Acta Universitatis Sapientiae

Informatica

Volume 2, Number 2, 2010

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **2**, 2 (2010) 117–134

Communication model of emuStudio emulation platform

Peter Jakubčo
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
email: peter.jakubco@tuke.sk

Slavomír Šimoňák
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
email: slavomir.simonak@tuke.sk

Norbert Ádám
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
email: norbert.adam@tuke.sk

Abstract. Within the paper a description of communication model of plug-in based emuStudio emulation platform is given. The platform mentioned above allows the emulation of whole computer systems, configurable to the level of its components, represented by the plug-in modules of the platform. Development tasks still are in progress at the home institution of the authors. Currently the platform is exploited for teaching purposes within subjects aimed at machine-oriented languages and computer architectures. Versatility of the platform, given by its plug-in based architecture is a big advantage, when used as a teaching support tool. The paper briefly describes the emuStudio platform at its introductory part and then the mechanisms of inter-module communication are described.

1 Introduction

Emulation is an imitation of internal structure of the system, by which we simulate its behavior or functionality. An emulator can be implemented in

Computing Classification System 1998: D.0
Mathematics Subject Classification 2010: 68N01
Key words and phrases: emulation, plug-ins, communication

a form of software, which emulates computer hardware its architecture and functionality as well.

Development of a fully-fledged emulator is connected with many areas of computer science, like a theory of compilers (needed mainly at instructions decoding in emulated processor), theory of emulation (includes different algorithms of emulation, methods of abstraction of real hardware into its software model), programming languages and programming techniques (required for performance improvements) and obviously detailed knowledge of emulated hardware.

The goal of our effort connected with the emuStudio was an emulation platform able to emulate different computers, which are similar by their structure. Such a tool was intended to be a valuable utility supporting the teaching process in areas like machine-oriented languages and computer architectures, so simplicity and configurability were the properties also considered. Thanks to the universal model of plug-in communication, we were able to create emulators of different computers like a real MITS Altair8800 [7] in two variations, or an abstract RAM machine [6], and others. More information on this topic can be found in [8, 4].

Not too much universal emulators are available nowadays. Partial success with standardizing emulated hardware was achieved within a project M.A.M.E. [2], which is oriented toward preserving historical games for video game consoles. In 2009 a medium scale research project co-financed by the European Union's Seventh Framework Programme started with the aim to develop an Emulation Access Platform (KEEP) [3]. As far as we know, the ability provided by the emuStudio platform to choose the configuration of emulated system by the user dynamically is unique.

2 Architectures for emulation

Computer architecture is a system characteristic, which unifies the function and the structure of its components [5]. Most widely known architectures are Harvard architecture [9] and Princeton architecture (also known as von Neumann architecture) [1], which is the core of most modern computers.

Versatility of the platform is oriented towards a liberty in choosing the configuration, rather than architecture of emulated computer. Configuration choice is given by the selection of components (plug-in modules) and their connections (a way of communication).

As a basic architecture for emulation, the von Neumann architecture type

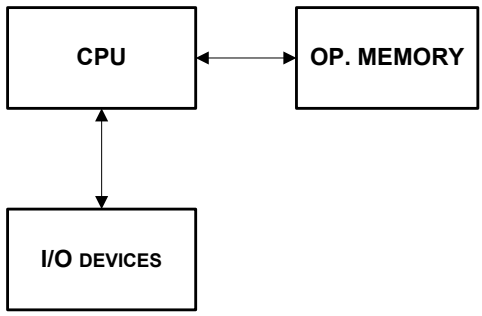


Figure 1: Computer architecture of von Neumann type

was chosen (Figure 1). Communication model (methods, protocol) thus was adapted for this type of architecture.

The core component of the architecture is a processor (CPU), which executes instructions, communicates with main memory, from which it fetches instructions to execute. Main memory, except the instructions mentioned, also stores data. CPU also communicates with peripheral devices. An extension of the emulator, compared to a basic von Neumann concept is that peripheral devices are allowed to communicate each other without the interaction of CPU and also with main memory.

The selection of computer configuration to emulate is left to the user. Required configuration thus can be composed by picking and connecting available components by the user. From the configuration composed a platform creates an instance, when the one is selected at the startup. By this selection, the virtual architecture arises, ready for emulation.

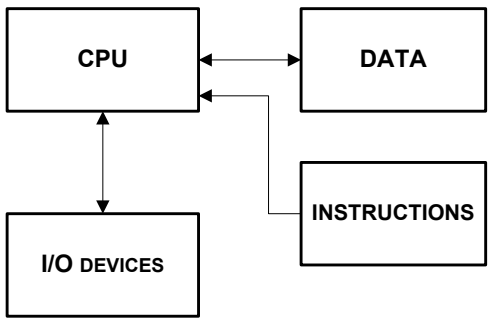


Figure 2: Computer architecture of Harvard type

The fact that the communication model is adapted for configurations of von Neumann type, does not guarantee any support for creating architectures of different type, but also it does not exclude it. As an example can serve the implementation of RAM machine emulator at our platform, which in fact uses the architecture of Harvard type.

Main difference between the two architectures mentioned is, that computers with Harvard architecture (Figure 2) use two types of memory first for storing instructions, second for storing data.

3 The structure of the platform

Basic components of a computer from any of architecture types mentioned above can be subdivided into three types:

- Processor (CPU),
- Operating memory,
- Input/output peripheral devices.

Particular components of real computers are interconnected by communication elements, like buses. Except that, components make use a number of support circuits, performing auxiliary functions.

When abstractions of real computers are considered (what emulators surely are), such elements usually are omitted, because they are not essential for proper functionality at given level of abstraction. Although emulation of buses would take us closer to real computer structure, the communication of components emulated would be non-effective (bus as a useless component in between other components) and can introduce possible difficulties (e.g. when data of greater size than the bus width are to be transferred). That's why buses are not used and components emulated use different way of communication (e.g. direct call of components' operations). When identifying, what would be and what would be not implemented in an emulator, it is necessary to take into account the emulator properties required. When, for example, the communication path tracking is required, then emulation of buses and communication elements is necessary.

The structure of emuStudio platform is depicted in Figure 3. Within the scheme, the arrow direction has the following meaning: let's have two objects O_1 and O_2 from the scheme. When the arrow points from O_1 to O_2, ($O_1 \rightarrow O_2$), then object O_1 is allowed to call operations of object O_2, but not in the

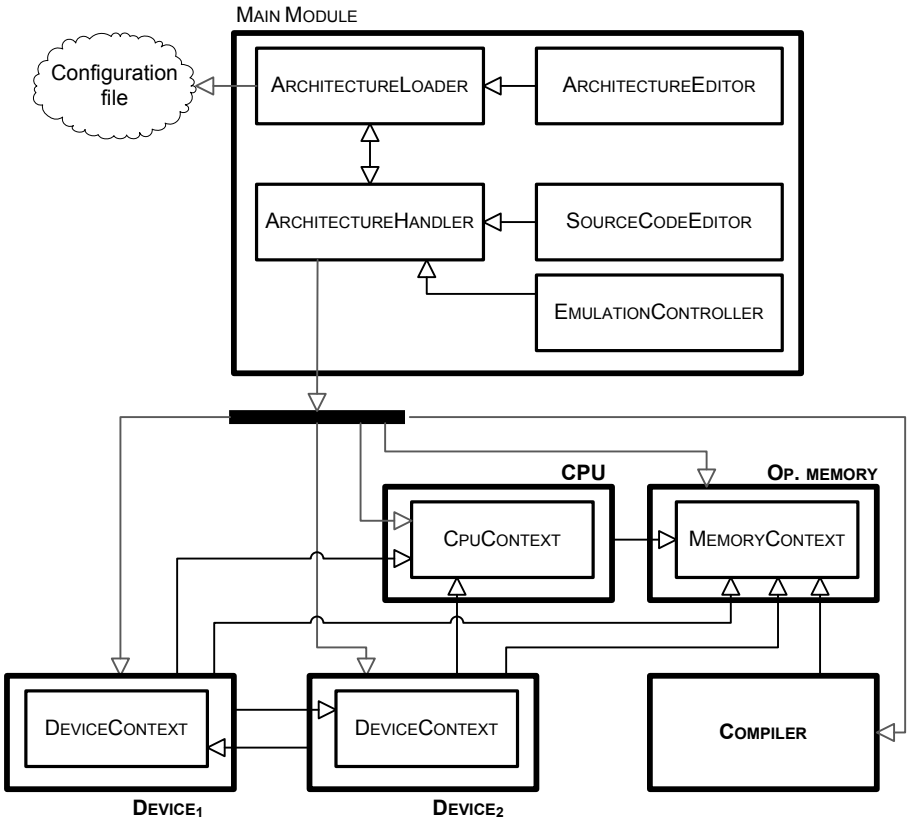


Figure 3: The structure of emuStudio platform

opposite direction (O_2 only can return the result of an operation). According to the scheme, four types of plug-in modules exist in the platform:

- Processors (CPU),
- Memories,
- Input/output peripheral devices,
- Compilers.

The emuStudio platform is implemented using the Java programming language, so one of advantages is the portability at a machine-code (byte-code) level. On the other side, Java programs itself are emulated too, by the Java virtual machine, so the emulator performance is decreased by this choice.

3.1 Context

As it can be seen in Figure 3, plug-in modules, except the compiler, contain a special component called context. Lines connecting modules, start from the edge of the module (e.g. DEVICE2), but point to the context of another module (CPUCONTEXT). It means that plug-in module, which requests another one, has no full access to this module. It can access the context of this module only.

Reasons for creating a component context are three: *safety*, *functionality encapsulation* and allowing for *non-standard functionality* of plug-in modules.

Plug-in module within the emuStudio platform is safe, if its functionality cannot be abused. We mean by this the functionality of module itself (e.g. unwanted change of internal parameters of the module, defined by the user), but also the functionality of the platform as a whole (e.g. controlling the emulation, terminating the application, or unwanted "dynamic" changes in virtual configuration).

For that reason the main module is the only one that has an access to all plug-in operations, and plug-ins consider the main module to be trusted.

Besides, communication model and API (*Application programming interface*) of plug-ins are open and free, so practically the plug-ins can be designed by anyone, by what the credibility of plug-in decreases. The safe functionality therefore should be separated, what has implied to context creation.

On the other hand, it is also not good idea if the plug-ins allow to use a functionality by another plug-ins that the other side doesn't need. The transparency fades out and there again arises the risk of improper use of the operations. The encapsulation principle used in Object oriented programming paradigm therefore claims to hide such operations, what is solved by the use of the context, too.

It is enough if the context will define only the standard functionality (in the form of communication operations) for plug-ins of equal types. However a situation can arise there, wherein this communication standard doesn't have to universally cover all the requirements of each concrete plug-in.

The context is therefore an ideal environment, where such non-standard functionality can be implemented. The fact that the context is realized inside a plug-in, enables to add operations that are not included in the standard context, into the plug-in implementation. Plug-ins that use the non-standard functionality have to know the form of a non-standard context - otherwise they cannot use the non-standard functionality.

3.2 Main module

The core of the platform is the main module. It consists of several components:

ArchitectureLoader - the configuration manager. It manages configuration file (stores and loads defined computer configurations), and creates an instance of virtual configuration (through ArchitectureHandler component).

ArchitectureEditor - configuration editor. The user uses this component to choose and connect components of defined computer architecture. This selection and connection is realized in a visual way by drawing of abstract schemas. The component allows creating, editing and deleting the abstract schemas, and it cooperates with ArchitectureLoader component.

ArchitectureHandler - the virtual architecture instance manager. It offers plug-ins instances to other components (other plug-ins) and implements an interface for storing/loading of plug-ins' settings.

SourceCodeEditor - the source code editor. It allows creating and editing the source code for chosen compiler, it supports syntax highlighting, rows labeling and directly communicates with the compiler (through ArchitectureHandler component).

EmulationController - the emulation manager. It controls the whole emulation process, and it stands in the middle of the interaction between virtual architecture and the user.

3.3 Compiler

The compiler plug-in represents a translator of source code into a machine code for concrete CPU. The structure of compiler's language is not limited at all, therefore the language doesn't have to be an assembler.

The compiler is chosen by the user in the configuration design process of emulated computer (such as other components are). The logical is a choice of compiler that compiles the source code into machine code for a processor chosen.

The output of the compiler should be a file with a machine code and optionally the output is redirected into operating memory, too. It depends on a concrete compiler, how the output will be realized.

3.4 CPU

Central processing unit (CPU) is a component of digital computer that interprets instructions of computer program and process data. CPU provides fundamental computer property of programmability, and it is one of the most significant components found in computers of each era, together with operating memory and I/O devices.

CPU plug-in represents a virtual processor. It is a base for whole emulation, because the control of emulation run in the main module actually means the control of processor run. The main activity of a CPU is the instruction execution. These instructions can be emulated by arbitrary emulation technique [8] (depending on implementation of a concrete plug-in).

The plug-in contains a special component called *processor context*, operations of what are specific for concrete CPU (besides the standard operations, there can be specified more operations by the programmer). Devices that need to have an access to the CPU get only its context available. Therefore the context for devices represents a "sandbox" that prohibits interfering with sensible settings and the control of processor's run. If such a device has to be connected with CPU, the context should contain operations that allow device connections.

3.5 Operating memory

The operating memory (OP) represents a virtual main store (storage space for data and instructions). Generally an OP consists from cells, where format, type, size and value of cells are not closely defined. Cells are placed sequentially, therefore it is possible to determine unique location of any cell in the memory (the location is called an *address*).

OP contains a component called *memory context* that besides the standard operations (reading from and writing to memory cells) can include specific operations (e.g. support of segmentation, paging and other techniques), too, of a concrete plug-in.

Devices (and a compiler) that need to have an access to OP (e.g. devices that use direct access into memory), get only this memory context. Devices get the context in a virtual architecture initialization process, and compiler (when needs to write compiled code directly into memory) when calling the *compile* method. Therefore operations in the context have to be safe (from usability's point of view) for other plug-ins.

3.6 Peripheral devices

Peripheral devices are virtual devices that emulate functionality of real devices. Generally the purpose of the devices is not closely defined, nor standardized, so plug-ins do not really need represent real devices.

The main idea of device communication is that all information within the communication process go into the device though its input(s) and go out from the device as one or more outputs. The devices then can be input, output, or input/output.

In some detail (that detail is not limited), the devices can work independently (and reacts to events of connected plug-ins), eventually interacts with the user. Devices can communicate with CPU, and/or OP, and/or other devices.

The communication model supports hierarchical device connections (the devices are therefore enabled to communicate to each other without the CPU attention/support).

Every device (following the Figure 3) contains one or more components, called *device context*. The device context can be extended by a concrete plug-in with non-standard operations. Other devices that need an access to this device get one or more contexts of the device (that mechanism ensures that one device can be connected to more devices). The operations available in the context have to be safe (from usability's point of view) for other plug-ins.

4 Communication realization in emuStudio platform

As could be seen, plug-in contexts solve some concrete problems of communication module. In this section, communication model will be described in more detail, and a way how the communication is realized between the main module and plug-ins. The communication model represents a collection of standardized methods, and by calling of them individual sides will communicate with each other.

Let's consider two objects that want to communicate with each other. In the case when communication sides are independent and separated systems (one side cannot directly call the other side), it is necessary to design a communication protocol and realization mechanism of the communication (e.g. medium) that are somewhat "bridge above the communication gap" (e.g. network) between the objects (Figure 4).

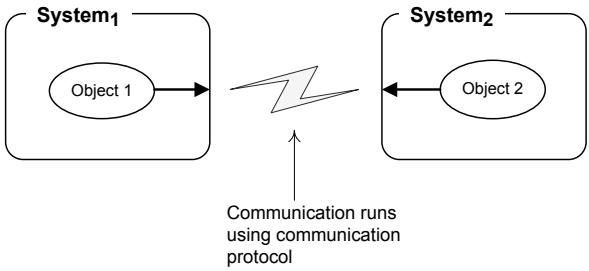


Figure 4: Communication realization between two independent and separated objects

The other case arises if the first object can directly access to the second object. The communication in this case will run directly, i.e. objects will directly call operations of the other objects. If the objects are separated and independent, two questions can arise.

At first, how the objects get access to other objects? The solution is to use another, third system, that will cover both subsystems (where the objects reside), or if you like will have direct access to both communicating objects, and the system will *provide* these objects each to another. Just in that way the communication in emuStudio platform works (Figure 5).

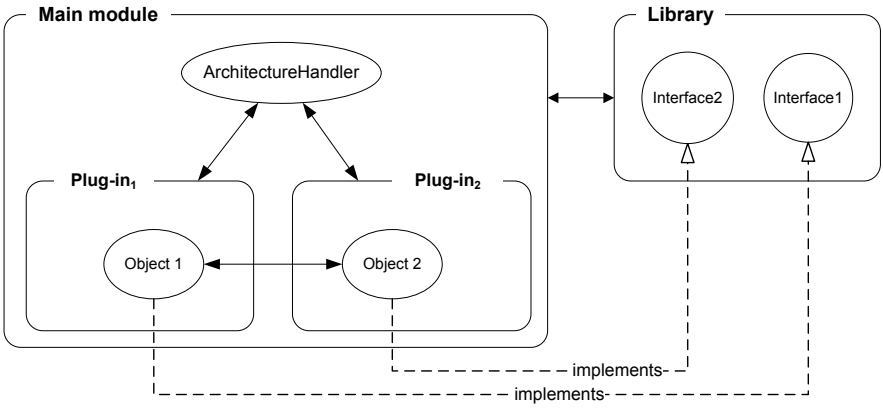


Figure 5: Communication realization in emuStudio platform

The second question is, how can the objects communicate together, that truly are of the same type, but belong to different systems with different implementations?

The solution is to create a standard model of operations that all the objects of given type will implement and it will be well-known to all objects. This model has to unify both the syntax and semantics of communication operations.

The main module represents a system of higher level, covering subsystems plug-ins. The objects of plug-ins the main module will get in virtual architecture instance creation process.

Communication operations are well-known both by the main module and by plug-ins. This is ensured by the fact that *prototypes* of the operations lies in external library, where the access is granted to both the main module and plug-ins. The operations are *ordered* according to the plug-in *type* into interfaces (an *interface* is defined as a structure containing a list of operations without their implementation). Each plug-in type has its own set of interfaces that corresponding plug-in has to implement.

4.1 External library structure

Figure 6 shows the structure of the external library, and contains all the prototypes (interfaces) for plug-ins.

Besides the packages and interfaces intended for the plug-ins to use, it contains a class called `runtime.StaticDialogs`, too. The class contains static methods that should disburden the plug-ins from common, fundamental and very often used methods implementation.

4.2 Standard operations – Compiler

Every compiler generally consists of these parts:

- Lexical analyzer,
- Syntactic analyzer (parser) that builds abstract syntactic tree,
- Semantic analyzer that verifies types usage and other semantic information,
- Code generator that generates a machine code using abstract syntactic tree.

Only some of these parts are important for interaction with the main module and plug-ins. Lexical and syntactic analyzer need to have the access to the source code itself. Semantic analyzer can work with knowledge gained from the phase of syntactic analysis (abstract syntactic tree), it means that

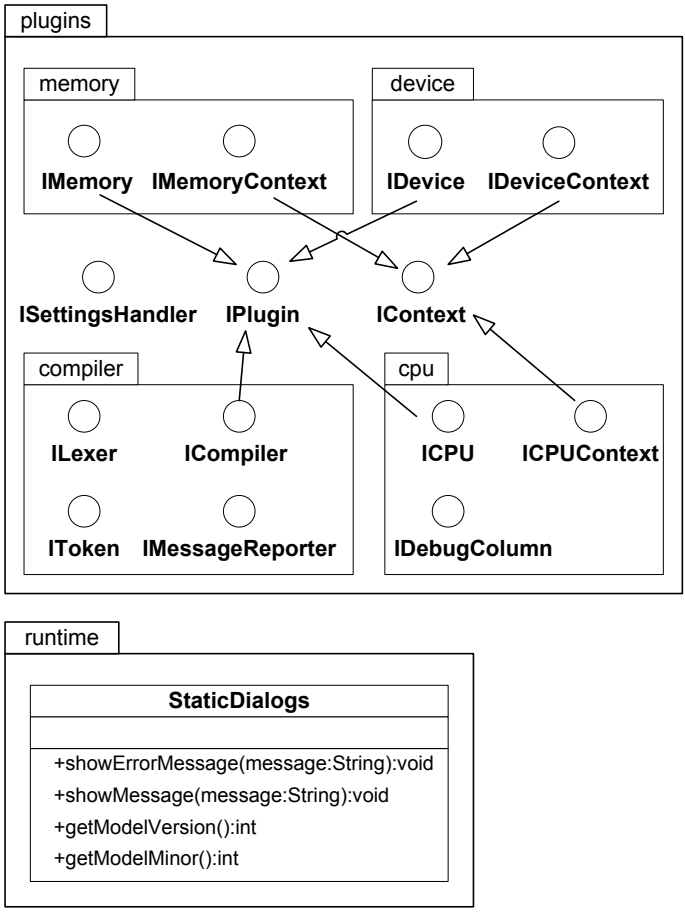


Figure 6: Library structure

semantic analyzer won't be in direct interaction with main module or other plug-ins. Therefore it is possible to skip all considerations of assigning it into a communication model.

Machine code generator can have an access to operating memory, too if the user asks to redirect the compiler output into operating memory. On the other hand, the main module needs to have an access to lexical analyzer, in order to make possible to use syntax highlighting in source code editor. Finally, the main module needs to call the compile operation itself.

In Table 1 basic standard operations are described that are important from the communication point of view.

Operation	**Description**
`Compile`	Source code compiling
`GetLexer`	Gets an lexical analyzer object
`GetStartAddress`	Gets absolute starting address of compiled program. The address can be later used as starting address for the program counter after the CPU RESET signal.

Table 1: Standard compiler operations

4.3 Standard CPU operations

Processor, or if you like the CPU, is a core of the architecture. It realizes the execution of the whole emulation, because its main activity is instruction execution. It also interacts with peripheral devices and with operating memory. Communication model does not limit the usage of the emulation technique for the processor emulation.

The CPU plug-in in the emuStudio platform besides the emulation itself, it has to co-operate with the user by the interaction using debugger and status windows (however operations related to the interaction will not be described here). In the status window the CPU should show the values of its registers, flags, actual CPU's running state and eventually other characteristics. The plug-in includes complete status window implementation so with different CPU the content of the status window will change accordingly.

Generally each CPU plug-in consists of following parts:

- The processor emulation implementation,
- Processor context that extends its functionality,
- Instruction disassembler,
- Status window GUI.

The CPU plug-in design demands the programmer to know the hardware that he is going to implement and to "answer the questions correctly" when the interface methods implementation are considered.

The work-flow cycle of each processor plug-in for the emuStudio platform is shown in Figure 7. As it can be seen from the figure, the processor can be found in one of four states. The RESET state is that state in which the

processor re-initializes itself and immediately after finishing that it sets itself to the BREAKPOINT state.

For the processor execution only the three states are meaningful:

- BREAKPOINT - the processor is temporally inactive (paused)
- RUNNING - the processor is running (executing instructions)
- STOPPED - the processor is stopped (waits for RESET signal)

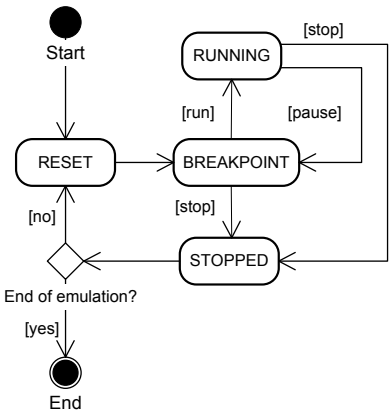


Figure 7: Processor work-flow cycle

The Table 2 describes basic operations to control the processor execution in the communication model. These operations tell how the CPU behavior can be influenced. However all of the operations are not supported in the real CPU's world and by contrast there definitely exist some CPU control operations that are not covered by the communication model. But mostly such operations are not common for all CPUs; therefore their support is optional within the scope of CPU context.

4.4 Standard operations – Operating memory

Operating memory (OP) is not a computer component that directly affects other computer components. It means that the memory is not "demanding" for services it is not acting like a communication initiator with the CPU, nor with the other devices (according to von Neumann conception). This fact is covered by the communication model all connections with the OP are one-directional, and the OP is always plugged *into* the device (or into a processor), and not in the other way. It means that the device (or processor) can use

Operation	**Description**
`Reset`	Re-initialization. The operation sets the CPU into a state in which it should be right after CPU launch (RESET in Figure 7).
`Step`	Emulation step. The CPU executes one instruction and then returns to the BREAKPOINT state.
`Stop`	The operation stops running or paused emulation. The CPU leads itself into a state in which it is not able to execute instructions anymore, till its reset (STOPPED in Figure 7).
`Pause`	Block/pause running emulation. The CPU leads itself to a state in which it stops to execute instructions, but its state (register values, flags and other settings) is unchanged after the last executed instruction (BREAKPOINT in Figure 7). From this state the CPU can be launched again.
`Execute`	The operation launches paused/blocked emulation. The CPU leads itself to a state in which permanently executes instructions (RUNNING in Figure 7). The stop of the CPU in this state can be activated by the user, otherwise the CPU stops spontaneously (e.g. after the execution of halt instruction).

Table 2: Some of the standard CPU operations

services of OP, but the OP cannot use services of the device the OP doesn't need to have an access to any plug-in.

The programmers can use the memory context also for the implementation of methods that allow attaching devices into operating memory. Such type of connection can be useful, if a device needs to be informed of the OP changed status (e.g. DMA technology).

Each OP implementation has to include a graphical user interface (GUI) so each memory should provide a graphical view to its content for a user (the content is represented by the values of its cells) and eventually to provide another manipulation with memory (e.g. address or value searching, memory content export into a file, etc.). Executed processor instructions description and the graphical view of memory content are basic interaction resources that the user has a contact with.

Summarizing previous paragraphs there can be named all components that each OP plug-in must contain:

- Memory context,

- Implementation of main interface - the memory functionality itself,
- Graphical user interface (GUI).

Basic operations that have to be implemented in each operating memory are described in Table 3.

Operation	Description
`Read`	Reading from operating memory - either one or more cells at once starting from given address.
`Write`	Writing into operating memory - either one or more cells at once starting from given address.
`ShowGUI`	The operation shows graphical user interface (GUI) of memory content.

Table 3: Some of the operating memory standard operations

4.5 Standard operations - Peripheral devices

There are known input, output and input-output devices. Their category can be identified easily according to a way how they are connected with other components of the configuration and to the direction of the connection (direction of data flow).

It is possible to implement virtual devices that communicate with real devices, but also fictive and abstract devices can be implemented. The device can interact with the user through its own graphical interface (GUI). Not all devices have to have GUI, but on the other hand there are such devices that their input and/or output are realized using the user interaction (e.g. terminals, displays). The devices can communicate with CPU, OP and with other devices, too.

A single device can be connected multiple times with other components. For this reason the devices can have several contexts, with possible different implementations. For example a serial card can have several physical ports, into which it is possible to plug in various devices (into each port can be plugged a single device, and each port is represented by a single context).

Communication model solves the following problems:

- How to connect devices to each other,
- How to realize input/output.

The basic idea of interconnection of two devices in the meaning of implementation is their contexts exchange with each other. All input/output operations that the devices will use for the communication resides in the context. In such a way the bidirectional connection is realized. Each device contains an operation intended for attaching of another device (op. `attachDevice`), that as a parameter takes the context of connecting device. This connection operation does not reside in the context, in order to ensure that the plug-ins couldn't change the structure of the virtual architecture. The connection job itself does the main module that performs the interconnection only in the virtual architecture creation process.

The input and output operations (`in` and `out`) reside in the device context, because by calling them the communication is performed. Transferred data type in these operations is not specified in the communication model, but it is defined by the plug-ins. The java `Object`s are therefore transferred (they are returned by the `in` method and the `out` method uses it as a parameter).

5 Conclusions

As far as we know, the emuStudio platform is the first attempt of the implementation of both the universal and interactive emulation platform with the emulated components realized via plug-ins. In the present time the platform is used as a teaching support tool for chosen subjects at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia, in its still expanding form for more than two years.

The versatility and configurability allows creating plug-ins of various levels of quality and purpose - they can be intended for pedagogic or even for scientific purposes - e.g. the implementation of plug-ins that emulate the real hardware with the support of measurement of various characteristics, or as one of the phases of design of new hardware or for its testing, etc.

For ensuring the platform's versatility it is important to stabilize the requirements, to standardize components and mainly to design a way of communication in the form of communication protocol, language or other mechanism.

The paper describes the mechanism of communication used in the emuStudio platform at the basic level. The communication mechanism still is not in its final form. Till the present time the 8-bit architectures (MITS Altair8800 and its modification) and two abstract machines (Random Access Machine, BrainDuck - our own architecture) are implemented only.

We believe that in the future the platform will be enhanced and the communication model finished and formally verified. There still is a free space for expanding the platform by adding new emulated computer architectures.

References

[1] J. von Neumann, *First Draft of a Report on the EDVAC*, 1945. ⇒118

[2] N. Salmoria et al., *MAME. The official site of the MAME development team.* ⇒118

[3] E. Freyre et al., *Keeping emulation environments portable (KEEP).* ⇒118

[4] P. Jakubčo, S. Šimoňák, emuStudio – a plugin based emulation platform, *J. Information, Control and Management Systems*, **7,** 1 (2009) 33–46. ⇒118

[5] M. Jelšina, *Architectures of computer systems: principles, structuring organisation, function* (in Slovak), Elfa, Košice, 2002, 567 p. ⇒118

[6] T. Kasai, Computational complexity of multitape Turing machines and random access machines, *Publ. Res. Inst. Math. Sci.*, **13,** 2 (1977) 469–496. ⇒118

[7] MITS, Inc., *Altair 8080 Operators Manual*, 1975. ⇒118

[8] S. Šimoňák, P. Jakubčo, Software based CPU emulation, *Acta Electrotechnica et Informatica*, **8,** 4 (2008) 50–59. ⇒118, 124

[9] L. Vokorokos et al., *Digital computer principles* (in Slovak), Elfa, Košice, 2008, 322 p. ⇒118

Received: March 12, 2010 • Revised: June 25, 2010

Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 135–153

Testing of sequences by simulation

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

Balázs Novák
Eötvös Loránd University
Department of Computer Algebra
H-1117, Budapest, Hungary
Pázmány sétány 1/C
email: psziho@inf.elte.hu

Abstract. Let ξ be a random integer vector, having uniform distribution

$$\mathbf{P}\{\xi = (i_1, i_2, \ldots, i_n) = 1/n^n\} \text{ for } 1 \le i_1, i_2, \ldots, i_n \le n.$$

A realization $(i_1, i_2, \ldots, i_n)$ of ξ is called *good*, if its elements are different. We present algorithms Linear, Backward, Forward, Tree, Garbage, Bucket which decide whether a given realization is good. We analyse the number of comparisons and running time of these algorithms using simulation gathering data on all possible inputs for small values of n and generating random inputs for large values of n.

1 Introduction

Let ξ be a random integer vector, having uniform distribution

$$\mathrm{P}\{\xi = (i_1, i_2, \ldots, i_n)\} = 1/n^n$$

for $1 \le i_1, i_2, \ldots, i_n \le n$.

A realization $(i_1, i_2, \ldots, i_n)$ of ξ is called *good*, if its elements are different. We present six algorithms which decide whether a given realization is good.

This problem arises in connection with the design of agricultural [4, 5, 57, 72] and industrial [34] experiments, with the testing of Latin [1, 9, 22, 23, 27, 32,

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 68M20
Key words and phrases: random sequences, analysis of algorithms, Latin squares, sudoku squares

53, 54, 63, 64] and sudoku [3, 4, 6, 12, 13, 14, 15, 16, 17, 20, 21, 22, 26, 29, 30, 31, 41, 42, 44, 46, 47, 51, 55, 59, 61, 64, 66, 67, 68, 69, 70, 72, 74] squares, with genetic sequences and arrays [2, 7, 8, 18, 24, 28, 35, 36, 37, 38, 45, 48, 49, 50, 56, 65, 71, 73, 75], with sociology [25], and also with the analysis of the performance of computers with interleaved memory [11, 33, 39, 40, 41, 43, 52].

Section 2 contains the pseudocodes of the investigated algorithms. In Section 3 the results of the simulation experiments and the basic theoretical results are presented. Section 4 contains the summary of the paper.

Further simulation results are contained in [62]. The proofs of the lemmas and theorems can be found in [43].

2 Pseudocodes of the algorithms

This section contains the pseudocodes of the investigated algorithms Linear, Backward, Forward, Tree, Garbage, and Bucket. The psudocode conventions described in the book [19] written by Cormen, Leiserson, Rivest, and Stein are used.

The inputs of the following six algorithms are n (the length of the sequence $\mathbf{s}$) and $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, a sequence of nonnegative integers with $0 \leq s_i \leq n$ for $1 \leq i \leq n$) in all cases. The output is always a logical variable g (its value is True, if the input sequence is good, and False otherwise).

The working variables are usually the cycle variables i and j.

2.1 Definition of algorithm Linear

Linear writes zero into the elements of an n length vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, then investigates the elements of the realization and if $v[s_i] > 0$ (signalising a repetition), then stops, otherwise adds 1 to $v[s[i]]$.

Linear$(n, \mathbf{s})$

```
01 g ← True
02 for i ← 1 to n
03     do v[i] ← 0
04 for i ← 1 to n
05     do if v[s[i]] > 0
06           then g ← False
07                return g
08           else v[s[i]] ← v[s[i]] + 1
09 return g
```

2.2 Definition of algorithm BACKWARD

BACKWARD compares the second (i_2), third (i_3), ..., last (i_n) element of the realization **s** with the previous elements until the first collision or until the last pair of elements.

BACKWARD($n, \mathbf{s}$)

```
01 g ← TRUE
02 for i ← 2 to n
03     do for j ← i − 1 downto 1
04             do if s[i] = s[j]
05                    then g ← FALSE
06                         return g
07 return g
```

2.3 Definition of algorithm FORWARD

FORWARD compares the first (s_1), second (s_2), ..., last but one (s_{n-1}) element of the realization with the following elements until the first collision or until the last pair of elements.

FORWARD($n, \mathbf{s}$)

```
01 g ← TRUE
02 for i ← 1 to n − 1
03     do for j ← i + 1 to n
04             do if s[i] = s[j]
05                    then g ← FALSE
06                         return g
07 return g
```

2.4 Definition of algorithm TREE

TREE builds a random search tree from the elements of the realization and finishes the construction of the tree if it finds the following element of the realization in the tree (then the realization is not good) or it tested the last element too without a collision (then the realization is good).

TREE($n, \mathbf{s}$)

```
01 g ← TRUE
02 let s[1] be the root of a tree
03 for i ← 2 to n
```

```
04     if [s[i] is in the tree
05        then g ← False
06             return
07        else insert s[i] in the tree
08 return g
```

2.5 Definition of algorithm Garbage

This algorithm is similar to Linear, but it works without the setting zeros into the elements of a vector requiring linear amount of time.

Beside the cycle variable i Garbage uses as working variable also a vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$. Interesting is that $\mathbf{v}$ is used without initialisation, that is its initial values can be arbitrary integer numbers.

The algorithm Garbage was proposed by Gábor Monostori [58].

Garbage($n, \mathbf{s}$)

```
01 g ← True
02 for i ← 1 to n
03     do if v[s[i]] < i and s[v[s[i]]] = s[i]
04           then g ← False
05                return g
06           else v[s[i]] ← i
07 return g
```

2.6 Definition of algorithm Bucket

Bucket handles the array $Q[1 : m, 1 : m]$ (where $m = \lceil\sqrt{n}\rceil$ and puts the element s_i into the rth row of Q, where $r = \lceil s_i/m\rceil$ and it tests using linear search whether s_j appeared earlier in the corresponding row. The elements of the vector $\mathbf{c} = (c_1, c_2, \ldots, c_m)$ are counters, where c_j $(1 \le j \le m)$ shows the number of elements of the ith row.

For the simplicity we suppose that n is a square.

Bucket($n, \mathbf{s}$)

```
01 g ← True
02 m ← √n
03 for j ← 1 to m
04     do c[j] ← 1
05 for i ← 1 to n
06     do r ← ⌈s[i]/m⌉m
```

```
07        for j ← 1 to c[r] − 1
08            do if s[i] = Q[r, j]
09                  then g ← FALSE
10                       return g
11                  else Q[r, c[r]] ← s[i]
12                       c[r] ← c[r] + 1
13 return g
```

3 Analysis of the algorithms

3.1 Analysis of algorithm LINEAR

The first algorithm is LINEAR. It writes zero into the elements of an n length vector $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, then investigates the elements of the realization sequentially and if $i_j = k$, then adds 1 to v_k and tests whether $v_k > 0$ signalizing a repetition.

In best case LINEAR executes only two comparisons, but the initialization of the vector $\mathbf{v}$ requires $\Theta(n)$ assignments. It is called LINEAR, since its running time is $\Theta(n)$ in best, worst and so also in expected case.

Theorem 1 *The expected number* $C_{exp}(n, \text{LINEAR}) = C_L$ *of comparisons of* LINEAR *is*

$$C_L = 1 - \frac{n!}{n^n} + \sum_{k=1}^{n} \frac{n!k^2}{(n-k)!n^{k+1}}$$

$$= \sqrt{\frac{\pi n}{2}} + \frac{2}{3} + \kappa(n) - \frac{n!}{n^n},$$

where

$$\kappa(n) = \frac{1}{3} - \sqrt{\frac{\pi n}{2}} + \sum_{k=1}^{n} \frac{n!k}{(n-k)!n^{k+1}}$$

tends monotonically decreasing to zero when n *tends to infinity.* $n!/n^n$ *also tends monotonically decreasing to zero, but their difference* $\delta(n) = \kappa(n) - n!/n^n$ *is increasing for* $1 \le n \le 8$ *and is decreasing for* $n \ge 8$.

Theorem 2 *The expected running time* $T_{exp}(n, \text{LINEAR}) = T_L$ *of* LINEAR *is*

$$T_L = n + \sqrt{2\pi n} + \frac{7}{3} + 2\delta(n),$$

n	C_L	$\sqrt{\pi n/2} + 2/3$	$n!/n^n$	$\kappa(n)$	$\delta(n)$
1	1.000000	1.919981	1.000000	0.080019	−0.919981
2	2.000000	2.439121	0.500000	0.060879	−0.439121
3	2.666667	2.837470	0.222222	0.051418	−0.170804
4	3.125000	3.173295	0.093750	0.045455	−0.048295
5	3.472000	3.469162	0.038400	0.041238	+0.002838
6	3.759259	3.736647	0.015432	0.038045	+0.022612
7	4.012019	3.982624	0.006120	0.035515	+0.029395
8	4.242615	4.211574	0.002403	0.033444	+0.031040
9	4.457379	4.426609	0.000937	0.031707	+0.030770
10	4.659853	4.629994	0.000363	0.030222	+0.029859

Table 1: Values of C_L, $\sqrt{\pi n/2} + 2/3$, $n!/n^n$, $\kappa(n)$, and $\delta(n) = \kappa(n) - n!/n^n$ for $n = 1,\ 2,\ \ldots,\ 10$

where

$$\delta(n) = \kappa(n) - \frac{n!}{n^n}$$

tends to zero when n *tends to infinity, further*

$$\delta(n+1) > \delta(n) \text{ for } 1 \le n \le 7 \text{ and } \delta(n+1) < \delta(n) \text{ for } n \ge 8.$$

Table 1 shows some concrete values connected with algorithm Linear.

3.2 Analysis of algorithm Backward

The second algorithm is Backward. This algorithm is a naive comparison-based one. Backward compares the second (i_2), third (i_3), ..., last (i_n) element of the realization with the previous elements until the first repetition or until the last pair of elements.

The running time of Backward is constant in the best case, but it is quadratic in the worst case.

Theorem 3 *The expected number* $C_{exp}(n, \text{Backward}) = C_B$ *of comparisons of the algorithm* Backward *is*

$$C_B = n + \sqrt{\frac{\pi n}{8}} + \frac{2}{3} - \alpha(n),$$

where $\alpha(n) = \kappa(n)/2 + (n!/n^n)((n+1)/2)$ *monotonically decreasing tends to zero when* n *tends to* ∞.

Table 2 shows some concrete values characterizing algorithm BACKWARD.

n	C_B	$n-\sqrt{\pi n/8}+2/3$	$(n!/n^n)((n+1)/2)$	$\kappa(n)$	$\alpha(n)$
1	0.000000	1.040010	1.000000	0.080019	1.040010
2	1.000000	1.780440	0.750000	0.060879	0.780440
3	2.111111	2.581265	0.444444	0.051418	0.470154
4	3.156250	3.413353	0.234375	0.045455	0.257103
5	4.129600	4.265419	0.115200	0.041238	0.135819
6	5.058642	5.131677	0.054012	0.038045	0,073035
7	5.966451	6.008688	0.024480	0.035515	0.042237
8	6.866676	6.894213	0.010815	0.033444	0.027536
9	7.766159	7.786695	0.004683	0.031707	0.020537
10	8.667896	8.685003	0.001996	0.030222	0.017107

Table 2: Values of C_B, $n-\sqrt{\pi n/8}+2/3$, $(n!/n^n)((n+1)/2)$, $\kappa(n)$, and $\alpha(n) = \kappa(n)/2 + (n!/n^n)((n+1)/2)$ for $n = 1,\ 2,\ \ldots,\ 10$

The next assertion gives the expected running time of algorithm BACKWARD.

Theorem 4 *The expected running time* $T_{exp}(n, \text{BACKWARD}) = T_B$ *of the algorithm* BACKWARD *is*

$$T_B = n + \sqrt{\frac{\pi n}{8}} + \frac{4}{3} - \alpha(n),$$

where $\alpha(n) = \kappa(n)/2 + (n!/n^n)((n+1)/2)$ *monotonically decreasing tends to zero when* n *tends to* ∞.

3.3 Analysis of algorithm FORWARD

FORWARD compares the first (s_1), second (s_2), ..., last but one (s_{n-1}) element of the realization with the next elements until the first collision or until the last pair of elements.

Taking into account the number of the necessary comparisons in line 04 of FORWARD, we get $C_{best}(n, \text{FORWARD}) = 1 = \Theta(1)$, and $C_{worst}(n, \text{FORWARD}) = B(n, 2) = \Theta(n^2)$.

The next assertion gives the expected running time.

Theorem 5 *The expected running time* $T_{exp}(n, \text{FORWARD}) = T_F$ *of the algorithm* FORWARD *is*

$$T_F = n + \Theta(\sqrt{n}). \tag{1}$$

Although the basic characteristics of FORWARD and BACKWARD are identical, as Table 3 shows, there is a small difference in the expected behaviour.

n	number of sequences	number of good sequences	C_F	C_W
2	4	2	1.000000	1.000000
3	27	6	2.111111	2.111111
4	256	24	3.203125	3.156250
5	3 125	120	4.264000	4.126960
6	46 656	720	5.342341	5.058642
7	823 543	5 040	6.326760	5.966451
8	16 777 216	40 320	7.342926	6.866676
9	387 420 489	362 880	8.354165	7.766159

Table 3: Values of n, the number of possible input sequences, number of good sequences, expected number of comparisons of FORWARD (C_F) and expected number of comparisons of BACKWARD (C_W) for $n = 2, 3, \ldots, 9$

3.4 Analysis of algorithm TREE

TREE builds a random search tree from the elements of the realization and finishes the construction of the tree if it finds the following element of the realization in the tree (then the realization is not good) or it tested the last element too without a collision (then the realization is good).

The worst case running time of TREE appears when the input contains different elements in increasing or decreasing order. Then the result is $\Theta(n^2)$. The best case is when the first two elements of $\mathbf{s}$ are equal, so $C_{best}(n, \text{TREE}) = 1 = \Theta(1)$.

Using the known fact that the expected height of a random search tree is $\Theta(\lg n)$ we can get that the order of the expected running time is $\sqrt{n} \log n$.

Theorem 6 *The expected running time* T_T *of* TREE *is*

$$T_T = \Theta(\sqrt{n} \lg n). \tag{2}$$

n	number of good inputs	number of comparisons	number of assignments
1	100 000.000000	0.000000	1.000000
2	49 946.000000	1.000000	1.499460
3	22 243.000000	2.038960	1.889900
4	9 396.000000	2.921710	2.219390
5	3 723.000000	3.682710	2.511409
6	1 569.000000	4.352690	2.773160
7	620.000000	4.985280	3.021820
8	251.000000	5.590900	3.252989
9	104	6.148550	3.459510
10	33	6.704350	3.663749
11	17	7.271570	3.860450
12	3	7.779950	4.039530
13	3	8.314370	4.214370
14	0	8.824660	4.384480
15	2	9.302720	4.537880
16	0	9.840690	4.716760
17	0	10.287560	4.853530
18	0	10.719770	4.989370
19	0	11.242740	5.147560
20	0	11.689660	5.279180

Table 4: Values of n, number of good inputs, number of comparisons, number of assignments of TREE for $n = 1, 2, \ldots, 10$

Table 4 shows some results of the simulation experiments (the number of random input sequences is 100 000 in all cases).

Using the method of the smallest squares to find the parameters of the formula $a\sqrt{n}\log_2 n$ we received the following approximation formula for the expected number of comparisons:

$$C_{exp}(n, \text{TREE}) = 1.245754\sqrt{n}\log_2 n - 0.273588.$$

3.5 Analysis of algorithm GARBAGE

This algorithm is similar to LINEAR, but it works without the setting zeros into the elements of a vector requiring linear amount of time.

Beside the cycle variable i GARBAGE uses as working variable also a vector

$\mathbf{v} = (v_1, v_2, \ldots, v_n)$. Interesting is that $\mathbf{v}$ is used without initialisation, that is its initial values can be arbitrary integer numbers.

The worst case running time of GARBAGE appears when the input contains different elements and the garbage in the memory does not help, but even in this case $C_{worst}(n, \text{GARBAGE}) = \Theta(n)$. The best case is when the first element is repeated in the input and the garbage helps to find a repetition of the firs element of the input. Taking into account this case we get $C_{best}(n, \text{GARBAGE}) = \Theta(1)$.

According to the next assertion the expected running time is $\Theta(\sqrt{n})$.

Lemma 7 *The expected running time of* GARBAGE *is*

$$T_{exp}(n, \text{GARBAGE}) = \Theta(\sqrt{n}). \tag{3}$$

3.6 Analysis of algorithm BUCKET

Algorithm BUCKET divides the interval $[1, n]$ into $m = \lceil \sqrt{n} \rceil$ subintervals $I_1, I_2, \ldots, I_m$, where $I_k = [(k-1)m + 1, km)]$, and assigns a bucket B_k to interval I_k. BUCKET sequentially puts the input elements i_j into the corresponding bucket: if i_j belongs to the interval I_k then it checks whether i_j is contained in B_k or not. BUCKET works up to the first repetition. (For the simplicity we suppose that $n = m^2$.)

In best case BUCKET executes only 1 comparison, but the initialization of the buckets requires $\Theta(\sqrt{n})$ assignments, therefore the best running time is also $\sqrt{n}$. The worst case appears when the input is a permutation. Then each bucket requires $\Theta(n)$ comparisons, so the worst running time is $\Theta(n\sqrt{n})$.

Lemma 8 *Let* b_j $(j = 1, 2, \ldots, m)$ *be a random variable characterising the number of elements in the bucket* B_j *at the moment of the first repetition. Then*

$$E\{b_j\} = \sqrt{\frac{\pi}{2}} - \mu(n)$$

for $j = 1, 2, \ldots, m$, *where*

$$\mu(n) = \frac{1}{3\sqrt{n}} - \frac{\kappa(n)}{\sqrt{n}},$$

and $\mu(n)$ *tends monotonically decreasing to zero when* n *tends to infinity.*

Table 5 contains some concrete values connected with $E\{b_1\}$.

n	$E\{b_1\}$	$\sqrt{\pi/2}$	$1/(3\sqrt{n})$	$\kappa(n)/\sqrt{n}$	$\mu(n)$
1	1.000000	1.253314	0.333333	0.080019	0.253314
2	1.060660	1.253314	0.235702	0.043048	0.192654
3	1.090055	1.253314	0.192450	0.029686	0.162764
4	1.109375	1.253314	0.166667	0.022727	0.143940
5	1.122685	1.253314	0.149071	0.018442	0.130629
6	1.132763	1.253314	0.136083	0.015532	0.120551
7	1.147287	1.253314	0.125988	0.013423	0.112565
8	1.147287	1.253314	0.117851	0.011824	0.106027
9	1.152772	1.253314	0.111111	0.010569	0.100542
10	1.157462	1.253314	0.105409	0.009557	0.095852

Table 5: Values of $E\{b_1\}$, $\sqrt{\pi/2}$, $1/(3\sqrt{n})$, $\kappa(n)/\sqrt{n}$, and $\mu(n) = 1/(3\sqrt{n}) - \kappa(n)/\sqrt{n}$ of BUCKET for $n = 1,\ 2,\ \ldots,\ 10$

Lemma 9 *Let f_n be a random variable characterising the number of comparisons executed in connection with the first repeated element. Then*

$$E\{f_n\} = 1 + \sqrt{\frac{\pi}{8}} - \eta(n),$$

where

$$\eta(n) = \frac{\frac{1}{3} + \sqrt{\frac{\pi}{8}} - \frac{\kappa(n)}{2}}{\sqrt{n} + 2},$$

and $\eta(n)$ tends monotonically decreasing to zero when n tends to infinity.

Theorem 10 *The expected number $C_{exp}(n, \text{BUCKET}) = C_B$ of comparisons of algorithm* BUCKET *in 1 bucket is*

$$C_B = \sqrt{n} + \frac{1}{3} - \sqrt{\frac{\pi}{8}} + \rho(n),$$

where

$$\rho(n) = \frac{5/6 - \sqrt{9\pi/8} - 3\kappa(n)/2}{\sqrt{n} + 1}$$

tends to zero when n and tends to infinity.

Index and Algorithm	$C_{best}(n)$	$C_{worst}(n)$	$C_{exp}(n)$
1. Linear	$\Theta(1)$	$\Theta(n)$	$\Theta(\sqrt{n})$
2. Backward	$\Theta(1)$	$\Theta(n^2)$	$\Theta(n)$
3. Forward	$\Theta(1)$	$\Theta(n^2)$	$\Theta(n)$
4. Tree	$\Theta(1)$	$\Theta(n^2)$	$\Theta(\sqrt{n}\lg n)$
5. Garbage	$\Theta(1)$	$\Theta(n)$	$\Theta(\sqrt{n})$
6. Bucket	$\Theta(\sqrt{n})$	$\Theta(n\sqrt{n})$	$\Theta(\sqrt{n})$

Table 6: The number of necessary comparisons of the investigated algorithms in best, worst and expected cases

Theorem 11 *The expected running time* $T_B(n, \text{Bucket}) = T_B$ *of* Bucket *is*

$$T_B = \left(3 + 3\sqrt{\frac{\pi}{2}}\right)\sqrt{n} + \sqrt{\frac{25\pi}{8}} + \phi(n),$$

where

$$\phi(n) = 3\kappa(n) - \rho(n) - 3\eta(n) - \frac{n!}{n^n} - \frac{3\sqrt{\pi/8} - 1/3 - 3\kappa(n)/2}{\sqrt{n}+1}$$

and $\phi(n)$ *tends to zero when* n *tends to infinity.*

It is worth to remark that simulation experiments of B. Novák [62] show that the expected running time of Garbage is a few percent better, then the expected running time of Bucket.

4 Summary

Table 6 contains the number of necessary comparisons in best, worst and expected cases for all investigated algorithms.
Table 7 contains the running time in best, worst and expected cases for all investigated algorithms.

Acknowledgements. The authors thank Tamás F. Móri [60] for proving Lemma 8 and 9 and Péter Burcsi [10] for useful information on references, both are teachers of Eötvös Loránd University.

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

Index and Algorithm	$T_{best}(n)$	$T_{worst}(n)$	$T_{exp}(n)$
1. LINEAR	$\Theta(n)$	$\Theta(n)$	$n + \Theta(\sqrt{n})$
2. BACKWARD	$\Theta(1)$	$\Theta(n^2)$	$\Theta(n)$
3. FORWARD	$\Theta(1)$	$\Theta(n^2)$	$\Theta(n)$
5. TREE	$\Theta(1)$	$\Theta(n^2)$	$\Theta(\sqrt{n}\lg n)$
6. GARBAGE	$\Theta(1)$	$\Theta(n)$	$\Theta(\sqrt{n})$
7. BUCKET	$\Theta(\sqrt{n})$	$\Theta(n\sqrt{n})$	$\Theta(\sqrt{n})$

Table 7: The running times of the investigated algorithms in best, worst and expected cases

References

[1] P. Adams, D. Bryant, M. Buchanan, Completing partial Latin squares with two filled rows and two filled columns, *Electron. J. Combin.* **15,** 1 (2008), Research paper 56, 26 p. ⇒136

[2] M.-C. Anisiu, A. Iványi, Two-dimensional arrays with maximal complexity, *Pure Math. Appl.* (*PU.M.A.*) **17,** 3–4 (2006) 197–204. ⇒136

[3] C. Arcos, G. Brookfield, M. Krebs, Mini-sudokus and groups, *Math. Mag.* **83,** 2 (2010) 111–122. ⇒136

[4] R. A. Bailey, R. Cameron, P. J. Connelly, Sudoku, gerechte designs, resolutions, affine space, spreads, reguli, and Hamming codes, *American Math. Monthly* **115,** 5 (2008) 383–404. ⇒135, 136

[5] W. U. Behrens, Feldversuchsanordnungen mit verbessertem Ausgleich der Bodenunterschiede, *Zeitschrift für Landwirtschaftliches Versuchs- und Untersuchungswesen,* **2** (1956) 176–193. ⇒135

[6] D. Berthier, Unbiased statistics of a constraint satisfaction problem – a controlled-bias generator, in: S. Tarek et al. (eds.), *Innovations in computing sciences and software engineering,* Proc. Second International Conference on Systems, Computing Sciences and Software Engineering (SCSS'2009, December 4–12, 2009, Dordrecht). Springer, Berlin, 2010. pp. 91–97. ⇒136

[7] S. Brett, G. Hurlbert, B. Jackson, Preface [Generalisations of de Bruijn cycles and Gray codes], *Discrete Math.*, **309,** 17 (2009) 5255–5258. ⇒136

[8] A. A. Bruen, R. A. Mollin, Cryptography and shift registers, *Open Math. J.*, **2** (2009) 16–21. ⇒136

[9] H. L. Buchanan, M. N. Ferencak, On completing Latin squares, *J. Combin. Math. Combin. Comput.*, **34** (2000) 129–132. ⇒136

[10] P. Burcsi, Personal communication. Budapest, March 2009. ⇒146

[11] G. J. Burnett, E. G. Coffman, Jr., Combinatorial problem related to interleaved memory systems, *J. ACM,* **20,** 1 (1973) 39–45. ⇒136

[12] P. J. Cameron, *Sudoku – an alternative history*, Talk to the Archimedeans, Queen Mary University of London, February 2007. ⇒136

[13] A. Carlos, G. Brookfield, M. Krebs, Mini-sudokus and groups, *Math. Mag.,* **83,** 2 (2010) 111–122. ⇒136

[14] J. Carmichael, K. Schloeman, M. B. Ward, Cosets and Cayley-sudoku tables, *Math. Mag.,* **83,** 2 (2010) 130–139. ⇒136

[15] Ch.-Ch. Chang, P.-Y. Lin, Z.-H. Wang, M.-Ch. Li, A sudoku-based secret image sharing scheme with reversibility, *J. Commun.,* **5,** 1 (2010) 5–12. ⇒136

[16] Z. Chen, Heuristic reasoning on graph and game complexity of sudoku, ARXIV.org, 2010. 6 p. ⇒136

[17] Y.-F. Chien, W.-K. Hon, Cryptographic and physical zero-knowledge proof: From sudoku to nonogram, in: P. Boldi (ed.), *Fun with Algorithms,* (5th International Conference, FUN 2010, Ischia, Italy, June 2–4, 2010.) Springer, Berlin, 2010, *Lecture Notes in Comput. Sci.,* **6099** (2010) 102–112. ⇒136

[18] J. Cooper, C. Heitsch, The discrepancy of the lex-least de Bruijn sequence, *Discrete Math.,* **310,** 6–7 (2010), 1152–1159. ⇒136

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms,* Third edition. The MIT Press, Cambridge, 2009. ⇒136

[20] J. F. Crook, A pencil-and-paper algorithm for solving sudoku puzzles, *Notices Amer. Math. Soc.,* **56** (2009) 460–468. ⇒136

[21] G. Dahl, Permutation matrices related to sudoku, *Linear Algebra Appl.,* **430** (2009), 2457–2463. ⇒136

[22] J. Dénes, A. D. Keedwell, *Latin squares. New developments in the theory and applications*, North-Holland, Amsterdam, 1991. ⇒136

[23] T. Easton, R. G. Parker, On completing Latin squares, *Discrete Appl. Math.*, **113,** 2–3 (2001) 167–181. ⇒136

[24] C. H. Elzinga, S. Rahmann, H. Wung, Algorithms for subsequence combinatorics, *Theor. Comput. Sci.*, **409,** 3 (2008) 394–404. ⇒136

[25] C. H. Elzinga, Complexity of categorial time series, *Sociological Methods & Research,* **38,** 3 (2010) 463–481. ⇒136

[26] M. Erickson, *Pearls of discrete mathematics,* Discrete Mathematics and its Applications, CRC Press, Boca Raton, FL, 2010. ⇒136

[27] R. Euler, On the completability of incomplete Latin squares, *European J. Combin.* **31** (2010) 535–552. ⇒136

[28] S. Ferenczi, Z. Kása, Complexity for finite factors of infinite sequences, *Theoret. Comput. Sci.* **218**, 1 (1999) 177–195. ⇒136

[29] R. Fontana, F. Rapallo, M. P. Rogantin, Indicator function and sudoku designs, in: P. Gibilisco, E. Ricco-magno, M. P. Rogantin, H. P. Wynn (eds.) *Algebraic and Geometric Methods in Statistics*, pp. 203–224. Cambridge University Press, Cambridge, 2010. ⇒136

[30] R. Fontana, F. Rapallo, M. P. Rogantin, Markov bases for sudoku grids. Rapporto interno N. 4, marzo 2010, Politecnico di Torino. ⇒136

[31] A. F. Gabor, G. J. Woeginger, How *not* to solve a Sudoku. *Operation Research Letters*, **38,** 6 (2010) 582–584. ⇒136

[32] I. Hajirasouliha, H. Jowhari, R. Kumar, R. Sundaram, On completing Latin squares, *Lecture Notes in Comput. Sci.,* **4393** (2007), 524–535. Springer, Berlin, 2007. ⇒136

[33] H. Hellerman, *Digital computer system principles.* Mc Graw Hill, New York, 1967. ⇒136

[34] A. Heppes, P. Révész, A new generalization of the concept of latin squares and orthogonal latin squares and its application to the design of experiments (in Hungarian), *Magyar Tud. Akad. Mat. Int. Közl.,* **1** (1956) 379–390. ⇒135

[35] M. Horváth, A. Iványi, Growing perfect cubes, *Discrete Math.,* **308,** 19 (2008) 4378–4388. ⇒136

[36] A. Iványi, On the d-complexity of words, *Ann. Univ. Sci. Budapest. Sect. Comput.* **8** (1987) 69–90 (1988). ⇒136

[37] A. Iványi, Construction of infinite de Bruijn arrays, *Discrete Appl. Math.* **22,** 3 (1988/89), 289–293. ⇒136

[38] A. Iványi, Construction of three-dimensional perfect matrices, (Twelfth British Combinatorial Conference, Norwich, 1989). *Ars Combin.* **29C** (1990) 33–40. ⇒136

[39] A. Iványi, I. Kátai, Estimates for speed of computers with interleaved memory systems, *Ann. Univ. Sci. Budapest. Sect. Math.*, **19** (1976) 159–164. ⇒136

[40] A. Iványi, I. Kátai, Processing of random sequences with priority, *Acta Cybern.* **4,** 1 (1978/79) 85–101. ⇒136

[41] A. Iványi, I. Kátai, Quick testing of random variables, *Proc. ICAI'2010* (Eger, January 27–30, 2010). To appear. ⇒136

[42] A. Iványi, I. Kátai, Testing of uniformly distributed vectors, in: *Abstracts János Bolyai Memorial Conference*, (Budapest, August 28–30, 2010), p. 47. ⇒136

[43] A. Iványi, I. Kátai, Testing of random matrices, *Ann. Univ. Sci. Budapest. Sect. Comput.* (submitted). ⇒136

[44] A. Iványi, B. Novák, Testing of random sequences by simulation, in: *Abstracts 8th MACS* (Komárno, July 14–17, 2010). ⇒136

[45] A. Iványi, Z. Tóth, Existence of de Bruijn words, *Second Conference on Automata, Languages and Programming Systems* (Salgótarján, 1988), 165–172, DM, 88-4, Karl Marx Univ. Econom., Budapest, 1988. ⇒136

[46] I. Kanaana, B. Ravikumar, Row-filled completion problem for sudoku, *Util. Math.* **81** (2010) 65–84. ⇒136

[47] Z. Karimi-Dehkordi, K. Zamanifar, A. Baraani-Dastjerdi, N. Ghasem-Aghaee, Sudoku using parallel simulated annealing, in: Y. Tan et al.

(eds.), *Advances in Swarm Intelligence* (Proc. First International Conference, ICSI 2010, Beijing, China, June 12–15, 2010, Part II. *Lecture Notes in Comput. Sci.,* **6146** (2010) 461–467, Springer, Berlin, 2010. ⇒ 136

[48] Z. Kása, Computing the d-complexity of words by Fibonacci-like sequences, *Studia Univ. Babeş-Bolyai Math.* **35,** 3 (1990) 49–53. ⇒136

[49] Z. Kása, *Pure Math. Appl.* On the d-complexity of strings, *(PU.M.A.)* **9,** 1–2 (1998) 119–128. ⇒136

[50] Z. Kása, Super-d-complexity of finite words, *Proc. 8th Joint Conference on Mathematics and Computer Science*, (Komárno, Slovakia, July 14–17), 2010, To appear. ⇒136

[51] A. D. Keedwell, Constructions of complete sets of orthogonal diagonal sudoku squares, *Australas. J. Combin.* **47** (2010) 227–238. ⇒136

[52] D. E. Knuth, *The art of computer programming, Vol. 1. Fundamental algorithms* (third edition). Addison–Wesley, Reading, MA, 1997. ⇒136

[53] J. S. Kuhl, T. Denley, On a generalization of the Evans conjecture, *Discrete Math.* **308,** 20 (2008), 4763–4767. ⇒136

[54] S. R. Kumar S., A. Russell, R. Sundaram, Approximating Latin square extensions, *Algorithmica* **24,** 2 (1999) 128–138. ⇒136

[55] L. Lorch, Mutually orthogonal families of linear sudoku solutions, *J. Aust. Math. Soc.,* **87,** 3 (2009) 409–420. ⇒136

[56] M. Matamala, F. Moreno, Minimum Eulerian circuits and minimum de Bruijn sequences, *Discrete Math.,* **309,** 17 (2009) 5298–5304. ⇒136

[57] H.-D. Mo, R.-G. Xu, Sudoku square – a new design in field, *Acta Agronomica Sinica*, **34,** 9 (2008) 1489–1493. ⇒135

[58] G. Monostori, Personal communication, Budapest, May 2010. ⇒138

[59] T. K. Moon, J. H. Gunther, J. J. Kupin, Sinkhorn solves sudoku,*IEEE Trans. Inform. Theory*, **55,** 4 (2009) 1741–1746. ⇒136

[60] T. Móri, Personal communication, Budapest, April 2010. ⇒146

[61] P. K. Newton, S. A. deSalvo, The Shannon entropy of sudoku matrices, *Proc. R. Soc. Lond. Ser. A, Math. Phys. Eng. Sci.* **466** (2010) 1957-1975. ⇒136

[62] B. Novák, *Analysis of sudoku algorithms* (in Hungarian), MSc thesis, Eötvös Loránd University, Fac. of Informatics, Budapest, 2010. ⇒136, 146

[63] L.-D. Öhman, A note on completing Latin squares, *Australas. J. Combin.,* **45** (2009) 117–123. ⇒136

[64] R. M. Pedersen, T. L. Vis, Sets of mutually orthogonal sudoku Latin squares. *College Math. J.,* **40,** 3 (2009) 174–180. ⇒136

[65] R. Penne, A note on certain de Bruijn sequences with forbidden subsequences, *Discrete Math.*, **310,** 4 (2010) 966–969. ⇒136

[66] J. S. Provan, Sudoku: strategy versus structure, *Amer. Math. Monthly*, **116,** 8 (2009), 702–707. ⇒136

[67] T. Sander, Sudoku graphs are integral, *Electron. J. Combin.,* **16,** 1 (2009), Note 25, 7 p. ⇒136

[68] Y. Sato, H. Inoue, Genetic operations to solve sudoku puzzles, *Proc. 12th Annual Conference on Genetic and Evolutionary Computation* GECCO'10, July 7–11, 2010, Portland, OR, pp. 2111–21012. ⇒136

[69] M. J. Soottile, T. G. Mattson, C. E. Rasmussen, *Introduction to concurrency in programming languages,* Chapman & Hall/CRC Computational Science Series. CRC Press, Boca Raton, FL, 2010. ⇒136

[70] D. Thom, *SUDOKU ist NP-vollständig*, PhD Dissertation, Stuttgart, 2007. ⇒136

[71] O. G. Troyanskaya, O. Arbell, Y. Koren, G. M. Landau, A. Bolshoy, Sequence complexity profiles of prokaryotic genomic sequences: A fast algorithm for calculating linguistic complexity, *Bioinformatics*, **18,** 5 (2002) 679–688. ⇒136

[72] E. R. Vaughan, The complexity of constructing gerechte designs, *Electron. J. Combin.,* **16,** 1 (2009), paper R15, 8 p. ⇒135, 136

[73] X. Xu, Y. Cao, J.-M. Xu, Y. Wu, Feedback numbers of de Bruijn digraphs, *Comput. Math. Appl.,* **59,** 4 (2010), 716–723. ⇒136

[74] C. Xu, W. Xu, The model and algorithm to estimate the difficulty levels of sudoku puzzles, *J. Math. Res.* **11,** 2 (2009), 43–46. ⇒136

[75] W. Zhang, S. Liu, H. Huang, An efficient implementation algorithm for generating de Bruijn sequences, *Computer Standards & Interfaces,* **31,** 6 (2009) 1190–1191. ⇒136

Received: August 20, 2010 • Revised: October 15, 2010

Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 154–167

Testing by C++ template metaprograms

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

Abstract. Testing is one of the most indispensable tasks in software engineering. The role of testing in software development has grown significantly because testing is able to reveal defects in the code in an early stage of development. Many unit test frameworks compatible with C/C++ code exist, but a standard one is missing. Unfortunately, many unsolved problems can be mentioned with the existing methods, for example usually external tools are necessary for testing C++ programs.

In this paper we present a new approach for testing C++ programs. Our solution is based on C++ template metaprogramming facilities, so it can work with the standard-compliant compilers. The metaprogramming approach ensures that the overhead of testing is minimal at runtime. This approach also supports that the specification language can be customized among other advantages. Nevertheless, the only necessary tool is the compiler itself.

1 Introduction

Testing is the most important method to check programs' correct behaviour. Testing can reveal many problems within the code in development phase. Testing is cruicial from the view of software quality [5]. Many purposes of testing can be, for instance, quality assurance, verification and validation, or reliability estimation. Nonetheless, testing is potentially endless. It can never completely

Computing Classification System 1998: D.2.5
Mathematics Subject Classification 2010: 62N03
Key words and phrases: testing, C++, template metaprogramming

identify all the defects within the software. The main task is is to deliver faultless software [20].

Correctness testing and reliability testing are two major areas of testing. However, many different testing levels are used. In this paper we deal with unit tests that is about correctness. The goal of unit testing is to isolate each part of the program and to show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development phase. Unfortunately, most frameworks need external tools [10].

A testing framework is proposed in [3, 4] which is based on the C++0x – the C++ forthcoming standard. The framework takes advantage of *concepts* and *axioms*. These constructs support the generic programming in C++ as they enable to write type constraints in template parameters. By now, these constructs are removed from the draft of the next standard. Metaprogram testing framework has already been developed [16] too, but it deals with metaprogams, it is just the opposite of our approach.

C++ template metaprogramming is an emerging paradigm which enables to execute algorithms when ordinary C++ programs are compiled. The style of C++ template metaprograms is very similar to the functional programming paradigm. Metaprograms have many advantages that we can harness. Metalevel often subserves the validation [8].

Template metaprograms run at compilation-time, whereupon the overhead at runtime is minimal. Metaprograms' "input" is the runtime C++ program itself, therefore metaprograms are able to retrieve information about the hosting program. This way we can check many properties about the programs during compilation [12, 14, 21, 22].

Another important feature of template metaprograms is the opportunity of *domain-specific languages*. These special purpose languages are integrated into C++ by template metaprograms [7, 9]. Libraries can be found that support the development of domain-specific languages [11]. New languages can be figured out to write C++ template metaprograms [18]. Special specification languages can be used for testing C++ programs without external tools.

In this paper we present a new approach to test C++ code. Our framework is based on the metaprogramming facility of C++. We argue for testing by meta-level because of numerous reasons.

The rest of this paper is organized as follows. In Section 2 C++ template metaprograms are detailed. In Section 3 we present the basic ideas behind our approach, after that in Section 4 we analyze the advantages and disadvantages of our framework. Finally, the future work is detailed in Section 5.

2 C++ template metaprogramming

The template facility of C++ allows writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations of lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by type, thus, we have to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list from it. See the example below:

```
template<typename T>
struct list
{
  void insert( const T& t );
  // ...
};

int main()
{
  list<int> l;      //instantiation for int
  list<double> d;   // and for double
  l.insert( 42 );   // usage
  d.insert( 3.14 ); // usage
}
```

The list type has one template argument `T`. This refers to the parameter type, whose objects will be contained in the list. To use this list we have to generate an instance assigning a specific type to it. The process is called *instantiation*. During this process the compiler replaces the abstract type `T` with a specific type and compiles this newly generated code. The instantiation can be invoked explicitly by the programmer but in most cases it is done implicitly by the compiler when the new list is first referred to.

The template mechanism of C++ enables the definition of partial and full *specializations*. Let us suppose that we would like to create a more space efficient type-specific implementation of the `list` template for the `bool` type. We may define the following specialization:

```
template<>
struct list<bool>
```

```
{
  //type-specific implementation
};
```

The implementation of the specialized version can be totally different from the original one. Only the names of these template types are the same. If during the instantiation the concrete type argument is `bool`, the specific version of `list<bool>` is chosen, otherwise the general one is selected.

Template specialization is an essential practice for template metaprogramming too [1]. In template metaprograms templates usually refer to other templates, sometimes from the same class with different type argument. In this situation an implicit instantiation will be performed. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```
template<int N>
struct Factorial
{
  enum { value=N*Factorial<N-1>::value };
};

template<>
struct Factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int result = Factorial<5>::value;
}
```

To initialize the variable `result` here, the expression `Factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `Factorial` template with 5. The definition of `value` is `N * Factorial<N-1>::value`, hence the compiler has to instantiate `Factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler chooses the special version of `Factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of

5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram "runs" while the compiler compiles the code.

Template metaprograms therefore stand for the collection of templates, their instantiations and specializations, and perform operations at compilation time. The basic control structures like iteration and condition appear in them in a functional way [17]. As we can see in the previous example iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```
template<bool cond,class Then,class Else>
struct If
{
  typedef Then type;
};

template<class Then, class Else>
struct If<false, Then, Else>
{
  typedef Else type;
};
```

The `If` structure has three template arguments: a boolean and two abstract types. If the `cond` is false, then the partly-specialized version of `If` will be instantiated, thus the `type` will be bound to `Else`. Otherwise the general version of `If` will be instantiated and `type` will be bound to `Then`.

With the help of `If` we can delegate type-related decisions from design time to instantiation (compilation) time. Let us suppose, we want to implement a `max(T,S)` function template comparing values of type T and type S returning the greater value. The problem is how we should define the return value. Which type is "better" to return the result? At design time we do not know the actual type of the `T` and `S` template parameters. However, with a small template metaprogram we can solve the problem:

```
template <class T, class S>
typename If<sizeof(T)<sizeof(S),S,T>::type
  max( T x, S y)
  {
    return x > y ? x : y;
  }
```

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite forms of implementation of expression templates [24]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [11].

We define a typelist with the following recursive template:

```
class NullType {};

typedef Typelist<char,Typelist<signed char,
    Typelist<unsigned char,NullType> > >
  Charlist;
```

In the example we store the three character types in a typelist. We can use helper macro definitions to make the syntax more readable.

```
#define TYPELIST_1(x)
          Typelist< x, NullType>
#define TYPELIST_2(x, y)
          Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)
          Typelist< x, TYPELIST_2(y,z)>
// ...
typedef
TYPELIST_3(char,signed char,unsigned char)
  Charlist;
```

Essential helper functions – like `Length`, which computes the size of a list at compilation time – have been defined in Alexandrescu's Loki library [2] in pure functional programming style. Similar data structures and algorithms can be found in the metaprogramming library [11].

The examples presented in this section expose the different approaches of template metaprograms and ordinary runtime programs. Variables are represented by static constants and enumeration values, control structures are implemented via template specializations, functions are replaced by classes. We use recursive types instead of the usual data structures. Fine visualizer tools can help a lot to comprehend these structures [6].

3 Testing framework

In this section we present the main ideas behind our testing framework which takes advantage of the C++ template metaprogramming.

First, we write a simple type which makes connection between the compilation-time and the runtime data. This is the kernel of the testing framework. If the compilation-time data is not equal to the runtime data, we throw an exception to mark the problem.

```
struct _Invalid
{
  // ...
};

template < int N >
class _Test
{
  const int value;

public:

  _Test( int i ) : value( i )
  {
    if ( value!=N )
      throw _Invalid();
  }

  int get_value() const
  {
    return value;
  }
};
```

Let us consider that a runtime function is written, that calculates the factorial of its argument. This function is written in an iterative way:

```
int factorial( int n )
{
  int f = 1;
  for( int i = 1; i <= n; ++i)
```

```
  {
    f *= i;
  }
  return f;
}
```

It is easy to test the factorial function:

```
template <int N>
_Test<Factorial<N>::value> factorial_test( const _Test<N>& n )
{
  return factorial( n.get_value() );
}
```

When `factorial_test` is called, it takes a compile-time and runtime parameter. The constructor of `_Test` guarantees, that the two parameters are equal. We take advantage of the parameter conversions of C++. When an integer is passed as `_Test`, it automatically calls the constructor of `_Test` which tests if the runtime and compilation time parameters are the same. If the runtime and compilation time parameters disagree, an exception is raised. The return type of `factorial_test` describes that it must compute the `Factorial<N>`. When it returns a value, it also calls the constructor of `_Test`. At compilation time it is computed what the return should be according to the metaprogram specification – e.g. what the `Factorial<N>` is. Because the `factorial_test` takes a `_Test` parameter, two parameters cannot be different. When the `factorial_test` returns it is also evaluates if the result of compilation time algorithm is the same with the result of the runtime algorithm, and an exception raised if it fails. So, we have a runtime and compilation time input, first we calculate the result at compilation time from the compilation time input. At runtime we have the very same input and a runtime function, and evaluates if the runtime algorithm results in the very same output. If it fails an exception is thrown.

Of course, we have to call the `factorial_test` function:

```
int main()
{
  factorial_test< 6 >( 6 );
}
```

In this case, we write `Factorial` metafunction that counts the factorial at compilation time, but we do not have to write this metafunction with metapro-

grams. This metaprogram can be generated by the compiler from a specification that can be defined in EClean [18, 17], Haskell, or other domain-specific language [15].

Instead of return value, references are often used to transmit data to the caller:

```
void inc( int& i )
{
  ++i;
}
```

At this point, we cannot wrap the call of this function into a tester function. Hence, in this case we deal with a new local variable to test.

```
template < int N >
_Test<N+1> inc_test( const _Test<N>& n )
{
  int i = n.get_value();
  inc( i );
  return i;
}
```

Since doubles cannot be template arguments we have to map doubles to integers. The natural way to do this mapping is the usage of the significand and exponent. Here is an example, that presents this idea:

```
template <int A, int B>
struct MetaDouble
{
};


double f( double d )
{
  return d*10;
}

template < int A, int B >
class _Test
{
```

```
  const double d;
public:
  _Test( double x ): d( x )
  {
    if ( get_value() != d )
      throw _Invalid();
  }

  double get_value() const
  {
    return A * pow( 10.0L, B );
  }
};

template <int A, int B>
_Test<A,B+1> f_test(MetaDouble<A, B> d)
{
  double dt = A*pow( 10.0L, B );
  return f( dt );
}
```

This framework can be easily extended in the way of C++ Standard Template Library (STL) [19]. We may use functor objects instead of the equality operator to make the framework more flexible because it can test more relations. We can take advantage of default template parameters of template classes. The following code snippet can be applied to the integers:

```
template <int N, class relation = std::equal_to<int> >
class _Test
{
  const int value;
  const relation rel;

public:
  _Test( int i ) : value( i )
  {
    if ( rel(N, value) )
      throw _Invalid();
  }
```

```
  int get_value() const
  {
    return value;
  }
};
```

4 Evaluation

In this section we argue for our approach. We describe pros and cons and present scenarios where our method is more powerful then the existing ones.

One the most fundamental advantages is that our framework does not need external tools, the only necessary tool is the compiler itself. Nevertheless, another important feature, that we compute the result at compilation time, so the runtime overhead is minimal. Of course, the compilation time is increased. The performance analysis of C++ template metaprograms is detailed in [13].

Our approach is able to detect and pursue the changes external APIs' interface. For instance, the type of return value has been changed, we do not need to modify the specifications. Just like the `max` example in 2 section, metaprograms can determine the type of return values, etc.

Domain-specific languages can be developed with the assistance of template metaprograms. Therefore, specification languages can be easily adopted to our approach. Users can select a specification language from the exisiting ones or develop new domain-specific languages for the specification [23]. The usual specification methods support only one specification language at all.

Moreover, metaprograms are written in an functional way, but runtime C++ programs are written in an imperative way. Therefore, testing approach and implementation is quite different. It is easy to focus on the results this way. A very useful advantage is that that our framework can be used for legacy code too.

Albeit there are some typical approaches which cannot be tested with our method. For instance, metaprograms cannot access database servers and metaprograms cannot deal with other runtime inputs. Files and requests cannot be managed with metaprograms. On the other hand, we can test the business logic of the programs: is the result correct if the input would be the specificated one. Also, calls of virtual methods cannot be handled at compilation time.

Our approach cannot facilitate the testing of multithreaded programs. Test-

ing concurrent programs is hard, but the compiler acts as a single-threaded non-deterministic interpreter.

5 Conclusions and future work

Testing is one of the most important methods to ensures programs' correctness. In this paper we argue for a new approach to test C++ programs. Our solution takes advantage of C++ template metaprogramming techniques in many ways. We have examined the pros and cons of our method.

After all, the most important task is to work out a set of special specification languages and generate standard compliant C++ metaprograms from these specifications.

In this paper we argue for a method that manages runtime entities at compilation time. With this method we tested runtime functions. Many other interesting properties should be managed in this way, for instance, the runtime complexity or the prospective exceptions.

Another important task is developing mapping between the runtime and compile time advanced datastructures. Node-based datastructures (like trees, linked lists) are also available in metalevel, but we have not mapped these structures to runtime akins. User-defined classes also may be mapped to the their compilation-time counterparts.

An other opportunity is that we take advantage of the metalevel and generate testcases at compilation time. In our approach the users specificate the test cases. It would be more convenient if the compiler could generate testcases which covers most of execution paths.

References

[1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming*, Addison-Wesley, Reading, MA, 2004. ⇒157

[2] A. Alexandrescu, *Modern C++ design: Generic programming and design patterns applied*, Addison-Wesley, Reading, MA, 2001. ⇒159

[3] A. H. Bagge, V. David, M. Haveraaen, The axioms strike back: Testing with concepts and axioms in C++, *Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE'09)*, Denver, CO, 2009, The ACM Digital Library, 15–24. ⇒155

[4] A.H. Bagge, M. Haveraaen, Axiom-based transformations: Optimisation and testing, *Electronic Notes in Theoret. Comput. Sci.*, **238,** 5 (2009) 17–33. ⇒155

[5] M. Biczó, K. Pócza, Z. Porkoláb, I. Forgács, A new concept of effective regression test generation in a C++ specific environment, *Acta Cybern.*, **18** (2008) 481–512. ⇒154

[6] Z. Borók-Nagy, V. Májer, J. Mihalicza, N. Pataki, Z. Porkoláb, Visualization of C++ template metaprograms, *Proc. Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, Timişoara, Romania, 2010, pp. 167–176. ⇒159

[7] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, W. Taha, DSL implementation in MetaOCaml, template Haskell, and C++, *Lecture Notes in Comput. Sci.*, **3016** (2004) 51–72. ⇒155

[8] G. Dévai, Meta programming on the proof level, *Acta Univ. Sapientiae Inform.*, **1,** 1 (2009) 15–34. ⇒155

[9] J. (Y). Gil, K. Lenz, Simple and safe SQL queries with C++ templates, *Science of Computer Programming*, **75,** 7 (2010) 573–595. ⇒155

[10] P. Hamill, *Unit test frameworks*, O'Reilly Media, Inc., Sebastopol, CA, 2004. ⇒155

[11] B. Karlsson, *Beyond the C++ standard library: An introduction to boost*, Addison-Wesley, Reading, MA, 2005. ⇒155, 159

[12] J. Mihalicza, N. Pataki, Z. Porkoláb, Á. Sipos, Towards more sophisticated access control, *Proc. 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, Tampere, Finland, 2009, pp. 117–131. ⇒155

[13] Z. Porkoláb, J. Mihalicza, N. Pataki, Á. Sipos, Analysis of profiling techniques for C++ template metaprograms, *Ann. Univ. Sci. Budapest. Sect. Comput.* **30** (2009) 97–116. ⇒164

[14] Z. Porkoláb, J. Mihalicza, Á. Sipos, Debugging C++ template metaprograms, *Proc. 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, Portland, OR, 2006, The ACM Digital Library, 255–264. ⇒155

[15] Z. Porkoláb, Á. Sinkovics, Domain-specific language integration with compile-time parser generator library, *Proc. 9th International Conference on Generative Programming and Component Engineering (GPCE' 10)*, Eindhoven, The Netherlands, 2010, The ACM Digital Library, pp. 137–146. ⇒162

[16] Á. Sinkovics, Unit testing of C++ template metaprograms, *Abstracts 8th Joint Conference on Mathematics and Computer Science (MACS'10)*, Komárno, Slovakia, June 14–17, 2010. ⇒155

[17] Á. Sipos, Z. Porkoláb, N. Pataki, V. Zsók, Meta<Fun> - Towards a functional-style interface for C++ template metaprograms, *Proc. 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007)*, Freiburg, Germany, 2007, 489–502. ⇒158, 162

[18] Á. Sipos, V. Zsók, EClean – An embedded functional language, *Electronic Notes in Theoret. Comput. Sci.*, **238,** 2 (2009) 47–58. ⇒155, 162

[19] B. Stroustrup, *The C++ programming language (Special edition)*, Addison-Wesley, Reading, MA, 2000. ⇒163

[20] Cs. Szabó, L. Samuelis, Observations on incrementality principle within the test preparation process, *Acta Univ. Sapientiae Inform.*, **1,** 1 (2009) 63–70. ⇒155

[21] Z. Szűgyi, N. Pataki, J. Mihalicza, Z. Porkoláb, C++ method utilities, *Proc. Tenth International Scientific Conference on Informatics (Informatics 2009)*, Herl'any, Slovakia, pp. 112–117. ⇒155

[22] Z. Szűgyi, N. Pataki, Sophisticated methods in C++, *Proc. International Scientific Conference on Computer Science and Engineering (CSE 2010)*, Kosice, Slovakia, pp. 93–100. ⇒155

[23] N. Vasudevan, L. Tratt, Comparative study of DSL tools, *Proc. Workshop on Generative Technologies 2010 (WGT 2010)*, Paphos, Cyprus, pp. 67–76. ⇒164

[24] T. L. Veldhuizen, Expression templates, *C++ Report*, **7,** 5 (1995) 26–31. ⇒159

Received: August 5, 2010 • Revised: October 15, 2010

Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 168–183

Computerized adaptive testing: implementation issues

Margit Antal
Sapientia Hungarian University of Transylvania, Department of Mathematics and Informatics
Tîrgu Mureş
email: manyi@ms.sapientia.ro

Levente Erős
Sapientia Hungarian University of Transylvania, Department of Mathematics and Informatics
Tîrgu Mureş
email: ideges@gmail.com

Attila Imre
Sapientia Hungarian University of Transylvania, Department of Human Sciences
Tîrgu Mureş
email: imatex@ms.sapientia.ro

Abstract. One of the fastest evolving field among teaching and learning research is students' performance evaluation. Computer based testing systems are increasingly adopted by universities. However, the implementation and maintenance of such a system and its underlying item bank is a challenge for an inexperienced tutor. Therefore, this paper discusses the advantages and disadvantages of Computer Adaptive Test (CAT) systems compared to Computer Based Test systems. Furthermore, a few item selection strategies are compared in order to overcome the item exposure drawback of such systems. The paper also presents our CAT system along its development steps. Besides, an item difficulty estimation technique is presented based on data taken from our self-assessment system.

Computing Classification System 1998: K.3.1
Mathematics Subject Classification 2010: 97U50
Key words and phrases: computerized adaptive testing, item response theory

1 Introduction

One of the fastest evolving field among teaching and learning research is students' performance evaluation. Web-based educational systems with integrated computer based testing are the easiest way of performance evaluation, so they are increasingly adopted by universities [3, 4, 9]. With the rapid growth of computer communications technologies, online testing is becoming more and more common. Moreover, limitless opportunities of computers will cause the disappearance of Paper and Pencil (PP) tests. Computer administered tests present multiple advantages compared to PP tests. First of all, various multimedia can be attached to test items, which is almost impossible in PP tests. Secondly, test evaluation is instantaneous. Moreover, computerized self-assessment systems can offer various hints, which help students' exam preparation.

This paper is structured in more sections. Section 2 presents Item Response Theory (IRT) and discusses the advantages and disadvantages of adaptive test systems. Section 3 is dedicated to the implementation issues. The presentation of the item bank is followed by simulations for item selection strategies in order to overcome the item exposure drawback. Then the architecture of our web-based CAT system is presented, which is followed by a proposal for item difficulty estimation. Finally, we present further research directions and give our conclusions.

2 Item Response Theory

Computerized test systems reveal new testing opportunities. One of them is the adaptive item selection tailored to the examinee's ability level, which is estimated iteratively through the answered test items. Adaptive test administration consists in the following steps: (i) start from an initial ability level, (ii) selection of the most appropriate test item and (iii) based on the examinee's answer re-estimation of their ability level. The last two steps are repeated until some ending conditions are satisfied. Adaptive testing research started in 1952 when Lord made an important observation: ability scores are test independent whereas observed scores are test dependent [6]. The next milestone was in 1960 when George Rasch described a few item response models in his book [11]. One of the described models, the one-parameter logistic model, became known as the Rasch model. The next decades brought many new applications based on Item Response Theory.

In the following we present the three-parameter logistic model. The basic

component of this model is the item characteristic function:

$$P(\Theta) = c + \frac{(1-c)}{1+e^{-Da(\Theta-b)}}, \tag{1}$$

where Θ stands for the examinee's ability, whose theoretical range is from $-\infty$ to ∞, but practically the range -3 to $+3$ is used. The three parameters are: a, discrimination; b, difficulty; c, guessing. Discrimination determines how well an item differentiates students near an ability level. Difficulty shows the place of an item along the ability scale, and guessing represents the probability of guessing the correct answer of the item [2]. Therefore guessing for a true/false item is always 0.5. $P(\Theta)$ is the probability of a correct response to the item as a function of ability level [6]. D is a scaling factor and typically the value 1.7 is used.

Figure 1 shows item response function for an item having parameters $a = 1, \quad b = 0.5, \quad c = 0.1$. For a deeper understanding of the discrimination parameter, see Figure 2, which illustrates three different items with the same difficulty ($b = 0.5$) and guessing ($c = 0.1$) but different discrimination parameters. The steepest curve corresponds to the highest discrimination ($a = 2.8$), and in the middle of the curve the probability of correct answer changes very rapidly as ability increases [2].

The one- and two-parameter logistic models can be obtained from equation (1), for example setting $c = 0$ results in the two-parameter model, while setting $c = 0$ and $a = 1$ gives us the one-parameter model.

Compared to the classical test theory, it is easy to realize the benefits of the former, which is able to propose the most appropriate item, based on item statistics reported on the same scale as ability [6].

Another component of the IRT model is the item information function, which shows the contribution of a particular item to the assessment of ability [6]. Item information functions are usually bell shaped functions, and in this paper we used the following (recommended in [12]):

$$I_i(\Theta) = \frac{P_i'(\Theta)^2}{P_i(\Theta)(1-P_i(\Theta))}, \tag{2}$$

where $P_i(\Theta)$ is the probability of a correct response to item i computed by equation (1), $P_i'(\Theta)$ is the first derivative of $P_i(\Theta)$, and $I_i(\Theta)$ is the item information function for item i.

High discriminating power items are the best choice as shown in Figure 3, which illustrates the item information functions for the three items shown in

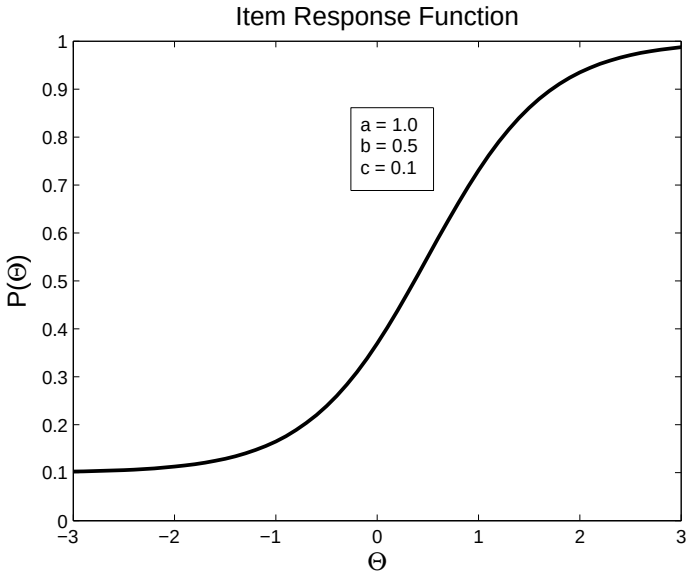


Figure 1: A three-parameter logistic model item characteristic function

Figure 2. All three functions are centered around the ability $\Theta = 0.5$, which is the same as the item difficulty.

Test information function I_{rr} is defined as the sum of item information functions. Two such functions are shown for a 20-item test selected by our adaptive test system: one for a high ability student (Figure 4) and another for a low ability student (Figure 5). The test shown in Figure 4 estimates students' ability near $\Theta = 2.0$, while the test in Figure 5 at $\Theta = -2.0$.

Test information function is also used for ability estimation error computation as shown in the following equation:

$$SE(\Theta) = \frac{1}{\sqrt{I_{rr}}}. \tag{3}$$

This error is associated with maximum likelihood ability estimation and is usually used for the stopping condition of adaptive testing.

For learner proficiency estimation Lord proposes an iterative approach [10], which is a modified version of the Newton-Raphson iterative method for solving equations. This approach starts with an initial ability estimate (usually a random value). After each item the ability is adjusted based on the response

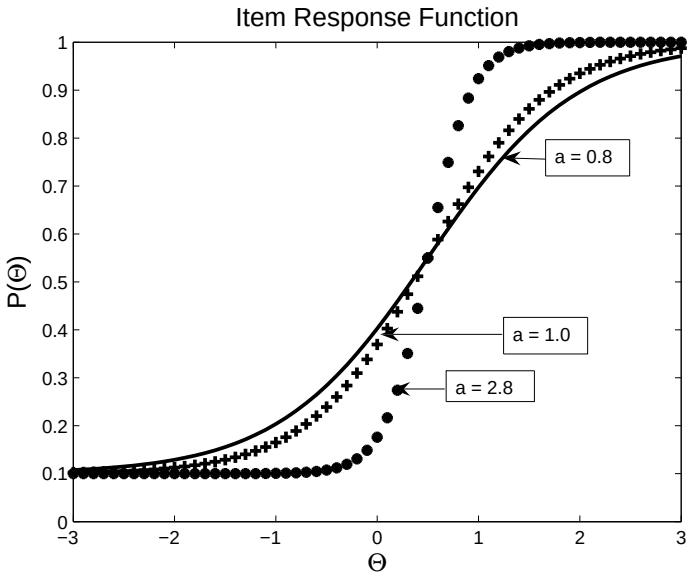


Figure 2: Item characteristic functions

given by the examinee. For example, after n questions the estimation is made according to the following equation:

$$\Theta_{n+1} = \Theta_n + \frac{\sum_{i=1}^{n} S_i(\Theta_n)}{\sum_{i=1}^{n} I_i(\Theta_n)} \tag{4}$$

where $S_i(\Theta)$ is computed using the following equation:

$$S_i(\Theta) = (u_i - P_i(\Theta)) \frac{P_i'(\Theta)}{P_i(\Theta)(1 - P_i(\Theta))}. \tag{5}$$

In equation (5) u_i represents the correctness of the ith answer, which is 0 for incorrect and 1 for correct answer. $P_i(\Theta)$ is the probability of correct answer for the ith item having the ability Θ (equation (2)), and $P_i'(\Theta)$ is its first derivative.

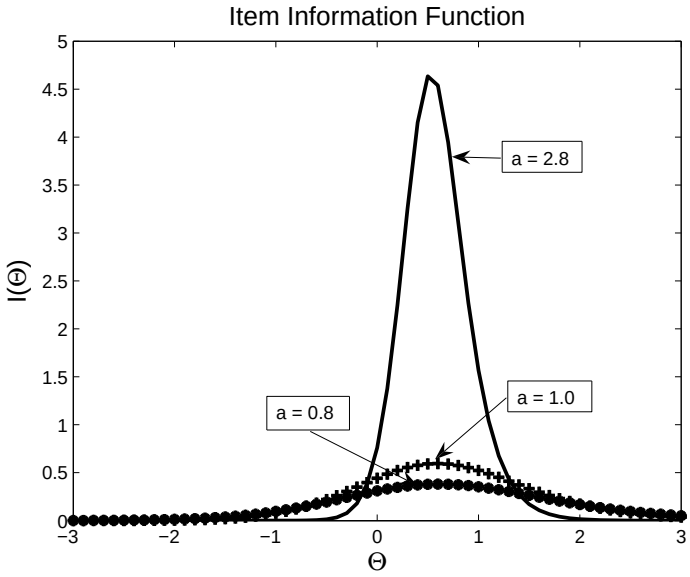


Figure 3: Item information functions

2.1 Advantages

In adaptive testing the best test item is selected at each step: the item having maximum information at the current estimate of the examinee's proficiency. The most important advantage of this method is that high ability level test-takers are not bored with easy test items, while low ability ones are not faced with difficult test items. A consequence of adapting the test to the examinee's ability level is that the same measurement precision can be realized with fewer test items.

2.2 Disadvantages

Along with the advantages offered by IRT, there are some drawbacks as well. The first drawback is the impossibility to estimate the ability in case of all correct or zero correct responses. These are the cases of either very high or very low ability students. In such cases the test item administration must be stopped after administering a minimum number of questions.

The second drawback is that the basic IRT algorithm is not aware of the test content, the question selection strategy does not take into consideration

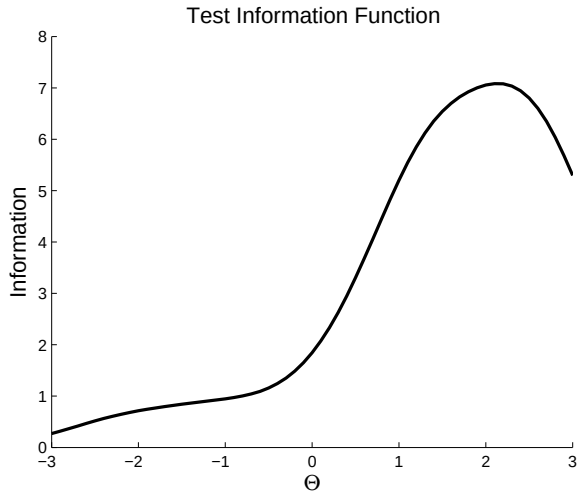


Figure 4: Test information function for a 20-item test generated for high ability students

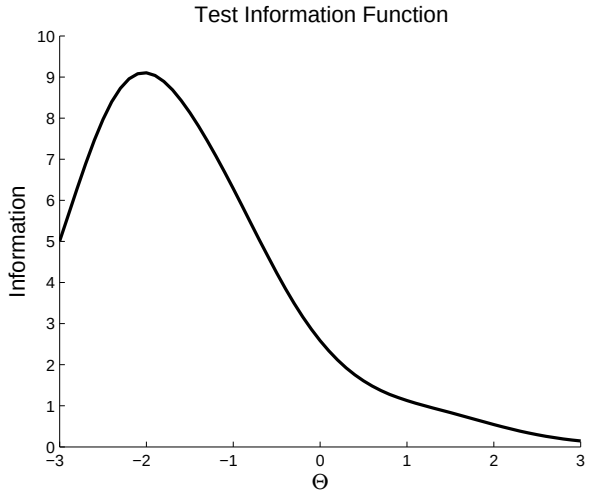


Figure 5: Test information function for a 20-item test generated for low ability students

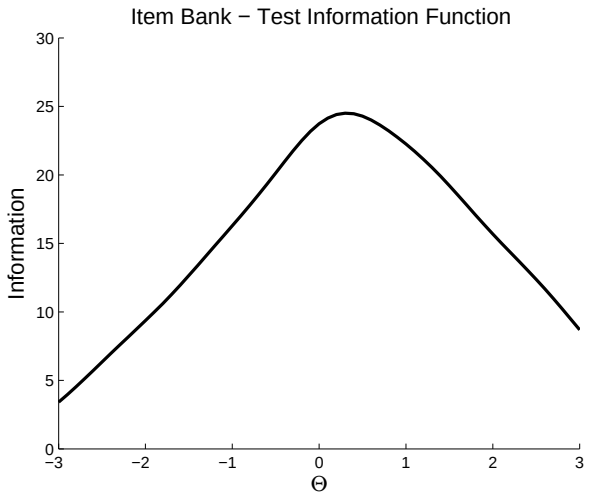


Figure 6: Test information function for all test items

to which topic a question belongs. However, sometimes this may be a requirement for generating tests assessing certain topics in a given curriculum. Huang proposed a content-balanced adaptive testing algorithm [7]. Another solution to the content balancing problem is the testlet approach proposed by Wainer and Kiely [15]. A testlet is a group of items from a single curriculum topic, which is developed as a unit. If an adaptive algorithm selects a testlet, then all the items belonging to that testlet will be presented to the examinee.

The third drawback, which is also the major one, is that IRT algorithms require serious item calibration. Despite the fact that the first calibration method was proposed by Alan Birnbaum in 1968 and has been implemented in computer programs such as BICAL (Wright and Mead, 1976) and LOGIST (Wingersky, Barton and Lord, 1982), the technique needs real measurement data in order to accurately estimate the parameters of the items. However, real measurement data are not always available for small educational institutions.

The fourth drawback is that several items from the item bank will be overexposed, while other test items will not be used at all. This requires item exposure control strategies. A good review of these strategies can be found in [5], discussing the strengths and weaknesses of each strategy. Stocking [13] made one of the first overviews of item exposure control strategies and classified them in two groups: (i) methods using a random component along the

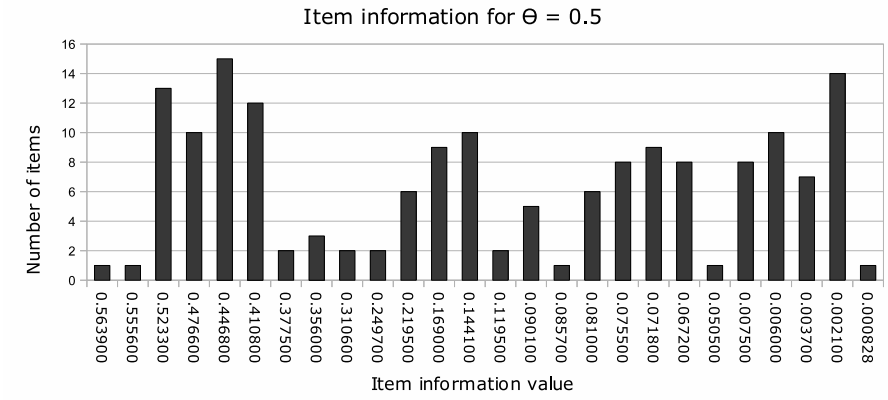


Figure 7: Item information clusters and their size

item selection method and (ii) methods using a parameter for each item to control its exposure. Randomization strategies control the frequency of item administration by selecting the next item from a group of items (e.g. out of the 5 best items). The second item exposure control strategy uses an exposure control parameter. In case of an item selection—due to its maximum information for the examinee's ability level—, the item will be administered only if its exposure control parameter allows it.

3 CAT implementation

3.1 The item bank

We have used our own item bank from our traditional computer based test system "Intelligent" [1]. The item bank parameters (a - discrimination, b - difficulty, c - pseudo guessing) were initialized by the tutor. We used 5 levels of difficulty from very easy to very difficult, which were scaled to the [-3,3] interval. The guessing parameter of an item was initialized by the ratio of the number of possible correct answers to the total number of possible answers. For example, it is 0.1 for an item having two correct answers out of five possible answers. Discrimination is difficult to set even for a tutor, therefore we used $a = 1$ for each item.

3.2 Simulations

In our implementation we have tried to overcome the disadvantages of IRT. We started to administer items adaptively only after the first five items. Ability (Θ) was initialized based on the number of correct answers given to these five items, which had been selected to include all levels of difficulty.

We used randomization strategies to overcome item exposure. Two randomization strategies were simulated. In the first one we selected the top ten items, i.e. the ten items having the highest item information. However, this is better than choosing the single best item, thus one must pay attention to the selection of the top ten items. There may be more items having the same item information for a given ability, therefore it is not always the best strategy choosing the first best item from a set of items with the same item information. To overcome this problem, in the second randomization strategy we computed the item information for all items that were not presented to the examinee and clustered the items having the same item information. The top ten items were selected using the items from these clusters. If the best cluster had less than ten items, the remainder items were selected from the next best cluster. If the best cluster had more than ten items, the ten items were selected randomly from the best cluster. For example, Figure 7 shows the 26 clusters of item information values constructed from 171 items for the ability of $\Theta = 0.5$. The best 10 items were selected by taking the items from the first two clusters (each having exactly 1 item) and selecting randomly another 8 items out of 13 from the third cluster.

Figure 8 shows the results from a simulation where we used an item bank with 171 items (test information function is shown in Figure 6 for all the 171 items), and we simulated 100 examinees using three item selection strategies: (i) best item (ii) random selection from the 10 best items (iii) random selection from the 10 best items and clustering. The three series in figure 8 are the frequencies of items obtained from the 100 simulated adaptive tests. Tests were terminated either when the number of administered items had exceeded 30 or the ability estimate had fallen outside the ability range. The latter were necessary for very high and very low ability students, where adaptive selection could not be used [2]. The examinee's answers were simulated by using a uniform random number generator, where the probability of correct answer was set to be equal to the probability of incorrect answer.

In order to be able to compare these item exposure control strategies, we computed the standard deviance of the frequency series shown in Figure 8. The standard deviance is $\sigma = 17.68$ for the first series not using any item

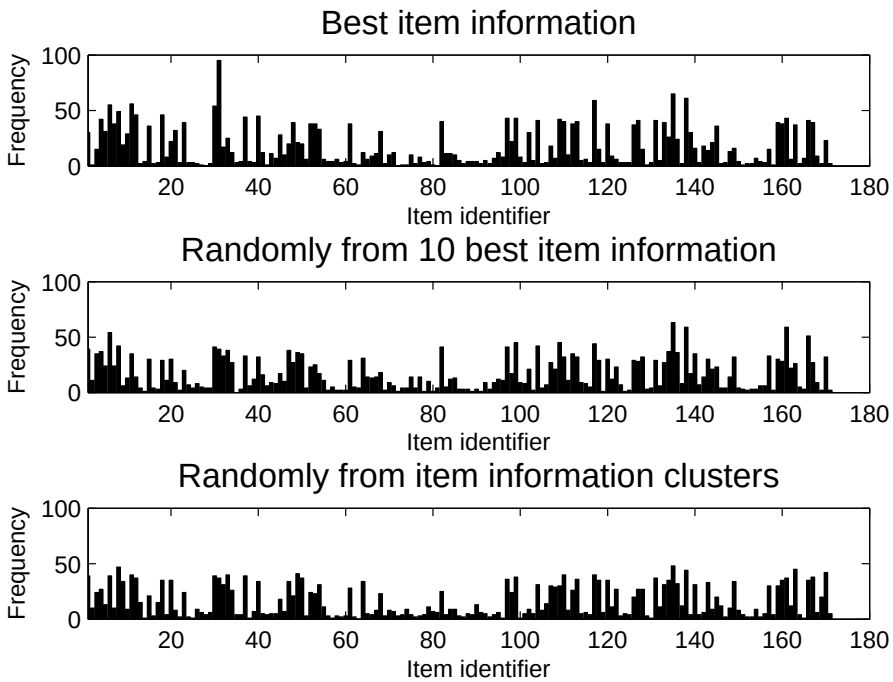


Figure 8: Item exposure with or without randomization control strategies

exposure control, it is $\sigma = 14.77$ for the second one, whereas for the third one is $\sigma = 14.13$. It is obvious that the third series is the best from the viewpoint of item exposure. Consequently, we will adopt this strategy in our distributed CAT implementation.

3.3 Distributed CAT

After the Matlab simulations we implemented our CAT system as a distributed application, using Java technologies on the server side and Adobe Flex on the client side. The general architecture of our system is shown in Figure 9. The administrative part is responsible for item bank maintenance, test scheduling, test results statistics and test termination criteria settings. In the near future we are planning to add an item calibration module.

The test part is used by examinees, where questions are administered ac-

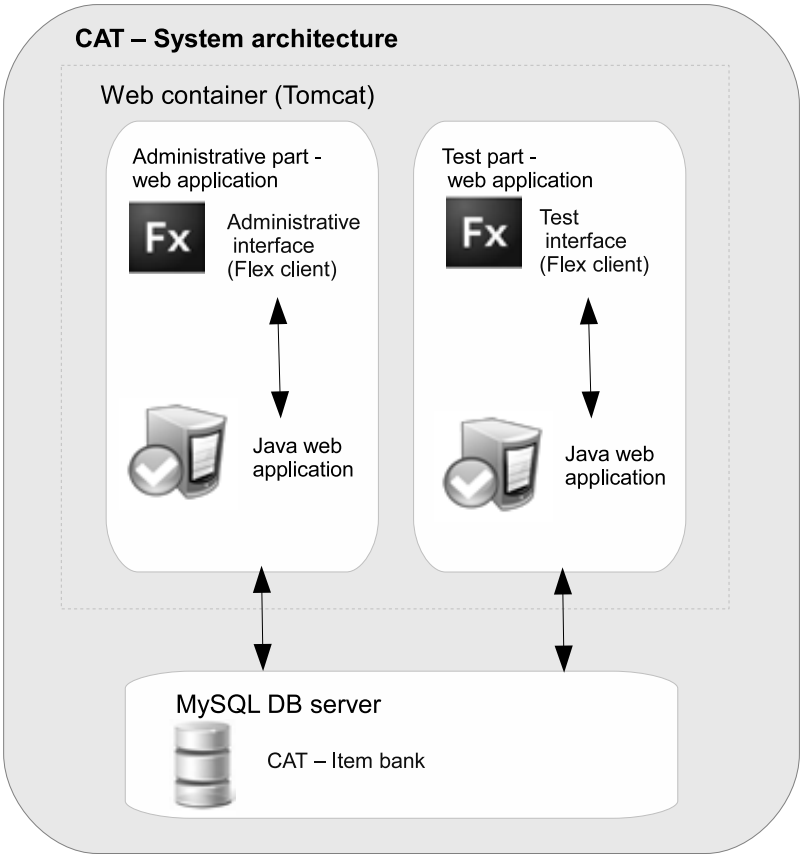


Figure 9: CAT-architecture

cording to settings. After having finished the test, the examinee may view both their test results and knowledge report.

3.4 Item difficulty estimation

Due to the lack of measurement data necessary for item calibration, we were not able to calibrate our item bank. However, 165 out of 171 items of our item bank were used in our self-assessment test system "Intelligent" in the previous term. Based on the data collected from this system, we propose a method for difficulty parameter estimation. Although there were no restrictions in using the self-assessment system, i.e. users could have answered an item several

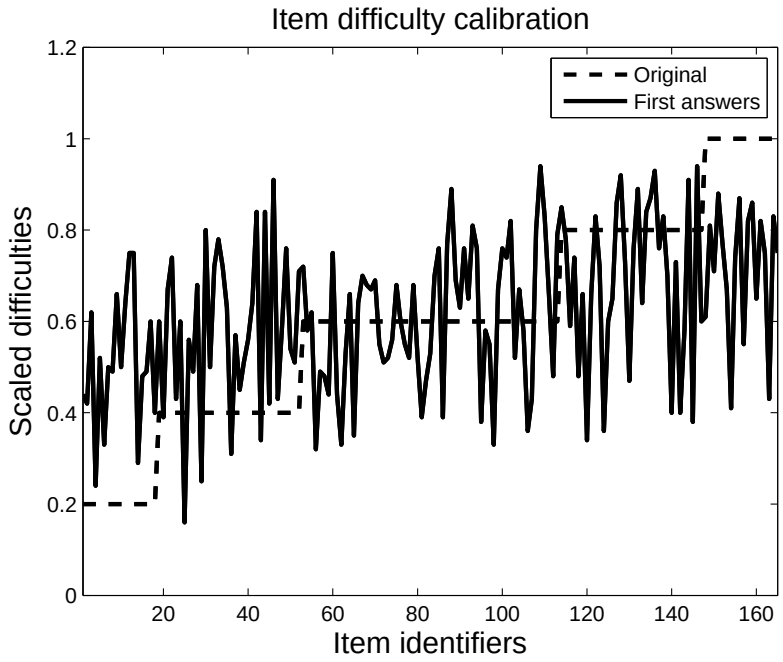


Figure 10: Item difficulty calibration

times, we consider that the first answer of each user could be relevant to the difficulty of the item.

Figure 10 shows the original item difficulty (set by the tutor) and the difficulty estimated by the first answer of each user. The original data series uses 5 difficulty levels scaled to the $[0, 1]$ interval. The elements of the "first answers" series were computed by the equation: $\frac{\text{all\_incorrect\_answers}}{\text{all\_answers}}$. We computed the mean difficulty for both series, and we obtained 0.60 for the original one and 0.62 for the estimated one. Conspicuous differences were found at the *very easy* and *very difficult* item difficulties.

4 Further research

At present we are working on the parameter estimation part of our CAT system. Although there are several item parameter calibration programs, this task must be taken very seriously because it influences measurement precision

directly. Item parameter estimation error is an active research topic, especially for fixed computer tests. For adaptive testing, this problem has been addressed by paper [8].

Researchers have empirically observed that examinees suitable for item difficulty estimations are almost useless when estimating item discrimination. Stocking [14] analytically derived the relationship between the examinee's ability and the accuracy of maximum likelihood item parameter estimation. She concluded that high ability examinees contribute more to difficulty estimation of difficult and very difficult items and less on easy and very easy items. She also concluded that only low ability examinees contribute to the estimation of guessing parameter and examinees, who are informative regarding item difficulty estimation, are not good for item discrimination estimation. Consequently, her results seem to be useful in our item calibration module.

5 Conclusions

In this paper we have described a computer adaptive test system based on Item Response Theory along its implementation issues. Our CAT system was implemented after one year experience with a computer based self-assessment system, which proved useful in configuring the parameters of the items. We started with the presentation of the exact formulas used by our working CAT system, followed by some simulations for item selection strategies in order to control item overexposure. We also presented the exact way of item parameter configuration based on the data taken from the self-assessment system.

Although we do not present measurements on a working CAT system, the implementation details presented in this paper could be useful for small institutions planning to introduce such a system for educational measurements on a small scale.

In the near future we would like to add an item calibration module to the administrative part of the system, taking into account the limited possibilities of small educational institutes.

References

[1] M. Antal, Sz. Koncz, Adaptive knowledge testing systems, *Proc. SZAMOKT XIX*, October 8–11, 2009, Tîrgu Mureş, Romania, pp. 159–164. ⇒176

[2] F. B. Baker, *The Basics of Item Response Theory*, ERIC Clearinghouse on Assessment and Evaluation, College Park, University of Maryland, 2001. ⇒170, 177

[3] M. Barla, M. Bielikova, A. B. Ezzeddinne, T. Kramar, M. Simko, O. Vozar, On the impact of adaptive test question selection for learning efficiency, *Computers & Educations*, **55,** 2 (2010) 846–857. ⇒169

[4] A. Baylari, G. A. Montazer, Design a personalized e-learning system based on item response theory and artificial neural network approach, *Expert Systems with Applications*, **36,** 4 (2009) 8013–8021. ⇒169

[5] E. Georgiadou, E. Triantafillou, A. Economides, A review of item exposure control strategies for computerized adaptive testing developed from 1983 to 2005, *Journal of Technology, Learning, and Assessment*, **5,** 8 (2007) 33 p. ⇒175

[6] R. K. Hambleton, R. W. Jones, Comparison of classical test theory and item response theory and their applications to test development, *ITEMS – Instructional Topics in Educational Measurement*, 253–262. ⇒169, 170

[7] S. Huang, A content-balanced adaptive testing algorithm for computer-based training systems, in: *Intelligent Tutoring Systems* (eds. C. Frasson, G. Gauthier, A. Lesgold), Springer, Berlin, 1996, pp. 306–314. ⇒175

[8] W. J. Linden, C. A. W. Glas, *Capitalization on item calibration error in computer adaptive testing*, LSAC Research Report 98-04, 2006, 14 p. ⇒181

[9] M. Lilley, T. Barker, C. Britton, The development and evaluation of a software prototype for computer-adaptive testing, *Computers & Education*, **43,** 1–2 (2004) 109–123. ⇒169

[10] F. M. Lord, *Application of item response theory to practical testing problems*, Lawrence Erlbaum Publishers, NJ, 1980. ⇒171

[11] G. Rasch, *Probabilistic models for some intelligence and attainment tests*, Danish Institute for Educational Research, Copenhagen, 1960. ⇒169

[12] L. M. Rudner, *An online, interactive, computer adaptive testing tutorial*, 1998, http://echo.edres.org:8080/scripts/cat/catdemo.htm ⇒170

[13] M. L. Stocking, *Controlling item exposure rates in a realistic adaptive testing paradigm*, Technical Report RR 3-2, Educational Testing Service, Princeton, NJ, 1993. ⇒175

[14] M. L. Stocking, Specifying optimum examinees for item parameter estimation in item response theory, *Psychometrika*, **55,** 3 (1990) 461–475. ⇒181

[15] H. Wainer, G. L. Kiely, Item clusters and computerized adaptive testing: A case for testlets, *Journal of Educational Measurement*, **24,** 3 (1987) 185–201. ⇒175

Received: August 23, 2010 • Revised: November 3, 2010

Acta Univ. Sapientiae, Informatica, **2**, 2 (2010) 184–193

Score lists in multipartite hypertournaments

Shariefuddin Pirzada
Department of Mathematics, University of Kashmir, India, and King Fahd University of Petroleum and Minerals, Saudi Arabia
email: sdpirzada@yahoo.co.in

Guofei Zhou
Department of Mathematics, Nanjing University, Nanjing, China
email: gfzhou@nju.edu.cn

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

Abstract. Given non-negative integers n_i and α_i with $0 \leq \alpha_i \leq n_i$ $(i = 1, 2, \ldots, k)$, an $[\alpha_1, \alpha_2, \ldots, \alpha_k]$-k-partite hypertournament on $\sum_1^k n_i$ vertices is a $(k+1)$-tuple $(U_1, U_2, \ldots, U_k, E)$, where U_i are k vertex sets with $|U_i| = n_i$, and E is a set of $\sum_1^k \alpha_i$-tuples of vertices, called arcs, with exactly α_i vertices from U_i, such that any $\sum_1^k \alpha_i$ subset $\cup_1^k U_i'$ of $\cup_1^k U_i$, E contains exactly one of the $\left(\sum_1^k \alpha_i\right)! \sum_1^k \alpha_i$-tuples whose entries belong to $\cup_1^k U_i'$. We obtain necessary and sufficient conditions for k lists of non-negative integers in non-decreasing order to be the losing score lists and to be the score lists of some k-partite hypertournament.

1 Introduction

Hypergraphs are generalizations of graphs [1]. While edges of a graph are pairs of vertices of the graph, edges of a hypergraph are subsets of the vertex set,

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 05C65
Key words and phrases: hypergraph, hypertournament, multi hypertournament, score, losing score.

consisting of at least two vertices. An edge consisting of k vertices is called a k-edge. A k-hypergraph is a hypergraph all of whose edges are k-edges. A k-hypertournament is a complete k-hypergraph with each k-edge endowed with an orientation, that is, a linear arrangement of the vertices contained in the hyperedge. Instead of scores of vertices in a tournament, Zhou et al. [13] considered scores and losing scores of vertices in a k-hypertournament, and derived a result analogous to Landau's theorem [6]. The score $s(v_i)$ or s_i of a vertex v_i is the number of arcs containing v_i and in which v_i is not the last element, and the losing score $r(v_i)$ or r_i of a vertex v_i is the number of arcs containing v_i and in which v_i is the last element. The score sequence (losing score sequence) is formed by listing the scores (losing scores) in non-decreasing order.

The following characterizations of score sequences and losing score sequences in k-hypertournaments can be found in G. Zhou et al. [12].

Theorem 1 *Given two positive integers n and k with $n \geq k > 1$, a non-decreasing sequence $R = [r_1, r_2, \ldots, r_n]$ of non-negative integers is a losing score sequence of some k-hypertournament if and only if for each j,*

$$\sum_{i=1}^{j} r_i \geq \binom{j}{k},$$

with equality when $j = n$.

Theorem 2 *Given positive integers n and k with $n \geq k > 1$, a non-decreasing sequence $S = [s_1, s_2, \ldots, s_n]$ of non-negative integers is a score sequence of some k-hypertournament if and only if for each j,*

$$\sum_{i=1}^{j} s_i \geq j\binom{n-1}{k-1} + \binom{n-j}{k} - \binom{n}{k},$$

with equality when $j = n$.

Some recent work on the reconstruction of tournaments can be found in the papers due to A. Iványi [3, 4]. Some more results on k-hypertournaments can be found in [2, 5, 9, 10, 11, 13]. The analogous results of Theorem 1 and Theorem 2 for $[h, k]$-bipartite hypertournaments can be found in [7] and for $[\alpha, \beta, \gamma]$-tripartite hypertournaments in [8].

Throughout this paper i takes values from 1 to k and j_i takes values from 1 to n_i, unless otherwise stated.

A k-partite hypergraph is a generalization of k-partite graph. Given non-negative integers n_i and α_i, $(i = 1, 2, \ldots, k)$ with $n_i \geq \alpha_i \geq 0$ for each i, an $[\alpha_1, \alpha_2, \ldots, \alpha_k]$-k-partite hypertournament (or briefly k-partite hypertournament) M of order $\sum_1^k n_i$ consists of k vertex sets U_i with $|U_i| = n_i$ for each i, $(1 \leq i \leq k)$ together with an arc set E, a set of $\sum_1^k \alpha_i$-tuples of vertices, with exactly α_i vertices from U_i, called arcs such that any $\sum_1^k \alpha_i$ subset $\cup_1^k U_i'$ of $\cup_1^k U_i$, E contains exactly one of the $\left(\sum_1^k \alpha_i\right)$ $\sum_1^k \alpha_i$-tuples whose α_i entries belong to U_i'.

Let $e = (u_{11}, u_{12}, \ldots, u_{1\alpha_1}, u_{21}, u_{22}, \ldots, u_{2\alpha_2}, \ldots, u_{k1}, u_{k2}, \ldots, u_{k\alpha_k})$, with $u_{ij_i} \in U_i$ for each i, $(1 \leq i \leq k, 1 \leq j_i \leq \alpha_i)$, be an arc in M and let $h < t$, we let $e(u_{1h}, u_{1t})$ denote to be the new arc obtained from e by interchanging u_{1h} and u_{1t} in e. An arc containing α_i vertices from U_i for each i, $(1 \leq i \leq k)$ is called an $(\alpha_1, \alpha_2, \ldots, \alpha_k)$-arc.

For a given vertex $u_{ij_i} \in U_i$ for each i, $1 \leq i \leq k$ and $1 \leq j_i \leq \alpha_i$, the score $d_M^+(u_{ij_i})$ (or simply $d^+(u_{ij_i})$) is the number of $\sum_1^k \alpha_i$-arcs containing u_{ij_i} and in which u_{ij_i} is not the last element. The losing score $d_M^-(u_{ij_i})$ (or simply $d^-(u_{ij_i})$) is the number of $\sum_1^k \alpha_i$-arcs containing u_{ij_i} and in which u_{ij_i} is the last element. By arranging the losing scores of each vertex set U_i separately in non-decreasing order, we get k lists called losing score lists of M and these are denoted by $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$ for each i, $(1 \leq i \leq k)$. Similarly, by arranging the score lists of each vertex set U_i separately in non-decreasing order, we get k lists called score lists of M which are denoted as $S_i = [s_{ij_i}]_{j_i=1}^{n_i}$ for each i $(1 \leq i \leq k)$.

2 Main results

The following two theorems are the main results.

Theorem 3 *Given* k *non-negative integers* n_i *and* k *non-negative integers* α_i *with* $1 \leq \alpha_i \leq n_i$ *for each* i $(1 \leq i \leq k)$*, the* k *non-decreasing lists* $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$ *of non-negative integers are the losing score lists of a* k*-partite hypertournament if and only if for each* p_i $(1 \leq i \leq k)$ *with* $p_i \leq n_i$*,*

$$\sum_{i=1}^{k} \sum_{j_i=1}^{p_i} r_{ij_i} \geq \prod_{i=1}^{k} \binom{p_i}{\alpha_i}, \tag{1}$$

with equality when $p_i = n_i$ *for each* i $(1 \leq i \leq k)$.

Theorem 4 *Given* k *non-negative integers* n_i *and* k *non-negative integers* α_i *with* $0 \le \alpha_i \le n_i$ *for each* i $(1 \le i \le k)$*, the* k *non-decreasing lists* $S_i = [s_{ij_i}]_{j_i=1}^{n_i}$ *of non-negative integers are the score lists of a* k*-partite hypertournament if and only if for each* p_i*,* $(1 \le i \le k)$ *with* $p_i \le n_i$

$$\sum_{i=1}^{k}\sum_{j_i=1}^{p_i} s_{ij_i} \ge \left(\sum_{i=1}^{k}\frac{\alpha_i p_i}{n_i}\right)\left(\prod_{i=1}^{k}\binom{n_i}{\alpha_i}\right)+\prod_{i=1}^{k}\binom{n_i-p_i}{\alpha_i}-\prod_{i=1}^{k}\binom{n_i}{\alpha_i}, \quad (2)$$

with equality when $p_i = n_i$ *for each* i $(1 \le i \le k)$.

We note that in a k-partite hypertournament M, there are exactly $\prod_{i=1}^{k}\binom{n_i}{\alpha_i}$ arcs and in each arc only one vertex is at the last entry. Therefore,

$$\sum_{i=1}^{k}\sum_{j_i=1}^{n_i} d_M^-(u_{ij_i}) = \prod_{i=1}^{k}\binom{n_i}{\alpha_i}.$$

In order to prove the above two theorems, we need the following Lemmas.

Lemma 5 *If* M *is a* k*-partite hypertournament of order* $\sum_1^k n_i$ *with score lists* $S_i = [s_{ij_i}]_{j_i=1}^{n_i}$ *for each* i $(1 \le i \le k)$*, then*

$$\sum_{i=1}^{k}\sum_{j_i=1}^{n_i} s_{ij_i} = \left[\left(\sum_{1=1}^{k}\alpha_i\right)-1\right]\prod_{i=1}^{k}\binom{n_i}{\alpha_i}.$$

Proof. We have $n_i \ge \alpha_i$ for each i $(1 \le i \le k)$. If r_{ij_i} is the losing score of $u_{ij_i} \in U_i$, then

$$\sum_{i=1}^{k}\sum_{j_i=1}^{n_i} r_{ij_i} = \prod_{i=1}^{k}\binom{n_i}{\alpha_i}.$$

The number of $[\alpha_i]_1^k$ arcs containing $u_{ij_i} \in U_i$ for each i, $(1 \le i \le k)$, and $1 \le j_i \le n_i$ is

$$\frac{\alpha_i}{n_i}\prod_{t=1}^{k}\binom{n_t}{\alpha_t}.$$

Thus,

$$\begin{aligned}\sum_{i=1}^{k}\sum_{j_i=1}^{n_i} s_{ij_i} &= \sum_{i=1}^{k}\sum_{j_i=1}^{n_i}\left(\frac{\alpha_i}{n_i}\right)\prod_{1}^{k}\binom{n_t}{\alpha_t} - \binom{n_i}{\alpha_i} \\ &= \left(\sum_{i=1}^{k}\alpha_i\right)\prod_{1}^{k}\binom{n_t}{\alpha_t} - \prod_{1}^{k}\binom{n_i}{\alpha_i} \\ &= \left[\left(\sum_{1=1}^{k}\alpha_i\right) - 1\right]\prod_{1}^{k}\binom{n_i}{\alpha_i}.\end{aligned}$$

□

Lemma 6 *If* $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$ $(1 \le i \le k)$ *are* k *losing score lists of a* k*-partite hypertournament* M*, then there exists some* h *with* $r_{1h} < \frac{\alpha_1}{n_1}\prod_1^k \binom{n_p}{\alpha_p}$ *so that* $R'_1 = [r_{11}, r_{12}, \ldots, r_{1h}+1, \ldots, r_{1n_1}]$, $R'_s = [r_{s1}, r_{s2}, \ldots, r_{st}-1, \ldots, r_{sn_s}]$ *(*$2 \le s \le k$*) and* $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$*, (*$2 \le i \le k$*),* $i \ne s$ *are losing score lists of some* k*-partite hypertournament,* t *is the largest integer such that* $r_{s(t-1)} < r_{st} = \ldots = r_{sn_s}$.

Proof. Let $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$ $(1 \le i \le k)$ be losing score lists of a k-partite hypertournament M with vertex sets $U_i = \{u_{i1}, u_{i2}, \ldots, u_{ij_i}\}$ so that $d^-(u_{ij_i}) = r_{ij_i}$ for each i $(1 \le i \le k,\ 1 \le j_i \le n_i)$.

Let h be the smallest integer such that

$$r_{11} = r_{12} = \ldots = r_{1h} < r_{1(h+1)} \le \ldots \le r_{1n_1}$$

and t be the largest integer such that

$$r_{s1} \le r_{s2} \le \ldots \le r_{s(t-1)} < r_{st} = \ldots = r_{sn_s}$$

Now, let

$$R'_1 = [r_{11}, r_{12}, \ldots, r_{1h} + 1, \ldots, r_{1n_1}],$$

$$R'_s = [r_{s1}, r_{s2}, \ldots, r_{st} - 1, \ldots, r_{sn_s}$$

$(2 \le s \le k)$, and $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$, $(2 \le i \le k)$, $i \ne s$.

Clearly, R'_1 and R'_s are both in non-decreasing order.

Since $r_{1h} < \frac{\alpha_1}{n_1}\prod_1^k \binom{n_p}{\alpha_p}$, there is at least one $[\alpha_i]_1^k$-arc e containing both u_{1h} and u_{st} with u_{st} as the last element in e, let $e' = (u_{1h}, u_{st})$. Clearly, R'_1, R'_s

and $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$ for each i $(2 \leq i \leq k)$, $i \neq s$ are the k losing score lists of $M' = (M - e) \cup e'$. □

The next observation follows from Lemma 6, and the proof can be easily established.

Lemma 7 *Let* $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$, $(1 \leq i \leq k)$ *be* k *non-decreasing sequences of non-negative integers satisfying (1). If* $r_{1n_1} < \frac{\alpha_1}{n_1} \prod_1^k \binom{n_t}{\alpha_t}$, *then there exists* s *and* t *(*$2 \leq s \leq k$*),* $1 \leq t \leq n_s$ *such that* $R'_1 = [r_{11}, r_{12}, \ldots, r_{1h} + 1, \ldots, r_{1n_1}]$, $R'_s = [r_{s1}, r_{s2}, \ldots, r_{st} - 1, \ldots, r_{sn_s}]$ *and* $R_i = [r_{ij_i}]_{j_i=1}^{n_i}$, *(*$2 \leq i \leq k$*),* $i \neq s$ *satisfy (1).*

Proof of Theorem 3. Necessity. Let R_i, $(1 \leq i \leq k)$ be the k losing score lists of a k-partite hypertournament $M(U_i, 1 \leq i \leq k)$. For any p_i with $\alpha_i \leq p_i \leq n_i$, let $U'_i = \{u_{ij_i}\}_{j_i=1}^{p_i} (1 \leq i \leq k)$ be the sets of vertices such that $d^-(u_{ij_i}) = r_{ij_i}$ for each $1 \leq j_i \leq p_i$, $1 \leq i \leq k$. Let M' be the k-partite hypertournament formed by U'_i for each i $(1 \leq i \leq k)$.

Then,

$$\sum_{i=1}^{k} \sum_{j_i=1}^{p_i} r_{ij_i} \geq \sum_{i=1}^{k} \sum_{j_i=1}^{p_i} d^-_{M'}(u_{ij_i}) = \prod_{1}^{k} \binom{p_t}{\alpha_t}.$$

Sufficiency. We induct on n_1, keeping $n_2, \ldots, n_k$ fixed. For $n_1 = \alpha_1$, the result is obviously true. So, let $n_1 > \alpha_1$, and similarly $n_2 > \alpha_2, \ldots, n_k > \alpha_k$. Now,

$$\begin{aligned} r_{1n_1} &= \sum_{i=1}^{k} \sum_{j_i=1}^{n_i} r_{ij_i} - \left(\sum_{j_1=1}^{n_1-1} r_{1j_1} + \sum_{i=2}^{k} \sum_{j_i=1}^{n_i} r_{ij_i} \right) \\ &\leq \prod_{1}^{k} \binom{n_t}{\alpha_t} - \binom{n_1 - 1}{\alpha_1} \prod_{2}^{k} \binom{n_t}{\alpha_t} \\ &= \left[\binom{n_1}{\alpha_1} - \binom{n_1 - 1}{\alpha_1} \right] \prod_{2}^{k} \binom{n_t}{\alpha_t} \\ &= \binom{n_1 - 1}{\alpha_1 - 1} \prod_{2}^{k} \binom{n_t}{\alpha_t}. \end{aligned}$$

We consider the following two cases.

Case 1. $r_{1n_1} = \binom{n_1-1}{\alpha_1-1}\prod_2^k \binom{n_t}{\alpha_t}$. Then,

$$\sum_{j_1=1}^{n_1-1} r_{1j_1} + \sum_{i=2}^{k}\sum_{j_i=1}^{n_i} r_{ij_i} = \sum_{i=1}^{k}\sum_{j_i=1}^{n_i} r_{ij_i} - r_{1n_1}$$

$$= \prod_1^k \binom{n_t}{\alpha_t} - \binom{n_1-1}{\alpha_1-1}\prod_2^k \binom{n_t}{\alpha_t}$$

$$= \left[\binom{n_1}{\alpha_1} - \binom{n_1-1}{\alpha_1-1}\right]\prod_2^k \binom{n_t}{\alpha_t}$$

$$= \binom{n_1-1}{\alpha_1}\prod_2^k \binom{n_t}{\alpha_t}.$$

By induction hypothesis $[r_{11}, r_{12}, \ldots, r_{1(n_1-1)}]$, $R_2, \ldots, R_k$ are losing score lists of a k-partite hypertournament $M'(U_1', U_2, \ldots, U_k)$ of order $\left(\sum_{i=1}^k n_i\right) - 1$. Construct a k-partite hypertournament M of order $\sum_{i=1}^k n_i$ as follows. In M', let $U_1' = \{u_{11}, u_{12}, \ldots, u_{1(n_1-1)}\}, U_i = \{u_{ij_i}\}_{j_i=1}^{n_i}$ for each i, $(2 \le i \le k)$. Adding a new vertex u_{1n_1} to U_1', for each $\left(\sum_{i=1}^k \alpha_i\right)$-tuple containing u_{1n_1}, arrange u_{1n_1} on the last entry. Denote E_1 to be the set of all these $\binom{n_1-1}{\alpha_1-1}\prod_2^k \binom{n_t}{\alpha_t}$ $\left(\sum_{i=1}^k \alpha_i\right)$-tuples. Let $E(M) = E(M') \cup E_1$. Clearly, R_i for each i, $(1 \le i \le k)$ are the k losing score lists of M.

Case 2. $r_{1n_1} < \binom{n_1-1}{\alpha_1-1}\prod_2^k \binom{n_t}{\alpha_t}$.

Applying Lemma 7 repeatedly on R_1 and keeping each R_i, $(2 \le i \le k)$ fixed until we get a new non-decreasing list $R_1' = [r_{11}', r_{12}', \ldots, r_{1n_1}']$ in which now $r_{1n_1}' = \binom{n_1-1}{\alpha_1-1}\prod_2^k \binom{n_t}{\alpha_t}$. By Case 1, R_1', R_i $(2 \le i \le k)$ are the losing score lists of a k-partite hypertournament. Now, apply Lemma 6 on R_1', R_i $(2 \le i \le k)$ repeatedly until we obtain the initial non-decreasing lists R_i for each i $(1 \le i \le k)$. Then by Lemma 6, R_i for each i $(1 \le i \le k)$ are the losing score lists of a k-partite hypertournament. □

Proof of Theorem 4. Let $S_i = [s_{ij_i}]_{j_i=1}^{n_i} (1 \le i \le k)$ be the k score lists of a k-partite hypertournament $M(U_i, 1 \le i \le k)$, where $U_i = \{u_{ij_i}\}_{j_i=1}^{n_i}$ with

$d^+_M(u_{ij_i}) = s_{ij_i}$, for each i, $(1 \le i \le k)$. Clearly,
$d^+(u_{ij_i}) + d^-(u_{ij_i}) = \frac{\alpha_i}{n_i} \prod_1^k \binom{n_t}{\alpha_t}$, $(1 \le i \le k, 1 \le j_i \le n_i)$.

Let $r_{i(n_i+1-j_i)} = d^-(u_{ij_i})$, $(1 \le i \le k, 1 \le j_i \le n_i)$.

Then $R_i = [r_{ij_i}]_{j_i=1}^{n_i} (i = 1, 2, \ldots, k)$ are the k losing score lists of M. Conversely, if R_i for each i $(1 \le i \le k)$ are the losing score lists of M, then S_i for each i, $(1 \le i \le k)$ are the score lists of M. Thus, it is enough to show that conditions (1) and (2) are equivalent provided $s_{ij_i} + r_{i(n_i+1-j_i)} = \left(\frac{\alpha_i}{n_i}\right) \prod_1^k \binom{n_t}{\alpha_t}$, for each i $(1 \le i \le k$ and $1 \le j_i \le n_i)$.

First assume (2) holds. Then,

$$\begin{aligned}
\sum_{i=1}^{k} \sum_{j_i=1}^{p_i} r_{ij_i} &= \sum_{i=1}^{k} \sum_{j_i=1}^{p_i} \left(\frac{\alpha_i}{n_i}\right) \left(\prod_1^k \binom{n_t}{\alpha_t}\right) - \sum_{i=1}^{k} \sum_{j_i=1}^{p_i} s_{i(n_i+1-j_i)} \\
&= \sum_{i=1}^{k} \sum_{j_i=1}^{p_i} \left(\frac{\alpha_i}{n_i}\right) \left(\prod_1^k \binom{n_t}{\alpha_t}\right) - \left[\sum_{i=1}^{k} \sum_{j_i=1}^{n_i} r_{ij_i} - \sum_{i=1}^{k} \sum_{j_i=1}^{n_i-p_i} s_{ij_i}\right] \\
&\ge \left[\sum_{i=1}^{k} \sum_{j_i=1}^{p_i} \left(\frac{\alpha_i}{n_i}\right) \left(\prod_1^k \binom{n_t}{\alpha_t}\right)\right] \\
&\quad - \left[\left(\left(\sum_1^k \alpha_i\right) - 1\right) \prod_1^k \binom{n_i}{\alpha_i}\right] \\
&\quad + \sum_{i=1}^{k} (n_i - p_i) \left(\frac{\alpha_i}{n_i}\right) \prod_1^k \binom{n_t}{\alpha_t} \\
&\quad + \prod_1^k \binom{n_i - (n_i - p_i)}{\alpha_i} - \prod_1^k \binom{n_i}{\alpha_i} \\
&= \prod_1^k \binom{n_i}{\alpha_i},
\end{aligned}$$

with equality when $p_i = n_i$ for each i $(1 \le i \le k)$. Thus (1) holds.

Now, when (1) holds, using a similar argument as above, we can show that (2) holds. This completes the proof. □

Acknowledgements

The research of the third author was supported by the European Union and the European Social Fund under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

[1] C. Berge, *Graphs and hypergraphs*, translated from French by E. Minieka, North-Holland Mathematical Library 6, North-Holland Publishing Co., Amsterdam, London, 1973. ⇒184

[2] D. Brcanov, V. Petrovic, Kings in multipartite tournaments and hypertournaments, *Numerical Analysis and Applied Mathematics: International Conference on Numerical Analysis and Applied Mathematics 2009*: **1, 2**, AIP Conference Proceedings, **1168** (2009) 1255–1257. ⇒185

[3] A. Iványi Reconstruction of complete interval tournaments, *Acta Univ. Sapientiae Inform.*, **1,** 1 (2009) 71–88. ⇒185

[4] A. Iványi, Reconstruction of complete interval tournaments II, *Acta Univ. Sapientiae Math.*, **2**, 1 (2010) 47–71. ⇒185

[5] Y. Koh, S. Ree, Score sequences of hypertournament matrices, On k-hypertournament matrices, *Linear Algebra Appl.*, **373** (2003) 183–195. ⇒185

[6] H. G. Landau, On dominance relations and the structure of animal societies. III. The condition for a score structure, *Bull. Math. Biophys.,* **15** (1953) 143–148. ⇒185

[7] S. Pirzada, T. A. Chishti, T. A. Naikoo, Score lists in $[\mathsf{h}, \mathsf{k}]$-bipartite hypertournaments, *Discrete Math. Appl.,* **19,** 3 (2009) 321–328. ⇒185

[8] S. Pirzada, T. A. Naikoo, G. Zhou, Score lists in tripartite hypertournaments, *Graphs and Comb.,* **23,** 4 (2007) 445–454. ⇒185

[9] S. Pirzada G. Zhou, Score sequences in oriented k-hypergraphs, *Eur. J. Pure Appl. Math.,* **1,** 3 (2008) 10–20. ⇒185

[10] S. Pirzada, G. Zhou, On k-hypertournament losing scores, *Acta Univ. Sapientiae Inform.,* **2,** 1 (2010) 5–9. ⇒185

[11] C. Wang, G. Zhou, Note on degree sequences of k-hypertournaments. *Discrete Math.,* **308,** 11 (2008) 2292–2296. ⇒185

[12] G. Zhou, T. Yao, K. Zhang, On score sequences of k-tournaments. *European J. Comb.*, **21**, 8 (2000) 993–1000. ⇒185

[13] G. Zhou, On score sequences of k-tournaments. *J. Appl. Math. Comput.,* **27** (2008) 149–158. ⇒185

Received: June 23, 2010 • Revised: October 21, 2010

Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 194–209

Pumping lemmas for linear and nonlinear context-free languages

Dedicated to Pál Dömösi on his 65th birthday

Géza Horváth
University of Debrecen
email: geza@inf.unideb.hu

Benedek Nagy
University of Debrecen
email: nbenedek@inf.unideb.hu

Abstract. Pumping lemmas are created to prove that given languages are not belong to certain language classes. There are several known pumping lemmas for the whole class and some special classes of the context-free languages. In this paper we prove new, interesting pumping lemmas for special linear and context-free language classes. Some of them can be used to pump regular languages in two place simultaneously. Other lemma can be used to pump context-free languages in arbitrary many places.

1 Introduction

The formal language theory and generative grammars form one of the basics of the field of theoretical computer science [5, 9]. Pumping lemmas play important role in formal language theory [3, 4]. One can prove that a language does not belong to a given language class. There are well-known pumping lemmas, for example, for regular and context-free languages. The first and most basic pumping lemma is introduced by Bar-Hillel, Perles, and Shamir in 1961 for context-free languages [3]. Since that time many pumping lemmas are introduced for various language classes. Some of them are easy to use/prove, some of them are more complicated. Sometimes a new pumping lemma is introduced to prove that a special language does not belong to a given language

Computing Classification System 1998: F.4.3
Mathematics Subject Classification 2010: 68Q45
Key words and phrases: context-free languages, linear languages, pumping lemma, derivation tree, regular languages

class. Several subclasses of context-free languages are known, such as deterministic context-free and linear languages. The linear language class is strictly between the regular and the context-free ones. In linear grammars only the following types of rules can be used: $A \to w$, $A \to uBv$ (A, B are non-terminals, $w, u, v \in V^*$). In the sixties, Amar and Putzolu defined and analysed a special subclass of linear languages, the so-called even-linear ones, in which the rules has a kind of symmetric shape [1] (in a rule of shape $A \to uBv$, i.e., with non-terminal at the right hand side, the length of u must equal to the length of v). The even-linear languages are intensively studied, for instance, they play special importance in learning theory [10]. In [2] Amar and Putzolu extended the definition to any fix-rated linear languages. They defined the k-rated linear grammars and languages, in which the ratio of the lengths of v and u equals to a fixed non-negative rational number k for all rules of the grammar containing non-terminal in the right-hand-side. They used the term k-linear for the grammar class and k-regular for the generated language class. In the literature the k-linear grammars and languages are frequently used for the metalinear grammars and languages [5], as they are extensions of the linear ones (having at most k nonterminals in the sentential forms). Therefore, for clarity, we prefer the term fix-rated (k-rated) linear for those restricted linear grammars and languages that are introduced in [2]. The classes k-rated linear languages are strictly between the linear and regular ones for any rational value of k. Moreover their union the set of all fixed-linear languages is also strictly included in the class of linear languages. In special case $k = 1$ the even-linear grammars and languages are obtained; while the case $k = 0$ corresponds to the regular grammars and languages. The derivation-trees of the k-rated linear grammars form pine tree shapes. In this paper we investigate pumping lemmas for these languages also. These new pumping lemmas work for regular languages as well, since every regular language is k-rated linear for every non-negative rational k. In this way the words of a regular language can be pumped in two places in a parallel way. There are also extensions of linear grammars. A context-free grammar is said to be k-linear if it has the form of a linear grammar plus one additional rule of the form $S \to S_1 S_2 \dots S_k$, where none of the symbols S_i may appear on the right-hand side of any other rule, and S may not appear in any other rule at all. A language is said to be k-linear if it can be generated by a k-linear grammar, and a language is said to be metalinear if it is k-linear for some positive integer k. The metalinear language family is strictly between the linear and context-free ones. In this paper we also introduce a pumping lemma for not metalinear context-free languages, which can be used to prove that the given language belongs to the class of the metalinear languages.

2 Preliminaries

In this section we give some basic concepts and fix our notation. Let $\mathbb{N}$ denote the non-negative integers and $\mathbb{Q}$ denote the non-negative rationals through the paper.

A grammar is an ordered quadruple $G = (N, V, S, H)$, where N, V are the non-terminal and terminal alphabets. $S \in N$ is the initial letter. H is a finite set of derivation rules. A rule is a pair written in the form $v \to w$ with $v \in (N \cup V)^* N (N \cup V)^*$ and $w \in (N \cup V)^*$.

Let G be a grammar and $v, w \in (N \cup V)^*$. Then $v \Rightarrow w$ is a direct derivation if and only if there exist $v_1, v_2, v', w' \in (N \cup V)^*$ such that $v = v_1 v' v_2$, $w = v_1 w' v_2$ and $v' \to w' \in H$. The transitive and reflexive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.

The language generated by a grammar G is $L(G) = \{w | S \Rightarrow^* w \wedge w \in V^*\}$. Two grammars are equivalent if they generate the same language modulo the empty word (λ). (From now on we do not care whether $\lambda \in L$ or not.)

Depending on the possible structures of the derivation rules we are interested in the following classes [2, 5].

• type 1, or context-sensitive (CS) grammars: for every rule the next scheme holds: $uAv \to uwv$ with $A \in N$ and $u, v, w \in (N \cup V)^*, w \neq \lambda$.

• type 2, or context-free (CF) grammars: for every rule the next scheme holds: $A \to v$ with $A \in N$ and $v \in (N \cup V)^*$.

• linear (Lin) grammars: each rule is one of the next forms: $A \to v$, $A \to vBw$; where $A, B \in N$ and $v, w \in V^*$.

• k-linear (k-Lin) grammars: it is a linear grammar plus one additional rule of the form $S \to S_1 S_2 \dots S_k$, where $S_1, S_2, \dots, S_k \in N$, and none of the S_i may appear on the right-hand side of any other rule, and S may not appear in any other rule at all.

• metalinear (Meta) grammars: A grammar is said to be metalinear if it is k-linear for some positive integer k.

• k-rated linear (k-rLin) grammars: it is a linear grammar with the following property: there exists a rational number k such that for each rule of the form: $A \to vBw$: $\frac{|w|}{|v|} = k$ (where $|v|$ denotes the length of v).

Specially with $k = 1$:

• even-linear (1-rLin) grammars.

Specially with $k = 0$:

• type 3, or regular (Reg) grammars: each derivation rule is one of the following forms: $A \to w$, $A \to wB$; where $A, B \in N$ and $w \in V^*$.

The language family regular/linear etc. contains all languages that can be

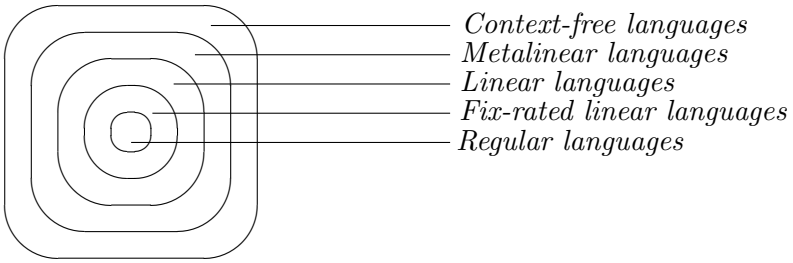


Figure 1: The hierarchy of some context-free language classes

generated by regular/linear etc. grammars. We call a language L fix-rated linear if there is a $k \in \mathbb{Q}$ such that L is k-rated linear. So the class of fix-rated linear languages includes all the k-rated linear language families. Moreover it is known by [2], that for any value of $k \in \mathbb{Q}$ all regular languages are k-rated linear.

The hierarchy of the considered language classes can be seen in Fig. 4. Further, when we consider a special fixed value of k, then we will also use it as $k = \frac{g}{h}$, where $g, h \in \mathbb{N}$ $(h \neq 0)$ are relatively primes.

Now we present normal forms for the rules of linear, k-rated linear and so, even-linear and regular grammars.

The following fact is well-known: Every linear grammar has an equivalent grammar in which all rules are in forms of $A \to aB, A \to Ba, A \to a$ with $a \in V, A, B \in N$.

Lemma 1 (Normal form for k-rated linear grammars) *Every k-rated $(k = \frac{g}{h})$ linear grammar has an equivalent one in which for every rule of the form $A \to vBw$: $|w| = g$ and $|v| = h$ such that g and h are relatively primes and for all rules of the form $A \to u$ with $u \in V^*$: $|u| < g + h$ holds.*

Proof. It goes in the standard way: longer rules can be simulated by shorter ones by the help of newly introduced nonterminals. □

As special cases of the previous lemma we have:

Remark 2 *Every even-linear grammar has an equivalent grammar in which all rules are in forms $A \to aBb, A \to a$, $A \to \lambda$ $(A, B \in N, a, b \in V)$.*

Remark 3 *Every regular language can be generated by grammar having only rules of types $A \to aB, A \to \lambda$ $(A, B \in N, a \in V)$.*

Derivation trees are widely used graphical representations of derivations in context-free grammars. The root of the tree is a node labelled by the initial symbol S. The terminal labelled nodes are leaves of the tree. The nonterminals, as the derivation continues from them, have some children nodes. Since there is a grammar in Chomsky normal form for every context-free grammar, every word of a context-free language can be generated such that its derivation tree is a binary tree.

In linear case, there is at most one non-terminal in every level of the tree. Therefore the derivation can go only in a linear (sequential) manner. There is only one main branch of the derivation (tree); all the other branches terminate immediately. Observing the derivations and derivation trees for linear grammars, they seem to be highly related to the regular case. The linear (and so, specially, the even-linear and fixed linear) languages can be accepted by finite state machines [1, 7, 8]. Moreover the k-rated linear languages are accepted by deterministic machines [8].

By an analysis of the possible trees and iterations of nonterminals in a derivation (tree) one can obtain pumping (or iteration) lemmas.

Further in this section we recall some well-known iteration lemmas.
The most famous iteration lemma works for every context-free languages [3].

Lemma 4 (Bar-Hillel lemma) *Let a context-free language L be given. Then there exists an integer $n \in \mathbb{N}$ such that any word $p \in L$ with $|p| \geq n$, admits a factorization $p = uvwxy$ satisfying*

1. $uv^i wx^i y \in L$ *for all* $i \in \mathbb{N}$
2. $|vx| > 0$
3. $|vwx| \leq n$.

Example 5 *Let $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$. It is easy to show with the Bar-Hillel lemma that the language L is not context-free.*

The next lemma works for linear languages [5].

Lemma 6 (Pumping lemma for linear languages) *Let L be a linear language. Then there exists an integer n such that any word $p \in L$ with $|p| \geq n$, admits a factorization $p = uvwxy$ satisfying*

1. $uv^i wx^i y \in L$ *for all integer* $i \in \mathbb{N}$
2. $|vx| > 0$
3. $|uvxy| \leq n$.

Example 7 *It is easy to show by using Lemma 6 that the language $L = \{a^i b^i c^j d^j | i, j \in \mathbb{N}\}$ is not linear.*

In [6] there is a pumping lemma for non-linear context-free languages that can also be effectively used for some languages.

Lemma 8 (Pumping lemma for non-linear context-free languages) *Let* L *be a non-linear context-free language. Then there exist infinite many words* $p \in L$ *which admit a factorization* $p = rstuvwxyz$ *satisfying*

1. $rs^i tu^i vw^j xy^j z \in L$ *for all integer* $i, j \geq 0$
2. $|su| \neq 0$
3. $|wy| \neq 0$.

Example 9 *Let*

$$H \subseteq \{1^2, 2^2, 3^2, \ldots\}$$

be an infinite set, and let

$$L_H = \{a^k b^k a^l b^l \mid k, l \geq 1;\ k \in H \text{ or } l \in H\} \cup \{a^m b^m \mid m \geq 1\}.$$

The language L_H *satisfies the Bar-Hillel condition. Therefore we can not apply the Bar-Hillel Lemma to show that* L_H *is not context-free. However the* L_H *language does not satisfy the condition of the pumping lemma for linear languages. Thus* L_H *is not linear. At this point we can apply Lemma 8, and the language* L_H *does not satisfy its condition. This means* L_H *is not context-free.*

Now we recall the well-known iteration lemma for regular case (see, for instance, [5]).

Lemma 10 (Pumping lemma for regular languages) *Let* L *be a regular language. Then there exists an integer* n *such that any word* $p \in L$ *with* $|p| \geq n$*, admits a factorization* $p = uvw$ *satisfying*

1. $uv^i w \in L$ *for all integer* $i \in \mathbb{N}$
2. $|v| > 0$
3. $|uv| \leq n$.

Example 11 *By the previous lemma one can easily show that the language* $\{a^n b^n | n \in \mathbb{N}\}$ *is not regular.*

Pumping lemmas are strongly connected to derivation trees, therefore they works for context-free languages (and for some special subclasses of the context-free languages).

In the next section we present pumping lemmas for the k-rated linear languages and for the not metalinear context-free languages.

3 Main results

Let us consider a k-rated linear grammar. Based on the normal form (Lemma 1) every word of a $k = \frac{g}{h}$-rated linear language can be generated by a 'pine-tree' shape derivation tree (see Fig. 2).

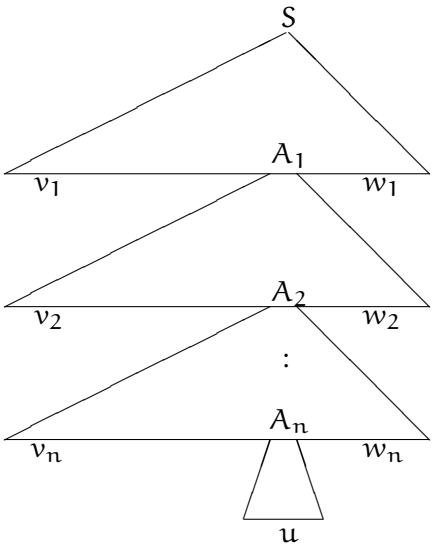


Figure 2: A 'pine-tree' shape derivation tree in a fix-rated linear grammar

Now we are ready to present our pumping lemmas for these languages.

Theorem 12 *Let* L *be a* ($\frac{g}{h} = k$)*-rated linear language. Then there exists an integer* n *such that any word* $p \in L$ *with* $|p| \geq n$*, admits a factorization* $p = uvwxy$ *satisfying*

1. $uv^i wx^i y \in L$ *for all integer* $i \in \mathbb{N}$
2. $0 < |u|, |v| \leq n\frac{h}{g+h}$
3. $0 < |x|, |y| \leq n\frac{g}{g+h}$
4. $\frac{|x|}{|v|} = \frac{|y|}{|u|} = \frac{g}{h} = k$.

Proof. Let $G = (N, V, S, H)$ be a k-rated linear grammar in normal form that generates the language L. Then let $n = (|N|+1)\cdot(g+h)$. In this way any word p with length at least n cannot be generated without any repetition of a nonterminal in the sentential form. Moreover, by the pigeonhole principle, there is a nonterminal in the derivation which occurs in the sentential forms during the first $|N|$ steps of the derivation and after the first occurrence it occurs also

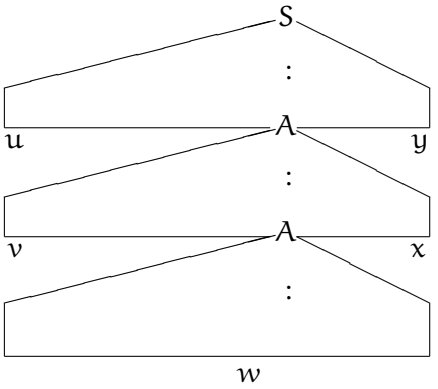


Figure 3: Pumping the subwords between the two occurrences of the nonterminal A.

in the next $|N|$ sentential forms. Considering the first two occurrences of this nonterminal A in the derivation tree, the word p can be partitioned to five parts in the following way. Let u and y be the prefix and suffix (respectively) generated by the first steps till the first occurrence of A. Let v and x be the subwords that are generated from the first occurrence of A till it appears secondly in the sentential form. Finally let w be the subword that is generated from the second occurrence of A in the derivation. (See also Fig. 3.) In this way the conditions 2, 3 and 4 of the theorem are fulfilled for the lengths of the partitions. Now let us consider the derivation steps between the first two occurrences of A. They can be omitted from the derivation; in this way the word uwy is obtained. This sequence of steps can also be repeated any time, in this way the words of the form uv^iwx^iy are obtained for any $i \in \mathbb{N}$. Thus the theorem is proved. □

Theorem 13 *Let L be a $(\frac{g}{h} = k)$-rated linear language. Then there exists an integer n such that any word $p \in L$ with $|p| \geq n$, admits a factorization $p = uvwxy$ satisfying*

1. $uv^iwx^iy \in L$ *for all integer* $i \in \mathbb{N}$
2. $0 < |v| \leq n\frac{h}{g+h}$
3. $0 < |x| \leq n\frac{g}{g+h}$
4. $0 < |w| \leq n$
5. $\frac{|x|}{|v|} = \frac{|y|}{|u|} = \frac{g}{h} = k$.

Proof. Let $G = (N, V, S, H)$ be a k-rated linear grammar in normal form that

generates the language L. Then let $n = (|N| + 1) \cdot (g + h)$. In this way any word p with length at least n cannot be generated without any repetition of a nonterminal in the sentential form. Moreover there is a nonterminal A in the derivation which occurs twice among the non-terminals of the last $|N + 1|$ sentential forms of the derivation. Considering these last two occurrences of A in the derivation tree the word p can be partitioned to five parts in the following way. Let u and y be the prefix and suffix (respectively) generated from the first steps till that occurrence of A which is the last but one during the derivation. Let v and x be the subwords that are generated by the steps between the last two occurrences of A. Finally let w be the subword that is generated from the last occurrence of A in the derivation. In this way the conditions 2, 3, 4 and 5 are fulfilled for the lengths of the partitions. Now let us consider the derivation steps between the these two occurrences of A. They can be omitted from the derivation; in this way the word uwy is obtained. This sequence of steps can also be repeated any time, in this way the words of the form $uv^i wx^i y$ are obtained for any $i \in \mathbb{N}$. Thus the theorem is proved. □

Remark 14 *In case of* $k = 0$ *the previous theorems give the well-known pumping lemmas for regular languages.*

Now we are presenting an iteration lemma for another special subclass of the context-free language family.

Theorem 15 *Let* L *be a context-free language which does not belong to any* k*-linear language for a given positive integer* k*. Then there exist infinite many words* $w \in L$ *which admit a factorization* $w = uv_0w_0x_0y_0 \dots v_kw_kx_ky_k$ *satisfying*

1. $uv_0^{i_0}w_0x_0^{i_0}y_0 \dots v_k^{i_k}w_kx_k^{i_k}y_k \in L$ *for all integer* $i_0, \dots, i_k \geq 0$
2. $|v_jx_j| \neq 0$ *for all* $0 \leq j \leq k$.

Proof. Let $G = (N, V, S, H)$ be a context-free grammar such that $L(G) = L$, and let $G_A = (N, V, A, H)$ for all $A \in N$. Because L is not k-linear, there exists $A_0, \dots, A_k \in V_N$ and $\alpha, \beta_0, \dots, \beta_k \in V^*$ such that $S \Rightarrow^* \alpha A_0\beta_0 \dots A_k\beta_k$, where all of the languages $L(G_{A_l})$, $0 \leq l \leq k$ are infinite. Then the words

$\{\alpha\}L(G_{A_0})\{\beta_0\} \dots L(G_{A_k})\{\beta_k\} \subseteq L$, and applying the Bar-Hillel Lemma for all $L(G_{A_l})$ we receive $\alpha a_0b_0^{i_0}c_0d_0^{i_0}e_0\beta_0 \dots a_kb_k^{i_k}c_kd_k^{i_k}e_k\beta_k \subseteq L$ for all $i_0 \geq 0, \dots, i_k \geq 0$. Let $u = \alpha a_0$, $v_l = b_l$, $w_l = c_l$, $x_l = d_l$, $y_l = e_l\beta_l$, and we have the above form. □

Remark 16 *With* $k = 1$ *we have a pumping lemma for non-linear context-free languages.*

Knowing that every k-linear language is metalinear for any $k \in \mathbb{N}$, we have:

Proposition 17 *Let* L *be a not metalinear context-free language. For all integers* $k \geq 1$ *there exist infinite many words* $w \in L$ *which admit a factorization* $w = uv_0w_0x_0y_0 \dots v_kw_kx_ky_k$ *satisfying*

1. $uv_0^{i_0}w_0x_0^{i_0}y_0 \dots v_k^{i_k}w_kx_k^{i_k}y_k \in L$ *for all integer* $i_0, \dots, i_k \geq 0$
2. $|v_jx_j| \neq 0$ *for all* $0 \leq j \leq k$.

4 Applications of the new iteration lemmas

As pumping lemmas are usually used to show that a language does not belong to a language class, we present an example for this type of application.

Example 18 *The DYCK language (the language of correct bracket expressions) is not* k*-linear for any value of* k *over the alphabet* $\{(,)\}$. *Let* $k \neq 1$ *be fixed as* $\frac{g}{h}$. *Let us consider the word of the form* $(^{(g+h)(n+2)})^{(g+h)(n+2)}$. *Then Theorem 12 does not work (if* $k \neq 1$*), the pumping deletes or introduces different number of* (*'s and*) *'s. To show that the DYCK language is not 1-rated (i.e., even-)linear let us consider the word* $(^{2n})^{2n}(^{2n})^{2n}$. *Using Theorem 13 the number of inner brackets can be pumped. In this way such words are obtained in which there are prefixes with more letters*) *than* (. *Since these words do not belong to the language, this language is not* k*-linear.*

In the previous example we showed that the DYCK language is not fixed linear.

In the next example we consider a deterministic linear language.

Example 19 *Let* $L = \{a^mb^m | m \in \mathbb{N}\} \cup \{a^mcb^{2m} | m \in \mathbb{N}\}$ *over the alphabet* $\{a, b, c\}$. *Let us assume that the language is fixed linear. First we show that this language is not fixed linear with ratio other than 1. On the contrary, assume that it is, with* $k = \frac{g}{h} \in \mathbb{Q}$ *such that* $k \neq 1$. *Let* n *be given by Theorem 12. Then consider the words of the form* $a^{m(g+h)}b^{m(g+h)}$ *with* $m > n$. *By the theorem any of them can be factorized to* $uvwxy$ *such that* $|uv| \leq \frac{2nh}{g+h}$. *Since* $g + h > 2$ *(remember that* $g, h \in \mathbb{N}$*, relatively primes and* $g \neq h$*),* $|uv| < nh$, *and therefore both* u *and* v *contains only* a*'s. By a similar argument on the length of* xy, x *and* y *contains only* b*'s. Since the ratio* $\frac{|x|}{|v|}$ *(it is fixed by the*

theorem) is not 1, by pumping we get words outside of the language. Now we show that this language is not even-linear. Assume that it is 1-rated linear ($g = h = 1$). *Let* n *be the value from Theorem 12. Let us consider the words of shape* $a^m c b^{2m}$ *with* $m > n$. *Now we can factorize these words in a way, that* $|uv| \leq n$ *and* $|xy| \leq n$ *and* $|v| = |x|$. *By pumping we get words* $a^{m+j} c b^{2m+j}$ *with some positive values of* j, *but they are not in* L. *We have a contradiction again. So this language is not fixed linear.*

In the next example we show a fixed-linear language that can be pumped.

Example 20 *Let* L *be the language of palindromes, i.e., of the words over* $\{a, b\}$ *that are the same in reverse order* ($p = p^R$). *We show that our pumping lemmas work for this language with the value* $k = 1$. *Let* $p \in L$, *then* $p = uvwxy$ *according to Theorem 12 or Theorem 13, such that* $|u| = |y|$ *and* $|v| = |x|$. *Therefore, by applying the main property of the palindromes, we have* $u = y^R$, $v = x^R$ *and* $w = w^R$. *By* $i = 0$ *the word* uwy *is obtained which is in* L *according to the previous equalities. By further pumping the words* $uv^i wx^i y$ *are obtained, they are also palindromes. To show that this language cannot be pumped with any other values, let us consider words of shape* $a^m b a^m$. *By Theorem 12 it can be shown in analogous way that we showed in Example 19 that enough long words cannot be pumped with ratio* $k \neq 1$.

Besides our theorems work for regular languages with $k = 0$ there is a non-standard application of them. As we already mentioned, all regular languages are k-rated linear for any values of $k \in \mathbb{Q}$. Therefore every new pumping lemma works for any regular language with any values of k. Now we show some examples.

Example 21 *Let the regular language* $(ab)^* aa(bbb)^* a$ *be given. Then we show, that our theorems work for, let us say,* $k = \frac{1}{2}$. *Every word of the language is of the form* $(ab)^n aa(bbb)^m a$ *(with* $n, m \in \mathbb{N}$). *For words that are long enough either* n *or* m *(or both of them) are sufficiently large. Now we detail effective factorizations* $p = uvwxy$ *of the possible cases. We give only those words of the factorization that have maximized lengths due to the applied theorem, the other words can easily be found by the factorization and, at Theorem 13, by taking into account the fixed ratio of some lengths in the factorization.*

- *Theorem 12 for* $k = \frac{1}{2}$*:*
  if $n > 3$ *and* $m > 0$ *: let* $u = ab$, $v = ababab$, $x = bbb$, $y = a$,
  if $m = 0$ *: let* $u = ababab$, $v = abab$, $x = ab$, $y = aaa$,

if $n = 3$ *: let* $u = ababababa$, $v = bb$, $x = b$, $y = bbba$,
if $n = 2$ *: let* $u = ababaa$, $v = bb$, $x = b$, $y = bba$,
if $n = 1$ *: let* $u = abaa$, $v = bb$, $x = b$, $y = ba$,
if $n = 0$ *: let* $u = aa$, $v = bb$, $x = b$, $y = a$.

- *Theorem 13 for* $k = \frac{1}{2}$*:*
if $n \leq 3m - 4$ *: let* $v = bb$, $w = b$ $x = b$,
if $n = 3m - 3$ *: let* $v = ababab$, $w = aabbbb$ $x = bbb$,
if $n = 3m - 2$ *: let* $v = ababab$, $w = abaabbbb$ $x = bbb$,
if $n = 3m - 1$ *: let* $v = ababab$, $w = ababaabbbb$ $x = bbb$,
if $n = 3m$ *: let* $v = ababab$, $w = aab$ $x = bbb$,
if $n = 3m + 1$ *: let* $v = ababab$, $w = abaab$, $x = bbb$,
if $n = 3m + 2$ *: let* $v = ababab$, $w = ababaab$, $x = bbb$,
if $n = 3m + 3$ *: let* $v = ababab$, $w = abababaab$, $x = bbb$,
if $n = 3m + 4$ *: let* $v = ababab$, $w = abababababaab$, $x = bbb$,
if $n = 3m + 5$ *: let* $v = ababab$, $w = abababababaaab$, $x = bbb$,
if $n \geq 3m + 6$, $n \equiv 0 (mod 3)$ *: let* $v = abab$, $w = \lambda$, $x = ab$,
if $n \geq 3m + 7$, $n \equiv 1 (mod 3)$ *: let* $v = abab$, $w = ab$, $x = ab$,
if $n \geq 3m + 8$, $n \equiv 2 (mod 3)$ *: let* $v = abab$, $w = abab$, $x = ab$.

In similar way it can be shown that pumping the words of a regular language in two places simultaneously with other values of k (for instance, $1, 5, \frac{7}{3}$ etc.) works.

In the next example we show that there are languages that can be pumped by the usual pumping lemmas for regular languages, but they cannot be regular since we prove that there is a value of k such that one of our theorems does not work.

Example 22 *Let* $L = \{a^r b a^q b^m | r, q, m \geq 2, \exists j \in \mathbb{N} : q = j^2\}$*. By the usual pumping lemmas for regular languages, i.e., by fixing* k *as 0, one cannot infer that this language is not regular. By* $k = 0$, $x = y = \lambda$ *and so* $p = uvw$*. Due to the* a*'s in the beginning, Theorem 12 works:* $u = a, v = a$*; and due to the* b*'s in the end Theorem 13 also works:* $v = b, w = b$.
Now we show that L *is not even-linear. Contrary, let us assume that Theorem 13 works for* $k = 1$*. Let* n *be the value for this language according to the theorem. Let* $p = a^2 b a^{(2n+5)^2} b^3$*. By the conditions of the theorem, it can be factorized to* $uvwxy$ *such that* $|v|, |w|, |x| \leq n$ *and* $|u| = |y|$*. In this way* vwx *must be a subword of* $a^{(2n+5)^2}$*, and so, the pumping decreases/increases only* q*. Since* $|v|, |x| \leq n$ *in the first round of pumping* $p' = a^2 b a^{(2n+5)^2 + |vx|} b^3$ *is*

obtained. But $(2n+5)^2 < (2n+5)^2 + |vx| \leq (2n+5)^2 + 2n < (2n+6)^2$, *therefore* $p' \notin L$.
Thus L *is not even-linear, and therefore it cannot be regular. Our pumping lemma was effective to show this fact.*

Usually pumping lemmas can be used only to show that some languages do not belong to the given class of languages. One may ask what we can say if a language satisfy our theorems. Now we present an example which shows that we cannot infer about the language class if a language satisfies our new pumping lemmas.

Example 23 *Let* $L = \{0^j1^m0^r1^i0^l1^i0^r1^m0^j | j, m, i, l, r \geq 1, r \text{ is prime}\}$. *One can easily show that this language satisfies both Theorem 12 and Theorem 13 with* $k = 1$*: one can find subwords to pump in the part of outer* 0*'s or* 1*'s (pumping their number form a given* j *or* m *to arbitrary high values), or in the middle part* 0*'s or* 1*'s (pumping their number from* i *or* l *to arbitrary high values), respectively. But this language is not even context-free, since intersected by the regular language* 010^*1010^*10 *a non semi-linear language is obtained. Since context-free languages are semi-linear (due to the Parikh theorem) and the class of context-free languages are closed under intersection with regular languages, we just proved that* L *cannot be linear or fix-rated linear.*

It is a more interesting question what we can say about a language for which there are values $k_1 \neq k_2$ such that all its enough long words can be pumped both as k_1-rated and k_2-rated linear language. We have the following conjecture.

Conjecture 24 *If a language* L *satisfies any of our pumping lemmas for two different values of* k*, then* L *is regular.*

If the previous conjecture is true, then exactly the regular languages form the intersection of the k-rated linear language families (for $k \in \mathbb{Q}$).

Regarding iteration lemma for the not metalinear case, we show two examples.

Example 25 *This is a very simple example, we can use our lemma to show that the language*

$$L_1 = \{a^l b^l a^m b^m a^n b^n \mid l, m, n \geq 0\}$$

is metalinear.

First of all, it is easy to show that L_1 *is context-free. The language* L_1 *does not satisfy the condition of the pumping lemma for not metalinear context-free languages, (Proposition 17,) so* L_1 *must be a metalinear context-free language.*

In our next example we show a more complicated language which satisfies the Bar-Hillel condition, and we use our pumping lemma to show that the language is not context-free.

Example 26 *Let*

$$H \subseteq \{2^k \mid k \in \mathbb{N}\}$$

be an infinite set, and let

$$L_2 = \{a^l b^l a^m b^m a^n b^n \mid l, m, n \geq 1;\ l \in H \text{ or } m \in H \text{ or } n \in H\} \cup$$

$$\cup \{a^i b^i a^j b^j \mid i, j \geq 1\}.$$

L_2 *satisfies the Bar-Hillel condition. Therefore we can not apply the Bar-Hillel Lemma to show that* L_2 *is not context-free. However it is easy to show that* L_2 *is not* 3*-linear language. Now we can apply Theorem 15, and the language* L_2 *does not satisfy its condition with* $k = 3$*. This means* L_2 *does not belong to the not* 3*-linear context-free languages, so the language* L_2 *is not context-free.*

5 Conclusions

In this paper some new pumping lemmas are proved for special context-free and linear languages. In fix-rated linear languages the lengths of the pumped subwords of a word depend on each other, therefore these pumping lemmas are more restricted than the ones working on every linear or every context-free languages. Since all regular languages are k-rated linear for any non-negative rational value of k, these lemmas also work for regular languages. The question whether only regular languages satisfy our pumping lemmas at least for two different values of k (or for all values of k) is remained open as a conjecture. We also investigated a special subclass of context-free language family and introduced iteration conditions which is satisfied only not metalinear context-free languages. These conditions can be used in two different ways. First they can be used to proove that a language is not context-free. On the other hand, we can also use them to show that the given language is belong to the metalinear language family.

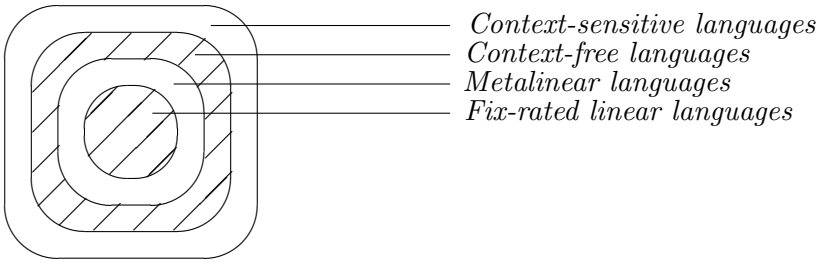


Figure 4: The target language classes of the new iteration lemmas

Acknowledgements

The work is supported by the Czech-Hungarian bilateral project (TéT) and the TÁMOP 4.2.1/B-09/1/KONV-2010-0007 project. The project is implemented through the New Hungary Development Plan, co-financed by the European Social Fund and the European Regional Development Fund.

References

[1] V. Amar, G. R. Putzolu, On a family of linear grammars, *Information and Control*, **7,** 3 (1964) 283–291. ⇒195, 198

[2] V. Amar, G. R. Putzolu, Generalizations of regular events, *Information and Control*, **8,** 1 (1965) 56–63. ⇒195, 196, 197

[3] Y. Bar-Hillel, M. Perles, and E. Shamir, On formal properties of simple phrase structure grammars, *Z. Phonetik. Sprachwiss. Komm.*, **14** (1961) 143–172. ⇒194, 198

[4] P. Dömösi, M. Ito, M. Katsura, C. Nehaniv, New pumping property of context-free languages, *Combinatorics, Complexity an Logic, Proc. International Conference on Discrete Mathemtics and Theoretical Computer Science – DMTCS'96*, Springer, Singapore, pp. 187–193. ⇒194

[5] J. E. Hopcroft, J. D. Ullman, *Introduction to automata theory, languages, and computation,* (2nd edition), Addison-Wesley, Reading, MA, 1979. ⇒ 194, 195, 196, 198, 199

[6] G. Horváth, New pumping lemma for non-linear context-free languages, *Proc. 9th Symposium on Algebras, Languages and Computation*, Shimane University, Matsue, Japan, 2006, pp. 160–163. ⇒199

[7] R. Lokunova, Linear context free languages, *Proc. ICTAC 2007, Lecture Notes in Comput. Sci.,* **4711** (2007) 351–365. ⇒198

[8] B. Nagy, On $5' \to 3'$ sensing Watson-Crick finite automata, *DNA 13, Revised selected papers, Lecture Notes in Comput. Sci.,* **4848** (2008) 256–262. ⇒198

[9] G. Rozenberg, A. Salomaa, (eds.) *Handbook of formal languages*, Springer, Berlin, Heidelberg, 1997. ⇒194

[10] J. M. Sempere, P. García, A characterization of even linear languages and its application to the learning problem, *Proc. Second International Colloquium, ICGI-94, Lecture Notes in Artificial Intelligence,* **862** (1994) 38–44. ⇒195

Received: October 5, 2010 • Revised: November 2, 2010

Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 210–230

Modelling dynamic programming problems by generalized d-graphs

Zoltán Kátai
Sapientia Hungarian University of Transylvania
Department of Mathematics and Informatics,
Tg. Mureş, Romania
email: katai_zoltan@ms.sapientia.ro

Abstract. In this paper we introduce the concept of generalized d-graph (admitting cycles) as special dependency-graphs for modelling dynamic programming (DP) problems. We describe the d-graph versions of three famous single-source shortest algorithms (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm), which can be viewed as general DP strategies in the case of three different class of optimization problems. The new modelling method also makes possible to classify DP problems and the corresponding DP strategies in term of graph theory.

1 Introduction

Dynamic programming (DP) as optimization method was proposed by Richard Bellman in 1957 [1]. Since the first book in applied dynamic programming was published in 1962 [2] DP has become a current problem solving method in several fields of science: Applied mathematics [2], Computer science [3], Artificial Intelligence [6], Bioinformatics [4], Macroeconomics [13], etc. Even in the early book on DP [2] the authors drew attention to the fact that some dynamic programming strategies can be formulated as graph search problems. Later this subject was largely researched. As recent examples: Georgescu and Ionescu introduced the concept of DP-tree [7]; Kátai [8] proposed d-graphs

Computing Classification System 1998: D.1
Mathematics Subject Classification 2010: 68W40, 90C39, 68R10, 05C12
Key words and phrases: dynamic programming, graph theory, shortest path algorithms

as special hierarchic dependency-graphs for modelling DP problems; Lew and Mauch [14, 15, 16] used specialized Petri Net models to represent DP problems (Lew called his model Bellman-Net).

All the above mentioned modelling tools are based on cycle free graphs. As Mauch [16] states, circularity is undesirable if Petri Nets represent DP problem instances. On the other hand, however, there are DP problems with "cyclic functional equation" (the chain of recursive dependences of the functional equation is cyclic). Felzenszwalb and Zabih [5] in their survey entitled *Dynamic programming and graph algorithms in computer vision* recall that many dynamic programming algorithms can be viewed as solving a shortest path problem in a graph (see also [9, 11, 12]. But, interestingly, some shortest path algorithms work in cyclic graphs too. Kátai, after he has been analyzing the three most common single-source shortest path algorithms (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm), concludes that all these algorithms apply cousin DP strategies [10, 17]. Exploiting this observation Kátai and Csiki [12] developed general DP algorithms for discrete optimization problems that can be modelled by simple digraphs (see also [11]). In this paper, modelling finite discrete optimization problems by generalized d-graphs (admitting cycles), we extend the previously mentioned method for a more general class of DP problems. The presented new modelling method also makes possible to classify DP problems and the corresponding DP strategies in term of graph theory.

Then again the most common approach taken today for solving real-world DP problems is to start a specialized software development project for every problem in particular. There are several reasons why is benefiting to use the most specific DP algorithm possible to solve a certain optimization problem. For instance this approach commonly results in more efficient algorithms. But a number of researchers in the above mentioned various fields of applications are not experts in programming. Dynamic programming problem solving process can be divided into two steps: (1) the functional equation of the problem is established (a recursive formula that implements the principle of the optimality); (2) a computer program is elaborated that processes the recursive formula in a bottom-up way [12]. The first step is reachable for most researchers, but the second one not necessary. Attaching graph-based models to DP problems results in the following benefits:

- it moves DP problems to a well research area: graph theory,
- it makes possible to class DP strategies in terms of graph theory,
- as an intermediate representation of the problem (that hides, to some

degree, the variety of DP problems) it enables to automate the programming part of the problem-solving process by an adequately developed software-tools [12],
- a general software-tool that automatically solves DP problems (getting as input the functional equation) should be able to save considerable software development costs [16].

2 Modelling dynamic programming problems

DP can be used to solve optimization problems (discrete, finite space) that satisfy the principle of the optimality: *The optimal solution of the original problem is built on optimal sub-solutions respect to the corresponding sub-problems.* The principle of the optimality implicitly suggests that the problem can be decomposed into (or reduced to) similar sub-problems. Usually this operation can be performed in several ways. The goal is to build up the optimal solution of the original problem from the optimal solutions of its smaller sub-problems. Optimization problems can often be viewed as special version of more general problems that ask for all solutions, not only for the optimal one (A so-called objective function is defined on the set of sub-problems, which has to be optimized). We will call this general version of the problem, all-solutions-version.

The set of the sub-problems resulted from the decomposing process can adequately be modelled by dependency graphs (We have proposed to model the problem on the basis of the previously established functional equation that can be considered the output of the mathematical part and the input of the programming part of the problem solving process). The vertices (continuous/dashed line squares in the optimization/all-solutions version of the problem; see Figures 2.a,b,c) represent the sub-problems and directed arcs the dependencies among them. We introduce the following concepts:

- *Structural-dependencies:* We have directed arc from vertex A to vertex B if solutions of sub-problem A *may directly depend* on solutions of sub-problem B (dashed arcs; see Figure 2.a).
- *Optimal-dependencies:* We have directed arc from vertex A to vertex B if the optimal solution of sub-problem A *directly depends* on the optimal solution of the *smaller (respect to the optimization process)* sub-problem B (continuous arcs; see Figure 2.b).
- *Optimization-dependencies:* We have directed arc from vertex A to vertex B if the optimal solutions of sub-problem A *may directly depend* on

the optimal solution of the *smaller (respect to the optimization process)* sub-problem B (dotted arcs; see Figure 2.c).

Since structural dependencies reflect the structure of the problem, the *structural-dependencies-graph* can be considered as input information (It can be built up on the basis of the functional equation of the problem). This graph may contain cycles (see Figure 2.a). According to the principle of the optimality the *optimal-dependencies-graph* is a rooted sub-tree (acyclic sub-graph) of the structural-dependencies-graph. Representing the structure of the optimal solution the optimal-dependencies-graph can be viewed as output information. Since optimization-dependencies are such structural-dependencies that are *compatible* with the principle of the optimality, the *optimization-dependencies-graph* is a maximal rooted sub-tree of the structural-dependencies-graph that includes the optimal-dependencies-tree. Accordingly, the vertices of the optimization-dependencies-graph (and implicitly the vertices of the optimal-dependencies-graph too) can be arranged on levels (hierarchic structure) in such a way that all its arcs are directed downward. The original problem (or problem-set) is placed on the highest level and the trivial ones on the lowest level. We consider that a sub-problem is structurally trivial if cannot be decomposed into, or reduced to smaller sub-sub-problems. A sub-problem is considered to be trivial regarding the optimization process if its optimal solution trivially results from the input data. If the structural-dependencies-graph contains cycles, then completing the hierarchic optimization-dependencies-graph to the structural-dependencies-graph some added arcs will be directed upward.

Let us consider, as an example, the following problem: Given the weighted undirected triangle graph OAB determine

- all paths from vertex O (origin) to the all vertices (O, A, B) of the graph (Figure 1.a),
- the maximal length paths from vertex O (origin) to the all vertices of the graph ($|OA| = 10, |OB| = 10, |AB| = 100$) (Figure 1.b),
- the minimal length paths from vertex O (origin) to the all vertices of the graph ($|OA| = 100, |OB| = 10, |AB| = 10$) (Figure 1.c).

Since path (O,A,B) includes path (O,A) and, conversely, path (O,B,A) includes path (O,B) the structural-dependencies-graph that can be attached to the problem is not cycle free (Figure 2.a). We have bidirectional arcs between vertices representing sub-problems A (determine all paths to vertex A) and B (determine all paths to vertex B). Since the maximizing version of the problem does not satisfy the principle of the optimality (the maximal path (O,B,A)

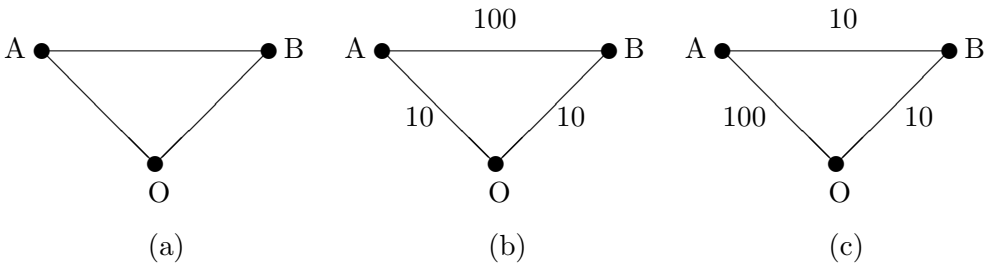


Figure 1: The triangle graph

includes path (O,B) that is not a maximal path too), in case b the optimal-dependencies-tree and the optimization-dependencies-tree are not defined. Figures 2.b and 2.c present the optimal- and optimization-dependencies-graphs attached to the minimizing version of the problem.

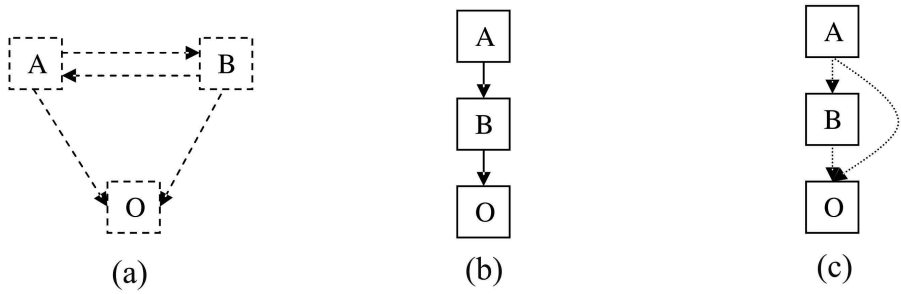


Figure 2: Structural/Optimal/Optimization-dependencies-graphs

3 d-graphs as special dependency graphs

Since decomposing usually means that the current problem is broken down into *two or more* immediate sub-problems ($1 \rightarrow N$ dependency) and since this operation can often be performed in *several* ways, Kátai [8] introduced d-graphs as special dependency graphs for modelling such problems. In this paper we define a generalized form of d-graphs as follows:

Definition 1 *The connected weighted bipartite finite digraph* $G_d(V, E, C)$ *is a d-graph if:*

- $V = V_p \bigcup V_d$ *and* $E = E_p \bigcup E_d$, *where*
  - V_p *is the set of the p-vertices,*
  - V_d *is the set of the d-vertices,*
  - *all in/out neighbours of the p-vertices (excepting the source/sink vertices) are d-vertices; each d-vertex has exactly one p-in-neighbour; each d-vertex has at least one p-out-neighbour,*
  - E_p *is the set of p-arcs (from p-vertices to d-vertices),*
  - E_d *is the set of d-arcs (from d-vertices to p-vertices),*
- *function* $C : E_p \rightarrow \mathbb{R}$ *associates a cost to every p-arc. We consider d-arcs of zero cost.*

If a d-graph is cycle-free, then its vertices can be arranged on levels (hierarchic structure) (see Figure 3). In [8] Kátai defines, respect to hierarchic d-graphs, the following related concepts: d-sub-graph, d-tree, d-sub-tree, d-spanning-tree, optimal d-spanning-tree and optimally weighted d-graph.

4 Modelling optimization problems by d-graphs

According to Kátai [8] a hierarchic d-graph can be viewed as representing the optimization-dependences-graph corresponding to the original problem and d-sub-graphs to the sub-problems. Since there is a one-to-one correspondence between p-vertices and d-sub-graph [8], these vertices also represent the sub-problems. The source p-vertex (or vertices) is attached to the original problem (or original problem-set), and the sink vertices to the structurally trivial sub-problems. A p-vertex has as many d-sons as the number of possibilities to decompose the corresponding sub-problem into its *smaller immediate* sub-sub-problems. A d-vertex has as many p-sons as the number of immediate smaller sub-problems (N) resulted through the corresponding *breaking-down* step ($1 \rightarrow N$ dependency between the p-grandfather-problem and the corresponding p-grandson-problems). We will say that a grandfather-problem is *reduced to* its grandson-problem if the intermediary d-vertex has a single p-son ($1 \rightarrow 1$ dependency). Parallel decomposing processes may result in identical sub-problems, and, consequently, the corresponding p-vertex has multiple p-grandfathers (through different d-fathers). Due to this phenomenon the

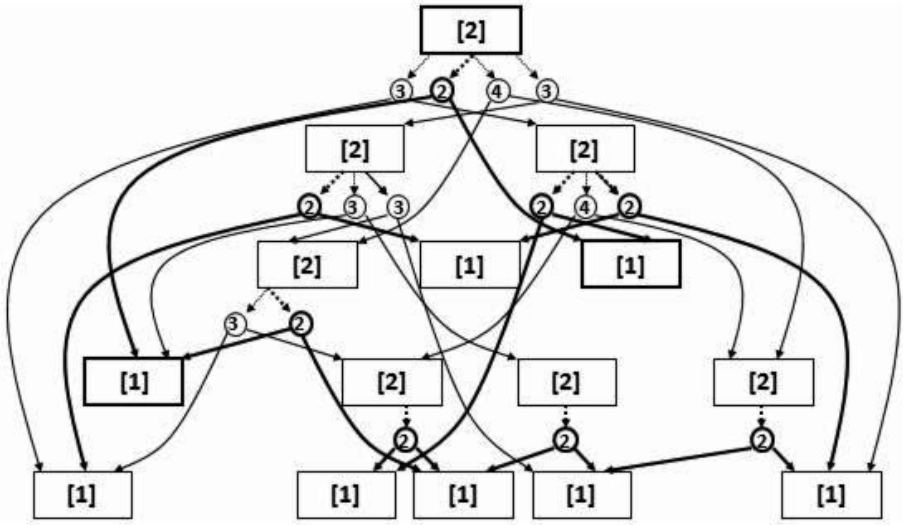


Figure 3: Hierarchic d-graph. p- and d-vertices are represented by rectangles and circles, respectively (We used bolded lines to emphasize the optimal d-spanning-(sub)trees)

number of the sub-problems may depend on the size of the input polynomially. The d-spanning-trees of the d-(sub)graphs represent the corresponding (sub)solutions, more exactly their tree-structure. The number of all solutions of the problem usually depends on the size of the input exponentially.

For example, if a p-vertex has n d-sons, these d-sons have $m_1, m_2, \ldots, m_n$ p-sons, and these p-son-problems have $(r_{1,1}, r_{1,2}, \ldots, r_{1,m1})$, $(r_{1,1}, r_{1,2}, \ldots, r_{1,m2})$, $(r_{1,1}, r_{1,2}, \ldots, r_{1,mn})$ solutions, respectively, then from the $\sum\sum r_{ij}$ solution of the p-grandson-problems results $\sum\prod r_{ij}$ solution for the common p-grandfather-problem. The number of solutions exponentially exceeds the number of sub-problems. The $\sum$-operator reflects the OR-connection between d-brothers and the $\prod$-operator the AND-connection between p-brothers.

5 Dynamic programming strategy on the optimization-dependencies d-graph

In the case of optimization problems we are interested only in the *optimal* solution of the original problem. Dynamic programming means building up

the optimal solutions of the *larger* sub-problems from the optimal solution of the already solved *smaller* sub-problems (starting with the optimal solution of the trivial sub-problems). Accordingly, (1) DP works on the hierarchic optimization-dependencies d-graph that can be attached to the problem, and (2) it deals with one solution per sub-problem, with the optimal one (DP strategies usually result in polynomial algorithms).

In line with this Kátai [8] defines two weight-functions ($w_p : V_p \to \mathbb{R}, w_d : V_d \to \mathbb{R}$) on the sets of p- and d-vertices of the attached hierarchic d-graph. Whereas the weight of a p-vertex is defined as the optimum (minimum/maximum) of the weights of its d-sons, the weight of a d-vertex is a function (depending on the problem to be modelled) of the weights of its p-sons. We consider the weight of a d-vertex to be optimal if is based on optimal the weights of its p-sons. The optimal weight of a p-vertex (excluding the sink vertices) is equal with the minimum/maximum of the optimal weights of its d-sons. The optimal weights of the p-sinks trivially result from the input data of the problem. Accordingly: the optimal weights of the p-vertices are computed (1) in *optimal way*, (2) on the basis of the *optimal weights* of their p-descendents. This means bottom-up strategy. Computing the optimal weights of the p-vertices we implicitly have their optimal d-sons (It is characteristic to DP algorithms that during the bottom-up building process it stores the already computed optimal p-weights in order to have them at hand in case they are needed to compute further optimal p-weights. If we also store the optimal d-sons of the p-vertices, then this information allows a quick reconstruction of the optimal d-spanning-tree in top-down way [17, 18]).

Defining the costs of the p-arcs as the absolute value of the weight-difference of its endpoints we get an optimally weighted d-graph with zero-cost minimal d-spanning-tree. We denote these kinds of p-arc-cost-functions by C^* [8]. Modelling optimization problems by a d-graphs $G_d(V, E, C^*)$ includes choosing functions w_p and w_d in such a way as the optimal weights of the p-vertices to represent the optimum values of the objective function respect to the corresponding sub-problems (These functions can be established on the basis of the functional equation of the problem; input information regarding the modelling process).

Proposition 2 *If an optimization problem can be modelled by a hierarchic d-graph* $G_d(V, E, C^*)$ *(as we described above), then it can be solved by dynamic programming.*

Proof. Since in an optimally weighted d-graph d-sub-trees of an optimal d-spanning-tree are also optimal d-spanning-trees respect to the d-sub-graphs

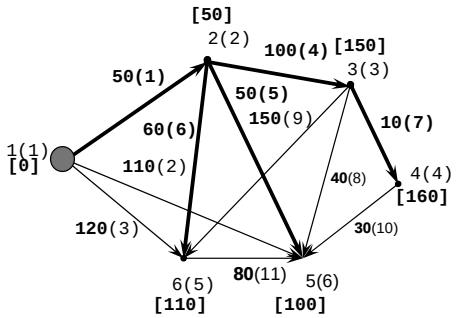
defined by their root-vertices, computing the optimal p- and d-weights according to a reverse topological order of the vertices (based on optimization-dependencies) implicitly identifies the optimal d-spanning-tree of the d-graph. This bottom-up strategy means DP and the optimal solution of the original problem will be represented by the weight of the source vertex (as value) and by the minimal d-spanning-tree (as structure). □

Computing the optimal weight of a p-vertex (expecting vertices representing trivial sub-problems) can be implemented as a gradual updating process based on the weights of its d-sons. The weights of p-vertices representing trivial sub-problems receive as starting-value their input optimal value. For other p-vertices we choose a proper starting-value according to the nature of the optimization problem (The weights of d-vertices are recomputed before every use). We define the following types of updating operations along p-arcs (if the weight of a certain d-son is "better" than the weight of his p-father, then the father's weight is replaced with the son's weight):

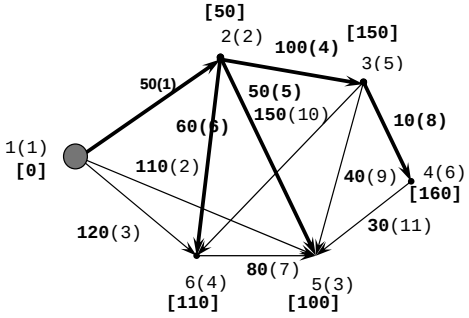
- *Complete:* based on the optimal value of the corresponding d-son.
- *Partial:* based on an intermediate value of the corresponding d-son.
- *Effective:* effectively improves the weight of the corresponding p-vertex.
- *Null:* dose not adjusts the weight of the corresponding p-vertex.
- *Optimal:* sets the optimal weight for the corresponding p-vertex. Optimal updates are complete and effective too.

6 d-graph versions of three famous single-source shortest-path algorithms

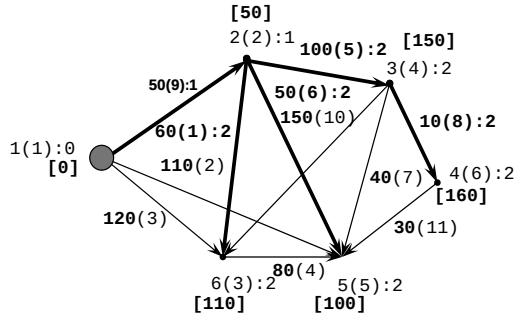
As we mentioned above, Kátai concludes that the three famous single-source shortest-path algorithms in digraphs (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm) apply cousin DP strategies [10, 17]. The common representative core of these DP algorithms is that the optimal weights (representing the optimal lengths to the corresponding vertices) are computed on account of updating these values along the arcs of the shortest-paths-tree according to their topological order (optimal-updating-sequence). Since this optimal tree is unknown (it represents the solution of the problem) all the three algorithms generate updating-sequences which contain, as subsequence, an optimal-updating-sequence necessary for the dynamic programming building process. The basic difference



(1,2), (1,5), (1,6), **(2,3)**, **(2,5)**, **(2,6)**, **(3,4)**, (3,5), (3,6), (4,5), (6,5)



(1,2), (1,5), (1,6), **(2,3)**, **(2,5)**, **(2,6)**, (6,5), **(3,4)**, (3,5), (3,6), (4,5)



1: (2,6), (1,5), (1,6), (6,5), (2,3), (2,5), (3,5), (3,4), **(1,2)**, (3,6), (4,5)

2: **(2,6)**, (1,5), (1,6), (6,5), **(2,3)**, **(2,5)**, (3,5), **(3,4)**, (1,2), (3,6), (4,5)

Figure 4: The strategies of the (a) Topological, (b) Dijkstra and (c) Bellman-Ford algorithms (we bolded the optimal-arc-subsequence of the generated arc-sequences)

among the three algorithms is the way they generate a proper arc-sequence and the corresponding updating-sequence.

In case the input digraph is acyclic we get a proper arc-sequence by ordering all the arcs of the graph topologically (this order can even be determined in advance). Dijkstra algorithm (working in cyclic graphs too, but without negative weighted arcs) determines the needed arc-sequence on the fly (parallel with the updating process). After the current weight of the next closest vertex has been confirmed as optimal value (greedy decision), the algorithm performs updating operations along the out-arcs of this vertex (This greedy choice can be justified as follows: if other out-neighbours of the growing shortest-paths-tree are farther - from the source vertex - than the currently closest one, then through these vertices cannot lead shortest paths to this). Bellman-Ford algorithm (working in cyclic graphs with negative weighted arcs too, but without feasible negative weighted cycles) goes through (in arbitrary order) all the arcs of the graph again and again until the arc-sequence generated in this way finally will contains, as sub-sequence, an optimal-updating-sequence (see Figure 4, [10]). The following d-graph algorithms implement DP strategies that exploit the core idea behind the above described single-source shortest-paths algorithms.

6.1 Building-up the optimization-dependencies d-graph in bottom-up way

Our basic goal is to perform updating operation along the p-arcs of the optimal-dependencies-tree according to their reverse topological order. We will call such arc sequences optimal-arc-sequence and the corresponding updating sequences optimal-updating-sequence. An optimal-updating-sequence surely results in building up the optimal value representing the optimal solution of the problem. Since the optimal-dependencies-tree is unknown (it represents the structure of the optimal solution to be determined), we should try to elaborate *complete arc sequences* that includes the desired optimal-updating-sequence (gratuitous updating operations have, at the worst, null effects).

We introduce the following colouring-convention:

- Initially all vertices are white.
- A p-vertex changes its colour to grey after the first attempt to update its weight. d-vertices automatically change their colour to grey if they do not have any more white p-sons.
- When the weight of a vertex riches its optimal value its colour is automatically changed to black.

We are facing a gratuitous updating operation if:

- along the corresponding p-arc was previously performed a complete update,
- the corresponding p-father is already black,
- the corresponding d-son is still grey or white.

Since the optimal values of trivial sub-problems automatically results from the input data of the problem, the corresponding p-vertices are automatically coloured with black.

The following propositions can be viewed as theoretical support for the below strategies that build up the optimal-dependencies d-graph (on the basis of the structural-dependencies-graph that can be considered input information) level-by-level in bottom-up way (At the beginning all p-vertices are places at level 0. All effective updates along the corresponding p-arcs move their p-end to higher level than the highest p-son of their d-end.).

Proposition 3 *If the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality has no black p-sources, then there exists at least one p-arc corresponding to an* effective complete *updating operation.*

Proof. Since the optimization problem satisfies the principle of the optimality the optimal-updating-sequence there exists and continuously warrants (while no black p-sources still exist) the existence of optimal updating operations, which are effective and complete too. □

Proposition 4 *Any p-arcs sequence (of the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality) that continuously applies non-repetitive complete updates (while such updating operations still exist) warrants that all p-sources become black-coloured. These p-arcs sequences contain arcs representing optimization-dependencies and surely include an optimal-arc-sequence.*

Proof. Since the optimization problem satisfies the principle of the optimality the optimal-updating-sequence there exists and warrants the continuous existence of optimal updating operations (which are also effective and complete) while no black p-sources still exist. Accordingly any p-arcs sequence that continuously applies non-repetitive complete updates includes an optimal-arc-sequence, and consequently results in colouring all p-sources with black. □

Proposition 5 *If the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality is cycle-free, then any reverse topological order of the all p-arcs continuously applies non-repetitive complete updates, and consequently, results in building up the optimal solution of the problem.*

Proof. Since the colours of the d-vertices surely become black after all their p-sons have already become black, any reverse topological order of all p-arcs continuously applies non-repetitive complete updates. According to the previous proposition these arc-sequences surely include an optimal-arc-sequence, and consequently results in building up the optimal solution of the problem. □

Proposition 6 *If an optimization problem satisfies the principle of the optimality, then there exists a finite multiple complete arc-sequence of the attached structural-dependencies d-graph that includes an optimal-arc-sequence, and consequently, the corresponding updating-sequence results in building up the optimal solution of the problem.*

Proof. The existence of such an arc-sequence immediately results from the facts that: (1) Any complete arc-sequence contains all arcs of the optimal-dependencies-tree; (2) The optimal-dependencies-tree is finite. If we repeat a complete arc-sequence that includes the arcs of the optimal-dependencies-tree according to their topological order (worst case), then we need as many updating-tours as the number of the p-arcs of the optimal-dependencies-tree is. □

6.1.1 Algorithm d-TOPOLOGICAL

If the structural-dependencies d-graph attached to the problem is cycle free (called: structurally acyclic DP problems), then this input graph can also be viewed as optimization-dependencies-graph. Considering a reverse topological order of the *all* vertices, all updating operations (along the corresponding p-arc-sequence) will be complete (see Proposition 5). Additionally, along the arcs of the optimal d-spanning-tree we have optimal updates. Accordingly, this algorithm (called d-TOPOLOGICAL) results in determining the optimal solution of the problem.

6.1.2 Algorithm d-DIJKSTRA

If the structural-dependencies d-graph contains cycles a proper vertices order involving complete updates along the corresponding p-arc-sequence can-

not be structurally established. In this case we should try to build up the optimization-dependencies d-graph (more exactly a reverse topological order of its all p-arcs) on the fly, parallel with the bottom-up optimization process.

Implementing a sequence of continuous complete updates presumes to identify at each stage of the building process the black d-vertices based on which we have not performed complete updating operations (Proposition 3 guaranties that such d-vertices exist continuously). A d-vertex is black only if all its p-sons are already black. Consequently, the basic question is: Can we identify the black p-vertices at each stage of the building process? As we mentioned above a p-vertex is certainly black after we have performed complete updates based on *all* its d-sons (The last effective update was performed on the basis of optimal d-son). Algorithms based on the topological order of the all arcs exploit this *structural* condition. However, a p-vertex may have become black before we have performed complete updating operation along all its p-out-arcs. Conditions making perceptible such black p-vertices may also be deduced from the principle of the optimality. For example, if the DP problem has a greedy character too, then it may work the following condition: the "best" d-vertex (having *relatively* optimal weight) among those based on which we have not performed complete updating operations can be considered black (Called: Cyclic DP problems characterized by greedy choices). Since Dijkstra algorithm applies this strategy, we call this algorithm: d-DIJKSTRA.

6.1.3 Algorithm d-BELLMAN-FORD

If we cannot establish one complete arc-sequence including an optimal-arc-sequence (we will call such problems: DP problems without 'negative cycles'), we are forced to repeat the updating-tour along a complete (even arbitrary) arc-sequence of the input graph (structural-dependencies d-graph) until this *multiple arc-sequence* will include the desired optimal updating sequence (see Proposition 6). An extra tour without any effective updates indicates that the optimal solution has been built up. If the arbitrary arc-sequence we have chosen includes the arcs of the optimal-dependencies-tree in topological order (worst case), then we need as many updating-tours as the number of the p-arcs of the optimal-dependencies-tree is. Since Bellman-Ford algorithm applies this strategy, we call this algorithm: d-BELLMAN-FORD.

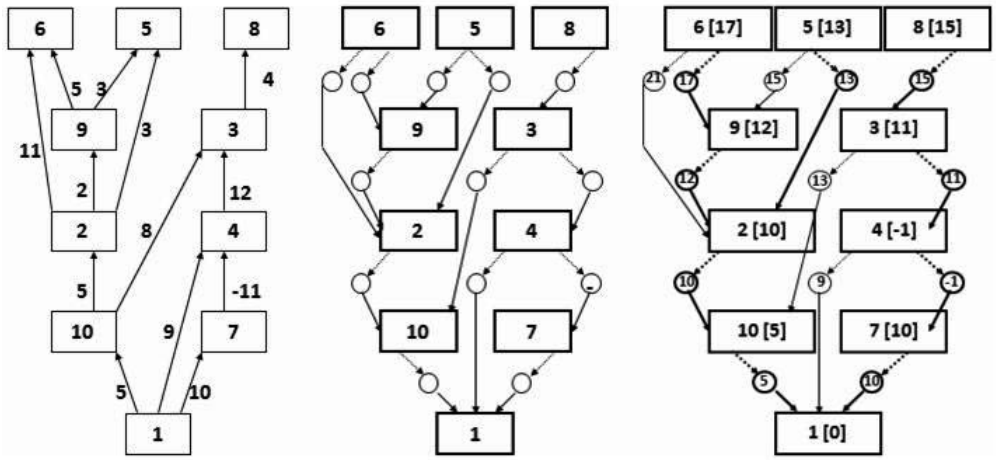
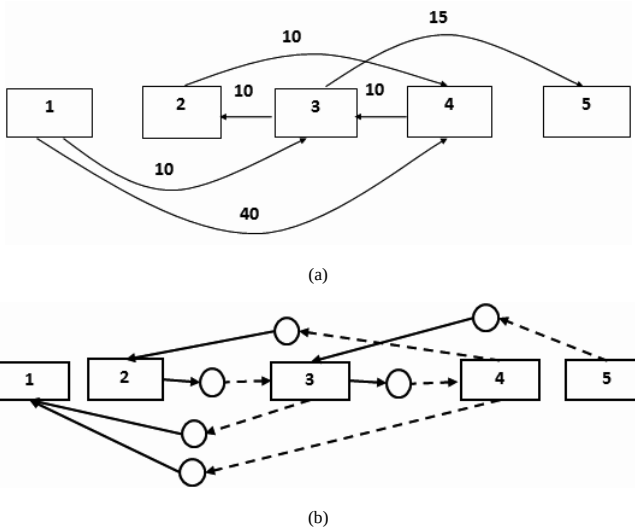


Figure 5: (a) Acyclic digraph; (b) Structural-dependencies d-graph; (c) Optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph)



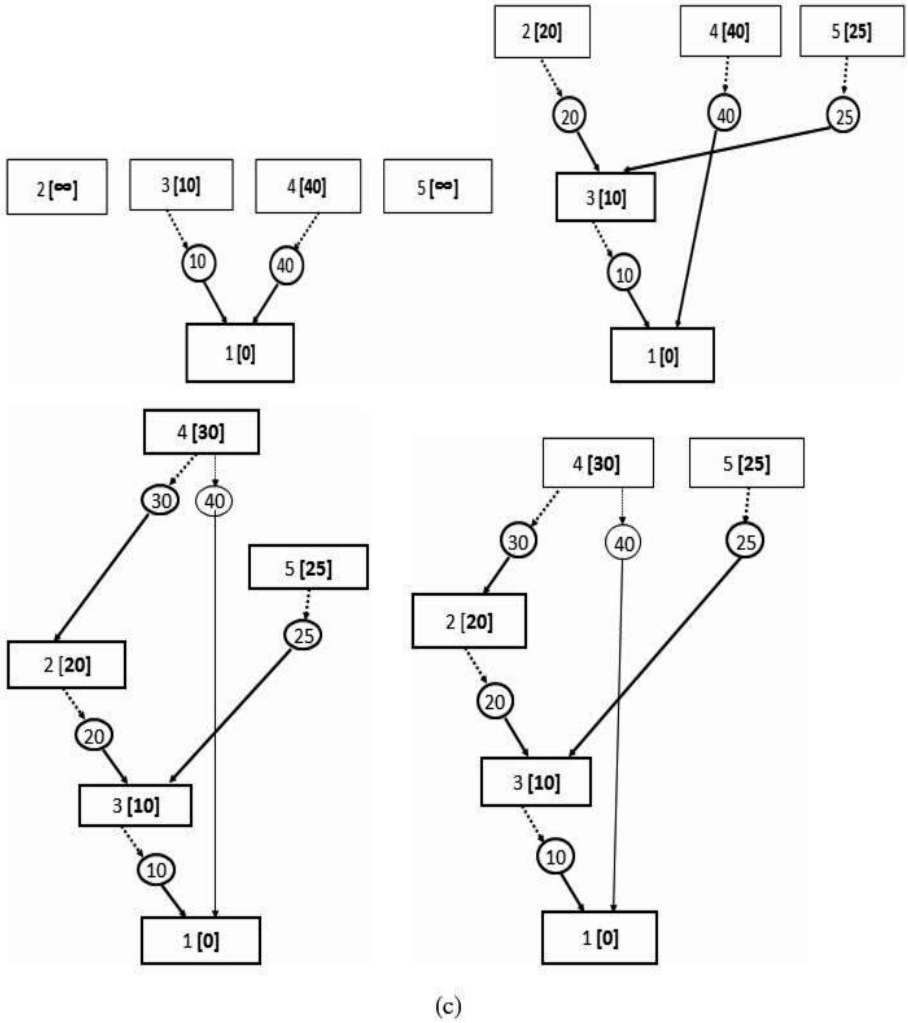
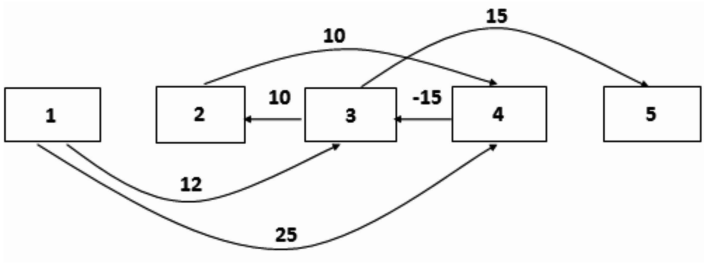
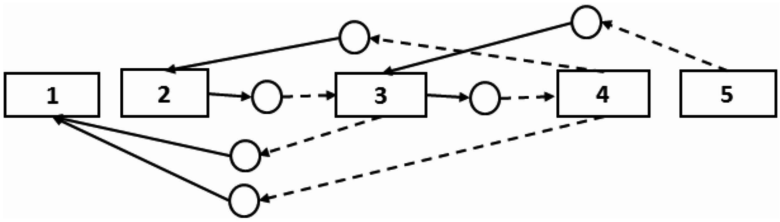


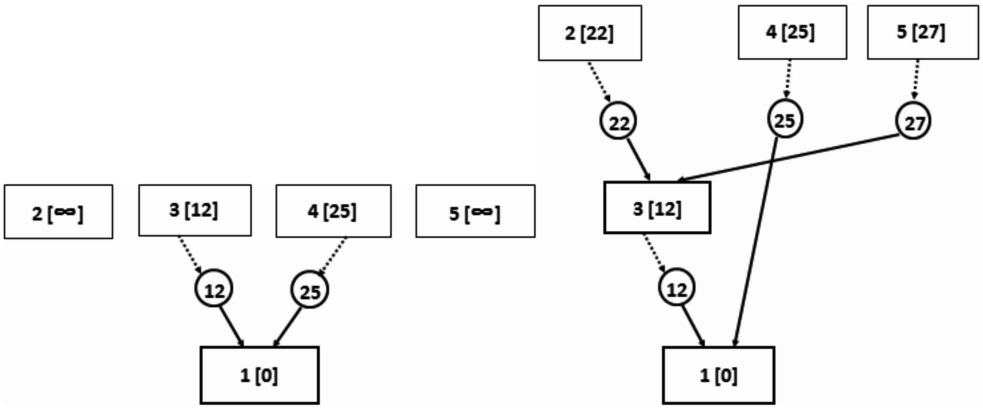
Figure 6: (a) Cyclic digraph without negative weighted arcs; (b) Cyclic structural-dependencies d-graph; (c) The bottom-up building-up process of the optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph)


15
10
1
2
10
3
-15
4
5
12
25

(a)


1
2
3
4
5

(b)


2 [∞]
3 [12]
4 [25]
5 [∞]
12
25
1 [0]
2 [22]
4 [25]
5 [27]
22
25
27
3 [12]
12
1 [0]

(c1) (c2)

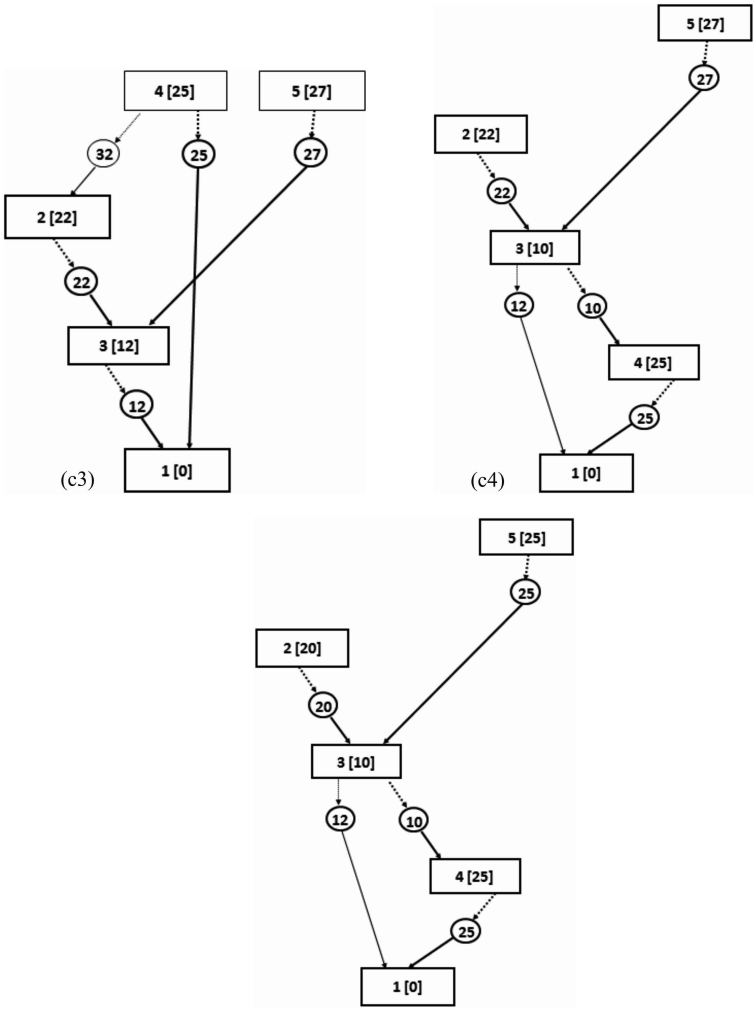


Figure 7: (a) Cyclic digraph with negative weighted arcs, but without negative cycles; (b) Cyclic structural-dependencies d-graph; The bottom-up building-up process of the optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph): (c1–c4) first updating-tour, (d) second updating-tour

6.1.4 A relevant sample problem

As an example we consider the single-source shortest problem: Given a weighted digraph determine the shortest paths from a source vertex to all the other vertices (destination vertices). The attached figures (see Figures 5, 6, 7) illustrate the level by level building process of the optimization-dependencies d-graph concerning to the algorithms d-TOPOLOGICAL, d-DIJKSTRA and d-BELLMAN-FORD (Regarding this problem we have only $1 \rightarrow 1$ dependencies between neighbour p-vertices).

7 Conclusions

Introducing the generalized version of d-graphs we received a more effective tool for modelling a larger class of DP problems (Hierarchic d-graphs introduced in [8] and Petri-net based models [14, 15, 16] work only in the case of structurally acyclic problems; Classic digraphs [11, 12] can be applied when during the decomposing process at each step the current problem is reduced to only one sub-problem). The new modelling method also makes possible to classify DP problems (Structurally acyclic DP problems; Cyclic DP problems characterized by greedy choices; DP problems without 'negative cycles') and the corresponding DP strategies (d-TOPOLOGICAL, d-DIJKSTRA, d-BELLMAN-FORD) in term of graph theory.

If we have proposed to develop a general software-tool that automatically solves DP problems (getting as input the functional equation) we should combine the above algorithms as follows:

- We represent explicitly the d-graph described implicitly by the functional equation.
- We try to establish the reverse topological order of the vertices by a DFS like algorithm (d-DFS). This algorithm can also detect possible cycles.
- If the graph is cycle free, we apply algorithm d-TOPOLOGICAL, else we try to apply algorithm d-DIJKSTRA.
- If no mathematical guarantees that we reached the optimal solution, then choosing as complete arc-sequence for algorithm d-BELLMAN-FORD the arc-sequence generated by algorithm d-DIJKSTRA (completed with unused arcs) in the first updating-tour we verify the d-DIJKSTRA result. We repeat the updating tours until no more effective updates.

Such a software-application should be able to save considerable software development costs.

8 Acknowledgements

This research was supported by the Research Programs Institute of Sapientia Foundation, Cluj, Romania.

References

[1] R. Bellman, *Dynamic programming*, Princeton University Press, Princeton, NJ, 1957. ⇒210

[2] R. Bellman, S. Dreyfus, *Applied dynamic programming*, Princeton University Press, Princeton, NJ, 1962. ⇒210

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms,* 3rd edition, The MIT Press, Cambridge, MA, USA, 2009. ⇒ 210

[4] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis,* Cambridge University Press, Cambridge, UK, 1998. ⇒210

[5] P. F. Felzenszwalb, R. Zabih, Dynamic programming and graph algorithms in computer vision, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Preprint, 19 July 2010, 51 p. http://doi.ieeecomputersociety.org/10.1109/TPAMI.2010.135 ⇒211

[6] Z. Feng, R. Dearden, N Meuleau, R. Washington, Dynamic programming for structured continuous Markov decision problems, *Proc. 20th Conference on Uncertainty in Artificial Intelligence, ACM International Conference Proceeding Series*, AUAI Press, **70** (2004) pp. 154–161. ⇒210

[7] H. Georgescu, C. Ionescu, The dynamic programming method, a new approach, *Studia Univ. Babeş-Bolyai Inform.*, **43,** 1 (1998) 23–38. ⇒210

[8] Z. Kátai, Dynamic programming and d-graphs, *Studia Univ. Babeş-Bolyai Inform.*, **51,** 2 (2006) 41–52. ⇒210, 214, 215, 217, 228

[9] Z. Kátai, Dynamic programming strategies on the decision tree hidden behind the optimizing problems, *Informatics in Education*, **6,** 1 (2007) 115–138. ⇒211

[10] Z. Kátai, The single-source shortest paths algorithms and the dynamic programming, *Teaching Mathematics and Computer Science*, **6,** Special Issue (2008) 25–35. ⇒211, 218, 220

[11] Z. Kátai, Dynamic programming as optimal path problem in weighted digraphs, *Acta Math. Acad. Paedagog. Nyházi*, **24,** 2 (2008) 201–208. ⇒ 211, 228

[12] Z. Kátai, A. Csiki, Automated dynamic programming, *Acta Univ. Sapientiae Inform.*, **1,** 2 (2009) 149–164. ⇒211, 212, 228

[13] I. King, *A simple introduction to dynamic programming in macroeconomic models*, 2002.
http://researchspace.auckland.ac.nz/bitstream/handle/2292/190/230.pdf
⇒210

[14] A. Lew, A Petri net model for discrete dynamic programming, *Proc. 9th Bellman Continuum: International Workshop on Uncertain Systems and Soft Computing,* Beijing, China, July 24–27, 2002, pp. 16–21. ⇒211, 228

[15] A. Lew, H. Mauch, Bellman nets: A Petri net model and tool for dynamic programming, *Proc. 5th Int. Conf. Modelling, Computation and Optimization in Information Systems and Management Sciences* (MCO), Metz, France, 2004, pp. 241–248. ⇒211, 228

[16] H. Mauch, DP2PN2Solver: A flexible dynamic programming solver software tool, *Control Cybernet.*, **35**, 3 (2006) 687–702. ⇒211, 212, 228

[17] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, *Control Cybernet.*, **35,** 3 (2006) 599–620. ⇒211, 217, 218

[18] D. Vagner, Power programming: dynamic programming, *The Mathematica Journal*, **5,** 4 (1995) 42–51. ⇒217

Received: September 1, 2010 • Revised: November 10, 2010

Contents
Volume 2, 2010

Acta Universitatis Sapientiae

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

Acta Universitatis Sapientiae, Informatica

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

Each volume contains two issues.

Sapientia University

Scientia Publishing House

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

Information for authors

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

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

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

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

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

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

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

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

Publication supported by

Printed by Gloria Printing House
Director: Péter Nagy