load.............ok
t/all_pod_files.......ok
t/cut-outside-block....ok
t/good................ok
t/item-ordering.......ok
t/load................ok
t/missing-file........ok
t/pod.................ok
t/selftest............ok
t/spaced-directives....skipped: Not written yet
t/unknown-directive....ok
All tests successful.
Files=11, Tests=19,  1 wallclock secs ( 0.11 usr  0.05 sys +  0
.84 cusr  0.18 csys =  1.17 CPU)
Result: PASS
```

Figure 1: Example test execution upon a module still being developed

Sequential processes do not share any resources during their execution, therefore change propagation might not affect the other process' results. Incrementality is observable e.g. in MDD [7], where test case skeletons are created first, then these skeletons are processed mostly separately. This process includes more detailed specification of test cases. Using independent test cases, the incrementality principle appears only in the phase of their specification. In the case of hierarchical test structure, the dependencies are active through the whole preparation and execution process.

Non-incremental software processes use always sequential execution that

induces a less complicated first-time test preparation but a more complicated regression testing during maintenance. Maintenance is the life-cycle phase when the regression test selection and test review and modification appear to be incremental. Figures 1 and 2 present practical examples of test grouping to ease test selection and regression testing.

```
Running make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/00-load.........1/3 # Testing LaTeX::Parser 1.00, Perl 5.0080
06, /usr/local/bin/perl
t/00-load.........ok
t/01-html.........ok
t/pod-coverage....ok
t/pod.............ok
All tests successful.
Files=4, Tests=20,  0 wallclock secs ( 0.05 usr  0.03 sys +  0.
23 cusr  0.14 csys =  0.45 CPU)
Result: PASS
```

Figure 2: Example test execution upon a final version of a program that was developed using processes of the waterfall SWLC

3 Observation two: agile processes

Agile software development methods use best practices of software development to improve old school methods in selected areas. Extreme programming focuses on rapid delivery and high frequency of iterations with customers involved into the development process, feature and behavior driven programming emphasize on requirement tracing and fulfillment [9]. The test driven development (TDD) [10] approach uses tests as requirement representation and the logical interconnection between tests and program code is very tight. Therefore, changes are propagated across all code not only in the case of intended intervention, but due to the test refactoring as well.

Figure 3 shows a test case reflecting functionality and architecture of the designed implementation. The test case says *Address should contain Street, Number, City, PostalCode attributes; Street should be changeable.* After the implementation of a program that satisfies this requirement, the test case will be extended by other requirements on *Address* one-by-one that incrementally changes it. After any change within the tests, these tests are run to check

```
public class AddressTest extends TestCase {
 public void testStreet() {
  Address address = new Address("Letna","9","Kosice","04200");
  assertTrue(address.getStreet().compareTo("Letna") == 0);

  address.setStreet("Main");
  assertTrue(address.getStreet().compareTo("Letna") == 0);
 }

 protected void runTest() throws Throwable {
  testStreet();
 }
}
```

Figure 3: Example code snippet of JUnit test for TDD

implementation consistency. I.e. changes are propagated from tests to implementation. Considering the next requirement as *requesting Street must not return bad value*, after extending the test case, probably no change within the implementation will be needed.

4 Observation three: cowboy coding

Cowboy coding technique is an approach, where the emphasis is on writing code. It is a term used to describe software development where the developers have autonomy over the development process. This includes control of the project's schedule, algorithms, tools, and coding style. A cowboy coder can be a lone developer or part of a group of developers with either no external management or management that controls only non-development aspects of the project, such as its nature, scope, and feature set. (The *what*, but not the *how*.) An example is the .NET environment where source code in C#, Visual Basic and other languages are used together in programming task solutions. These sources are mostly translated into Common Intermediate Language (CIL) that allows besides bytecode execution a good analysis of the program [11] due to the meta-data that are stored within that bytecode.

In such a case, testing is covered within the project goals as satisfying the requirement of producing a "deliverable" product, or final product evaluation.

For the evaluation phase, acceptance tests are weighted before any other kind of tests. Acceptance test failures are reflected into functional test creation. Functional test failures indicate coding failures that should be detected

by writing and running test case code. Figure 4 reflects relations between the mentioned activities. Incrementality within tests occurs in unit testing phase where old tests are revised and modified.

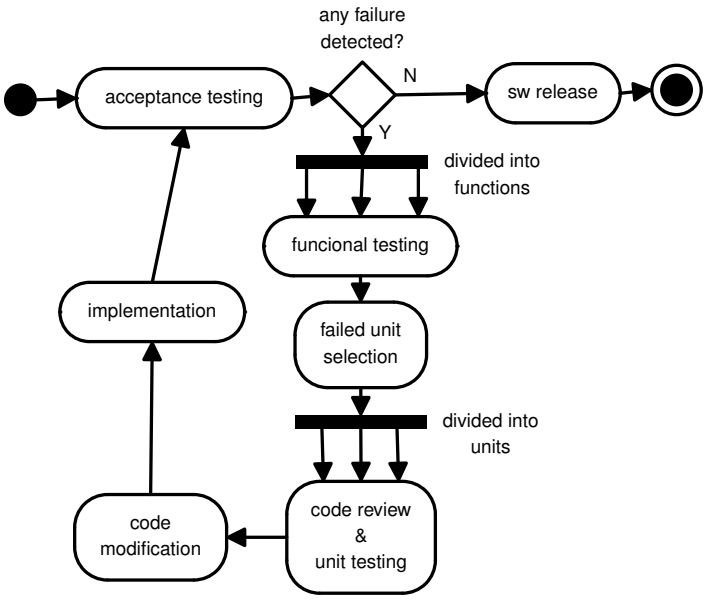


Figure 4: Example testing activities of cowboy coding

5 Conclusion

We highlighted the presence of incrementality principle in software projects and proved that it is a not replaceable feature of test preparation.

Summarizing observations gained with old school techniques, test preparation is distributed across more phases and runs in parallel to other streams of development. Communication between these streams is one-way: changes in the design and/or implementation indicate changes within test plan and/or test cases.It happens because of the main top-down development principle that introduces changes in the way of refinements at current or next (lower) levels of abstraction. On the other hand, the bottom-up strategy needs more independent tests first at implementation level of components. The upper (more abstract) level tests are a kind of an incomplete set of integration tests con-

sidering all possible integrations and all reachable interfaces of components. Change propagation can be observed in cases, when changes are introduced at bottom (or any other low) level.

The second group of observations tells us about the incrementality principle within projects being solved agile. Extreme programming focuses on rapid delivery of software products, iterations incrementally extend the software solution that implicitly starts change propagation. This is propagation in both ways, test might influence the design (the main principle of test driven development), and design/implementation tasks might result into changes in test plan structure or test case behavior.

The third case studies cowboy coding and points to the presence of incrementality principle even in this strategy. Test preparation is based on requirements, these requirements build the main bridge for change propagation. At the highest level, incrementality shows up in the way changed requirements effect the implementation and test cases. At lower level, when a change on the code is made, tests are created/changed only in the case of failure of tests from higher level. Cowboy coding is a fast development technique omitting the most things that are not strictly needed for successful delivery of the software product. Even testing is intended to be as independent from the application implementation as possible, but there are observations presenting some dependencies defined at top level through requirements and at lower levels by failure removal. Cowboy coding was the only methodology allowing less incrementality within our observations, but even there are maintenance and development tasks that indicate change propagation in the indirect way – by making tests fail.

The future of this research might be focused on selection and specification of common parameters to create meta-level descriptions of these dependencies that, in further, might lead to rule discovery by abstraction on these dependencies.

Recently, we run a few projects of quite simple programming tasks being different in the type of used SWLC and being common in the goal to collect test preparation incrementality characteristics. These characteristics will provide attributes and numerical values for further analysis and metrics definition [11].

Acknowledgments

This work was supported by project VEGA No. 1/0350/08 “Knowledge-Based Software Life Cycle and Architectures.”

References

[1] D. Bell, I. Morrey, J. Pugh, *The Essence of Program Design*, Prentice Hall Europe, 1st edition, 1997, Hungarian translation: *Programtervezés*, Kiskapu Kft., 2003.

[2] Z. Juhász, Á. Sipos, Implementation of a finite state machine with active libraries in C++, *Proc. of the 7th International Conference on Applied Informatics*, Eger, Hungary, Vol. 2, 247–255, 2007.

[3] M. M. Lehman, J. F. Ramil, Towards a Theory of Software Evolution – And its Practical Impact, *International Symposium on Principles of Software Evolution,* ISPSE (2000) pp. 2.

[4] L. Samuelis, Cs. Szabó, Notes on the role of the incrementality in software engineering, *Studia Univ. Babeş-Bolyai Inform.*, **51**, 2 (2006) 11–18.

[5] Cs. Szabó, L. Samuelis, Notes on the software evolution within test plans, *Acta Electrotechnica et Informatica*, **8**, 2 (2008) 56–63.

[6] Cs. Szabó, L. Samuelis, Software evolution within test plans – how, when and why? *Egyptian Computer Science Journal*, **29**, 2 (2007) 1–10.

[7] B. Hailpern, P. Tarr, Model-driven development: The good, the bad, and the ugly, *IBM Systems Journal*, **45**, 3 (2006) 451–461.

[8] J. Arlow, I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley, 2nd edition, 2005.

[9] P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, *Agile software development methods – Review and analysis*, Otamedia Oy, Espoo, VTT Publications, 2002.

[10] K. Beck, *Test Driven Development: By Example*, The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[11] Z. Porkoláb et al., Application of OO metrics to estimate .NET project software size, *Proc. of the 7th International Conference on Applied Informatics*, Eger, Hungary, Vol. 2, 293–299, 2007.

Received: October 13, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 71–88

Reconstruction of complete interval tournaments

Antal Iványi
Eötvös Loránd University,
Department of Computer Algebra
1117 Budapest, Pázmány Péter sétány 1/C.
email: tony@compalg.inf.elte.hu

Abstract. Let a, b and n be nonnegative integers ($b \geq a$, $b > 0$, $n \geq 1$), $\mathcal{G}_n(a, b)$ be a multigraph on n vertices in which any pair of vertices is connected with at least a and at most b edges and $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ be a vector containing n nonnegative integers. We give a necessary and sufficient condition for the existence of such orientation of the edges of $\mathcal{G}_n(a, b)$, that the resulted out-degree vector equals to $\mathbf{v}$. We describe a reconstruction algorithm. In worst case checking of $\mathbf{v}$ requires $\Theta(n)$ time and the reconstruction algorithm works in $O(bn^3)$ time. Theorems of H. G. Landau (1953) and J. W. Moon (1963) on the score sequences of tournaments are special cases $b = a = 1$ resp. $b = a \geq 1$ of our result.

1 Introduction

Ranking of objects is a typical practical problem. One of the popular ranking methods is the pairwise comparison of the objects. If the result of a comparison is expressed by dividing points between the corresponding objects, then directed graphs serve as natural tools to represent the results: vertices correspond to the objects, arcs to the points and out-degrees serve as basis for ranking. Another natural tool to represent the results is a point table.

In this paper the terminology of D. E. Knuth [9] and the pseudocode of T. H. Cormen and his coauthors [2] are used.

AMS 2000 subject classifications: 05C20, 68C25
CR Categories and Descriptors: F.2 [Theory of Computation]: Subtopic – Analysis of algorithms and problem complexity
Key words and phrases: score sequences, tournaments, efficiency of algorithms

Let a, b and n be nonnegative integers ($b \geq a$, $n \geq 1$), $\mathcal{T}_n(a,b)$ be a directed multigraph on n vertices in which any pair of vertices is connected with at least a and at most b arcs. Then $\mathcal{T}_n(a,b)$ is called **interval** or (a,b)**-tournament**, its vertices are called **players,** the out-degree sequence $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is called **score vector** and the comparisons are called **matches**.

For the simplicity we suppose that $v_1 \leq v_2 \leq \cdots \leq v_n$. The increasingly ordered score vector is called **score sequence** and is denoted by $\mathbf{s} = (s_1, s_2, \ldots, s_n)$.

If any integer partition of the points is permitted, then the tournament is **complete**, otherwise **incomplete** [7].

If $a = b \geq 1$, then we get multitournaments $\mathcal{T}_n(a)$ and if $a = b = 1$, then we get the well-known concept of tournaments $\mathcal{T}_n$.

In 1953 H. G. Landau [10] proved the following popular theorem. About ten proofs are summarised by K. B. Reid [14] and two recent ones are due to J. Griggs and K. B. Reid [4], resp. to K. B. Reid and C. Q. Zhang [15]. Pirzada, Shah and Naikoo investigated similar problems [13]. Several exercises on tournaments can be found in the recent book of D. E. Knuth [8].

Theorem 1 *A sequence* $(s_1, s_2, \ldots, s_n)$ *satisfying* $0 \leq s_1 \leq s_2 \leq \ldots \leq s_n$ *is the score sequence of some tournament* $\mathcal{T}_n(1)$ *if and only if*

$$\sum_{i=1}^{k} s_i \geq B_k, \quad 1 \leq k \leq n, \tag{1}$$

with equality when $k = n$.

In 1963 J. W. Moon in [11] proved the following generalisation of the Landau's theorem.

Theorem 2 *A sequence* $(s_1, s_2, \ldots, s_n)$ *satisfying* $0 \leq s_1 \leq s_2 \leq \cdots \leq s_n$ *is the score sequence of some a-tournament* $\mathcal{T}_n(a)$ *if and only if*

$$\sum_{i=1}^{k} s_i \geq aB_k, \ 1 \leq k \leq n, \tag{2}$$

with equality when $k = n$.

Figure 1 shows the point table of a tournament $\mathcal{T}_6(2,10)$. The score sequence of this tournament is $\mathbf{s}$ = (9,9,19,20,32,34).

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	Score
$\mathcal{P}_1$	—	1	5	1	1	1	9
$\mathcal{P}_2$	1	—	4	2	0	2	9
$\mathcal{P}_3$	3	3	—	5	4	4	19
$\mathcal{P}_4$	8	2	5	—	2	3	20
$\mathcal{P}_5$	9	9	5	7	—	2	32
$\mathcal{P}_6$	8	7	5	6	8	—	34

Figure 1: The results of the matches of six players.

We wish to decide whether there exist tournaments with a given score sequence and if yes, then we wish to reconstruct one of them.

Our problems can be formulated also as follows [3]. Let $\mathcal{G}_n$ be a multigraph in which the number of connecting edges lies between a and b for any pair of vertices. Design effective algorithms to decide whether there exist an orientation of the edges guaranteeing a prescribed out-degree sequence and to reconstruct a corresponding digraph.

We remark that Gyárfás et al. [5] and Brualdi [1] published quick algorithms for 1-tournaments.

Also it is worth to remark that many enumeration type results are known. In connection with classical tournaments it is known due to P. Tetali [16] that only a few score sequences permit the reconstruction in a unique way: typical is the large number of nonisomorph reconstructions. G. Péchy and L. Szűcs [12] proposed a parallel algorithm for generation of all possible score sequences of the 1-tournaments of n players.

The aim of this paper is to solve the decision and reconstruction problems [6] for complete (a, b)-tournaments.

2 Necessary conditions for (a, b)-tournaments

It is easy too see the following necessary condition, where B_n is the binomial coefficient n over 2 for $n = 1, 2, \ldots$.

Lemma 1 *If* $(s_1, s_2, \ldots, s_n)$ *is the score sequence of some* (a, b)*-tournament* $T_n(a, b)$*, then*

$$\sum_{i=1}^{k} s_i \geq aB_k \quad (1 \leq k \leq n) \tag{3}$$

and

$$\sum_{i=1}^{n} s_i \le bB_n. \tag{4}$$

If $a = 2$ and $b = 10$, then the sequence $\mathbf{s}= (1, 1, 21)$ shows that the requirements of Lemma 1 are not sufficient. Since $\mathcal{P}_1$ and $\mathcal{P}_2$ divided only 2 points, they lost at least 8 points and so the sum of the scores can be at most 22 instead of $b\mathcal{B}_3 = 30$. This remark can be extended to a general condition.

We define a loss function L_k ($k = 0, 1, 2, \ldots, n$) by the following recursion: $L_0 = 0$ and if $1 \le k \le n$, then

$$L_k = \max\left(L_{k-1},\ bB_k - \sum_{i=1}^{k} s_i\right). \tag{5}$$

Now L_k gives a lower bound for the number of lost points in the matches among the players $\mathcal{P}_1,\ \mathcal{P}_2,\ \ldots,\ \mathcal{P}_k$ (not always the exact value since the players $\mathcal{P}_1,\ \mathcal{P}_2,\ \ldots,\ \mathcal{P}_k$ could win points against $\mathcal{P}_{k+1}, \ldots, \mathcal{P}_n$).

Lemma 2 *If* $(s_1, s_2, \ldots, s_n)$ *is the score sequence of some* (a, b)*-tournament* $\mathcal{T}_n(a, b)$*, then*

$$\sum_{i=1}^{k} s_i + (n-k)s_k \le bB_n - L_k \quad (1 \le k \le n). \tag{6}$$

Proof. The member $(n-k)s_k$ of the left side is due to the monotonicity of $\mathbf{s}$. The loss function L_k takes into account the lost points of the matches among the players $\mathcal{P}_1, \ldots, \mathcal{P}_k$. ■

These lemmas imply the following assertion.

Lemma 3 *If* $(s_1, s_2, \ldots, s_n)$ *is the score sequence of some* (a, b)*-tournament* $\mathcal{T}_n(a, b)$*, then*

$$a\mathcal{B}_k \le \sum_{i=1}^{k} s_i \le bB_k - L_k - (n-k)s_k \quad (1 \le k \le n). \tag{7}$$

Proof. (7) is an algebraic consequence of (3) and (6). ■

3 Definition of the algorithms

We describe the proposed new algorithms in words, by examples and by the pseudocode used in [2].

Algorithm ScoreCheck uses Lemma 3. Algorithm ScoreSlicing is an extended version of Ryser's construction method [14], and algorithm Main organises the work of ScoreSlicing.

At first let's consider the small tournament $\mathcal{T}_3(2,10)$ whose point table is shown in Figure 2. The score sequence of this tournament is $\mathbf{s} = (3,4,5)$.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	Score
$\mathcal{P}_1$	—	3	0	3
$\mathcal{P}_2$	0	—	4	4
$\mathcal{P}_3$	4	1	—	5

Figure 2: The results of the matches of three players.

According to (5) we have $L_0 = 0,\ L_1 = 0,\ L_2 = bB_2 - S_2 = 3$, and $L_3 = bB_3 - S_3 = 18$. The requirements of Lemma 3 are $aB_1 = 0 \le S_1 \le bB_3 - 2s_1 = 24$, $aB_2 = 2 \le S_2 \le bB_3 - L_2 - s_2 = 23$ and $aB_3 = 6 \le S_3 \le bB_3 - L_3 = 12$. These inequalities hold.

Let's try to construct a possible point table. The number of points of $\mathcal{P}_i$ against $\mathcal{P}_j$ is denoted by $r_{i,j}$ $(1 \le i,\ j \le n)$. Provisionally we suppose $r_{i,j} = b = 10$, if $j > i$, and $r_{i,j} = 0$ otherwise (in the main diagonal of the table $r_{ij} = 0$ is represented by –).

We begin with the possible results of the player $\mathcal{P}_3$ having the largest number of points. We fix such results for $\mathcal{P}_3$ that after removing of its results from the point table the score sequence (s'_1, s'_2) of the remaining players is monotone and satisfies (7).

$\mathcal{P}_3$ has only $s_3 = 5$ points instead of the possible maximum $(n-1)b = 20$, so $M_3 = 20 - 5 = 15$ points are missing. These points are win by other players or are lost. At first we determine the points win by other players, then the points lost by $\mathcal{P}_3$.

How many is the maximal permitted value of $r_{2,3}$? Since we investigate a (2,10)-tournament, $r_{2,3} \le b = 10$. $\mathcal{P}_1$ and $\mathcal{P}_2$ play a match where they together have to win at least $a = 2$ points, therefore they can win against $\mathcal{P}_3$ at most $A_2 = s_1 + s_2 - aB_1 = 5$ additional points, so $r_{2,3} \le A_2 = 5$. A natural requirement is $r_{2,3} \le s_2 = 4$. The monotonicity requires $r_{2,3} \le s_2 - s_1 = 1$.

The strongest requirement is $r_{2,3} \leq 1$, therefore let $r_{2,3} = 1$. So we founded place for 1 point from the 15 missing points of $\mathcal{P}_3$, the score sequence of the modified $\mathcal{T}_2$ is (3,3), $\mathcal{P}_1$ and $\mathcal{P}_2$ have $A'_2 = s'_1 + s'_2 - aB_1 = 4$ additional points and $M'_3 = 14$.

We divide these additional points between $\mathcal{P}_1$ and $\mathcal{P}_2$ and get $r''_{2,3} = 1+2 = 3$, $r''_{1,3} = 0 + 2 = 2$ and $M''_3 = 10$. These numbers imply $r'_{3,2} = b - r''_{2,3} = 7$ and $r'_{3,1} = b - r''_{1,3} = 8$. Since $A_2 = 0$, that is $\mathcal{P}_1$ and $\mathcal{P}_2$ have no further additional points, they can not win further points from $\mathcal{P}_3$. $\mathcal{P}_3$ lost $r_{2,3} + r_{1,3} = 3 + 2 = 5$ points, so we found 5 of the missing $M_3 = 15$ points. Now we determine $r'_{3,2}$ trying to decrease M''_3 as possible. Since $r''_{2,3}$ is large enough to guarantee $r_{2,3} + r_{3,2} \geq a$ and $M''_3 = 10$ is also large enough, let $r'_{3,2} = 0$ implying $M'''_3 = 10 - 7 = 3$. The next step is to fix $r''_{3,1} = r'_{3,1} - M'''_3 = 8 - 3 = 5$. Now $\mathcal{P}_3$ has the obligatory 5 points, and $\mathcal{P}_1$ needs further $s''_1 = s'_1 - r''_{1,3} = 1$ point, and $\mathcal{P}_2$ needs further $s''_2 = s'_2 - r''_{2,3} = 1$ point. So we can remove $\mathcal{P}_3$ receiving a tournament $\mathcal{T}_2(2, 10)$ with a score sequence $\mathbf{s}'' = (1, 1)$ and we can finish the construction setting $r_{1,2} = 1$ and $r_{2,1} = 1$.

The following Figure 3 shows the reconstructed tournament.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	Score
$\mathcal{P}_1$	—	1	2	3
$\mathcal{P}_2$	1	—	3	4
$\mathcal{P}_3$	5	0	—	5

Figure 3: The reconstructed results of the matches of three players.

In this simple example we can answer the question: how many possible reconstructions are possible? Since $r_{1,2}$ and s_1 determine $r_{1,3}$, $r_{2,1}$ and s_2 determine $r_{2,3}$, $r_{3,1}$ and s_3 determine $r_{3,2}$, we have at most $(s_1 + 1) \times (s_2 + 1) \times (s_3 + 1) = 120$ reconstructions.

The exact value of the number of the possible reconstructions is smaller. For example the permitted values of $r_{1,2}$ are 0, 1, 2, and 3. But if $r_{1,2} = 2$, then $r_{1,3} = s_1 - r_{1,2} = 1$. Now $r_{3,1} + r_{1,3} \geq a = 2$ and $r_{3,1} \leq s_5$ allow only 1, 2, 3, 4 and 5 for $s_{3,1}$, that is there are only 5 possibilities instead of 6.

3.1 Definition of the checking algorithm

Input. a and b: minimal and maximal number of points divided after each match;
n =: the number of players ($n \geq 2$);

$\mathbf{s} = (s_1, s_2, \ldots, s_n)$: a nondecreasing sequence of integers.
Output. One of the following messages:
"i-th score is too small";
"i-th score is too large";
"the sequence satisfies both necessary conditions";
$\mathbf{B} = (B_0, B_1, \ldots, B_n)$: the sequence of the binomial coefficients;
$\mathbf{L} = (L_0, L_1, \ldots, L_n)$: the sequence of the values of the loss function;
$\mathbf{S} = (S_0, S_1, \ldots, S_n)$: the sequence of the sums of the i smallest scores.
Working variables. i: cycle variable.

```
ScoreCheck(n, a, b, B, L, s, S)
01 L_0 ← 0
02 S_0 ← 0
03 B_0 ← 0
04 for i ← 1 to n
05     do S_i ← S_{i−1} + s_i
06        B_i ← B_{i−1} + i − 1
07        L_i ← max(L_{i−1}, bB_i − S_i)
08        if S_i < aB_i
09           then return i"-th score is too small"
10        if S_i > bB_n − L_i − s_i(n − i)
11           then return i"-th score is too large"
12 return "the sequence satisfies both necessary conditions"
```

Figure 1 shows the point table of a tournament of 6 players. In this case the score sequence is $\mathbf{s} = (9, 9, 19, 20, 32, 34)$, $L_0 = 0$, $L_1 = 0$, $L_2 = 0$, $L_3 = 0, L_4 = 3, L_5 = 11$, and $L_6 = 27$. The requirements of (7) are fulfilled: $0 \leq S_1 = 9 \leq 105$, $2 \leq S_2 = 18 \leq 114$, $6 \leq S_3 = 37 \leq 93$, $12 \leq S_4 = 57 \leq 107$, $20 \leq S_5 = 89 \leq 107$, $30 \leq S_6 = 123 \leq 123$. Therefore the conditions in lines 08 and 10 of this program never hold, so the algorithm returns the message of line 12.

3.1.1 Complexity analysis of the checking algorithm

The running time of ScoreCheck is $\Theta(n)$ in worst case.

For incorrect sequences the running time of ScoreCheck can be small. For example if $s_1 = s_2 = (n-1)b$ or $a > 0$ and $s_1 = s_2 = 0$, then the running time is $O(1)$.

We remark that adding a linear time sorting algorithm [2] ScoreCheck can be extended for score vectors too (saving the linear running time).

The memory requirement of ScoreCheck is $\Theta(n)$. If the stepwise input of the scores is permitted, then we can implement this algorithm using only $O(1)$ memory.

3.2 Definition of the main algorithm

The work of the slicing program is managed by the following program Main.

Input. a and b: minimal and maximal number of points divided after each match;
$\mathbf{B} = B_0, B_1, \ldots, B_n$): the sequence of the binomial coefficients;
$\mathbf{L} = (L_0, L_1, \ldots, L_n)$: the values of the loss function;
n: the number of players ($n \geq 2$);
$\mathbf{s} = (s_1, s_2, \ldots, s_n)$: a nondecreasing sequence of integers satisfying (7);
$\mathbf{S} = (S_1, S_2, \ldots, S_n)$: the sums of the scores.

Output. $R = [r_{i,j}]_{n \times n}$: point table of the reconstructed tournament $\mathcal{T}_n(a, b)$.

Working variables. g, i, k: cycle variables;
$\mathbf{p} = (p_1, p_2, \ldots, p_n)$: a provisional score sequence;
$\mathbf{p}_k = (p_1, p_2, \ldots, p_k)$ ($k = 1, 2, \ldots, n$): prefixes of the provisional score sequence $\mathbf{p}$;
$\mathbf{q} = (q_1, q_2, \ldots, q_{k-1}) = (r_{1,k}, r_{2,k}, \ldots, r_{k-1,k})$;
$\mathbf{r} = (r_1, r_2, \ldots, r_{k-1}) = (r_{k,1}, r_{k,2}, \ldots, r_{k,k-1})$.

During the reconstruction process we have to take into account the following bounds:

$$a \leq r_{i,j} + r_{j,i} \leq b \quad (1 \leq i,\ j \leq n, i \neq j); \tag{8}$$

$$\text{modified scores have to satisfy (7);} \tag{9}$$

$$r_{i,j} \leq p_i \ (1 \leq i,\ j \leq n, i \neq j); \tag{10}$$

$$\text{the monotonicity } p_1 \leq p_2 \leq \ldots \leq p_k \text{ has to be saved } (1 \leq k \leq n). \tag{11}$$

Main$(a, b, n, \mathbf{B}, \mathbf{L}, \mathbf{p}, \mathcal{R})$

01 for $i \leftarrow 1$ **to** n
02 **do** $\mathcal{R}_{i,i} \leftarrow 0$
03 $p_i \leftarrow s_i$

04 if $n \geq 3$
05 **then for** $k \leftarrow n$ **downto** 3
06 **do** ScoreSlicing(a, b,**B**,**L**,$k, \mathbf{p}_{k-1}, \mathbf{p}_k$)
07 **for** $g \leftarrow 1$ **to** $k-1$
08 **do** $R_{g,k} \leftarrow q_g$
09 $R_{k,g} \leftarrow r_g$
10 $r_{1,2} \leftarrow \lfloor (p_1 + p_2)/2 \rfloor$
11 $r_{2,1} \leftarrow \lceil (p_1 + p_2)/2 \rceil$
12 return $\mathcal{R}$

3.3 Definition of the slicing algorithm

The key part of the reconstruction is the following algorithm ScoreSlicing.

Input. a, b: minimal and maximal number of points divided after each match;
$\mathbf{B} = (B_1, B_2, \ldots, B_n)$: the sequence of the binomial coefficients;
$\mathbf{L} = (L_1, L_2, \ldots, L_k)$: the values of the loss function;
k: the number of the actually investigated players ($k > 2$);
$\mathbf{p}_k = (p_1, p_2, \ldots, p_k)$: provisional score sequence;
$\mathbf{s} = (s_1, s_2, \ldots, s_k)$: a nondecreasing sequence of integers satisfying (7);
$\mathbf{S} = (S_1, S_2, \ldots, S_k)$: the sums of the scores.

Output: $\mathbf{p}_{k-1} = (p_1, p_2, \ldots, p_{k-1})$: a provisional score sequence;
$\mathbf{q} = (q_1, q_2, \ldots, q_{k-1}) = (r_{1,k}, r_{2,k}, \ldots, r_{k-1,k})$;
$\mathbf{r} = (r_1, r_2, \ldots, r_{k-1}) = (r_{k,1}, r_{k,2}, \ldots, r_{k,k-1})$.

Working variables. $\mathbf{A} = (A_1, A_2, \ldots, A_n)$ the number of the additional points;
d: difference of the maximal increasable scores and the following largest score;
e: number of sliced points per player;
f: frequency of the number of maximal values among the scores $p_1, p_2, \ldots, p_{k-1}$;
g, h, i: cycle variables;
m: maximal amount of sliceable points;
M: missing points: the difference of the number of actual points and the number of maximal possible points of $\mathcal{P}_k$;
$\mathbf{p}_0$: number of points of the hypothetical "negative player" $\mathcal{P}_0$ used in line 15;
$\mathbf{P} = (P_1, P_2, \ldots, P_n)$: the sums of the provisional scores;
x: the maximal index i with $i < k$ and $r_{i,k} < b$.

SCORESLICING($a, b, \mathbf{B}, \mathbf{L}, n, \mathbf{p}_{k-1}, \mathbf{p}_k$)

```
01 p_0 ← 0
02 P_0 ← 0
03 for i ← 1 to k − 1
04     do P_i ← P_{i−1} + p_i
05        A_i ← P_i − aB_i
06 for g ← 1 to k − 1
07     do r_{g,k} ← 0;
08        r_{k,g} ← b;
09 M ← (k − 1)b − p_k
10 while M > 0 and A_{k−1} > 0
11        do x ← k − 1
12           while r_{x,k} = b
13                 do x ← x − 1
14        f ← 1
15        while p_{x−f+1} = p_{x−f}
16              do f = f + 1
17        d ← p_{x−f+1} − p_{x−f}
18        m ← min(b, d, ⌈A_x/f⌉, ⌈M/f⌉)
19        for g ← f downto 1
20           do y ← min(b − r_{x+1−g,k}, m, M, A_{x+1−g}, p_{x+1−g})
21              r_{x+1−g,k} ← r_{x+1−g,k} + y
22              p_{x+1−g} ← p_{x+1−g} − y
23              r_{k,x+1−g} ← b − r_{x+1−g,k}
23              M ← M − y
24           for h ← g downto 1
25              A_{x+1−h} ← A_{x+1−h} − y
26 if M = 0
27   then for g ← 1 to k − 1
28             do r_{g,k} ← max(r_{g,k}, 0)
29                r_{k,g} ← min(r_{k,g}, b)
30                go to 41
```

```
31 if A_x = 0
32    then for g ← k − 1 downto 1
33               do r_{g,k} ← max(r_{g,k}, 0)
34                  for g ← k − 1 downto 1
35                      do y ← max(a − r_{g,k}, 0)
36                         if M ≥ b − y
37                            then r_{k,g} ← y
38                                 M ← M − (b − y)
39                            else r_{k,g} ← b − M
40                                 M ← 0
41 for g ← 1 to 1
42     do q_g ← r_{g,k}
43        r_g ← r_{k,g}
44 return p,  q,  r
```

Let's demonstrate the work of MAIN and SCORESLICING by the reconstruction of the tournament whose point table is shown in Figure 1.

The basic idea is that MAIN slices (partitions) the points of $\mathcal{P}_6, \mathcal{P}_5, \ldots, \mathcal{P}_1$ by repeated calls of SCORESLICING.

The details are as follows. After assigning zeros to the elements of the main diagonal of $\mathcal{R}$ (in lines 01–03) MAIN calls SCORESLICING with $k = 6$. Then SCORESLICING computes the sequence of the additional points $\mathbf{A}$, further the provisional last column and the provisional last row of $\mathcal{R}$ (lines 03-09). The results of the execution of lines 03–08 of SCORESET are represented in Figure 4.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	$\mathbf{p}_6$	$\mathbf{A}$
$\mathcal{P}_1$	—					0	9	9
$\mathcal{P}_2$		—				0	9	16
$\mathcal{P}_3$			—			0	19	31
$\mathcal{P}_4$				—		0	20	45
$\mathcal{P}_5$					—	0	32	69
$\mathcal{P}_6$	10	10	10	10	10	—	34	93

Figure 4: The results of lines 04–08 of SCORESLICING.

Line 09 yields the actual number of the missing points M, then in the lines 10–43 the sequences $\mathbf{p}_{k-1}$, $\mathbf{q}$, and $\mathbf{r}$ are determined.

The steps of the reconstruction of the tournament are shown in Figure 5 in digital form. The second column of the figure contains the starting state of the reconstruction — the score sequence $\mathbf{p}_6 = (9, 9, 19, 20, 32, 34)$.

	$\mathbf{p}_6$	$\mathbf{p}_6$	$\mathbf{p}_6$	$\mathbf{p}_6$	$\mathbf{p}_5$	$\mathbf{p}_5$	$\mathbf{p}_5$	$\mathbf{p}_4$	$\mathbf{p}_3$	$\mathbf{p}_2$
$\mathcal{P}_1$	9	9	9	9	9	9	9	8*	2*	1*
$\mathcal{P}_2$	9	9	9	9	9	9	9	8*	2*	1*
$\mathcal{P}_3$	19	19	19	16*	16	16	9*	8*	2*	–
$\mathcal{P}_4$	20	20	19*	17*	17	16*	9*	9	–	–
$\mathcal{P}_5$	32	22*	22	22	22	22	22	–	–	–
$\mathcal{P}_6$	34	34	34	34	–	–	–	–	–	–

Figure 5: Steps of the reconstruction (stars denote changes).

The second column of Figure 6 contains the actual parameters k, x, A_x, M, f, d, m, and y.

Parameter/k	6	6	6	6	5*	5	5	5	4*	3*	2*
x	5	4*	4	4	4	4	4	4	3*	2*	–
A_x	69	59*	58*	53*	39*	38*	24*	12*	18*	2*	–
M	16	6*	5*	0*	18*	17*	3*	0*	21*	18*	–
f	1	1	2*	–	1*	2*	4*	–	3*	2*	–
d	12	1*	10*	–	1*	9*	9*	–	8*	2*	–
m	10	1*	3*	–	1*	7*	1*	–	6*	0*	–
y	10	1*	2*	–	1*	7*	1*	–	6*	0*	–

Figure 6: Parameters of the reconstruction (stars denote changes).

$\mathcal{P}_5$ has $A_5 = 69 > 0$ additional points (computed in line 05) and $\mathcal{P}_6$ has $M = 16 > 0$ missing points (computed in line 9), therefore ScoreSlice executes lines 10–25. The algorithm determined in lines 11–13 that $\mathcal{P}_x = \mathcal{P}_5$ is the first player who can get from the missing points of $\mathcal{P}_6$. The frequency of players having p_x points is $f = 1$ (computed in lines 14–16). The difference $p_{6,5} - p_{6,4} = 12$ (computed in line 17). At the moment we can slice at most $m = 10$ points per player (computed in line 18). Since A_5 is large enough we get $y = 10$ (computed in line 20), and decrease the number of points of $\mathcal{P}_5$ by $y = 10$ points (in line 21). Therefore the updated new values are

$r_{5,6} = 10,\ r_{6,5} = 0,\ M = 6$ and $A_5 = 59$. The new score vector $\mathbf{p}_6 = (9, 9, 19, 20, 22^*, 34)$ is in the third column of Figure 5 (stars denote changes).

Since $M = 6 > 0$ and $A_5 = 59 > 0$, we use again lines 11–25 and since $r_{5,6} = 10$, we get a new, smaller value $x = 4$. f remains 1, $d = 1$, $m = y = 1$, so $r_{4,6} = 1,\ p_4 = 19,\ r_{6,4} = 9,\ M = 5,\ A_4 = 58$. The new parameters are in the third column of Figure 6, the new score vector $\mathbf{p}_6 = (9, 9, 19, 19^*, 22, 34)$ appears in the fourth column of Figure 5.

Now $M = 5 > 0$ and $A_5 = 58 > 0$, so continuing with lines 10–25 x remains 4 but the frequency is now $f = 2$, the difference $d = 10$, the small M allows only $m = 3$ and $y = 3$ (see fourth column of Figure 6). So it follows $r_{3,6} = 3,\ p_3 = 16,\ r_{4,6} = 1 + 2 = 3,\ p_4 = 17,\ M = 0,\ A_5 = 53$, and $\mathbf{p}_6 = (9, 9, 16^*, 17^*, 22, 34)$ is shown in the fifth column of Figure 5. Since M decreased to zero, ScoreSlicing continues in line 26 and executing line 44 returns to Main the sequences $\mathbf{p}_5 = (9, 9, 16^*, 17^*, 22)$, $\mathbf{q} = (10, 10, 7, 7, 0)$, and $\mathbf{r} = (0, 0, 3, 3, 10)$ shown in the sixth column of Figure 5, resp. in seventh line and seventh column of Figure 7.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	Score
$\mathcal{P}_1$	—	0	0	0	0	**0**	9
$\mathcal{P}_2$	10	—	0	0	0	**0**	9
$\mathcal{P}_3$	10	10	—	0	0	**3**	19
$\mathcal{P}_4$	10	10	10	—	0	**3**	20
$\mathcal{P}_5$	10	10	10	10	—	**10**	32
$\mathcal{P}_6$	**10**	**10**	**7**	**7**	**0**	—	34

Figure 7: The partially reconstructed results of the matches of six players of the given tournament $\mathcal{T}_6(2, 10)$ after determining of the results of $\mathcal{P}_6$, where bold numbers denote final values.

After updating $\mathcal{R}$ Main calls SliceScoring with the parameter $k = 5$.

The parameters determined in lines 11–16 are shown in the sixth column of Figure 6. Since $M = 18 > 0$ and $A_4 = 39 > 0$, the algorithm executes lines 11–25 and gets $r_{4,5} = 1,\ p_4 = 16,\ r_{5,4} = 9, M = 17$, and $A_4 = 38$. The new score vector $\mathbf{p}_4 = (9, 9, 16^*, 16, 22)$ is shown in the seventh column of Figure 6.

Since $M = 17 > 0$ and $A_4 = 38 > 0$, the algorithm in lines 11–16 computes the values shown in the seventh column of Figure 6 and then in lines 18–23 gets $r_{3,6} = 1 + 7 = 8,\ p_3 = 9,\ r_{6,3} = 2,\ r_{4,6} = 0 + 7 = 7,\ p_4 = 9,\ r_{6,4} = 3,\ M = 3,$

and $A_4 = 24$. The new score vector $\mathbf{p}_5 = (9^*, 9^*, 9, 9, 22)$ is shown in the tenth column of Figure 6.

Now $M = 3 > 0$ and $A_4 = 24 > 0$, therefore the algorithm continues in line 11 and gets the parameter values contained in the eighth column of Figure 6. These values imply in lines 18–25 $r_{1,5} = 1,\ p_1 = 8,\ r_{2,5} = 1,\ p_2 = 8,\ r_{3,5} = 1,\ p_3 = 8,\ \mathbf{p}_4 = (8, 8, 8, 9)$ and $M = 0$. Since $M = 0$, the algorithm continues in line 26 and in lines 26–30 gets $\mathbf{q} = (1, 1, 8, 8)$ and $\mathbf{r} = (9, 9, 2, 2)$. ScoreSlicing returns these vectors to Main and it finishes the filling of the sixth line and sixth column of $\mathbf{R}$. The resulted $\mathbf{R}$ is shown in Figure 8.

Main continues by calling ScoreSlicing for $k = 4$. Since $M = 21 > 0$ and $A_3 = 18 > 0$, the algorithm gets in lines 11–16 the parameters shown in the ninth column of Figure 6. Line 20 results $y = 6$ due to the small amount of additional points of $\mathcal{P}_3$. So we get $r_{1,4} = 6,\ p_1 = 2,\ r_{4,1} = 4,\ r_{2,4} = 6,\ p_2 = 2,\ r_{4,2} = 4,\ r_{3,4} = 6,\ p_3 = 2,\ r_{4,3} = 4,\ M = 0$, then $\mathbf{p} = (2, 2, 2)$, $\mathbf{q} = (6, 6, 6)$ and $\mathbf{r} = (3, 3, 3)$. Using the returned vectors Main fills the fifth row and the fifth column of $\mathbf{R}$ as Figure 9 shows.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	Score
$\mathcal{P}_1$	—	0	0	0	**1**	**0**	9
$\mathcal{P}_2$	10	—	0	0	**1**	**0**	9
$\mathcal{P}_3$	10	10	—	0	**8**	**3**	19
$\mathcal{P}_4$	10	10	10	—	**8**	**3**	20
$\mathcal{P}_5$	**9**	**9**	**2**	**2**	—	**10**	32
$\mathcal{P}_6$	**10**	**10**	**7**	**7**	**0**	—	34

Figure 8: The partially reconstructed results of the matches of six players of the given tournament $\mathcal{T}_6(2, 10)$ after determining of the results of $\mathcal{P}_5$, where bold numbers denote final values.

Main continues by calling ScoreSlicing for $k = 3$. Since now $M = 18 > 0$, and $A_2 = 2 > 0$, the algorithm gets in lines 11–16 the parameters shown in the tenth column of Figure 6. So lines 18-25 give the results $r_{1,3} = 1, p_1 = 1,\ r_{2,3} = 1,\ p_2 = 1$, and $M = 0$. Then we get in lines 26–29 that $\mathbf{q} = (1, 1)$ and $\mathbf{r} = (1, 1)$. Using the returned vectors Main fills the fifth row and the fifth column of $\mathbf{R}$, then in lines 10–11 determines $r_{1,2}$ and $r_{2,1}$.

Figure 10 shows the point table of the reconstructed tournament.

Figure 11 shows the rounds of the reconstruction in graphical form.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	Score
$\mathcal{P}_1$	—	0	0	**6**	**1**	**0**	9
$\mathcal{P}_2$	10	—	0	**6**	**1**	**0**	9
$\mathcal{P}_3$	10	10	—	**6**	**8**	**3**	19
$\mathcal{P}_4$	**3**	**3**	**3**	—	**8**	**3**	20
$\mathcal{P}_5$	**9**	**9**	**2**	**2**	—	**10**	32
$\mathcal{P}_6$	**10**	**10**	**7**	**7**	**0**	—	34

Figure 9: The partially reconstructed results of the matches of six players of the given tournament $\mathcal{T}_6(2, 10)$ after determining of the results of $\mathcal{P}_4$, where bold numbers denote final values.

Player/Player	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	Score
$\mathcal{P}_1$	—	1	1	6	1	0	9
$\mathcal{P}_2$	1	—	1	6	1	0	9
$\mathcal{P}_3$	1	1	—	6	8	3	19
$\mathcal{P}_4$	3	3	3	—	8	3	20
$\mathcal{P}_5$	9	9	2	2	—	10	32
$\mathcal{P}_6$	10	10	7	7	0	—	34

Figure 10: The fully reconstructed results of the matches of players of the given tournament $\mathcal{T}_6(2, 10)$.

3.3.1 Complexity analysis of ScoreSlicing and Main

The running time of this algorithm equals to $O(bn^3)$, since the sum of the missing points M_k is $O(bk^2)$, and the sum of the additional points A_k is $O(bk^2)$, and the sum of the scores s_i is $O(bn^2)$, and the processing of a missing point, of an additional point and also of a win point requires $O(n)$ steps.

The memory requirement of ScoreSlicing equals to $\Theta(n^2)$.

The running time of lines 01–03 of Main is $\Theta(n)$. In lines 04–09 algorithm Scoreslicing is executed $\Theta(n)$ times, so the running time of Main depends on the running time of Scoreslicing and is $O(bn^3)$.

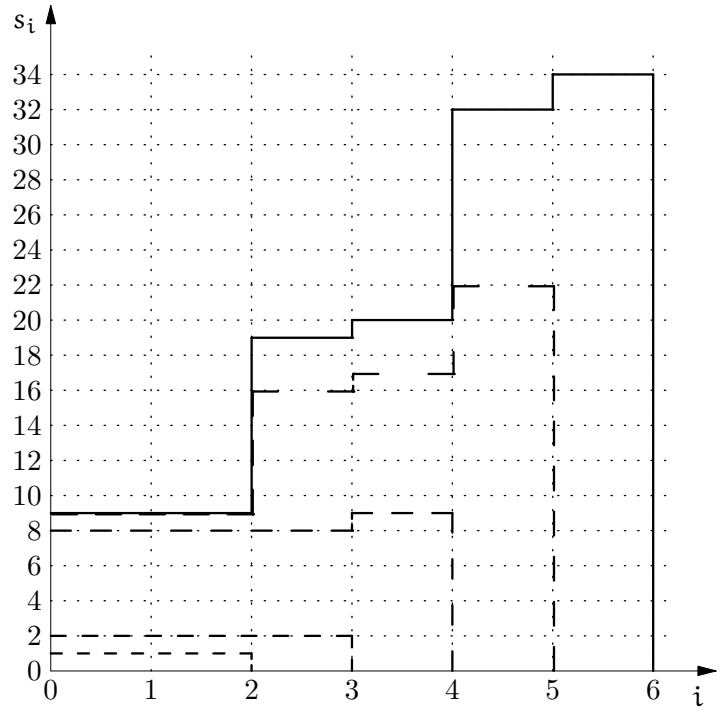


Figure 11: The staircase functions of the score sequences $\mathbf{p}_6 = (9, 9, 19, 20, 32, 34)$, $\mathbf{p}_5 = (9, 9, 16, 17, 22)$, $\mathbf{p}_4 = (8, 8, 8, 9)$, $\mathbf{p}_3 = (2, 2, 2)$, and $\mathbf{p}_2 = (1, 1)$.

4 Necessary and sufficient condition for (a, b)-tournaments

Theorem 3 *A sequence* $(s_1, s_2, \ldots, s_n)$ *satisfying* $0 \leq s_1 \leq s_2 \leq \cdots \leq s_n$ *is the score sequence of some tournament* $\mathcal{T}_n(a, b)$ *if and only if*

$$aB_k \leq \sum_{i=1}^{k} s_i \leq bB_n - L_k - (n-k)s_i \ (1 \leq k \leq n).$$

Proof. Lemma 3 implies the necessity of these inequalities.

The sufficiency of these inequalities can be shown by induction based on the correctness of the reconstruction algorithm.

If $n = 2$, then $a \leq s_1 + s_2 \leq b$ due to 6 and then the scores $r_{1,2} \leftarrow \lfloor S_2/2 \rfloor$ and $r_{2,1} \leftarrow \lceil S_2/2 \rceil$ received by lines 10 and 11 of MAIN are correct values.

Let now $n > 2$. It is sufficient to show that ScoreSlicing reduces the input problem of size n to the reconstruction of the scores of $n-1$ players.

$A_k = S_k - aB_k \leq bB_k - aB_k$ and $M = b(n-1)$ imply $\min(A_k, M) \leq \min((b-a)B_k, b(n-1)) \leq bn(n-1)/2$. This minimum decreases at least by 1 in each execution of the while cycle in lines 23 and 25 – or at least one of M and A_k becomes to zero (if $f = 1$, then $A_k > 0$ due to line 10, and if $f \geq 2$, then $A_{x+1-g} > 0$, since otherwise $A_{x-g} < 0$, what is impossible) and ScoreSlicing ends quickly in lines 26–30 or in lines 31-40.

The inequality (8) is guaranteed by lines 18, 20, and 35.

The inequality (9) is guaranteed by lines 18 and 20.

The inequality (10) is guarantedd by line 20.

The inequality (11) is guaranteed by line 19–23. ■

Acknowledgement. The author thanks Bence Sári (Eötvös Loránd University of Budapest) and Tibor Liska (Computer and Automation Research Institute of HAS) for the computer experiments, András Gyárfás (Computer and Automation Research Institution of HAS) and Béla Vizvári (Eötvös Loránd University of Budapest) for their interest and useful comments.

References

[1] A. R. Brualdi, J. Shen, Landau's inequalities for tournament scores and a short proof of a theorem on transitive sub-tournaments, *J. Graph Theory* **38**, 4 (2001) 244–254.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms. Eleventh corrected printing*, MIT Press/McGraw Hill, Cambridge/New York, 2008.

[3] A. Frank, T. Király, Combined connectivity augmentation and orientation problems, *EGRES Technical Report* No. 2001-17. Egerváry Research Group, Budapest, 2001, pp. 18.

[4] J. Griggs, K. B. Reid, Landau's theorem revisited, *Australas. J. Comb.* **20** (1999) 19–24.

[5] B. Guiduli, A. Gyárfás, S. Thomassé, P. Weidl, 2-partition-transitive tournaments, *J. Combin. Theory Ser. B.* **72**, 2 (1998) 181–196.

[6] A. Iványi, Score vectors of tournaments, In *Proc. of 25th Hungarian Conf. on Operation Research* (Debrecen, October 17–20, 2001), p. 52 (in Hungarian).

[7] A. Iványi, Maximal tournaments, *Pure Math. Appl.* **13** (1–2) (2002), 171–183.

[8] D. E. Knuth, *The Art of Computer Programming. Volume 4, Fascicle 3. Generation of All Combinations and Partitions.* Addison-Wesley, Upper Saddle River, 2005.

[9] D. E. Knuth, *The Art of Computer Programming. Volume 4, Fascicle 0. Introduction to Combinatorial Algorithms and Boolean Functions.* Addison-Wesley, Upper Saddle River, 2008.

[10] H. G. Landau, On dominance relations and the structure of animal societies. II. The condition for a score sequence, *Bull. Math. Biophys.* **15** (1953) 143–148.

[11] J. W. Moon, An extension of Landau's theorem on tournaments, *Pacific J. Math.* **13** (1963) 1343–1345.

[12] G. Péchy, L. Szűcs, Parallel verification and enumeration of tournaments, *Studia Univ. Babeş-Bolyai Inform.* **45**, 2 (2000) 11–26.

[13] S. Pirzada, T. A. Naikoo, N. A. Shah, Score sequences in oriented graphs. *J. Appl. Math. Comput.* **23**, 1–2 (2007) 257–268.

[14] K. B. Reid, Tournaments: Scores, kings, generalisations and special topics, *Congr. Numer.* **115** (1996) 171–211.

[15] K. B. Reid, C. Q. Zhang, Score sequences of semicomplete digraphs, *Bull. Inst. Combin. Appl.* **24** (1998) 27–32.

[16] P. Tetali, Unique tournaments, *J. Comb. Theory, Ser. B.* **72**, 1 (1998) 157–159.

Received: November 2, 2008

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 89–108

Towers of Hanoi – where programming techniques blend

Zoltán Kátai
Sapientia University
email: katai_zoltan@ms.sapientia.ro

Lehel István Kovács
Sapientia University
email: klehel@ms.sapientia.ro

Abstract. According to the literature, the Towers of Hanoi puzzle is a classical *divide et conquer* problem. This paper presents different ways to solve this puzzle. We know that the time-complexity of the puzzle is $2^n - 1$. However, the question is how much time is needed to run each implementation.

1 The legend

Once upon in time, in India, in the reign of Fo Hi, , monks in the Siva-temple of Banares (which marks the center of the world) have to move a pile of 64 sacred golden disks from one diamond peg to another. The disks are fragile; only one can be carried at a time. A disk may not be placed on top of a smaller, less valuable disk. And, there is only one other diamond peg in the temple (besides the source and destination pegs) sacred enough that a pile of disks can be placed there. So, the monks start moving disks back and forth, between the original pile, the pile at the new location, and the intermediate location, always keeping the piles in order (largest on the bottom, smallest on the top) [9]. The legend claims that once the monks are finished, the world will end. So we need to figure out how long it is going to take the monks to finish the puzzle. How many moves will it take to transfer n disks from the source peg to the destination peg?

AMS 2000 subject classifications: 68W40
CR Categories and Descriptors: D.1 [Programming Techniques]
Key words and phrases: Towers of Hanoi, programming techniques, backtracking, greedy, divide and conquer, dynamic programing, recursive, iterative

The puzzle was introduced in 1883 by N. Claus (de Siam) Professor at University of Li-Sou-Stian, an anagram pseudonym for douard Lucas (D'Ameins) Professor at Lyce Saint-Louis [1].

According to the literature – as we can see in the proof –, the Towers of Hanoi puzzle is a classical *divide et conquer* problem. The puzzle can be solved by reducing the problem to smaller, but similar sub-problems; the key-move is the transfer of the largest disc.

In this train of thought we must to transfer $n-1$ discs from the first peg to the third, than we can move the largest disc to the second peg, finally we must to transfer $n-1$ discs from the third peg to the second. According to this recursive move-sequence, it is easy to see, that we need 2^n-1 steps to solve the n-discs puzzle.

So, the monks needs to make $2^{64}-1$ (= 18 446 744 073 709 551 615) moves to solve the puzzle. Assuming, that they are capable to move one disc per second, the end of the world comes approximately in 590 000 000 000 years (according to estimations the Universe is 13,7 billion years old!).

2 The *min/max* problem

The Tower of Hanoi problem has four parameters: $H(n, s, d, h)$

- n: the number of discs
- s: source peg
- d: destination peg
- d: h: "helping peg"

Move n discs from peg s to peg d using the peg h. Initially the source peg is a, the destination peg is b, and the helping peg is c.

The problem, as optimization task, has two versions ($H_{min}(n, a, b, c)$ / $H_{max}(n, a, b, c)$): find the shortest/longest moves-sequence. (Target function: minimize/maximize the number of the moves). In the case of the "maximum version", obviously, we are interested only in cycle free solutions.

We introduce the following notations:

'$\rightarrow$': this symbol shows a direct move (one-step move) of a given disc between to pegs ($a \rightarrow b$: move a given disc from peg a to peg b directly).

'>>': this symbol represents a move-sequence that transposes a disc-tower between to pegs ($a >> b$: transpose a given disc-tower from peg a to peg b).

The key idea behind both solutions is to focus on the largest disc. In case of the "minimum version" the largest disc is moved from peg a to peg b in one (minimum) step ($a \rightarrow b$). This action supposes that, previously, the other

$(n-1)$ discs have already been transposed from peg a to peg c ($a >> c$). After the largest disc has made the move ($a \to b$), the other $(n-1)$ discs have to be transposed again, but at this time from peg c to peg b ($c >> b$). Naturally, in order to achieve the optimal solution, the two moves of the $(n-1)$ discs have to also be performed in minimum number of steps. Consequently, we get the following recursive formula (f_{min}) for the minimum version of the problem (the optimisation is present in the formula only implicitly):

- if $n = 1$, then $H_{min}(n, a, b, c) = a \to b$
- if $n > 1$, then $H_{min}(n, a, b, c) = H_{min}(n-1, a, c, b), a \to b, H_{min}(n-1, c, b, a)$

Denoting with m_n the number of moves needed to solve the minimum version of the n-size problem, we have:

$m_n = 2m_{n-1} + 1, m_1 = 1$
$m_n = 2(2m_{n-2} + 1) + 1 = 2^2 m_{n-2} + 2 + 1$
$m_n = 2(2(2m_{n-3} + 1) + 1) + 1 = 2^3 m_{n-3} + 2^2 + 2 + 1$
...
$m_n = 2^{n-1} m_1 + 2^{n-2} + ... + 2^2 + 2 + 1 = 2^{n-1} + 2^{n-2} + ... + 2^2 + 2 + 1$
$m_n = 2^n - 1$

In case of the maximum version of the problem, the largest disc is moved from peg a to peg b in two (maximum) steps ($a \to c$, $c \to b$). As we exposed above, these moves entail that before, between and after them the other $(n-1)$ discs are transposed in optimal way from peg a to peg b ($a >> b$), from peg b to peg a ($b >> a$), and again, from peg a to peg b ($a >> b$). The recursive formula (f_{max}) that describes the optimal solution for the maximum version of the problem is the following:

- if $n = 1$, then $H_{max}(n, a, b, c) = a \to c, c \to b$
- if $n > 1$, then $H_{max}(n, a, b, c) = H_{max}(n-1, a, b, c), a \to c, H_{max}(n-1, b, a, c), c \to b, H_{max}(n-1, a, b, c)$

Denoting with M_n the number of moves needed to solve the maximum version of the n-size problem, we have:

$M_n = 3M_{n-1} + 2, M_1 = 2$
$M_n = 3(3M_{n-2} + 2) + 2 = 3^2 M_{n-2} + 2 \cdot 3 + 2$
$M_n = 3(3(3M_{n-3} + 2) + 2) + 2 = 3^3 M_{n-3} + 2 \cdot 3^2 + 2 \cdot 3 + 2$
...
$M_n = 3^{n-1} M_1 + 2 \cdot 3^{n-2} + ... + 2 \cdot 3^2 + 2 \cdot 3 + 2 = 2(3^{n-1} + 3^{n-2} + ... + 3^2 + 3 + 1)$
$M_n = 2((3^n - 1)/(3 - 1))$
$M_n = 3^n - 1$

Notice that there are only these two possibilities: the largest disc is moved from the source peg to the destination peg in one or two steps. If we always

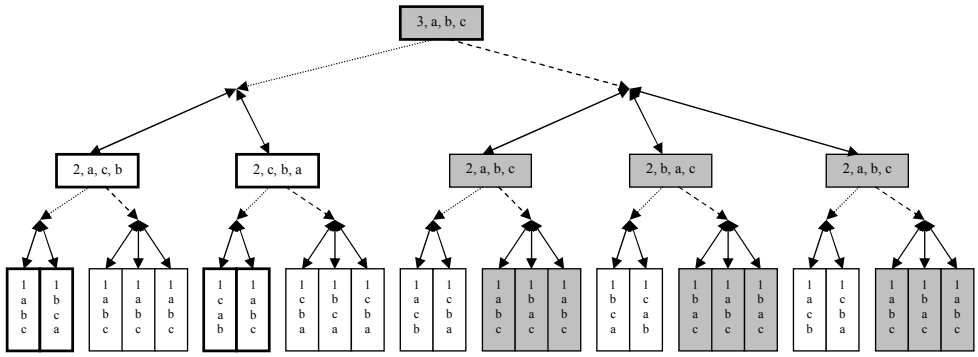


Figure 1: The n=3 case.

choose the one-step solution, we get the minimum-solution. Choosing always the second possibility we have the maximum-solution. If we combine the two possibilities we receive the other solutions. All these can be represented by a rooted tree (see Figure 1). The root-node represents the n-size problem that can be reduced in two ways ("minimum-branch"/"maximum-branch") to two or three $(n-1)$-size sub-problems, together to five sub-problems. This means, that we have a complete n-level tree with 5^{k-1} node at each level k $(k = 1, 2, \ldots, n)$. This tree has $5^n - 1$ nodes. On the one hand, the complete n-level sub-tree that contains only "minimum-branches" (dotted-arcs) correspond to the optimal solution of the minimum version of the problem. On the other hand, the complete n-level sub-tree with only "maximum-branches" (dashed-arcs) represents the maximum-solution of the problem. Notice that the "minimum sub-tree" (bold-line rectangles) has $2^n - 1$ nodes, and the maximum one (filled rectangles) $3^n - 1$ nodes. Furthermore, since each complete n-level sub-tree represents a solution, the problem has as many solutions as such sub-trees exist. For $n = 3$ there are one minimum, one maximum and 1870 other solutions.

The recursive formulas can be seen as the ones that describe the structure of the optimal solutions. These structures can be represented with rooted trees, which correspond to the "minimum sub-tree" and "maximum sub-tree" of the tree shown in Figure 1. Figure 2 shows the tree attached to the minimum version of the 3-size problem ($H_{min}(3, a, b, c)$). The sub-trees represent the optimal sub-solutions. The leaves correspond to the moves. The leaf-sequence represented by dotted arcs corresponds to the optimal move-sequence. The root-node (white-filled rectangles) of each sub-tree has its "own leaf" (dashed-

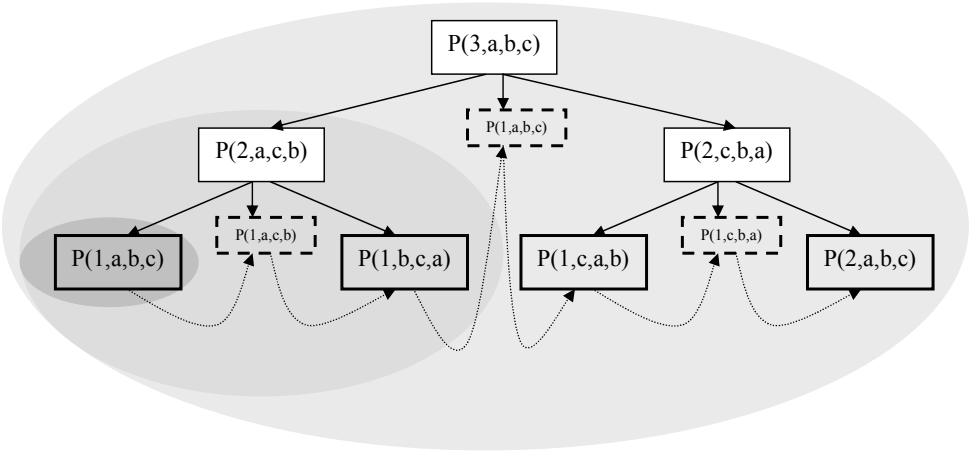


Figure 2: The $H_{min}(3, a, b, c)$ problem.

line rectangles) that represent the move of the largest disc of the corresponding "tower". We consider that these leaves are at the same level with their parent nodes (the "root-nodes" "assimilate" their "own leaves"). If we denote the discs (from bottom to top) and the tree-levels (from root to leaves) with $1, 2, 3, \ldots, n$, respectively, then it can be observed that leaves from a given level k correspond to the moves of disc k. Disc k makes 2^{k-1} moves. Since every second leaf (bolded-line rectangles) of the leaf-sequence is placed on the n^{th} level, these moves are performed by the smallest disc.

3 The recursive implementation

The below recursive procedures (P_recursive_min, P_recursive_max) are direct transcriptions of the recursive formulas.

```
void P_recursive_min(int k, char s, char d, char h)
{
  if(k==1)
    printf("%c -> %c\n", s, d);
  else
    {
      P_recursive_min(k-1, s, h, d);
```

```
      printf("%c -> %c\n", s, d);
      P_recursive_min(k-1, h, d, s);
    }
}


void P_recursive_max(int k, char s, char d, char h)
{
  if(k==1)
    printf("%c -> %c\n%c -> %c\n", s, h, h, d);
  else
    {
      P_recursive_max(k-1, s, d, h);
      printf("%c -> %c\n", s, h);
      P_recursive_max(k-1, d, s, h);
      printf("%c -> %c\n", h, d);
      P_recursive_max(k-1, s, d, h);
    }
}
```

What programming technique(s) applies these implementations?

4 Divide and conquer or greedy?

At the first sight the above implementations are pure divide and conquer algorithms: the problem to be solved is divided in two/three simpler, similar subproblems. However, why is the problem divided in sub-problems even in these specific ways? These are direct consequences of the decision we are made with respect to the largest disc, decisions that are greedy choices. (The largest disc is moved from the source peg to the destination peg in minimum/maximum steps) Consequently, from this point of view, the recursive implementations are such divide and conquer strategies that apply greedy decisions in the dividing stage of the method.

On the other hand it can be stated that the "principle solving" of each subproblem starts with a greedy decision (relative to the largest disc) that reduces the current sub-problem to two/three simpler, similar sub-sub-problems. Particularly in the case of this problem (Tower of Hanoi) the two (minimum version) sub-sub-problems have to be solved before, and after (respectively) the greedy decision is implemented. As a consequence, the order the greedy

decisions have to be implemented follows a kind of in-order traverse of the tree representing the optimal solution (see figure 2). Accordingly, from this point of view, the recursive implementations are greedy strategy that use divide and conquer like methods (in-order DFS algorithm) in order to establish the implementation order of the greedy decisions.

The standard greedy algorithm first implements the greedy decision and after this it solves the "reduced sub-problem(s)". The pre-order DFS and the BFS traverses of the "solution-tree" allow of such a greedy strategy. In order to establish the proper move-order, these implementations need to store the moves. Unfortunately, there is a major concern relating this idea. The size of the solution-code depends exponentially on the problem-size (n). Coding the six possible moves $a \to b$, $a \to c$, $b \to c$, $c \to b$, $c \to a$, $b \to a$ with digits 0, 1, 2, 3, 4, 5 the code of the optimal solution of a 64-size problem has $2^{64} - 1$ digits. No computer capable to store that huge amount of data. Assuming that n takes moderate-values the optimal move-sequence can be stored in array moves[0..(2^n-1)-1]. Bi-dimensional array code stores the move-codes. Procedure P_greedy_min determines the elements of array moves according to the BFS traverse of the tree. Array q[0..(2^n-1)-1] implements the queue necessary for the BFS. Variables first and last indicate the beginning and the end of the queue. The items of the queue q store the source (s), destination (d) and helping (h) pegs of the corresponding sub-problem, and the left (l) and right (r) margins of the current segment in array moves. The array-segment [0..(2^n-1)-1] (the whole array moves) is attached to the original problem. At each step procedure P_greedy_min stores the greedy-move (s→d) corresponding to the sub-problem from the front of the queue in the middle element (m = (q[first].l+q[first].r)>>1) of the corresponding array-segment ([q[first].l .. q[first].r]). The array-sub-segments [q[first].l .. m-1] and [m+1 .. q[first].r] are attached to the left and right son-sub-problems.

```
void P_greedy_min(int n, char a, char b, char c, int* moves, item* q,
int(*code)[3])
{
  int min_step_nr =(1<<n)–1;
  int first = 0;
  q[0].s = a; q[0].d = b; q[0].h = c;
  q[0].l = 0; q[0].r = min_step_nr – 1;
  int last = 1;
  int m;
```

```
  while (first < last)
  {
    m = (q[first].l + q[first].r) >> 1;
    moves[m] = code[q[first].s-'a'][q[first].d-'a'];
    if (q[first].l < q[first].r)
    {
      q[last].s = q[first].s; q[last].d = q[first].h; q[last].h = q[first].d;
      q[last].l = q[first].l; q[last].r = m - 1;
      ++last;
      q[last].s = q[first].h; q[last].d = q[first].d; q[last].h = q[first].s;
      q[last].l = m + 1; q[last].r = q[first].r;
      ++last;
    }
  ++first;
  }
  for(i = 0; i < min_step_nr; ++i)
    printf("%d, ", moves[i]);
}
```

5 Dynamic programming

The recursive formulas can be interpreted as the ones that describe the way the optimal solution is built by optimal sub-solutions (principal of the optimality). The principle of the optimality was introduced by Richard Bellman [10], who called the corresponding recursive formulas as the functional equations of the problem. Dynamic programming follows this strategy: it starts from the optimal solutions of the trivial sub-problems and builds the optimal solutions of the more and more complex sub-problems and eventually of the original problem. In case of this problem the bottom-up way means that we solve the problem for k=1, 2, 3, ..., n. Although for a specific k there are six different k-size sub-problems (P(k, a, b, c), P(k, a, c, b), P(k, b, c, a), P(k, b, a, c), P(k, c, a, b), P(k, c, b, a)), their solutions can be obtained from one another by simple letter-changing. For instance, the move-sequence that solves optimally the $H_{\min}(k, c, a, b)$ sub-problem can be obtained from the optimal solution of the $H_{\min}(k, a, b, c)$ sub-problem by changing letter a with c, letter b with a and letter c with a. Accordingly, for each k=1, 2, 3, ..., n only one "variant" of the k-size sub-problems would be enough to be solved.

The main difficulty of dynamic programming is that it is often nontrivial to establish what sub-problems in what order have to be solved. In Figure 2 the imbricated ellipses illustrate the bottom-up strategy the dynamic programming follows. Notice, that the optimal solution of the $H_{min}(3, a, b, c)$ problem implies only two 2-size and five 1-size optimal sub-solutions. The "dynamic programming order" the sub-problems (for n=3) have to be solved is the following: 1. $H_{min}(1, a, b, c)$; 2. $H_{min}(2, a, c, b)$ (built up from the optimal solutions of the $H_{min}(1, a, b, c)$ and $H_{min}(1, b, c, a)$ 1-size "variants"); 3. $H_{min}(3, a, b, c)$ (built up from the optimal solutions of the $H_{min}(2, a, c, b)$ and $H_{min}(2, c, b, a)$ 2-size "variants"). The growing sub-tree sequence included in the imbricated ellipse-sequence represents the "increasing" sub-problem sequence that, according to the dynamic programming strategy, has to be solved. In this sequence the current sub-problem succeeds its left-son-sub-problem, and precedes its father-sub-problem. For n odd/even these sub-problem sequences are the followings (in case of the minimum version of the problem):

$H_{min}(1, a, b, c)$, $H_{min}(2, a, c, b)$, $H_{min}(3, a, b, c)$, $H_{min}(4, a, c, b)$, ..., $H_{min}(n, a, b, c)$.

$H_{min}(1, a, c, b)$, $H_{min}(2, a, b, c)$, $H_{min}(3, a, c, b)$, $H_{min}(4, a, b, c)$, ..., $H_{min}(n, a, b, c)$.

If it is difficult to build an iterative algorithm that determines this "dynamic programming order", than, it is advisable to try to use the recursive formula dictated order. Unfortunately the direct transcription of the recursive formula into recursive procedure usually results in inefficient divide and conquer algorithm. Since dynamic programming problems are often characterized by overlapping sub-problems, the standard divide and conquer approach commonly results in repeated evaluation of the identical sub-problems. To avoid this ingredient the so-called "recursion with result caching" (memoization) technique can be applied. According to this technique ones a sub-problem has been solved its optimal solution (often the optimal value of the target function) is stored (memorized), and whenever later the recursive algorithm meets again the same sub-problem its stored solution is simply retrieved.

As specificness, in case of Towers of Hanoi problem solving a sub-problem *means* to solve its son-sub-problems *effectively*. Consequently, all sub-problems have to be solved as many times as the algorithm meets them (as in case of divide and conquer problems). Nevertheless we could do to store the code of the optimal solutions of the solved sub-problems. (This method also works only for moderate values of parameter n) Moreover, as we pointed out above, it would be enough to store the code of the optimal solution of only one variant for each k=1, 2, 3, ..., n. Unfortunately, managing (storing, retrieving,

generating from one another, printing) these exponential-size move-sequences has the same time-complexity as generating them again (The optimal solution of the k-size problem has the same number of moves as the number of nodes of the corresponding k-level binary tree). Consequently, the memoization technique, in this case, do not decrease (compared to the direct, divide and conquer like implementations of the formulas f_{min} and f_{max}) the time complexity of the algorithms.

Returning to the "recursion with result caching" technique, notice that in the sub-problem sequence to be solved there are only to type of problems: $H_{min}(k, a, b, c)$ and $H_{min}(k, a, c, b)$. In case of first-type sub-problems ($H_{min}(k, a, b, c)$) the move of the largest disc is $a \rightarrow b$ (coded with 0). Furthermore, after the optimal solution of the left-son-sub-problem ($H_{min}(k-1, a, c, b)$) has been stored, the optimal move-sequence for the right-son-sub-problem ($H_{min}(k-1, c, b, a)$) is generated by the following letter-changes: a is changed with c, b is changed with a and c is changed with b. According to these letter-changes we have the following move-changes: change move 0 with move 4, move 1 with move 3, move 2 with move 0, move 3 with move 5, move 4 with move 2, move 5 with move 1. For the second type problems ($H_{min}(k, a, c, b)$) the largest-disc-move is $a \rightarrow c$ (coded with 1). The corresponding move-changes are the followings: change move 0 with move 2, move 1 with move 5, move 2 with move 4, move 3 with move 1, move 4 with move 0, move 5 with move 3.

Procedure P_memoization_min applies the memoization technique. Rows 0 and 1 of array move_changing[0..1][0..5] store the two move-changing patterns to be applied (alternatively). (Binary variable pattern indicates the largest-disc-move and the move-changing-pattern to be applied)

```
void P_memozation_min(int k, char s, char d, char h, int n, int *moves)
{
  int i, p, pattern;
  pattern = ((n+k)&1);
  if(k==1)
    moves[0]=pattern;
  else
  {
    P_memozation_min(k-1, s, h, d, n, moves);
    p=(1<<(k-1))-1; //the number of moves in the son-sub-problems
    moves[p]=pattern; //the move of the largest disc
```

```
    for(i=0;i<p;++i) //generating the right-son-solution
      moves[p+1+i] = move_changing[pattern][moves[i]];
  }
}
```

What can we say as a conclusion for this ? Since the recursive formulas implemented by procedures P_recursive_min (P_recursive_max) and P_memoization_min are direct materializations of the principle of the optimality, and what is more, they accomplish (along the back-way of the recursion) the bottom-up building process prescribed by this principle, the strategy these procedures implement can be considered dynamic programming. More exactly: recursive dynamic programming. Most exactly (especially in case of procedures P_recursive_min and P_recursive_max): divide and conquer like recursive dynamic programming algorithms.

Remark: If we analyse the recursive formulas in top-down direction, then they describe the way the greedy decisions reduce the problem to similar, simpler sub-problems. The bottom-up analysis of the same formulas shows the way the optimal solution (of the problem) is built on the score of the optimal sub-solutions (of the sub-problems). This is why the algorithm can be seen both greedy and dynamic programming strategy.

6 Backtracking

According to the backtracking strategy, we try to find the optimal solution as a move-sequence. At each stage of the problem-solving process there are three possible moves. Considering the three pegs "large", "medium" and "small" according to the size of their top-disc, the three moves are: "small"→"medium", "small"→"large", "medium"→"large". The solution-space of the problem can also be represented by a rooted-tree. The nodes correspond to the stages of the problem. Each stage can be characterised by a set-triplet. The sets of a give triplet contain the discs from the corresponding pegs. Since at each stage there are three possible moves, all nodes (expecting the leaves) have three son-nodes. The root-node represents the initial stage of the problem when all discs are on peg a: $\{(1, 2, \ldots, n);();()\}$. In the final stage, corresponding to the solution-leaves, all discs are on peg b: $\{();(1, 2, \ldots, n);()\}$. The shortest/longest root – ,,solution-leaf" path, represents the optimal solution.

The backtracking algorithms apply depth-first search (usually implemented recursively), and choose the optimal solution by the standard minimum-/maximum-search method. In case of the Towers of Hanoi problem the backtracking

strategy, in its primitive form, avoids only the loops and it is very inefficient. For example, the tree representing the structure of the optimal solution of the minimum version of the problem (Figure 2) has $2^n - 1$ nodes (supposing that "root-nodes" assimilate their "own leaves"), equal with the numbers of the moves along the optimal move-sequence. In case of the solution-space-tree, only its shortest root – "solution-leaf" path contains so many nodes. Furthermore, since the backtracking algorithm also needs to store the code of the current move-sequence, this algorithm also works only for moderate-values of parameter n.

The elements of array pegs[0..2] store the number of discs on the corresponding peg (pegs[i].nr), and the discs themselves (pegs[i].discs[0..(n-1)]). The source and destination pegs corresponding to the six possible moves are memorized in array move_types[0..5]. Arrays moves and states store the current moves- and state-sequence. (In a given stage of the solution building process the state of the problem can be described by the peg-sequence corresponding to the positions the discs occupy) Function valid verifies if move i is valide as next step (k). Procedures move_forward and move_backward move and remove the current disc.

```
void P_backtrack_min(int k, ITEM_P*pegs, ITEM_MT*move_types,
int*moves, ITEM_ST*states)
{
  if (states[k]==ENDSTATE)
  {if(k<kmin){kmin=k; copy(opt_solution, moves, kmin);}}
  else
  {
    for(i=0; i<6;++i)
    {
      if(valid(i, k, pegs, move_types, states))
      {
        move_forward(i, k, pegs, move_types, states, moves);
        P_backtrack_min(k+1, pegs, move_types, moves, states);
        move_backward(i, k, pegs, move_types, states);
      }
    }
  }
}
```

To optimise a backtracking algorithm means to reduce the traversed part of

the solution-space-tree. An utterly optimised backtracking algorithm traverses only the optimal root – "solution-leaf" path. This means that we are able to establish at each stage the optimal move. However such an algorithm would be rather greedy or dynamic programming than backtracking. (See "The iterative implementation" section.)

7 The iterative implementation

In the literature [6], [7], [8] and others debates on non-recursive dynamic programming algorithms, as solutions of the Towers of Hanoi puzzle. The ideas analysed in previous sections raise the following question: Is it possible to generate the move-sequence that represents the optimal solution iteratively? It is not hard to realize that procedure P_memozation_min can be easily transcribed to an iterative dynamic programming algorithm.

According to the "dynamic programming order" presented above, the algorithm advances from father to father. The optimal solution of any father-sub-problem can be determined on the score of the optimal solution of its left-son-sub-problem. If the current sub-problem in the bottom-up building process is $H_{min}(k, x, y, z)$, then $H_{min}(k, y, z, x)$ is its right-brother-sub-problem, $H_{min}(k+1, x, z, y)$ is its father-sub-problem, and the move the largest disc of the father-sub-problems has to perform is $x \rightarrow z$. As we previously mentioned, the optimal solution of the right-brother-sub-problem can be found from the solution of the left-brother-sub-problem by simply letter changing. To solve the $H_{min}(k+1, x, z, y)$ father-sub-problem means, that, after the current left-son-sub-problem ($H_{min}(k, x, y, z)$) has been solved, we move disc $(k+1)$ from peg x to peg z, and than we generate the solution of the right-son-sub-problem ($H_{min}(k, y, z, x)$). The starting problem is $H_{min}(1, a, b, c)$ or $H_{min}(1, a, c, b)$ depending if n is odd or even. Procedure P_iterative_DP_min follows this iterative dynamic programming strategy.

```
void P_iterative_DP_min(int n, int *moves)
{
  int i, k, pattern, p;
  moves[0] = pattern = !(n&1); //the first move
  for(k=2;k<=n;++k)
  {
    p=(1<<(k–1))–1;
    pattern  ^ = 1; //we change the pattern
    moves[p]=pattern; //the move of the largest disc
```

```
                    //corresponding to the father problem
    for(i=0;i<p;++i) //generating the righ-brother-solution
      moves[p+1+i] = move_changing[pattern][moves[i]];
  }
}
```

The main concern about procedure P_iterative_DP_min is related (as we described above) with the limited memory capacity of the computers. How can we eliminate this ingredient? The solution is based on the following observations (We have considered the minimum version of the problem; We consider the three pegs "large", "medium" and "small" according to the size of their top-disc):

• In any intermediate state of the problem solving process there are three possible moves: "small"→"medium", "small"→"large", "medium"→"large".

• The top-disc of the "small-peg" is always the smallest disc. It is not allowed two consecutive moves with the smallest disc. (To avoid the loops, and to maintain the minimal character of the move-sequence). So, the smallest and the medium-size top-discs move alternatively.

• If the next to move is the medium-size top-disc, than it is clear that the "medium"→"large" move has to be performed. The ,,own leaves" of the root-nodes represent these moves. (Figure 2)

• The smallest disc (disc n) moves according to the patterns (a, b, c, a, b, c, ...) or (a, c, b, a, c, b, ...) depending on the odd or even character of parameter n. Consequently, the moves of the smallest-disc are also unambiguously. (There is only one optimal-size solution)

What reasoning lies behind the last remark? These move-patterns can be established by a careful analysis of the bottom-up building process, and the correctness of them is proved by mathematical induction. We assume that the problem to be solved is $H_{min}(n, a, b, c)$, and n is odd. We will prove that in this case the smallest disc follows the (a, b, c, a, b, c, ..., c, a, b) 2^{n-1} long move-sequence.

For n=1 we have only the smallest disc and its move is $a \to b$. Assume that for a given odd $n > 1$ the 2^{n-1} long move-sequence of the smallest disc is: a, b, c, a, b, c, ..., c, a, b. The right-brother sub-problem of problem $H_{min}(n, a, b, c)$ is $H_{min}(n, b, c, a)$. The corresponding move-sequence for the smallest disc is: b, c, a, b, c, a, ..., a, b, c. Concating these patterns, we get the 2^n long move-sequence corresponding to the father-sub-problem $H_{min}(n+1, a, c, b)$: a, b, c, a, b, c, ..., a, b, c. Repeating this procedure we get for the $H_{min}(n+2, a, b, c)$ grandfather-sub-problem the 2^{n+1} long a, b,

c, a, b, c, ..., c, a, b move-sequence-pattern. We can use the same train of thought for n even.

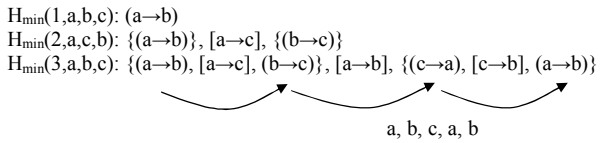


Figure 3: We used round brackets for the moves of the smallest disc and square brackets for the moves of the largest disc of the corresponding sub-problem ($k > 1$). The curly brackets-pairs represent the brother sub-problems.

Consequently: the strategy applied by this iterative algorithm is mainly dynamic programming due two the following reasons:

• The algorithm generates the move-sequence that implements the bottom-up solution-building process of the dynamic programming strategy.

• The way the move-sequence is established is rooted in optimalisations included in the principal of the optimality.

• The move-pattern the smallest disc follows can be determined by bottom-up analyses of the recursive formulas.

Interestingly, the move-patters the smallest disc has to follows can also be determined by a greedy approach of the problem. Solving the $H_{min}(n, a, b, c)$ problem we have the following moving-patterns:

• all n discs get from peg a to peg b: $a >> b$; (between stages a and b may also be other stages)

• disc 1 moves: $a \rightarrow b$ (disc 1 moves directly from peg a to peg b; greedy-move)

• top (n-1) discs: $a >> c >> b$ (top (n-1) discs pass through stages a, c, b)

• disc 2 moves: $a \rightarrow c \rightarrow b$, greedy-moves

• top (n-2) discs: $a >> b >> c >> a >> b$;

• disc 3 moves: $a \rightarrow b \rightarrow c \rightarrow a \rightarrow b$, greedy-moves

• top (n-3) discs: $a >> c >> b >> a >> c >> b >> a >> c >> b$;

• disc 4 moves: $a \rightarrow c \rightarrow b \rightarrow a \rightarrow c \rightarrow b \rightarrow a \rightarrow c \rightarrow b$, greedy-moves

• ...

Since for all $k > 1$ sub-problem $H_{min}(k, x, y, z)$ is reduced to sub-problems $H_{min}(k-1, x, z, y)$ and $H_{min}(k-1, z, y, x)$ the next moving-pattern is generated from the current one by intercalating between all consecutive stages the

"third stage". Notice that there are only two patters, one for odd-discs (a, b, c, a, b, c, ...) and one for even-discs (a, c, b, a, c, b, ...).

The fact that the $p_1 >> p_2 >> \ldots >> p_m$ stage-sequence of the k-size tower in case of its largest disc means $p_1 \to p_2 \to \ldots \to p_m$ direct move-sequence is based on greedy decisions. Furthermore, the above-presented "pattern generating process" is hand-by-hand with the top-down greedy strategy described previously. Accordingly, we can state, that (form this point of view) the move-patterns the iterative algorithm follows, can also be established in greedy way.

Procedure P_iterative_min implements the above-presented iterative algorithm. The current values in array state[1..n] represent the current state of the problem. Element state[i] store the peg corresponding to the position of disc i. Variable i represents the current disc that moves. During odd steps (k is odd) disc n moves. At even steps (k is even) the smallest disc, that is on different peg than disc n, moves. For odd i disc i follows the "increasing circular pattern": a, b, c, a, b, c,... In cases when i is even disc i has to move according to the "decreasing circular pattern": c, b, a, c, b, a, ...

```
void P_iterative_min(int n, char a, char b, char c)
{
  char p = (char*)calloc(n+1, sizeof(char));
  for(i=1;i<=n;++i) p[i] = a;
  int nr=(1<<n)-1;
  int k=1;
  while(k<=nr)
  {
    if(k&1) i=n; //the smalest disc moves
    else
      for(i=n-1;p[i]==p[n];--i);
      //the smallest disc, that is on different peg than disc n, moves
    printf("%c -> ",p[i]); //move from peg p[i]
    p[i]+=1-((i&1)<<1);
    // according to the odd/even character of i,
    // the ,,increasing/decreasing" pattern is followed
    if(p[i]==c+1) p[i]=a;
    elseif(p[i]==a-1) p[i]=c; // circular pattern are followed
    printf(" %c\n",p[i]); //move to peg p[i]
    ++k;
  }
}
```

8 Analysis of programs

The Towers of Hanoi puzzle was analyzed in the literature in many ways, such [2], [3], [4], [5]. In this paper we solved the problem according to the principles of the four major programming techniques. Including the iterative solution we have got 5 programs (P_recursive_min – divide and conquer solution, P_greedy_min – greedy solution, P_memozation_min – dynamic programming, P_backtrack_min – backtracking solution, and P_iterative_min – the iterative solution).

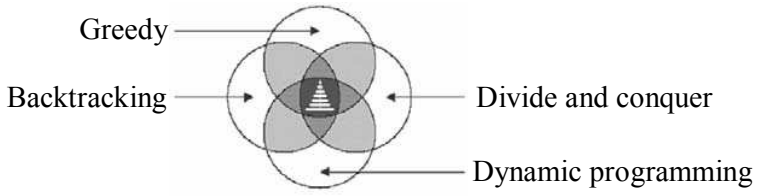


Figure 4: Towers of Hanoi – where programming techniques blend.

In the above mentioned programs, we used a time measuring sequence, as follows:

```
#include <windows.h>
__int64 freq, tStart, tStop;
unsigned long TimeDiff;
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
QueryPerformanceCounter((LARGE_INTEGER*)&tStart);
```

With this sequence we started the timer, then we called the program solving sequence. At the end, we calculate the time difference:

```
QueryPerformanceCounter((LARGE_INTEGER*)&tStop);
TimeDiff = (unsigned long)(((tStop − tStart) ∗ 1000000) / freq);
```

Using the 1 000 000 multiplier, we got the time in microsecond – a very precise chronometer. Because of the time sharing algorithm of the operating system, we did execute 20 times each program, and calculate an average of time-need.

We tested the programs for 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, and 20 discs. During the tests we disabled any I/O operation (ex. printf).

Table 1 concludes the tests, and the time-needs.

Technical information: We measured the time in microsecond. The computer: Intel Pentium 4; 2,40 GHz CPU; 1,50 GB RAM; the programs: Mi-

n	*steps* 2^n-1	*recursive*	*greedy*	*backtracking*	*dynamic*	*iterative*
1	1	1	1	5	1	1
2	3	1	1	15	1	1
3	7	1	1	143	1	1
4	15	2	2	29 022	2	2
5	31	3	3	1 791 167 126	2	2
6	63	5	5	,,∞”	2	2
7	127	8	8	,,∞”	3	3
8	255	14	15	,,∞”	4	4
9	511	27	30	,,∞”	6	6
10	1023	51	63	,,∞”	10	11
15	32 767	1599	2071	,,∞”	250	320
20	1 048 575	52 080	70 112	,,∞”	10 511	10 849

Table 1: Comparative analysis of programs.

crosoft Visual C++ 6.0, Win32 Console Application.

According to the comparative analysis, we can conclude that the dynamic programming and the iterative solutions are the fastest (aprox. 5-times faster than the recursive, divide and conquer solution), the classic backtracking is the slowest. There is a very small difference between the iterative solution and the recursive dynamic programming solution.

9 Conclusions

As we pointed out in this paper the principle solving process of Towers of Hanoi problem mainly follows a greedy strategy. Then again, the way the optimal solution is built mostly follows the dynamic programming “way of thinking”. The greedy algorithms produce only one decision-sequence, a sequence of greedy decisions. The dynamic programming strategies usually generate several optimal sub-sequences. The fact, that both the recursive and iterative algorithms generate only one decision-sequence is another reason why these solutions can be considered as greedy strategies implemented in dynamic programming way.

The recursive formulas are born in top-down greedy way, but they are implemented in bottom-up way as dynamic programming strategies. Furthermore, considering them in top-down / bottom-up way, they are materializa-

tions of the principal of greedy-chooses / optimality. Although the current sub-problem at each step is reduced to two/three sub-sub-problems (that is characteristic to divide and conquer strategies) the problem solving processes (both the principle and the implementation) are directed by greedy and dynamic programming optimalisations included in the recursive formulas. The fact that the problem solving process pretends the repeated "evaluation" of the overlapping sub-problems is also a divide and conquer feature of the problem.

With respect to the time complexity of the implementations, they are exponential, since the size of the optimal solutions depends exponentially on the size of the input. All algorithms traverse the same optimal-solution-tree with (2^n-1) or (3^n-1) nodes (expecting the backtracking algorithm that traverses the whole solution-space-tree). The running time differences between the recursive and iterative implementations can mainly be explained by the loss of time that arises due to the recursive calls. Whereas procedure P_recursive_min implements the bottom-up building process on the back-way of the recursion, procedure P_iterative_min performs this directly. Since the optimal solution has exponential size, even minor variances between implementations are reflected in exponential way with respect to the running time of the algorithms.

References

[1] N. Claus (pseudonym for Édouard Lucas), *La Tour d'Hanoï: V'eritable Cassetête Annamite*, Original instruction sheet printed by Paul Bousrez, Tours, 1883.

[2] Z. Kátai, *Algoritmusok felülnézetből*, Stientia Kiadó, Kolozsvár, 2006.

[3] D.T. Barnard, The Towers of Hanoi: An Exercise in Non-Recursive Algorithm Development, *Tech. Report 80-103*, Queen's University, 1980.

[4] J.P. Bode, A.M. Hinz, Results and open problems on the Tower of Hanoi, *Congr. Numer.*, **139** (1999) 113–122.

[5] P. Cull, E.F. Ecklund, Jr., Towers of Hanoi and analysis of algorithms, *Amer. Math. Monthly*, **92** (1985) 407–420.

[6] B. Eggers, The Towers of Hanoi: Yet another nonrecursive solution, *SIGPLAN Notices*, **20**, 9 (1985) 32–42.

[7] F.O. Ikpotokin, S.C. Chiemeke, E.O. Osaghae, Alternative approach to the optimal solution of tower of Hanoi, *J. Inst. Math. Comput. Sci.*, **15** (2004) 229–244.

[8] M. Sniedovich, OR/MS Games: 2. Towers of Hanoi Puzzle, *INFORMS Transactions on Education*, **3**, 1 (2002) 34–51.

[9] Hanoi tornyai, http://mattort.fvt.hu/, *Abacus Matematikai Lapok*, 2001-2002. .

[10] R.E. Bellman, *Dynamic Programming*, Princeton, New Jersey, 1957.

Received: November 18, 2008.

Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 109–118

Generating and ranking of Dyck words

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

Abstract. A new algorithm to generate all Dyck words is presented, which is used in ranking and unranking Dyck words. We emphasize the importance of using Dyck words in encoding objects related to Catalan numbers. As a consequence of formulas used in the ranking algorithm we can obtain a recursive formula for the nth Catalan number.

1 Introduction

Let $B = \{0, 1\}$ be a binary alphabet and $x_1x_2 \dots x_n \in B^n$. Let $h : B \to \{-1, 1\}$ be a valuation function with $h(0) = 1$, $h(1) = -1$, and $h(x_1x_2 \dots x_n) = \sum_{i=1}^{n} h(x_i)$.

A word $X = x_1x_2 \dots x_{2n} \in B^{2n}$ is called a *Dyck word* [4] if it satisfies the following conditions:

$$h(x_1x_2 \dots x_i) \geq 0, \text{ for } 1 \leq i \leq 2n-1$$
$$h(x_1x_2 \dots x_{2n}) = 0.$$

n is the semilength of the word.

AMS 2000 subject classifications: 68R05
CR Categories and Descriptors: G.2.1. [Combinatorics]: Subtopic - Combinatorial algorithms.
Key words and phrases: Dyck words, generating and ranking algorithms, Catalan numbers

2 Lexicographic order

The algorithm that generates all Dyck words in lexicographic order is obvious. Let us begin with 0 in the first position, and add 0 or 1 each time the Dyck-property remains valid. In the following algorithm $2n$ is the length of a Dyck word, n_0 counts the 0s, and n_1 the 1s.
There are the following cases :

Case 1: $(n_0 < n)$ and $(n_1 < n)$ and $(n_0 > n_1)$ (We can continue by adding 0 and 1.)
Case 2: $(n0 < n)$ and $(n_1 < n)$ and $(n_0 = n_1)$ (We can continue by adding 0 only.)
Case 3: $(n_0 < n)$ and $(n_1 = n)$ (We can continue by adding 0 only.)
Case 4: $(n_0 = n)$ and $(n_1 < n)$ (We can continue by adding 1 only.)
Case 5: $(n_0 = n_1 = n)$ (A Dyck word is obtained.)

Let us use the following short notations:

Dyck 0 for
$x_i \leftarrow 0$
$n_0 \leftarrow n_0 + 1$
LexDyckWords(X, i, n_0, n_1)
$n_0 \leftarrow n_0 - 1$

Dyck 1 for
$x_i \leftarrow 1$
$n_1 \leftarrow n_1 + 1$
LexDyckWords(X, i, n_0, n_1)
$n_1 \leftarrow n_1 - 1$

The algorithm is the following:

LexDyckWords(X, i, n_0, n_1)
1 **if** Case 1
2 **then** $i \leftarrow i + 1$
3 *Dyck 0*
4 *Dyck 1*
5 **if** Case 2 **or** Case3
6 **then** $i \leftarrow i + 1$
7 *Dyck 0*
8 **if** Case 4
9 **then** $i \leftarrow i + 1$
10 *Dyck 1*
11 **if** Case 5
12 **then** Visit $x_1 x_2 \dots x_n$
13 **return**

The recursive call:
$x_1 \leftarrow 0$, $n_0 \leftarrow 1$, $n_1 \leftarrow 0$
LexDyckWords$(X, 1, n_0, n_1)$

For $n = 4$ we obtain:

00001111, 00010111, 00011011, 00011101, 00100111, 00101011, 00101101, 00110011, 00110101, 01000111, 01001011, 01001101, 01010011, 01010101.

This algorithm obviously generates all Dyck words.

3 Generating the positions of 1s

Let $b_1 b_2 \dots b_n$ be the positions of 1s in the Dyck word $x_1 x_2 \dots x_{2n}$. E.g. for $x_1 x_2 \dots x_8 = 01010011$ we have $b_1 b_2 b_3 b_4 = 2478$.

To be a Dyck word of semilength n, the positions $b_1 b_2 \dots b_n$ of 1s of the word $x_1 x_2 \dots x_{2n}$ must satisfy the following conditions:

$$2i \le b_i \le n + i, \quad \text{for} \quad 1 \le i \le n.$$

Following the idea of generating combinations by positions of 0s in the corresponding binary string [5] we propose a similar algorithm that generates the positions $b_1 b_2 \dots b_n$ of 1s.

PosDyckWords(n)

```
 1  for i ← 1 to n
 2      do b_i ← 2i
 3  repeat
 4      Visit b_1 b_2 ... b_n
 5      IND ← 0
 6      for i ← n − 1 downto 1
 7          do if b_i < n + i
 8                then b_i ← b_i + 1
 9                     for j ← i + 1 to n − 1
10                         do b_j ← max(b_{j−1} + 1, 2j)
11                     IND ← 1
12                     break (for)
13  until IND = 0
14  return
```

For $n = 4$ we obtain:

2468, 2478, 2568, 2578, 2678, 3468, 3478, 3568, 3578, 3678, 4568, 4578, 4678, 5678.

The corresponding Dyck words are:

01010101, 01010011, 01001101, 01001011, 01000111, 00110101, 00110011, 00101101, 00101011, 00100111, 00011101, 00011011, 00010111, 00001111.

Because all values of positions that are possible are taken by the algorithm, it generates all Dyck words. Words are generated in reverse lexicographic order.

4 Generating by changing 10 in 01

The basic idea [2] is to change the first occurence of 10 in 01 to get a new Dyck word. We begin with $0101\ldots01$.

```
DyckWords(X, k)
 1  i ← k
 2  while i < 2n
 3        do Let j be the position of the first occurence of 10 in x_i x_{i+1} ... x_{2n},
             or 0 if such a position doesn't exist.
 4           if j > 0
 5              then Let Y ← X
 6                   Change y_i with y_{i+1}.
 7                   Visit y_1 y_2 ... y_{2n}
 8                   DyckWords(Y, j − 1)
 9                   i ← j + 2
10  return
```

The first call is DyckWords$(X, 1)$, if $X = 0101\ldots01$.

For $X = 01010101$, the algoritm generates:

01010101, 00110101, 00101101, 00011101, 00011011, 00010111, 00001111, 00101011, 00100111, 00110011, 01001101, 01001011, 01000111, 01010011.

Can this algorithm always generate all Dyck words? To prove this we show that any Dyck word can be tranformed to $(01)^n$ by several changing of 01 in 10. Let us consider the leftmost subword of the form 0^i1, for $i > 0$. Changing 01 in 10 $(i-1)$ times, we will obtain a leftmost subword of the form $0^{i-1}1$. So, all subwords of this form can be avoided.

5 Ranking Dyck words

Ranking Dyck words means [6] to determine the position of a Dyck word in a given ordered sequence of all Dyck words.

Algorithm PosDyckWords generates all Dyck word in reverse lexicographic order. For ranking these words we will use the following function [7], where $f(i,j)$ represents the number of paths between (0,0) and (i,j) not crossing the diagonal $x = y$ of the grid.

$$f(i,j) = \begin{cases} 1, & \text{for } 0 \le i \le n, j = 0 \\ f(i-1,j) + f(i,j-1), & \text{for } 1 \le j < i \le n \\ f(i,i-1), & \text{for } 1 \le i = j \le n \\ 0, & \text{for } 0 \le i < j \le n \end{cases} \tag{1}$$

Some values of this function are given in the following table.

j											
9										4862	
8									1430	4862	
7								429	1430	3432	
6							132	429	1001	2002	
5						42	132	297	572	1001	
4					14	42	90	165	275	429	
3				5	14	28	48	75	110	154	
2			2	5	9	14	20	27	35	44	
1		1	2	3	4	5	6	7	8	9	
0	1	1	1	1	1	1	1	1	1	1	
	0	1	2	3	4	5	6	7	8	9	i

It is easy to prove that if C_n is the nth Catalan number then

$$C_{n+1} = f(n+1,n) = \sum_{i=0}^{n} f(n,i), \quad n \ge 0 \tag{2}$$

$$f(n+1,k) = \sum_{i=0}^{k} f(n,i), \quad n \ge 0, n \ge k \ge 0.$$

Using this function the following ranking algorithm results.

```
RANKING(b_1 b_2 ... b_n)
1  c_1 ← 2
2  for j ← 2 to n
3       do c_j ← max(b_{j-1} + 1, 2j)
4  nr ← 1
5  for i ← 1 to n − 1
6       do for j ← c_i to b_i − 1
7             do nr ← nr + f(n − i, n + i − j)
8  return nr
```

For example, if b =4 5 8 9 10, we get c =2 5 6 9 10, and $nr = 1 + f(4,4) + f(4,3) + f(2,2) + f(2,1) = 1 + 14 + 14 + 2 + 2 = 33$.
This algorithm can be used for ranking in lexicographic order too.

6 Unranking Dyck words

The unranking algorithm for a given n will map a number between 1 and C_n to the corresponding Dyck word represented by positions of 1s. Here the Dyck words are considered in reverse lexicographic order too.

```
UNRANKING(nr)
1  b_0 ← 0
2  nr ← nr − 1
3  for i ← 1 to n
4       do b_i ← max(b_{i-1} + 1, 2i)
5          j ← n + i − b_i
6          while (nr ≥ f(n − i, j)) and (b_i < n + i)
7                do nr ← nr − f(n − i, j)
8                   b_i ← b_i + 1
9                   j ← j − 1
10 return b_1 b_2 ... b_n
```

If $n = 6$ and $nr = 93$, we will have: $92 - f(5,5) - f(5,4) - f(3,3) - f(2,2) - f(1,1) = 92 - 42 - 42 - 5 - 2 - 1$, so the corresponding Dyck word represented by positions of 1's is: b = 4 5 7 9 11 12. Are changed from the initial values 2i the following: position 1 by 2, position 3 by 1, position 4 by 1 and position 5 by 1.

7 Applications of Dyck words

If $\mathcal{O}$ is a set of C_n objects, Dyck words can be used for encoding the objects of $\mathcal{O}$. The importance of such an encoding currently is not suitably accentuated. We present here an encoding and decoding algorithms for binary trees, based on [1].

Algorithm for encoding a binary tree
Let B_L be the left and B_R the right subtree of the binary tree B. $w01$ means the concatenation of word w with 01, and w is considered a global variable.

```
EncodingBT(B)
 1  if B_L ≠ ∅ and B_R = ∅
 2     then w ← w01
 3          EncodingBT(B_L)
 4  if B_L = ∅ and B_R ≠ ∅
 5     then w ← w10
 6          EncodingBT(B_R)
 7  if B_L ≠ ∅ and B_R ≠ ∅
 8     then w ← w00
 9          EncodingBT(B_L)
 8          w ← w11
 9          EncodingBT (B_R)
10  return
```

Call:

```
w ← 0
EncodingBT(B)
w ← w1
```

For all trees of $n = 4$ vertices the result of the algorithm is given in Fig. 1.

Algorithm to decode a Dyck word into a binary tree
At the beginning the root of the generated binary tree is the current vertex. When an edge is drawn, its endvertex becomes the current vertex.

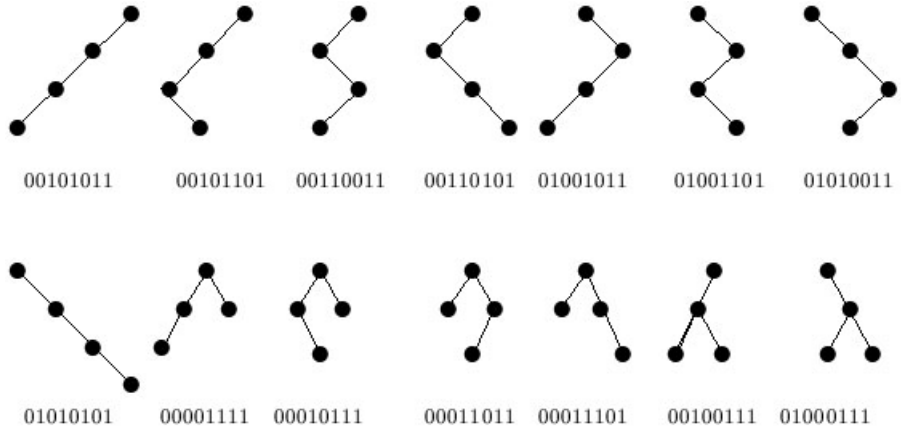


Figure 1: Encoding of binary trees for $\mathtt{n} = 4$.

DecodingBT(w)

```
1  Let ab be the first two letters of w.
2  Delete ab from w.
3  if ab = 01
4     then draw a left edge from the current vertex
5          DecodingBT(w)
6  if ab = 10
7     then draw a right edge from the current vertex
8          DecodingBT(w)
9  if ab = 00
10    then put in the stack the position of the current vertex
11         draw a left edge from the current vertex
12         DecodingBT(w)
13 if ab = 11
14    then get from the stack the position of the new current vertex
15         draw a right edge from the current vertex
16         DecodingBT(w)
17 return
```

Call:

delete 0 from the beginning and 1 from the end of the input word w
draw a vertex (the root of the tree) as current vertex
DecodingBT(w)

For some other objects related to Catalan numbers the corresponding coding can be found in [1] and at http://www.ms.sapientia.ro/˜kasa/CodingDyck.pdf.

8 A consequence

As a consequence of formulas (1) and (2) the following formula for the $(n+1)$th Catalan number results:

$$C_{n+1} = 1 + \sum_{k\geq 0} (-1)^k \binom{n-k}{k+1} C_{n-k}. \tag{3}$$

We can prove that

$$f(n, n-k) = \sum_{i=0}^{n} (-1)^i \binom{k-i}{i} C_{n-i}$$

for appropriate n and k, using mathematical induction on n and k, and formula (1) in the form

$$f(n, n-k) = f(n, n-k+1) - f(n-1, n-k+1).$$

Now, from (2)

$$\begin{aligned} C_{n+1} &= \sum_{i=0}^{n} f(n,i) = f(n,0) + \sum_{i=1}^{n} f(n,i) = 1 + \sum_{i=0}^{n-1} f(n, n-i) \\ &= 1 + \sum_{i=0}^{n-1} \left(\sum_{k=0}^{n} (-1)^k \binom{i-k}{k} C_{n-k} \right) \\ &= 1 + \sum_{k=0}^{n} (-1)^k C_{n-k} \left(\sum_{i=0}^{n-1} \binom{i-k}{k} \right) \\ &= 1 + \sum_{k=0}^{n} (-1)^k \binom{n-k}{k+1} C_{n-k}. \end{aligned}$$

In the last line $\binom{k}{k} + \binom{k+1}{k} + \cdots + \binom{n-1-k}{k} = \binom{n-k}{k+1}$ was used.

References

[1] A. Bege, Z. Kása, Coding objects related to Catalan numbers, *Studia Univ. Babeş-Bolyai Infor.* **46**, 1 (2001) 31–40.

[2] A. Bege, Z. Kása, *Algoritmikus kombinatorika és számelmélet*, Presa Universitară Clujeană, 2006.

[3] E. Deutsch, Dyck path enumeration, *Discrete Math.* **204**, 1–3 (1999) 167–202.

[4] P. Duchon, On the enumeration and generation of generalized Dyck words, *Discrete Math.* **225**, 1–3 (2000) 121–135.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 4. Fasc. 3, Generating All Combinations and Partitions*, Addison-Wesley Reading MA., 2005.

[6] J. Liebehenschel, Ranking and unranking of lexicographically ordered words: An average-case analysis. *J. Autom. Lang. Comb.*, **2**, 4 (1997) 227–268

[7] W. Yang, *Discrete Mathematics*, http://www.cis.nctu.edu.tw/˜wuuyang/, manuscript.

Received: November 18, 2008

Acta Universitatis Sapientiae

The scientific journal of the Sapientia University publishes original papers and deep surveys in several areas of sciences written in English.
Information about the appropriate series can be found at the Internet address
http://www.acta.sapientia.ro.

Acta Universitatis Sapientiae, Informatica

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

This 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 deep surveys in all field of Computer Science. All papers will be 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 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, AMS Subject Classification codes (`http://www.ams.org/msc/`) and CR Categories and Description codes (`http://oldwww.acm.org/class/1998`). References should be listed alphabetically using the following examples:

For papers in journals:

A. Hajnal, V. T. Sós, Paul Erdős is seventy, *J. Graph Theory*, **7**, 4 (1983) 391–393.

For books:

D. Stanton, D. White, *Constructive Combinatorics*, Springer, New York, 1986.

For papers in contributed volumes:

Z. Csörnyei, Compilers in *Algorithms of Informatics, Vol. 1. Foundations* (ed. A. Iványi), mondAt Kiadó, Budapest, 2007. pp. 80–130.

For internet sources:

E. Ferrand, An Analogue of the Thue-Morse Sequence, *Electron. J. Comb.* **14** (2007) #R30, http://www.combinatorics.org/

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

Authors are encouraged to submit papers not exceeding 15 pages, but no more than 10 pages are preferable.

Each author is entitled to an issue containing his paper free of charge. No reprints will be available.

Publication supported by