Acta Universitatis Sapientiae

Informatica

Volume 1, Number 2, 2009

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 125–134

Some notes on drawing twofolds in 4-dimensional Euclidean space

Dedicated to the memory of Professor Elemér Kiss (1929-2006), who would be 80 this year.

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

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

Abstract. In the present paper we give an elementary and illustrative proof that in $\mathbf{E}^4$, the complete surfaces with constant positive curvature are not isomorphic. It is well-known, if two surfaces in $\mathbf{E}^3$ are complete with the same positive curvature they are global isomorphic. The same statement is not true in $\mathbf{E}^4$, although these surfaces remain global isometric. We will illustrate our proof with some nice examples.

1 Introduction

Many articles [9, 17, 18] presented the techniques of drawing objects in higher dimension than three, and they also highlighted the educational importance of them.

The subject of this paper is related to these drawing techniques and we want to state that the drawings should reflect the fact that the closed, compact,

AMS 2000 subject classifications: 53A05, 53A07
CR Categories and Descriptors: I.3.5. [Computational Geometry and Object Modeling]
Key words and phrases: surface representations

smooth surfaces in three dimensions will remain closed, compact and smooth in four dimensions too.

In the 4-dimensional Euclidean space there is an unsolved problem, namely whether there exists a complete analytical hyperbolic plane in $\mathbf{E}^4$. We know that there is no surface having this property in $\mathbf{E}^3$ (Hilbert theorem), but in $\mathbf{E}^5$ [6, 10, 15] and in $\mathbf{E}^6$ [1] it is possible. Thus the question is more exciting in $\mathbf{E}^4$ and proving either the existence or the non-existence would be an important result.

There is another interesting question: what is the graphical image in $\mathbf{E}^4$ for a compact, complete analytical surface which has negative constant curvature? Such a surface exists because it was given by Ōtsuki [13]. The surface constructed by him in $\mathbf{E}^4$ has negative curvature but it is not constant. On the other hand, with the constructed surface Ōtsuki [12] demonstrated that there are compact and complete surfaces with negative curvature in $\mathbf{E}^4$.

Furthermore, we are studying only the surfaces with positive constant curvature. It is well-known from Cohn-Vossen and Herglotz theorem [3, 8] that if two surfaces are complete with the same positive curvature, they are global isomorphic. Our aim is to give an elementary proof that in higher dimension than three, the complete surfaces with constant positive curvature will not remain rigid. Here rigid means that a complete surface with constant positive curvature could not be transformed into itself by one parameter movement. We also illustrate the proof with some examples using different drawing techniques.

2 The basic idea

It is well-known that a surface of revolution is a surface generated by rotating a plane curve about an axis. By definition, the axis of the surface of revolution is a straight line, although the axis of rotation can be imagined as a space curve. In the latter case we find a generalization of the surface of revolution, called canal surface. In other words, the canal surface is a surface formed as the envelope of a family of spheres whose centers lie on a space curve. If the sphere centers lie on a straight line, the channel surface is a surface of revolution. For example, the sphere is a special canal surface, whose axis is a straight line.

In the next part we use a simple mathematical deduction to prove that complete surfaces with constant positive curvature are not global isomorphic.

Let $\mathfrak{p}(\mathfrak{u}) = (\mathfrak{x}(\mathfrak{u}), \mathfrak{y}(\mathfrak{u}))$ be a planar curve, parameterized by arc length.

The corresponding Frenet formulas have the following form:

$$\begin{aligned} p'(u) &= e(u), \\ e'(u) &= \kappa(u)n(u), \\ n'(u) &= -\kappa(u)e(u), \end{aligned}$$

where the tangent vector for the curve p is $e = e(u)$, the normal vector is $n = n(u)$ and the binormal vector is $b = b(u)$. We suppose $b'(u) = 0$, which means that p is a planar curve. The canal surface of the planar curve p has the following form:

$$f(u, v) = p(u) + r(u)(n(u)\cos(v) + b(u)\sin(v)),$$

where $r(u)$ is the radius of the spheres from the definition of the canal surface.

According to our aim, we put the condition that the surface has positive Gaussian curvature. This means that

$$G(u) = +1 \tag{1}$$

equation must hold, where G is the curvature of the surface. In order to solve this equation first we try to calculate the curvature of the canal surface using the next fundamental forms of it:

$$\begin{aligned} g_{11} &= r'^2(u) + (1 - \kappa(u)r(u)\sin(v))^2, \\ g_{12} &= 0, \\ g_{22} &= r^2(u). \end{aligned}$$

Furthermore, if we put the condition $\kappa(u) = 0$, then we get the Gauss curvature:

$$G(u) = -\frac{r''(u)}{r(u)(1 + r'^2(u))^2}. \tag{2}$$

By replacing the found expression into formula (1), we get the following equation:

$$r''(u) = -r(u)(1 + r'^2(u))^2. \tag{3}$$

Equation (3) can be solved by integrating elementary, but we are interested in a result, which gives us the sphere as solution, so we get the next particular result: $r(u) = \sqrt{2u - u^2}$, where $u \in [0, 2]$.

Furthermore, we repeat the previous sequence of ideas in 4 dimensions, and we choose a space curve in $\mathbf{E}^3$ with this form: $p(u) = (x(u), y(u), z(u))$. Then we get by Frenet formulas in $\mathbf{E}^4$:

$$\begin{aligned} p'(u) &= e_1(u), \\ e_1'(u) &= \kappa(u)e_2(u), \\ e_2'(u) &= -\kappa(u)e_1(u) + \tau(u)e_3(u), \\ e_3'(u) &= -\tau(u)e_2, \\ e_4'(u) &= 0, \end{aligned}$$

where $\{e_1, e_2, e_3, e_4\}$ is the Frenet orthonormal basis. The canal surfaces in $\mathbf{E}^4$ have the following form:

$$f(u, v) = p(u) + r(u)(e_3(u)\cos(v) + e_4(u)\sin(v))$$

The Gauss fundamental forms for the surface f are the following equations:

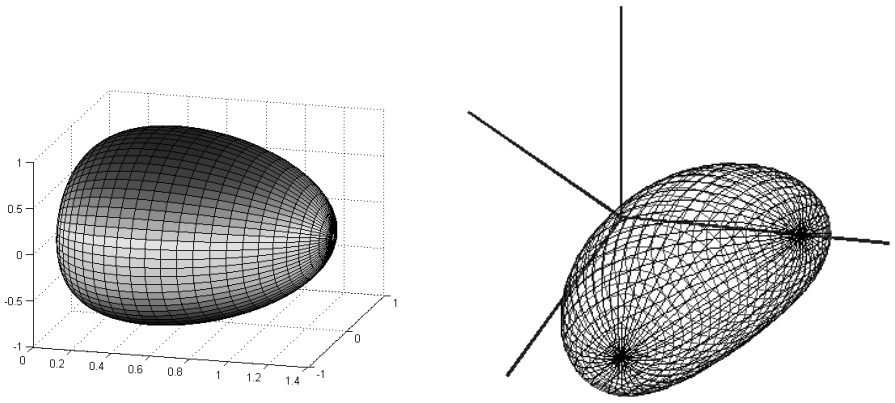
$$\begin{aligned} g_{11} &= r'^2(u) + 1 + r^2(u)\tau^2(u)\cos^2(v), \\ g_{12} &= 0, \\ g_{22} &= r^2(u). \end{aligned}$$

Furthermore, we put the condition that the torsion of the space curve has null value. This means that the space curve is a plane curve. By continuing the calculations, we get the curvature formula (2) for the surface and we are again interested in those solutions of equation (1), which give us complete surfaces.

The calculations reflect the fact that the curvature for these surfaces is independent of the form of the planar curve in $\mathbf{E}^4$, which is the axis of the surface. In other words, the axes of the canal surface can be chosen in many ways, hence there is an infinite number of surfaces with positive constant curvatures. To summarize our results, we state the following theorem:

Theorem 1 *For each surface of revolution with positive constant curvature in* E^3 *there are corresponding infinite number of canal surfaces with positive constant curvature in* E^4.

These results have a geometric interpretation, too. For example, let us consider the 3-dimensional sphere. It is well-known that we get it from the rotation of the circle around its diameter. If we take the sphere by its north and



(a) (b)

Figure 1: The Ōtsuki sphere.

south pole and if we "bend" the rotation axis towards the fourth dimension, the surface will still have a constant curvature which is reflected in the calculations, but we get spheres which will not be isomorphic in the 4-dimensional space. If the shape of the rotation axis is a quarter of the asteroid then we get the famous sphere of Ōtsuki [11] (see Fig. 1) represented by the (4)–(7) equations:

$$x_1(u,v) = \frac{4}{3}\cos^3\frac{u}{2}, \tag{4}$$

$$x_2(u,v) = \frac{4}{3}\sin^3\frac{u}{2}, \tag{5}$$

$$x_3(u,v) = \sin(u)\cos(v), \tag{6}$$

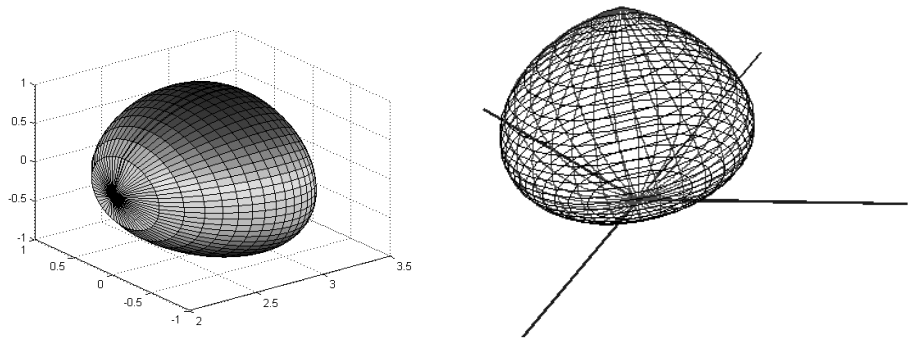
$$x_4(u,v) = \sin(u)\sin(v), \tag{7}$$

where $u \in [0,\pi], v \in [0,2\pi]$.

Furthermore, we give three other examples of complete surfaces with constant positive curvature in $\mathbf{E}^4$:

Example 1

$$\begin{aligned} x_1(u,v) &= x_1(u) = 2\arcsin(u/2) + \sqrt{4-u^2}, \\ x_2(u,v) &= x_2(u) = \sqrt{2}\sqrt{(2+u)u} - \sqrt{2}\ln(1+u+\sqrt{(2u+u^2)}), \\ x_3(u,v) &= \sqrt{u(2-u)}\sin v, \\ x_4(u,v) &= \sqrt{u(2-u)}\cos v, \quad u \in [0,2], v \in [0,2\pi]. \end{aligned}$$



(a) (b)

Figure 2: Drawings for Example 1: (a) Intersection with a hyperplane. (b) Axonometric mapping from $\mathbf{E}^4$ into $\mathbf{E}^2$.

Example 2

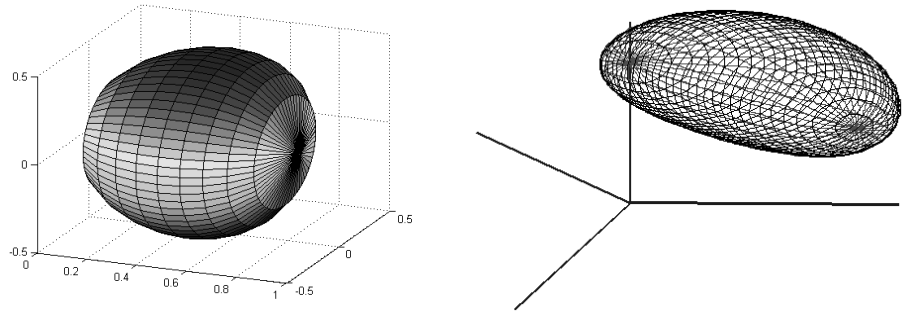
$$\begin{aligned} x_1(u,v) &= x_1(u) = \sin u, \\ x_2(u,v) &= x_2(u) = \cos u, \\ x_3(u,v) &= \sqrt{u(1-u)}\sin v, \\ x_4(u,v) &= \sqrt{u(1-u)}\cos v, \quad u \in [0,1], v \in [0,2\pi]. \end{aligned}$$

Example 3

$$\begin{aligned} x_1(u,v) &= x_1(u) = 2\sin(u/2), \\ x_2(u,v) &= x_2(u) = 2\cos(u/2), \\ x_3(u,v) &= \sqrt{u(2-u)}\sin v, \\ x_4(u,v) &= \sqrt{u(2-u)}\cos v, \quad u \in [0,2], v \in [0,2\pi]. \end{aligned}$$

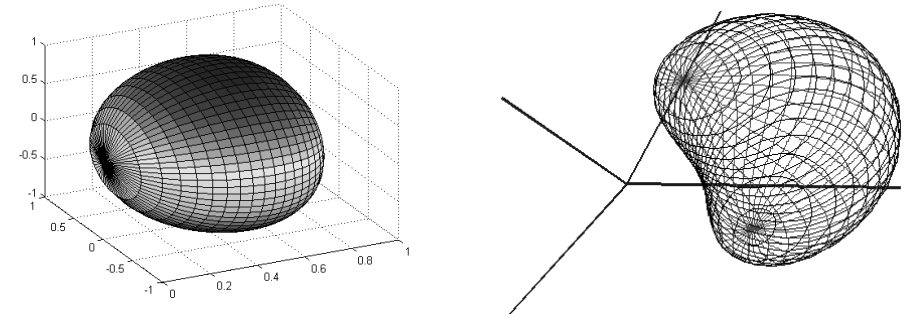
3 The used drawing technigues

We have drawn two kinds of figures using MATLAB programming language. The first type of drawings are intersections in $\mathbf{E}^4$ with a hyperplane. This means that we omit one of the four coordinates from the surface representation, and after that we apply an axonometry by mapping the three-dimensional figure onto the plane (see Fig. 1(a), 2(a), 3(a), 4(a)).



(a) (b)

Figure 3: Drawings for Example 2: (a) Intersection with a hyperplane. (b) Axonometric mapping from $\mathbf{E}^4$ into $\mathbf{E}^2$.



(a) (b)

Figure 4: Drawings for Example 3: (a) Intersection with a hyperplane. (b) Axonometric mapping from $\mathbf{E}^4$ into $\mathbf{E}^2$.

The second type of figures are axonometric mappings from $\mathbf{E}^4$ into $\mathbf{E}^2$ (see Fig. 1(b), 2(b), 3(b), 4(b)). The transformation has the following form:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \begin{pmatrix} x_1(u,v) \\ x_2(u,v) \\ x_3(u,v) \\ x_4(u,v) \end{pmatrix},$$

where the rank of the transformation matrix $[a_{ij}]_{2\times 4}$ is equal to 2. These kinds of mapping techniques were studied by Szabó [17, 18], and he proved that the objects can be also represented in $\mathbf{R}^n$, both in axonometric and perspective way. These mappings keep their straight lines and proportion in case of axonometry and in case of perspective, they keep their straight lines and cross-ratio.

On the other hand, Stiefel [16] showed that in $\mathbf{E}^4$ the Pohlke's theorem (i.e. the axonometric image of a shape is similar to the parallel projection of the shape) is not valid. Nevertheless, some properties remain valid. For example the close parameter lines on the surfaces in $\mathbf{E}^4$ are transformed into closed curves as you can see in the figures.

4 Conclusions

In this paper we considered constant positive curvature surfaces from the 4-dimensional Euclidean space. Many surfaces with constant positive curvature have the interesting property that they are not global isomorphic in $\mathbf{E}^4$, while in $\mathbf{E}^3$ the same property is not true. We have proved this property mathematically and also illustrated with some nice examples.

References

[1] D. Blanuša, Über die Einbettung hyperbolischer Raume in euklidische Raume, *Monatsh. Math.*, **59**, 3 (1955) 217–229.

[2] S. Brian, X. Frederico, Efimov's theorem in dimension greater than two, *Invent. Math.*, **90**, 3 (1987) 443–450.

[3] S. Cohn-Vossen, The isometric deformability of surfaces in the large, *Uspehi Mat. Nauk*, **1**, 1 (1936), 33–76 (in Russian).

[4] N. V. Efimov, Generation of singularities on surfaces of negative curvature, *Mat. Sb. (N.S.)*, **64** (1964) 286–320.

[5] W. Henke, Isometrische Immersion des n-dim. hyperbolischen Raumes H_n in E^{4n-3}, *Manuscripta Math.*, **34**, 2–3 (1981) 265–278.

[6] W. Henke, Isometric immersion of n-dim. hyperbolic spaces into (4n-3)-dim. standard spheres and hyperbolic spaces, *Math. Ann*, **258**, 3 (1982) 341–348.

[7] W. Henke, W. Nettekoven, The hyperbolic n-space as a graph in Euclidean $(6n-6)$ space, *Manuscripta Math.*, **59**, 1 (1987) 13–20.

[8] G. Herglotz, Über die Starrheit der Eiflachen, *Abh. Math. Sem. Univ. Hamburg* **15**, 1 (1943) 127–129.

[9] Z. Kovács, L. Kozma, Assimilation of mathematical knowledge using Maple, *Teaching Mathematics and Computer Science* **1**, 2 (2003) 321–331.

[10] R. Oláh-Gál: The n-dimensional hyperbolic spaces in $\mathbf{E}^{4n-3}$, *Publ. Math. Debrecen*, **46**, 3–4 (1995) 205–213.

[11] T. Ōtsuki, Surfaces in the 4-dimensional euclidean space isometric to a sphere, *Kodai Math. Sem. Rep.*, **18**, 2 (1966) 101–115.

[12] T. Ōtsuki, On the total curvature of surfaces in Euclidean spaces, *Japan. J. Math.*, **35** (1966) 61–71.

[13] T. Ōtsuki, A construction of closed surfaces of negative curvature in $\mathbf{E}^4$, *Math. J. Okayama Univ.*, **3**, 2 (1954) 95–108.

[14] A. Ros, Compact hypersurfaces with constant scalar curvature and a congruence theorem, *J. Differential Geom.*, **27**, 2 (1988) 215–223.

[15] E. R. Rozendorn, A realization of the metric $ds^2 = du^2 + f^2(u)\ dv^2$ in five-dimensional Euclidean space, *Akad. Nauk Armjan. SSR Dokl.*, **30** (1960) 197–199 (in Russian).

[16] E. Stiefel, Zum Satz von Pohlke, *Comment. Math. Helv.*, **10**, 1 (1937) 208–225.

[17] J. Szabó, Die Verallgemeinerung des Eckhartschen Einschneiderfahrens auf den n-dimensionalen Fall, *Publ. Math. Debrecen*, **15** (1968) 181–187.

[18] J. Szabó, On the axonometrical projection in the computer graphics, *Acta Math. Acad. Paedagog. Nyíregyháziensis*, **17**, 3 (2001) 179–183.

[19] G. Vranceanu, G. G. Vranceanu, Surfaces with positive constant or null curvature in $\mathbf{E}^4$, *Rev. Roumaine Math. Pures Appl.*, **22** (1977) 281–287.

[20] G. Vranceanu, Surfaces de rotation dans $\mathbf{E}^4$, *Rev. Roumaine Math. Pures Appl.*, **22** (1977) 857–862.

Received: February 23, 2009

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 135–148

On confluent drawings: visualizing graphs using an ortho-confluent system

Irina Honciuc
Al. I. Cuza University of Iaşi
Computer Science Faculty
email: irina.honciuc@info.uaic.ro

Cornelius Croitoru
Al. I. Cuza University of Iaşi
Computer Science Faculty
email: croitoru@info.uaic.ro

Abstract. Confluent drawing is a technique that allows visualizing non-planar graphs in a crossing-free manner. Its central idea is very simple: subsets of graph's edges are merged into confluence points and drawn as smooth curved lines, similar to train tracks. This approach eliminates edge crossings and offers an aesthetically pleasant representation for the initial graph. This article presents the ortho-confluence technique, which introduces the idea of local orthogonal system relative to a graph's node. The concept of ortho-confluence was successfully implemented in our application named ConfluentViz, and the results are presented in the final part of this article.

1 Introduction

In 2003, Dickerson, Eppstein, Goodrich and Meng introduced confluent drawing, and with it a heuristic that is able to generate confluent drawings for some graphs [4]. The problem of finding a proper representation without edge crossings for a non planar graph is not very straight forward. But there are heuristics that determine whether a non-planar graph can be efficiently drawn in a confluent way.

A particular approach in confluent drawings is ortho-confluence. This represents a way of drawing a graph in a confluent manner, such that some edges

AMS 2000 subject classifications: 68R10
CR Categories and Descriptors: G.2.2. [Graph Theory]: Graph algorithms
Key words and phrases: graph theory, graph visualization, confluence, ortho-confluence

of the initial graph are merged with a horizontal or vertical track corresponding to a local cartesian coordinate system that has its origin in another node called parent. The parent of a node in a confluent representation is, in general, the parent node given by the BFS order. Depending on the graph structure, a parent node in the confluent representation can also be a node that has a very large degree or a node that is connected with other nodes by a long edge. This article offers a close look on ortho-confluence technique as it is successfully implemented in the ConfluentViz application. This is a graph editor tool that enables the user to create and manage ortho-confluent representations for various graph categories: trees, forests, graphs containing circuits, graphs containing large cliques (bicliques).

The first sections of this paper present the context of the topic and the existing work including some important results. The following sections are related to the ortho-confluence concept. Furthermore, the implementation results and issues are explained in detail. At the end of our paper, we present some conclusions and further work.

2 Motivation and problem description

Graph visualization is a vast area of research [1, 11, 12, 13, 14]. Developing an intelligent tool that produces visual representations for different graph categories is a challenging process; it implies finding a suitable algorithm that takes a graph as an input and outputs an equivalent representation. This final representation must satisfy some aesthetic criteria in order to be relevant. It is highly desired to have less crossing edges, a good positioning for the vertices and edges, optimal angles, all these on a minimum of drawing area. Perhaps the most important criteria is edge crossings minimization, because crossed edges make the relations in the representation difficult to identify. The ideal output would be the one with no edge crossings at all.

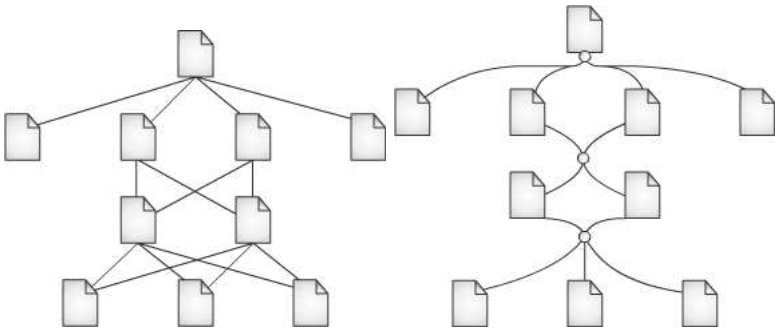
Graphs that can be represented in a standard way on a plane surface with no edge crossings are called planar graphs [15]. There are efficient algorithms that produce representations with no edge crossings for planar graphs [1]. Unfortunately, most of the graphs that appear in real life models are not planar. Thus, most of these graphs cannot be represented in a standard way without edge crossings. There are algorithms for minimizing edge crossings in non planar graphs, but the general problem of representing a non-planar graph in a standard way that minimizes edge crossings is NP-hard [7].

However, representing non planar graphs nicely is a common problem in

many domains. For example, in software visualization there are diagrams for representing application architecture, class diagrams, method calls, data flow processes, object interactions. In these diagrams the components, the entities or the objects are drawn as simple shapes: circles, squares, triangles, etc.

An important advantage of confluent representations is that in such diagrams we can easily identify source and destination nodes for the edges that share a common portion. These common structures could indicate in a method-call diagram separate methods that can be joined together for efficiency. Similarly, structures in which many sources communicate all with many destinations could indicate the need for refactoring or could offer new perspectives for changing the software design.

The navigation rules in a web application are represented in Fig. 1 in a standard way and in a confluent manner.



(a) in a standard way (b) in a confluent way

Figure 1: Two representations of a web site navigation rules

Other applications for graph visualization also include different airline maps, subway maps, social networks, genealogy. We want to obtain this kind of representations automatically. Thus, we need efficient algorithms to generate software diagrams or maps that preserve the relations in the model and at the same time the output is pleasant for the human eye.

3 Existing work

M. Dickenson, D. Eppstein, M. Goodrich, and J. Meng introduced [4] the concept of confluent representations as a way of visualizing non planar diagrams in a planar way and presented algorithms that output confluent representations for both directed and undirected graphs, mainly for graphs that appear

frequently in software visualizations.

The concept is quite simple: some edges are merged together forming "tracks" so that their intersections become overlapping paths. The resulting graphs are easier to comprehend, yet keeping a high degree of connectivity information. Some airline companies already use these confluent representations for displaying route maps. Also, similar diagrams are present in surface topology.

It is well-known that every non-planar graph contains a subgraph homeomorphic to the complete graph of five vertices, K_5, or the complete bipartite graph between two sets of three nodes, $K_{3,3}$ [15]. On the other hand, every K_n or $K_{n,m}$ admits a confluent representation as it is indicated in Fig. 2.

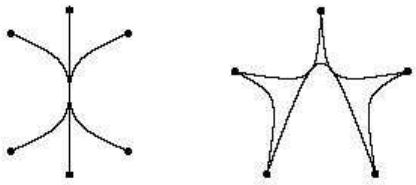


Figure 2: Confluent representations for $K_{3,3}$ and K_5

A curve is locally-monotone if it contains no self intersections and no sharp turns. Confluent representations are a way of drawing graphs on a plane surface by merging edges into paths that are unions of locally-monotone curves. An undirected graph G is confluent if and only if there exists a drawing A such that:

- There is a one-to-one mapping between the vertices in G and A, so that, for each vertex $v \in V(G)$, there is a corresponding vertex $v' \in A$, which has a unique point placement in the plane. In other words, there is a bijective function between the vertices in G and A.
- There is an edge $(v_i, v_j) \in E(G)$ if and only if there is a locally-monotone curve e' connecting v_i' and v_j' in A.
- A is planar. That is, while locally-monotone curves in A can share overlapping portions, no two of them can cross.

In a confluent representation A, a confluence point is defined as the point in plane, where two or more locally-monotone curves are merged together.

Directed confluent representations are defined similarly, except that in such drawings the locally-monotone curves are directed and the tracks formed by union curves must be oriented consistently.

There are two important visual elements that are used in confluent representations: *traffic circles* and *switches*. A switch is a common point for two or more curves or a point in which these curves change direction. A traffic circle could be defined as a confluent representation of a clique so that all the locally-monotone curves share a common portion with a circular track. This way, the clique property is preserved, that is any node is reachable from any other node (Fig. 3). Traffic circles partially solve the crossing edges problem, offering a simplified view of the representation and also a suggestive way of representing multiple connections between nodes. For example, many important cities reduced the traffic problems by eliminating cross intersections, replacing them with traffic circles.

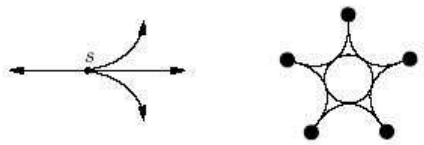


Figure 3: A switch and a traffic circle (representing K_5)

Although testing the planarity of a graph can be done in linear time, the problem of deciding whether a graph has a confluent representation is quite difficult. The main idea of the algorithm that outputs a confluent representation for a graph G is to find all the clique and biclique subgraphs of G and replace them with traffic circles. This algorithm applies especially on sparse graphs. It has been shown that the time complexity of this algorithm is $O(n)$.

Another important result is that there are large classes of non-planar graphs that can be drawn in a planar way using the confluent approach. These classes are:

- interval graphs;
- complements of trees;
- cographs;
- complements of n-cycles.

For example, a complement of a tree or an interval graph admits a confluent representation even though they are non planar graphs. The proof for each confluence theorem for the graph classes above is done especially by construction. Still, it has been demonstrated that there are some graphs that cannot

be drawn in a confluent way [4]. Among these we have the 4-dimensional cube, a certain subgraph of the Petersen graph and the Petersen graph itself. The subgraph obtained by eliminating a node from the Petersen graph is the smallest non-confluent graph we know. Also, if we divide each edge of a graph by adding a new node, the resulted graph is non-confluent. Similarly, by adding a node on each edge of a non-planar graph and connecting it to both endpoints of that edge, the result is also non-confluent. In general, all the chordal graphs are not confluent.

In 2005, David Eppstein, Michael Goodrich and Jeremy Yu Meng introduced delta-confluent drawings [6]. Delta-confluent graphs are a generalization of tree-confluent graphs. These classes of graphs and distance-hereditary graphs, a well-known class of graphs, coincide. The idea of tree-confluent graphs was published by Hui, Schaefer and Štefankovič [10]. A graph is tree-confluent if and only if it is represented by a train track system which is topologically a tree. It is also shown in their paper that the class of tree-confluent graphs is equivalent to the class of chordal bipartite class.

- A Δ-junction is a structure where three paths are united in three distinct points. Each of these points is called a junction port.
- A Λ-junction is a structure where two of the three paths in a Δ-junction are disconnected. The two paths that are disconnected are called *tails* and the remaining one is called *head*.

Figure 4: A Δ-junction (left) and a Λ-junction (right)

A Δ-confluent drawing is a confluent drawing in which each junction is either a Δ-junction or a Λ-junction and if we replace every junction in the drawing with a new vertex, the result is a tree.

4 Ortho-confluence

We can define ortho-confluence similarly to confluent representations. We say that a graph G is ortho-confluent if and only if there is a representation A

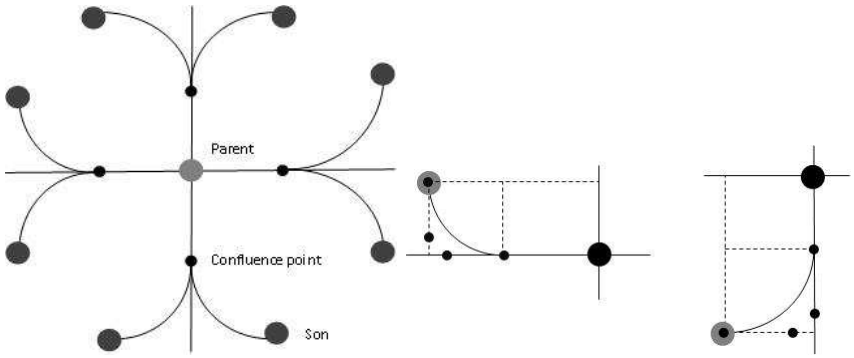
such that:

- There is a one-to-one mapping among the vertices in G and A, so that, for each vertex $v \in V(G)$, there is a corresponding vertex $v' \in A$, which has a unique point placement in the plane. In other words, there is a bijective function between the vertices in G and A.
- There is an edge $(v_i, v_j) \in E(G)$ if and only if there is a locally-monotone curve e' connecting v_i' and v_j' in A.
- A is planar. That is, while locally-monotone curves in A can share overlapping portions, no two of them can cross.
- The confluence points from any subset of curves in A must be positioned on either a vertical or a horizontal axe.

The graph classes that can be drawn both simply confluently and ortho-confluently are those in which the confluence points can be positioned on a grid. The distance between grid lines is the smallest distance between confluence points for the represented graph. Graph classes such as n-cycles complements, path complements, tree complements and interval graphs can all be represented ortho-confluently. Also, Δ-confluent graphs can be represented in an ortho-confluent manner. In general, any non-planar graph that admits a confluent representation can also be drawn ortho-confluently by applying some minor modifications:

- Traffic circles, Δ-junctions, Λ-junctions and other predefined confluence structures should be treated as nodes on a grid when representing them.
- The tangents in the endpoints of each smooth curve should form a 90 degrees angle.

However, there are some important elements that define an ortho-confluent representation. We can consider nodes in an ortho-confluent representation as being parent nodes and son nodes. The parent nodes are traversed by two tracks: a vertical track and a horizontal track that together form a local coordinate system. These tracks separate the drawing surface in four distinct quadrangles. Depending on the positioning of the son node in one of these four quadrates, there are 8 cases in which the son node can be confluently connected with its parent (Fig. 5(a)). In order to represent the tracks we used Bézier curves like in Fig. 5(b) (the control points are chosen at three quarters from the son-track distance).



(a) confluently attaching son to parent

(b) horizontal confluence (left) and vertical confluence (right)

Figure 5: The ortho-confluent system

Having a graph that does not contain large cliques, the algorithm that outputs an ortho-confluent representation has the following steps:

1. Find a parent node. This node is one of the two endpoints of the longest edge in the graph.
2. Apply a BFS on the graph and maintain an order list (the node's order given by the BFS) and a parent list (containing for each node his parent in BFS).
3. Confluently attach each node to his parent, in the order given at step 2.
4. Maintain two lists – processed and confluencePoints – that contain, for each node in the graph, whether it was included in the ortho-confluent representation and its confluence point with the vertical or the horizontal track of the parent.

5 Implementation issues

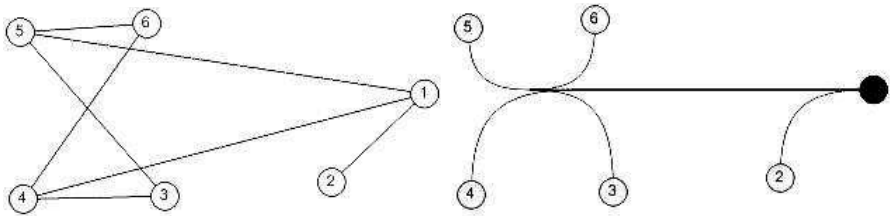
At the moment, there is no commercial graph editor to offer the possibility of representing graphs in a confluent layout. The application we developed in order to illustrate ortho-confluence representation is called ConfluentViz. The main purpose of this tool is to enable users to edit graphs and also to generate aesthetically pleasant representations for different graph classes: trees, graphs that contain circuits, graphs that contain large cliques and bicliques.

The application is developed in C# programming language and uses a free, open source system for designing diagrams and 2D user interface applications – Piccolo.NET. This has monolithic [3] class architecture: it primarily uses compile-time inheritance to extend functionality instead of using run-time composition to extend functionality. It offers the possibility of designing applications that require Zoomable User Interface (ZUI) and animations. It is developed for the .NET framework 2.0 and it is based on the classes and methods collection in GDI+ (Graphics Device Interface) for representing geometric shapes in .NET. We extended this system obtaining a graph editor with confluent layout creation support. The main functionalities of ConfluentViz are:

- graph editing;
- ortho-confluent representation;
- XML storage for graphs;
- obtaining JPEG or PostScript images from the actual representations.

The graph classes that can be represented confluently using ConfluentViz are: trees, forests, graphs containing circuits, graphs containing large cliques or bicliques. Moreover, this application offers the possibility to create a confluent layout automatically, after editing the graphs, or in an assisted manner.

An example of an ortho-confluent representation for an ordinary graph is presented below in Fig. 6. The second graph is a valid confluent representation of the first one, because the connections between nodes (represented by straight lines in the first case and smooth curves in the second case) are preserved. We can see that we can reach nodes 6 and 3 from node 5, by traversing the horizontal track that connects parent node with son nodes. The same is for nodes 4, 6 and 3. Similarly, node 5 cannot be reached from node 4 because the track that connects them is not a smooth one. All the curves are locally monotone, that is they do not have sharp turn backs or crossings.



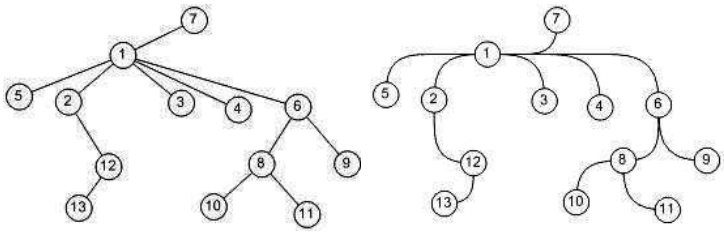
(a) the standard representation (b) the ortho-confluent representation

Figure 6: A graph represented in a standard way and in an ortho-confluent way

Algorithm 1 offers an ortho-confluent representation for trees and forests and it also uses BFS. We can mention that trees are planar graphs, thus they admit a confluent representation. The complexity of the algorithm is $O(n)$, where n is the number of nodes in the graph, and this is because we use BSF and adjacency lists. An example of an ortho-confluent representation for a tree is given in Fig. 7. We obtained good results using this algorithm especially on trees that have large degree nodes: the incident edges are better distinguished when they are represented as Bézier curves and merged together in confluence points.

Algorithm 1 ConfluentComponentTree(g)

```
1.  rootIndex <- MaxDegreeNode(g) //keep the max. degree vertex
2.  bfs <- BreadthFirstSearch(g, rootIndex)
3.  orderBFS<- bfs.order // keep the order of the nodes in BFS
4.  parentsBFS<-bfs.parents // keep the parent of each node in
    the BFS
5.  foreach i=0,VerticesCount
6.      parentIndex <- orderBFS[i]
7.      node <- editor.nodeLayer[orderBFS[i]];
8.      if (parentIndex >= 0) then
9.            parent <- editor.nodeLayer[parentIndex];
10.           Merge(parent, node);
11.     endif
12. end foreach
```



(a) the standard representation (b) the ortho-confluent representation

Figure 7: A tree represented in a standard way and in an ortho-confluent way

Cliques are complete subgraphs of a graph. In order to obtain a confluent representation for cliques we have to:

1. place each vertex of the clique on the support circle, so that we obtain a regular polygon,

2. in the middle of this support circle represent a traffic circle,

3. determine the confluence points coordinates (the intersection of the lines that unite the middle of each polygon edge and the center of the support circle with the traffic circle),

4. draw the tracks (connect each node with the closest confluence point determined at step 3).

In Fig. 8(b) we can see a traffic circle in the middle that replaces a part of the crossing edges. This traffic circle in not an actual node in the graph, it is a structure that has a visual role, facilitating the confluent representation. Bicliques are other structures that can be represented confluently. Similar to clique's case, their usual representation can have many edge crossings that make the relation in the drawing hard to identify. In order to obtain a confluent representation for bicliques we have to follow the next set of steps:

1. identify the 2 partition sets of the biclique subgraph using BFS,

2. determine the largest partition set and the node that has the highest $\mathfrak{y}$ coordinate in this partition,

3. align the centers of the nodes in the largest partition vertically,

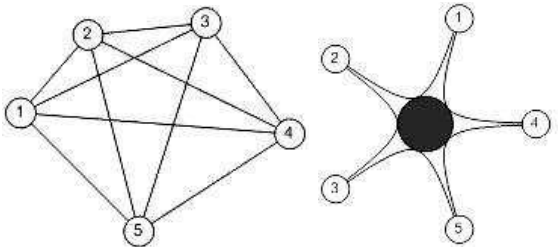
4. determine the middle nodes of the two partitions,

5. align the middle nodes horizontally,

6. align the nodes vertically in the smallest partition,

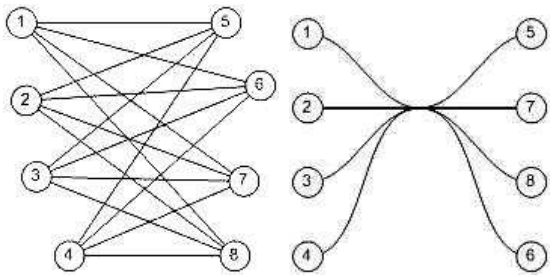
7. connect each vertex in the two partition sets with the middle of the line that unites the nodes determined at step 4.

An example of a biclique that was represented in a confluent manner using the above set of steps is given in Fig. 9.



(a) the standard representation (b) the ortho-confluent representation

Figure 8: K_5 represented in a standard way and in an ortho-confluent way



(a) the standard representation (b) the ortho-confluent representation

Figure 9: $K_{4,4}$ represented in a standard way and in an ortho-confluent way

Having an algorithm that outputs a confluent representation for structures like cliques and bicliques, we can easily obtain a confluent representation for a non-planar graph that contains large cliques or bicliques.

6 Further work

At the moment, ConfluentViz application does not use a planarity test. This would be a nice feature to have if we want to implement a general algorithm that outputs a confluent representation for any non-planar graph, similarly to HeuristicDrawUndirected algorithm presented previously.

Related to ortho-confluence, we saw that this is a particular type of conflu-

ent representation that introduces the notion of local orthogonal coordinate system. It would be interesting to determine which type of confluent representation produces better results for different graph classes. Moreover, having a very large graph, with a very complicated structure, we could use together different confluent techniques to produce a confluent representation.

There are also other types of graphs on which we can successfully apply confluent techniques. For example, directed hypergraphs [8] are a generalization for directed graphs and they can model binary relations among the subsets of a given set. These types of relations are common in different areas in Computer Science such as: data base systems, expert systems, parallel programming, scheduling, routing in dynamic networks, data mining and bioinformatics. The edges of the directed graph are called hyperarcs and they connect distinct subsets of nodes. A solution for visualizing the hypergraphs could be confluent representation. In this case, the arcs that form a hyperarc are merged together in a confluence point (grouping origin and destination of a hyperarc).

7 Conclusion

In this article, we presented a new method of visualizing different graph categories called confluent representation. This can be very useful in software visualization, topology, airline maps and subway maps or in designing site navigation rules. We introduced ortho-confluence and we identified some efficient algorithms that output aesthetic drawings for different classes of graphs. The results obtained with ConfluentViz application satisfy the main aesthetic criteria for graph visualization.

References

[1] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis, *Graph drawing: algorithms for the visualization of graphs*, Prentice Hall, 1998.

[2] B. B. Bederson, *Piccolo.NET: a scalable structured graphics toolkit*, Microsoft Faculty Summit, August 2, 2004.

[3] B. B. Bederson, J. Grosjean, J. Meyer, Toolkit Design for Interactive Structured Graphics, *IEEE Transactions on Software Engineering*, **30**, 8 (2004) 535–546.

[4] M. T. Dickerson, D. Eppstein, M. T. Goodrich, J. Y. Meng, Confluent drawings: visualizing non-planar diagrams in a planar way, *Proc. 11th Int. Symp. Graph Drawing (GD 2003)*, *Lecture Notes in Computer Science*, **2912**, 2003, pp. 1–12.

[5] D. Eppstein, M. T. Goodrich, J. Y. Meng, Confluent layered drawings, *Proc. 12th Int. Symp. Graph Drawing (GD 2004)*, *Lecture Notes in Computer Science* (ed. J. Pach), **3383**, 2004, pp. 184–194.

[6] D. Eppstein, M. T. Goodrich, J. Y. Meng, Delta-confluent drawings, *Proc. 13th Int. Symp. Graph Drawing (GD 2005)*, *Lecture Notes in Computer Science* (eds. P. Healy, N. S. Nikolov), **3843**, 2006, pp. 165–176.

[7] M. R. Garey, D. S. Johnson, Crossing number is *NP*-complete. *SIAM J. Algebraic Discrete Methods*, **4**, 3 (1983) 312–316.

[8] A. L. P. Guedes, L. Markenzon, *Directed Hypergraph Planarity*, Technical Report RT-DINF 003/2001, Departamento de Informtica – UFPR, December 2001.

[9] M. Hirsch, H. Meijer, D. Rappaport, Biclique edge cover graphs and confluent drawings, *Proc. 14th Int. Symp. Graph Drawing (GD 2006)*, *Lecture Notes in Computer Science*, **4372**, 2007, pp. 405–416.

[10] P. Hui, M. Schaefer, D. Štefankovič, Train tracks and confluent drawings, *Proc. 12th Int. Symp. Graph Drawing (GD 2004)*, *Lecture Notes in Computer Science* (ed. J. Pach), **3383**, 2004, pp. 318–328.

[11] M. Jünger, P. Mutzel, *Graph drawing software*, Springer-Verlag, 2003.

[12] M. Kaufmann, D. Wagner, *Drawing graphs – methods and models*, *Lecture Notes in Computer Science Tutorial*, **2025**, Springer-Verlag, 2001.

[13] T. Nishizeki, S. Rahman, *Planar graph drawing*, World Scientific, 2004.

[14] K. Sugiyama, *Graph drawing and applications for software and knowledge engineers*, World Scientific, 2002.

[15] W. T. Tutte, C. St. J. A. Nash-Williams, *Graph theory*, Cambridge University Press, 2001.

Received: April 4, 2009

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 149–164

Automated dynamic programming

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

Ágnes Csiki
INTELES SRL
Str. Gheorghe Pitut nr.1
Oradea, Romania
email: csiki_agi@yahoo.com

Abstract. Since the first book in dynamic programming was published in 1957, this algorithm design strategy has become a current problem solving method in several fields of science. The dynamic programming problem solving process can be divided into two steps. Firstly, we establish the functional equation of the problem, a recursive formula that implements the principle of the optimality (mathematical part). Secondly, a computer program is elaborated that processes the recursive formula in bottom-up way (programming part). In this paper we are going to present a method and a software tool that automates the programming part of the dynamic programming process in case of several problems.

1 Introduction

Dynamic programming as optimizing method was proposed by Richard Bellman. Since the first book [1] in dynamic programming was published in 1957, this algorithm design strategy has become a current problem solving method in several fields of science (Applied mathematics [2], Computer sciences [3], Artificial Intelligence [5], Bioinformatics [4], Macroeconomics [9], etc.). The dynamic programming problem solving process can be divided into two steps. Firstly, we establish the functional equation of the problem, a recursive formula that implements the principle of the optimality. Secondly, a computer

AMS 2000 subject classifications: 68W40, 90C39, 68R10, 05C12
CR Categories and Descriptors: D.1 [PROGRAMMING TECHNIQUES]
Key words and phrases: dynamic programing, graph theory, shortest path algorithms

program is elaborated that processes the recursive formula in bottom-up way. We will refer to these two steps as mathematical and programming parts of the dynamic programming. Numerous researchers in the above mentioned various fields of applications are not experts in programming. In this paper we are going to present a method and a software tool that automates the programming part of the dynamic programming process in case of several problems.

2 The mathematical part

Dynamic programming is often used to solve optimizing problems. The problem usually consists of a target function, which has to be optimized through an optimal sequence of decisions. The dynamic programming is built on the principle of optimality: the optimal solution is built by optimal sub-solutions. This principle is expressed by a recursive formula (functional equation), which describes mathematically the way the more and more complex optimal sub-solutions are built from the simpler ones. Obviously, this is a formula where the way of the optimal (minimum or maximum) decision making has been built in. Once the functional equation is established, the problem can be considered mathematically solved.

We assume that the recursive branches of the functional equation have the following general form: $c(A) = \min / \max\{f_A(c(B_i))|i = 1, 2, ..., n\}$, where c denotes the target function. $c(A)$ represents the optimum value attached to sub-problem A. This optimum directly depends on the optimum value of the one of sub-problems B_i. More exactly, it depends on $c(B_i)$, which optimizes (minimizes or maximizes) function f_A. Function f_A depends on the problem to be solved.

3 The programming part

The programming part of the problem solving process is built on another principle of the dynamic programming: the optimal values of the target function concerning the already solved sub-problems are stored (often in an array that we denote by C). According to the principle of the optimality we are interested only in the optimal solutions of the sub-problems. This technique, often called memoization or result catching, makes it possible to avoid the repeating computation for overlapped sub-problems, which are also characteristic for dynamic programming problems. The core of the computer program that implements the dynamic programming algorithm consists in computing

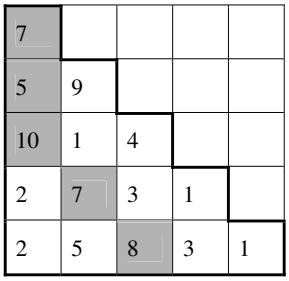


Figure 1: Array A associated to Triangle problem.

the corresponding elements of the array C in bottom-up way according to the strategy given by the recursive formula. An efficient strategy solves each sub-problem before its optimum value is needed by any other sub-problem. The complexity of this programming task varies from problem to problem. It is often nontrivial to write a code that evaluates the sub-problems in the most efficient order.

In [7] we presented three examples; since the computer program works on the elements of the array C, the recursive formula is generally drafted for these elements:

1. Triangle (International Olympiad in Informatics, Sweden, 1994): On and under the main diagonal of a square matrix with n rows there are natural numbers. We assume that the matrix is stored in the bi-dimensional array A. Determine the longest path from peak (element a_{11}) to the base (n-th row), considering the following:

- On a certain path element a_{ij} can be followed either by element $a_{i+1,j}$ (down), or by element $a_{i+1,j+1}$ (diagonally to the right), where $1 \leq i < n$ and $1 \leq j \leq i$.
- By the length of a path we mean the sum of the elements to be found along the path.

For example, should for $n = 5$ the matrix be the following (see Fig. 1.), then the longest path from the peak to the base is the shaded one and its length is 37.

2. Office-building_1: Let A be a matrix whose elements a_{ij} ($i = 1, \ldots, n, j = 1, \ldots, m$) represent an one-storied rectangular office building. The elements of

the matrix represent the offices and they store the taxes to be paid by anyone who enters the respective room. There is a door between any two neighbouring elements. You can enter the building only at office with position $(1, 1)$ and leave it only at position (n, m). Which is the minimal loss of money you can get through the building with?

For example, see Fig. 2 ($n = 5$, $m = 4$). The minimal loss of money is 14, which we got by following the shaded path.

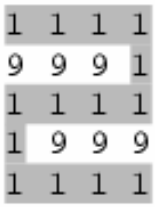


Figure 2: Array A associated to Office-building_1.

3. Office-building_2: The same problem with the following differences:

- There are offices where they do not take money, but they give a certain amount of money ("negative tax").
- There are one-way doors (with one-side door-handles). Array B, whose elements b_{ij} ($i = 1, \ldots, n, j = 1, \ldots, m$) are binary strings with 4 characters ('0' or '1'), stores the door-codes of the offices. The first binary character of the code represents the up-door, the second the right-door, the third the down-door and the fourth the left-door. For example, code "0101" means that we can leave the office only to right and left directions.
- We assume that there is no such office-tour of the building, going along which we could increase our amount of money.

Determine the most favorable way of getting through the building. For example, see Fig. 3 ($n = 5$, $m = 4$) and Fig. 4. The most favorable path goes through the same offices this time too, and means a loss of money of 7.

In the case of all the above-presented problems the array C is bi-dimensional, and the recursive formulas that implement the principle of optimality have the following forms:

1	1	1	1		0111	0111	0111	0011
19	19	19	1		1110	1111	1111	1011
3	1	3	1		1110	1111	1111	1011
-2	19	19	1		0110	1111	1111	1011
-6	-2	3	1		0100	1100	1101	1101

Figure 3: Array A and B associated to Office-building_2 problem.

0 1 **1** 1 1	0 1 **1** 1 1	0 1 **1** 1 1	0 1 **1** 0 1
1 0 **19** 1 1	1 1 **19** 1 1	1 1 **19** 1 1	1 1 **1** 0 1
1 0 **3** 1 1	1 1 **1** 1 1	1 1 **3** 1 1	1 1 **1** 0 1
0 0 **-2** 1 1	1 1 **19** 1 1	1 1 **19** 1 1	1 1 **1** 0 1
0 0 **-6** 1 0	1 0 **-2** 1 0	1 1 **3** 1 0	1 1 **1** 1 0

Figure 4: The Office-building.

Problem 1 (Triangle): (Element c_{ij} stores the length of the longest path from the position (i, j) to the nth row; The trivial sub-problems are represented by the cells from the n-th row, and the optimal value of the original problem is going to be stored in cell c_{11}.)

$c_{nj} = a_{nj}$, $1 \leq j \leq n$

$c_{ij} = a_{ij} + \max(c_{i+1,j}, c_{i+1,j+1})$, $1 \leq i < n, 1 \leq j \leq i$

Problems 2 and 3 (Office-buildings): (Element c_{ij} stores the length of the optimal path between the offices from the positions $(1, 1)$ and (i, j); The trivial sub-problem is represented by cell c_{11}, and the optimal value of the original problem is going to be stored in cell c_{nm}).)

$c_{11} = a_{11}$,
otherwise
$c_{ij} = a_{ij} + \min(c_{i-1,j}, c_{i,j+1}, c_{i+1,j}, c_{i,j-1})$
(assuming that the rooms with the respective positions exist,
and they have "proper doors")

In the first example (the triangle problem) the chain of the recursive calls is cycle-free. In such a situation, there is an elegant technique that does not require the programmer to establish the evaluation order of the sub-problems: recursion with result catching [11]. By catching the results of all recursive calls, the second and subsequent evaluations of any sub-problem become constant-time operations, reducing the overall running time considerably. Recursion with result catching is very easy to implement in softwares like Maple, Matlab, Mathematica, etc. (In Maple we use the `option remember` instruction.) These softwares and the recursion with result catching technique are within the programming reach of most of the researchers, even if they are not experts in programming.

For instance: On the one hand, procedure `triangle_A` is an immediate transcription of the recursive formula, but, unfortunately, this algorithm has exponential time complexity (inefficient divide and conquer strategy). On the other hand procedure `triangle_B` differs from `triangle_A` only in one line (`option remember;`) and it has polynomial complexity (dynamic programming technique).

```
triangle_A := proc(n, a, i, j)
  if i<n then
    return a[i,j]+max(triangle_A(n,a,i+1,j),
                      triangle_A(n,a,i+1,j+1));
  else return a[i,j];
  end if;
end proc:


triangle_B := proc(n, a, i, j)
  option remember;
  if i<n then
    return a[i,j]+max(triangle_B(n,a,i+1,j),
                      triangle_B(n,a,i+1,j+1));
```

```
   else return a[i,j];
   end if;
end proc:
```

In the case of the second and third sample-problems the chain of the recursive calls is circular. For example, the optimal value of cell c_{23} may depend on the optimal value of cell c_{33}, and, conversely, c_{33} may also depend on c_{23}. In such situation, the recursive approach is excluded (to avoid infinite recursive call). Furthermore, there are no easy dynamic programming solutions for these type of problems (for more details, see [7]). The method and software tool presented in this paper are especially useful in case of such problems.

4 Dynamic programming as optimal path algorithm in weighted digraphs

In the followings we are going to consider the recursive formula as an implicit description of a weighted digraph. By this approach several dynamic programming problems can be interpreted as optimal path problems between two specific vertices of this graph [8].

- The vertices of the graph represent the sub-problems. Thus, we can consider the used elements of array C storing the optimal values of the sub-problems as such ones, which represent the vertices of the graph.
- The arcs of the graph represent possible choices (optimize means here the choice of optimal). The graph has an arc from vertex B to vertex A if, the optimum value of the array-element corresponding to vertex A may directly depend on the optimum value of the array-element corresponding to vertex B, according to the recursive formula. For example, if $c_A = min/max\{f_A(c_{B_i})|i = 1, \ldots, n\}$, then there are arcs from vertices B_i to vertex A.
- The weights of the arcs reflect the weights of choices.
- The optimal sequence of decisions is represented by the optimal path between the vertex representing the trivial sub-problem and the vertex that represents the original problem. If the problem has more than one trivial sub-problem, we introduce a dummy trivial-node, which is connected to all trivial vertices.

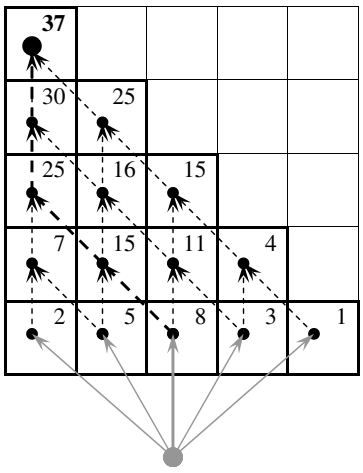


Figure 5: Cycle free digraph attached to the Triangle problem.

Figures 5 and 6 show the graphs behind the sample-problems. The big black node represents the original problem and the gray one the trivial sub-problem. In the case of the triangle problem we introduced a dummy trivial-node. The optimal paths are represented by thick arrows. The cells of array C store the optimum values of the corresponding sub-problems.

We distinguish three cases [8]:

- 1. The attached graph is cycle free. In this case the most efficient optimal path algorithm is based on the topological order of the vertices. The time complexity of this algorithm is $O(N + M)$ (N and M are the numbers of the vertices and arcs, respectively) [3].

- 2. The graph contains cycles, but there are no negative weight arcs. For this case the best choice is Dijkstra's shortest path algorithm. The time complexity of the most efficient implementation of this algorithm is $O(N\log N + M)$ [3].

- 3. The graph has negative arcs, but it has no negative weight cycles. This shortest path problem is solved by the Belmann-Ford algorithm ($O(NM)$) [3].

The optimal (shortest) path problem to be solved is the following. Given a weighted digraph ($G(V, E, w)$, V: set of vertices, E: set of arcs, $w : E \to R$

weight function) with N vertices $1, 2, \ldots, N$ and M arcs, determine the shortest paths from the vertex s (source) to all the other vertices (destinations).

We denote with $c(v)$ (the optimum value attached to vertex v) the weight of the shortest path from the source (s) to vertex v. ($c(s) = 0, c(u) = \infty$ for all vertices u that are not reachable from vertex s)

The above mentioned shortest path algorithms are based on the following lemmas and propositions: (For proofs and further details see [3] and [6])

Lemma 1. Parts of any shortest path are also shortest paths. (Principle of optimality)

Lemma 2. All s-source shortest paths constitute an s-rooted tree called optimal-paths-tree.

Lemma 3. The optimal-paths-tree can be built progressively starting with vertex s. At each step the tree is extended with a new arc that attaches to the tree a new vertex. (Implementation of the principle of optimality)

Lemma 4. If vertex u is the immediate predecessor of vertex v on the optimal path from s to v, then: $c(v) = c(u) + w(u, v)$. (The optimum value of vertex v is based on the optimum value of vertex u, and can be computed on the basis of the weight of arc (u, v))

Corollary 1. The optimum values of all vertices that are reachable from s depend on the optimum value of the one of their in-neighbours.

Lemma 5. The optimum values have to be computed according to a topological order of the vertices with respect to the optimal-paths-tree.

Lemma 6. Assuming $c(s) = 0$, the building process of the optimal-paths-tree consist in applying the formula from Lemma 4 on all arcs of the optimal-paths-tree in their topological order.

Assuming that the optimum values attached to the vertices are going to be generated in array C, we define the following updating operation (operator `update`) on the basis of arc $(u, v) \in E$:

```
update(u,v)
   if c_v > c_u + w(u,v) then c_v = c_u + w(u,v)
   end_if
end_update
```

Corollary 2. If $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying `operator update` on all arcs of the optimal-paths-tree in their topological order results in $c_u = c(u)$ for all $u \in V$.

Lemma 7. For all arcs $(u, v) \in E$ it is true that: $c_v \leq c_u + w(u, v)$.

Lemma 8. If $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying operator `update` on any arc-sequence that includes as sub-sequence the arc-sequence required by Lemma 6 results in $c_u = c(u)$ for all $u \in V$.

All the three shortest path algorithms mentioned above apply the following strategy:

- it generates such an arc-sequence that includes as sub-sequence the arc-sequence required by Lemma 6,
- it applies operator `update` on all arcs of the generated sequence.

Lemma 9. If G is cycle free, then the topological-sequence of all arcs of G includes as sub-sequence the arc-sequence required by Lemma 6.

Proposition 1. If G is cycle free, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then applying operator `update` on all arcs in their topological order results in $c_u = c(u)$ for all $u \in V$.

Lemma 10. If all arcs in G have non negative weights and vertex u is a predecessor of vertex v on the optimal path from s to v, then: $c(u) \leq c(v)$.

Corollary 3. If all arcs in G have non negative weights, then the optimum values of all vertices v that are reachable from s may only depend on in-neighbours that have optimum values less or equal than $c(v)$.

According to Lemma 10 and Corollary 3 Dijkstra's algorithm determines the shortest paths according to the ascending order of their weights.

Proposition 2. (Dijkstra's algorithm) If all arcs in G have non negative weights, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then the algorithm that

- starts with vertex s,
- in each step attaches the arc that links to the tree the vertex that is 'closest' to root s (according to the current values stored in array C) to the growing optimal-paths-tree,
- applies operator `update` on all out-arcs of the currently attached vertex,

results in $c_u = c(u)$ for all $u \in V$.

Proposition 3. (Bellman-Ford algorithm) If G has no negative cycles, $c_s = 0$ and $c_u = \infty$ for all $u \in V \backslash \{s\}$, then the algorithm that

- chooses an arbitrary sequence of all arcs in G,
- applies operator `update` to the chosen sequence, again and again, until no more changes in array C,

results in $c_u = c(u)$ for all $u \in V$.

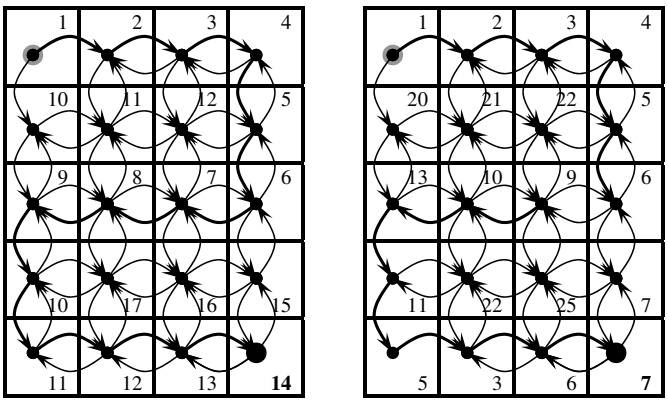


Figure 6: Digraphs (without negative arcs/without negative cycles) attached to the Office-buildings problems.

5 The method and the software tool

The core idea of the algorithm behind the software is that we represent explicitly the graph described implicitly by the recursive formula. Since the dimension of the array C varies from problem to problem we treat it as one-dimensional (row-major-index). The used cells (the vertices) of array C are going to store the optimum values of the corresponding sub-problems and pointers to their out-neighbour cells.

There are two strategies to transpose the functional equation of dynamic programming into an algorithm: the direct method (direct-conversion of the functional equation into an iterative/recursive procedure) and the successive approximation methods (after an initial approximation, the cells that are going to store the optimum values are successively updated –improved– either by the functional equation itself or by an equation related to it) [10].

Another classification of the dynamic programming strategies is based on the way the optimum values of the sub-problems are computed. The so-called pull-approach computes directly (not by an updating process) the optimum value of the current node on the basis of the already computed optimum values of its immediate predecessors. This approach is an immediate application of the functional equation, and can be used only for the acyclic graphs. The recursion with result catching technique applies this approach [10].

The key idea in the case of the push-approach (adaptable to all three cases) is to propagate any improvement that has been made in the current vertex u to its out-neighbors. More exactly, if v is an out-neighbor of the current vertex u, then cell c_v (the cell corresponding to the sub-problem that is represented by the vertex v) is updated on the basis of the arc (u, v). In other words, if value $f_v(c_u)$ 'is better' than the current value of the cell c_v, then c_v is updated with the value $f_v(c_u)$. The algorithm ends when any other improvements cannot be performed [10]. All the three optimum path algorithms we are using in the software apply successive approximation and push-approach.

The topological algorithm traverses the vertices of the graph (starting with the (dummy)trivial-node) according to their topological order, and updates the c_v value of all out-neighbors of the current vertex u on the basis of the arc (u, v). At the moment we have arrived to a vertex, the corresponding element in array C already stores the optimum value. The algorithm only confirms this optimum. The succession the optimum values of the vertices are determined is predestinated by the topological order of these vertices. The topological order of the vertices can be established by a Depth First Search (DFS) procedure. The algorithm attempts to approximate with each arc at most once [6].

If the graph has no negative weight arcs, then it can be observed that the optimum values of the vertices are in ascending order along the shortest paths. Consequently, the Dijkstra algorithm traverses the vertices according to this order, and updates the c_v values of all out-neighbors of the current vertex u on the basis of the arc (u, v). It is evident that in this case the order the optimum values of the vertices are determined is unpredictable. Therefore, Dijkstra's algorithm determines this order on the fly (during the algorithm); if the vertex v is the closest (according to the current values of the array C) out-neighbour of the already confirmed shortest-path-tree, then c_v is confirmed as the optimum value of node v. The algorithm attempts to approximate with each arc at most once [6].

The Bellman–Ford algorithm goes through (in arbitrary order) all the arcs of the graph (and attempts to approximate with them) again and again. It needs at most $(N - 1)$ tours. During a last extra-tour the algorithm realizes that all elements of the array C have reached their optimal values. (There were not any updates) [6].

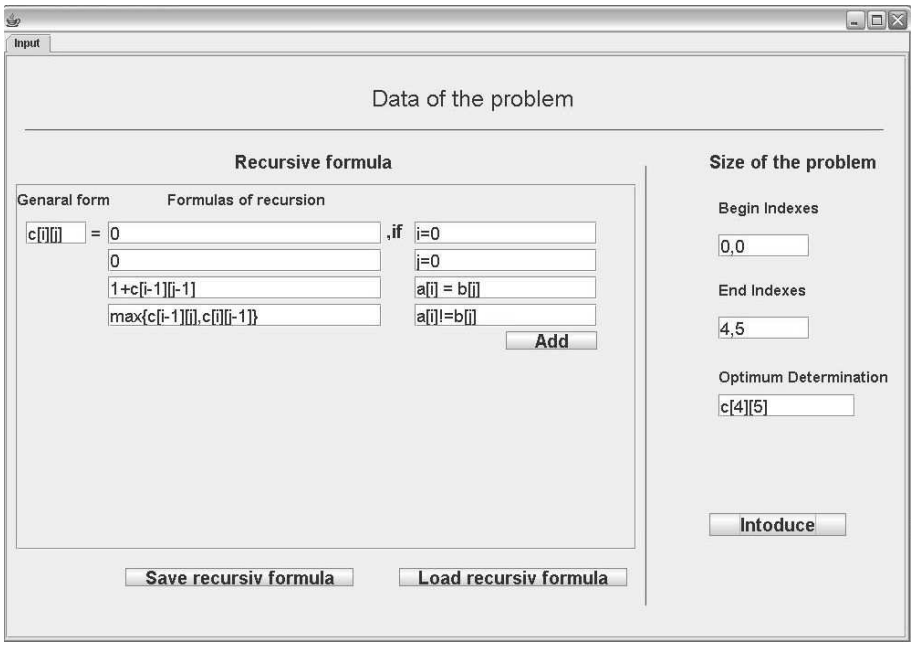


Figure 7: The Input-interface.

The algorithm is:

1. Input:
   (a) The recursive formula is introduced.
   (b) The index-limits (along every dimension) of the array C are introduced.
   (c) The indexes of the cell that represents the original problem are introduced.
2. The recursive formula is analyzed:
   (a) The software asks for the input data.
   (b) The digraph is built.
3. The type of the digraph is determined. (A DFS algorithm tests if the graph is acyclic or not, has negative arcs or not, and whether it contains negative cycles or not.)

4. The proper optimal path algorithm is applied.

5. The solution (the optimum value corresponding to the original problem, and the cell-indexes along the optimal path) is printed.

Fig. 7 shows the input interface that implements steps 1/a, 1/b and 1/c of the algorithm. As a sample problem we have:

Given two sequences in arrays a[1..4] and b[1..5], determine the longest common subsequence.

Fig. 8 shows the output interface that presents a simulation of the dynamic programming solution building process. The optimum value of sub-problem $(3,3)$ is computed on the basis of the optimum values of sub-problems $(2,3)$ and $(3,2)$.

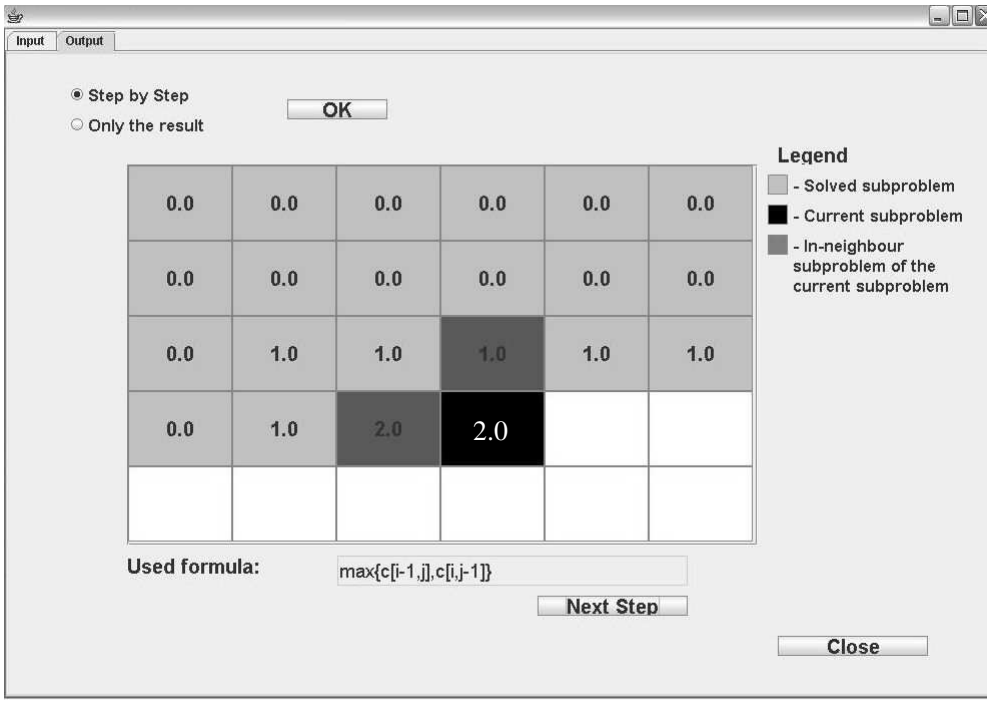


Figure 8: The Output-interface.

6 Conclusions

The above presented method treats different dynamic programming problems uniformly and in such a way that it makes possible the automation of the programming part of the problem solving process. The software is a very useful tool for all researchers who have to deal with dynamic programming problems, especially for those who are not experts in programming.

Acknowledgement

This research was supported by the Research Programs Institute of Sapientia University.

References

[1] R. Bellman, *Dynamic programming*, Princeton University Press, New Jersey, 1957.

[2] R. Bellman, S. Dreyfus, *Applied dynamic programming*, Princeton University Press, New Jersey, 1962.

[3] T. H. Cormen, C. E. Leirserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, The MIT Press, 2003.

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

[5] Z. Feng, R. Dearden, N Meuleau, R. Washington, Dynamic programming for structured continuous Markov decision problems, *Proceedings of the 20th conference on Uncertainty in artificial intelligence, ACM International Conference Proceeding Series,* AUAI Press, **70**, 2004, pp. 154–161.

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

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

[8] Z. Kátai, Dynamic programming as optimal path problem in weighted digraphs, *Acta Mathematica Academiae Paedagogicae Nyíregyháziensis*, **24**, 2 (2008) 201–208.

[9] I. King, *A Simple Introduction to Dynamic Programming in Macroeconomic Models*, 2002.
http://researchspace.auckland.ac.nz/handle/2292/190

[10] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, *Control and cybernetics*, **35**, 3 (2006) 599–620.

[11] D. Vagner, Power programming: Dynamic programming, *The Mathematica Journal*, **5**, 4 (1995) 42–51.

Received: May 5, 2009.

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 165–213

Extended structural recursion and XSLT

Balázs Kósa
Eötvös Loránd University
Faculty of Informatics
Department of Information Systems
email: balhal@inf.elte.hu

András Benczúr
Eötvös Loránd University
Faculty of Informatics
Department of Information Systems
email: abenczur@inf.elte.hu

Attila Kiss
Eötvös Loránd University, Faculty of Informatics
Department of Information Systems
email: kiss@inf.elte.hu

Abstract. In this paper we describe a simulation of a practically important fragment of XPath 1.0 [16] and XSLT 1.0 [17] with extended structural recursions, which in turn immediately offers us a top-down implementation strategy working in time $O(|D|^2|Q|)$. Here, $|D|, |Q|$ respectively denote the size of the data and the query. However, if the size of the variables is restricted with a constant, then the evaluation works in $O(|D||Q|)$ time. Structural recursions are insensitive to the order of the edges (in our XML model instead of nodes, edges represent elements); hence, in this respect, they are of a weaker expressive power than the more usual models of XML query languages [14]. Still, a large fragment of the most frequent scenarios appearing in practice can be captured with them, which underlines their importance.

1 Introduction

Structural recursion is a graph traversing and restructuring operation applied in many fields of computer science including syntax analysis, code generation

AMS 2000 subject classifications: 68U99
CR Categories and Descriptors: H.2.3 [Database Management]: Languages – XPath, XSLT; D.3.3 [Programming Languages]: Language Constructs and Features – Recursion
Key words and phrases: XML, XPath, XSLT, structural recursion

and program transformation. In the context of databases it was already recommended as a query language alternative in the early 90's to be able to overstep the limitations of the relational data model [6]. The rising of semistructured databases and XML [15] put structural recursions again in the limelight. It formed the basis of UnQL [7] and the core of XSLT [17], where in each step the children of the current node are selected to be processed in the next step. In [11] structural recursions were examined in the context of the typechecking problem. However, in all of these works only a simpler version of the operation was considered.

In [2] we offered a new way of defining the semantics using a special kind of intersection similar to its counterpart in automata theory. We also showed how this new approach intertwines with simple "typing systems" of semistructured data, where simulations are used to prescribe the structure of the data. In [3] we introduced not-isempty conditions in *if... then... else...* statements to be able to define different behaviours depending on the underlying subtree of the processed edge. We analyzed the complexity of the satisfiability and containment problem of such structural recursions.

In this paper going further we extend our model with registers, with which the results of different structural functions called on the same data fragment are connected (structural recursions consist of structural functions). To underpin the usefulness of this extension, we simulate a fragment of XPath 1.0 [16] (XPath_0) and XSLT 1.0 (XSLT_0 [5]).

In XPath_0 only the use of location paths with predicates is supported (i.e. there are no arithmetical or string operations). In the predicates the nonemptiness of such paths, the equality of their results with a constant using existential semantics can be checked and the Boolean combinations of such conditions can be taken. The simulation immediately offers us an implementation strategy in worst case working in time $O(|D|^2|Q|)$. However, if the size of variables, i.e., the size of the list of edges which the variable is equal to, is restricted with a constant, then the implementation works in time $O(|D||Q|)$ both for XPath_0 and XSLT_0. This means that our approach has the same efficiency as the method developed by Gottlob et al. in [8]. Here, $|D|, |Q|$ respectively denote the size of the data and query.

In this paper, we only consider axes `child, parent, ancestor`, `descendant`, but it is not difficult to extend the model to be able to handle the rest of the axes. We process XML trees in a top-down manner and we argue that our approach only processes those elements that are inevitably processed by such an evaluation strategy.

Structural recursions are insensitive to the order of the edges (in our XML

model instead of nodes, edges represent elements), hence in this respect they are of a weaker expressive power than the more usual models of XML query languages [14]. Still, a large fragment of the most frequent scenarios appearing in practice can be captured with them, which underlines their importance. For example in [5] three important features of XSLT_0 are highlighted, which are very useful in practical applications. Firstly, one can use variables to "look forward" in the document. Secondly, variables can be passed as parameters between templates. Thirdly, using modes one can process the same data fragment with different templates.

In Section 2 we introduce our data model, XPath_0 and XSLT_0. In the latter two descriptions we heavily rely on the results of [5, 9]. In Section 3 the syntax and semantics of structural recursions are given. The rewriting method of XPath_0 is presented in Section 4, while in Section 5 XSLT_0 is modelled.

2 Preliminaries

Data model

We consider XML documents as rooted, ordered, directed, unranked trees. However, in our setting, since it is more natural to define structural recursions in edge-labelled trees, we assume that instead of nodes, edges with labels represent tags. It is very easy to rewrite a node-labelled tree into an edge-labelled one and vice versa [1]. In the sequel we refer to these trees as *document trees*. In accordance with [15], we also assume that each document tree has a distinguished document edge, the only outgoing edge of the root, with label /. Furthermore, the document edge is followed by the root edge representing the first element of the corresponding document. The document tree of XML document

```
<a><b>xy</b><c>wz</c></a>
```

can be found in Fig. 2(a).

Formally, we introduce three constructors: the empty tree {} consisting of a node only, the singleton set $\{\mathtt{l} : \mathtt{t}\}$, which is a directed $\mathtt{l}$ edge with subtree $\mathtt{t}$ in its end node, and the append operation @. In $\mathtt{t}_1@\mathtt{t}_2$ the roots of $\mathtt{t}_1$ and $\mathtt{t}_2$ are pulled together. It is not difficult to see that by using these constructors every document tree can be built up [7]. For example $\{\mathtt{a} : \{\mathtt{c} : \{\}\}@\{\mathtt{d} : \{\}\}\}@\{\mathtt{b} : \{\mathtt{e} : \{\}\}\}$ stands for the tree of Fig. 2(b). Furthermore, this construction also gives us a notation to represent document trees. These representations are said to be *ssd-expressions* [1] (ssd: semistructured data). An edge $\mathtt{e}_1$ precedes $\mathtt{e}_2$ if

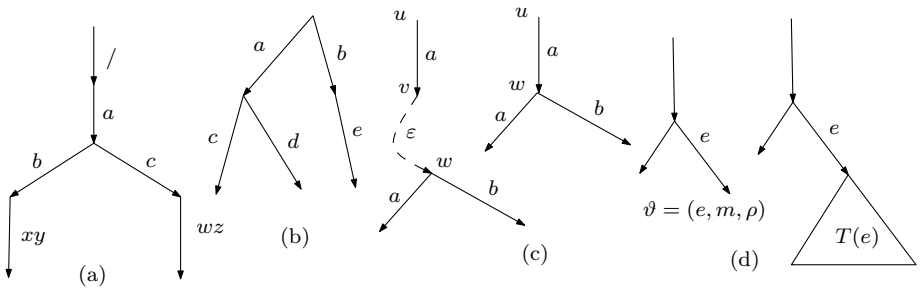


Figure 1: (a) The document tree of the XML document on Page 167. (b) The tree represented by the ssd-expression on Page 167. (c) Elimination of an ε edge. (d) Substitution of a local configuration with the appropriate subtree.

in the corresponding ssd-expression e_1 is written first. Note that this order corresponds to the document order of an XML document [15].

Now, let Σ be a finite alphabet. In what follows, we denote $\mathcal{T}^{\Sigma}$ the set of document trees with edge-labels from Σ. A finite sequence of such trees: $t_1, \ldots, t_m$ constitutes a forest, their set is denoted $\mathcal{F}^{\Sigma}$. With $\mathcal{T}^{\Sigma}(B), \mathcal{F}^{\Sigma}(B)$ we denote that the leaves are labelled with labels from set B. In the following, for graph G we denote with $V.G, E.G$ its node and edge set respectively. For a document tree t $lab(u)$ we will give the label of node u, $lab : E.t \to \Sigma$. On the other hand $T : \Sigma \cup \{*\} \to E$ ($E \subseteq E.t$) is defined s.t. $T(\sigma) = \{e \mid e \in E.t \wedge lab(e) = \sigma\}$ and $T(*) = E.t$.

We are to concentrate mainly on the graph traversing nature of XPath and XSLT, hence, except for the document edge, we suppose that all edges are of the same type. This means that we do not deal with attribute, processing instruction edges, etc. [16]. Our methods can be extended in a straightforward manner to handle edges with types. Furthermore, we assume that every edge has an associated value. Again, we suppose that these values are of the same type, and in contrast to [15] we do not specify how they are obtained. We denote their recursively enumerable set with $\mathcal{D}$. For a given edge in t, function $val : E.t \to \mathcal{D}$ gives its associated value. In accordance with [18], we assume that in a document tree every edge e has a unique identifier, $id(e)$ in notation.

For intermediate results of the constructions we shall consider forests from $\mathcal{F}^{\Delta \cup \varepsilon}$. Here, Δ denotes the set of output symbols. The role of an ε edge will be similar to the role of silent transitions in automata, and they will be eliminated similarly. Formally, for edges $(u, a, v), (v, \varepsilon, w)$ in tree t, an (u, a, w) edge is added to t, and the former two edges are deleted (cf. Fig. 2(c)).

e: Expression, p: Predicate, χ: Axis	
$e ::=$	$\chi :: \tau[p_1] \dots [p_n] \mid e_1/e_2$
$\chi ::=$	child | descendant | parent | ancestor | self
$p ::=$	$e \mid e = d \mid e = \$X \mid (p_1 \wedge p_2) \mid (p_1 \vee p_2) \mid \neg(p) \mid [p]$
$d \in \mathcal{D}, X \in \mathcal{V}, \tau \in \Sigma \cup \{*\}$	

ε: Expression$\rightarrow (\mathsf{E.t} \rightarrow 2^{\mathsf{E.t}})$, ε^p :Predicate$\rightarrow (\mathsf{E.t} \rightarrow \{\mathtt{true}, \mathtt{false}\})$, $\rho : \mathcal{V} \rightarrow 2^{\mathsf{E.t}}$, where $\rho(x)$ is a finite set, $\mathcal{V}$ is the set of variables	
$\varepsilon[\![\chi :: \tau[p_1] \dots [p_n]]\!](x) =$	$\{y \mid x \chi y \wedge y \in \mathsf{T}(\tau) \wedge \varepsilon^p[\![p_1]\!](y) = \mathtt{true} \wedge \dots \wedge \varepsilon^p[\![p_n]\!](y) = \mathtt{true}\}$
$\varepsilon[\![e_1/e_2]\!](x) =$	$\cup_{y \in \varepsilon[\![e_1]\!](x)} \varepsilon[\![e_2]\!](y)$
$\varepsilon^p[\![e]\!](x) =$	true iff $\varepsilon[\![e]\!](x)$ is not empty
$\varepsilon^p[\![e = d]\!](x) =$	true iff $\exists y, y \in \varepsilon[\![e]\!](x) \wedge \mathit{val}(y) = d$
$\varepsilon^p[\![e = X]\!](x) =$	true iff $\exists y, z$ s.t. $y \in \varepsilon[\![e]\!](x) \wedge z \in \rho(X) \wedge \mathit{val}(y) = \mathit{val}(z)$
$\varepsilon^p[\![(p_1 \theta p_2)]\!](x) =$	true iff $\varepsilon^p[\![p_1]\!](x) \theta\, \varepsilon^p[\![p_2]\!](x)$ is true $\theta \in \{\wedge, \vee\}$
$\varepsilon^p[\![\neg(p)]\!](x) =$	true iff $\varepsilon^p[\![p]\!](x)$ is false
$\varepsilon^p[\![[p]]\!](x) =$	true iff $\varepsilon^p[\![p]\!](x)$ is true
$x, y, z \in \mathsf{E.t}, d \in \mathcal{D}, X \in \mathcal{V}$	

Figure 2: The syntax (first table) and semantics (second table) of XPath_0.

XPath_0

XPath has already grown to be a widely known and applied language, thus we restrict ourselves to give only the syntax and semantics rules of the fragment we are going to examine (Fig. 2) with a short explanation. For a more formal and exhaustive presentation, consider [9].

We assume that axis names: `child`, `descendant`, `parent`, `ancestor`, `self` are self-describing. The basic building blocks of XPath expression are location steps: $\chi :: \tau[p_1] \dots [p_n]$. Here, χ is an axis, τ is called *edge test* (node test in [16]), which is from $\Sigma \cup \{*\}$, while p_i-s are predicates that are used to filter the returned set of edges. As an example, consider location step

```
child::a[child::b=5],
```

which returns those `a` children of the actual edge that have a `b` child.

In [9, 16] the semantics are given in terms of *contexts*. A context consists of a *context-edge*, a *context-position* and a *context-size*. However, here, owing to the restricted use of functions, e.g. in XPath_0 we do not use functions

`position()`, `first()` and `last()`, it is enough to consider the context-edge or *actual edge*, as it is frequently called. First, we assume there is a given edge on which the evaluation of the expression starts. This edge is usually determined by the host language. Then each location step selects a set of edges, which in turn serve as starting edges of the next location step. This mechanism is explained formally in Fig. 2 (second table).

Note the use of a variable assignment in the semantics. Usually, this is also given by the host environment. Here, $\mathcal{V}$ denotes the recursively enumerable set of variables. In accordance with the syntax of XPath instead of X we refer to a variable as $X. Finally, note also the use of brackets in predicates. These may be omitted, when the precedence rules among Boolean operators give the same evaluation order.

XSLT$_0$

Again, we do not explain XSLT in detail, but rather give an informal overview and a concise formal definition of the semantics. For a more detailed explanation, consider [5]. There, three important features of XSLT$_0$ are highlighted. Firstly, by means of variables one can "look forward" in the document. Secondly, variables can be passed as parameters between templates. Thirdly, using nodes one can process the same data fragment with different templates. All of these properties appear in the XSLT program of Fig. 3, which is called on XML documents of Fig. 4. The program firstly selects the `id`-s of those groups whose top manager is John. Then it selects the `id`-s of the "subgroups" of the former groups, in which an employee with name `Ann` works.

In T_1 we store in variable `X` the `id`-s of those groups that have an employee with name `Ann`. In T_2 groups with John as top manager are selected. Finally, in T_3 using variable `X` subgroups having an employee named `Ann` are chosen. Note that in the XPath expressions we have used the abbreviated syntax [16]. Rewriting the XPath$_0$ expressions with the syntax of Fig. 2 is straightforward.

An XSLT program consists of templates, i.e., `xsl:template` elements. At a given step, we assume that there is a list of edges, `E`, which has been chosen at the former step. At the beginning of the evaluation this set consists of the document edge of the document. We process the edges of `E` in document order. Suppose that the actual edge is e. First, we select that template, which fits e, i.e., e satisfies the condition given by the XPath expression of the `match` attribute of the `xsl:template` element. According to the specification of [17], this template should be unambiguous. Then e is processed by this template (it serves as the actual edge for the XPath$_0$ expressions), and another list of

```
<xsl:template match="/">                                    (T1)
  <xsl:variable name="X" select="//group[emp/name='Ann']/id"/>
  <result>
    <xsl:apply-templates select="//group" mode="top">
      <xsl:with-param name="X" select="$X"/>
    </xsl:apply-templates>
  </result>
</xsl:template>

<xsl:template match="group" mode="top">                     (T2)
  <xsl:param name="X"/>
  <xsl:if test="topMgr/name='John'">
    <topGroup>
      <id>
        <xsl:value-of select="id"/>
      </id>
    </topGroup>
    <xsl:apply-templates select="//group" mode="Ann">
      <xsl:with-param name="X" select="$X"/>
    </xsl:apply-templates>
  </xsl:if>
</xsl:template>

<xsl:template match="group" mode="Ann">                     (T3)
  <xsl:param name="X"/>
  <xsl:if test="id=$X">
    <id>
      <xsl:value-of select="id"/>
    </id>
  </xsl:if>
</xsl:template>
```

Figure 3: An XSLT program.

edges is selected by the `xsl:apply-template` element for further processing. When these edges are all processed, then the edge after e in E is considered. Note that in our example T_2 and T_3 may be called on the same elements.

Syntax. For the formal definition of semantics we use a more abstract representation of an XSLT$_0$ program [5]. A template is formalized as an (m, σ)-*rule* as follows:

```
<groups>
  <group>
    <id>G01</id>
    <name>sales</name>
    <topMgr>
      <id>03</id>
      <name>John</name>
    </topMgr>
    <emp>
      <id>04</id>
      <name>Tom</name>
    </emp>...
    <emp>...</emp>
    <group>
      <id>G02</id>
      <name>PR</name>
      <emp>
        <id>05</id>
        <name>Steven</name>
      </emp>
    </group>...
    <group>...</group>
  </group>
</groups>
```

Figure 4: A fragment of an XML document.

$$
\begin{aligned}
&\mathit{template}\ m(\sigma, x_1, \ldots, x_n)\\
&\quad \mathit{vardef}\\
&\qquad y_1 := r_1; \ldots ; y_s := r_s;\\
&\quad \mathit{return}\\
&\qquad \text{if } c_1 \text{ then } z_1; \ldots \text{ if } c_k \text{ then } z_k; \text{ else } z_{k+1};\\
&\mathit{end}.
\end{aligned}
$$

Here, m is a mode, $\sigma \in \Sigma$, the latter element shows that the template is called on σ elements (in our data model on σ edges). As it has been shown in [5] we may suppose that only XPath expressions comprising a single element name appear as values of `match` attributes of `xsl:template` elements. σ gives

```
<xs:schema>
  <xs:element name="groups" type="groupsType"/>
  <xs:element name="group" type="groupType"/>
  <xs:element name="topMgr" type="empType"/>
  <xs:element name="emp" type="empType"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="id" type="xs:string"/>
  <xs:complexType name="groupsType">
    <xs:sequence>
      <xs:element ref="group" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="groupType">
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" ref="id"/>
      <xs:element maxOccurs="1" minOccurs="1" ref="name"/>
      <xs:element maxOccurs="1" minOccurs="0" ref="topMgr"/>
      <xs:element maxOccurs="unbounded" minOccurs="1" ref="emp"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="group"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="empType">
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" ref="id"/>
      <xs:element maxOccurs="1" minOccurs="1" ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 5: The XML Schema which the XML document fragment of Fig. 4 conforms to.

this element name. $x_1, \ldots, x_n$ are the parameters, while $y_1, \ldots, y_s$ are local variables. r_i is either a constant in $\mathcal{D}$, or it is an XPath_0 expression ($1 \leq i \leq s$). Here, y_j may appear in r_i, if $j < i$. c_j-s are conditions that are either constants ($\mathtt{true}, \mathtt{false}$), or Boolean combinations of atomic conditions: $x = d$, e, $e = d$, $e = x$ ($1 \leq j \leq k, x \in \mathcal{V}, d \in \mathcal{D}, e$ is an XPath_0 expression). Note that all of these tests may appear in XPath_0 expressions (Fig. 2), and their meaning is the same as there. For instance the single condition e becomes true iff e

results a non-empty edge set. Finally, z_s-s are all forests in $\mathcal{F}^{\Delta}(\mathcal{AT})$, i.e., their edge-labels are from Δ (the alphabet of output symbols), and the leaves may be labelled with elements of $\mathcal{AT}$, where $\mathcal{AT}$ denotes the set of *apply-template*-, or shortly *at-expressions*.

An at-expression is of the form $m(p, \bar{x}, \rho)$, where m is mode, p is an XPath_0 expression, with which the next list of edges is selected for processing; $\bar{x}$ is the set of variables that are passed as parameters to the next instantiated template, and $\rho : \mathcal{V} \to 2^{E.t}$ is a variable assignment.

Example 1 *Consider the rewriting of the XSLT$_0$ program of Fig. 3.*

```
template st(/, ϵ)                                   (T1)
    vardef
        x :=//group[emp/name='Ann']/id;
    return
        if true then {result : {}}; at-expr: top(//group, x, ρ)
end.

template top(group, x)                              (T2)
    return
        if topMgr/name='John' then {topGroup : {id : {}}@{}};
            at-expr1: val(id, ϵ, ϵ), at-expr2: Ann(//group, x, ρ)
end.

template Ann(group, x)                              (T3)
    return
        if id= x then {id : {}}; at-expr1: val(id, ϵ, ϵ)
end.

template val(id, ϵ)                                 (T4)
    return
        if true then {val(id) : {}};
end.
```

Here, T_1 is called on the document edge /, in mode st with no parameters. z_1 consists of a single node, whose at-expression label is top(//group, x, ρ), where ρ assigns to x the result of the XPath expression of the vardef part. We return to the special role of the (st, /)-rule soon. In T_2 the at-expressions after at-expr1, at-expr2 respectively belong to the leaf of the id edge and the

leaf appended to this edge. Note the superficial nature of this construction. Furthermore, the `xsl:value-of` element returning the value of `id` children is represented here as a special template T_4 also returning the value of `id` elements. The at-expression of T_3 invokes T_4 on the `id` children of the actual edge. It is easy to verify that with this rewriting we simulate correctly the functioning of the former `xsl:value-of` element.

As we have already mentioned, according to [17], the conflict of templates should be avoided, hence only one (m, σ)-rule is allowed to give for each (m, σ)-pair. Furthermore, we assume that there is a distinguished *start mode*, $\mathtt{st}$, which is only used with the document edge. For sake of simplicity, we also assume that $(\mathtt{st}, /)$ is the only rule, where variable definitions are allowed to give. It has only one condition, where c_1 is $\mathtt{true}$, and z_1 constructs a $\mathtt{result}$ edge as the root edge of the output. (With this, we guarantee that the output is always a tree.) In the XSLT program of Fig. 3 (T_1) stands for this rule. Note that according to [17] it is not allowed to be given a mode to a template instantiated on the document edge. Nevertheless, this slight deviation carries no importance.

Semantics. A run of a program on t is given in terms of trees in $\mathcal{T}^{\Delta}(\mathsf{LC}(\mathsf{t})^*)$. Here, $\mathsf{LC}(\mathsf{t}) = \mathsf{E.t} \times \mathsf{M} \times \Psi$ denotes the set of *local configurations*, where M is the set of modes and Ψ is the set of partial variable assignments from $\mathcal{V}$ to $2^{\mathsf{E.t}}$ with a finite domain. $(e, \mathsf{m}, \rho) = \vartheta \in \mathsf{LC}(\mathsf{t})$ represents that case, when (m, σ)-rule T is to be applied on e, where $\mathtt{lab}(e) = \sigma$, and the parameters of T are in the domain of ρ. Here, e is the edge whose endnode is labelled with ϑ. For a tree in $\mathcal{T}^{\Delta}(\mathsf{LC}(\mathsf{t})^*)$, a leaf may be labelled with a sequence of local configurations.

Informally, suppose that a subtree of the result $\hat{\mathsf{t}} \in \mathcal{T}^{\Delta}(\mathsf{LC}(\mathsf{t})^*)$ has been already constructed. For sake of transparency we consider a leaf labelled with a single local configuration $\vartheta = (e, \mathsf{m}, \rho)$. This defines an (m, σ)-rule T to be applied. Here, e will be used as the actual edge of the XPath_0 expressions of T. When $\mathsf{T}(e)$ is evaluated, the result (in $\mathcal{F}^{\Delta}(\mathsf{LC}(\mathsf{t})^*)$) should be added to the leaf and ϑ should be deleted. If the leaf is labelled with a sequence containing more local configurations, the former method should be applied to each of them consecutively (cf. Fig. 2(d)).

The process starts with t_0, where t_0 consists of an edge with label /, and with leaf label $(/, \mathtt{st}, \epsilon)$ (this guarantees that the $(\mathtt{st}, /)$-rule is called first), and it stops, when there is no $\mathsf{LC}(\mathsf{t})^*$ label in the constructed output to process further. We shall assume that, if (m, σ)-rule T has been already applied to an edge e, then T is not allowed to be applied on e again. Thus, we avoid infinite loops. This feature is guaranteed when XSLT_0 programs are simulated with

structural recursions.

Formally, an XSLT$_0$ program is a tuple $P = (\Sigma, \Delta, M, st, R)$, where Σ and Δ are the labeling alphabets of the input and output respectively. M is the set of modes disjunct from Σ and Δ. st is the start mode and R is a finite set of (m, σ)-rules.

We define a *rewrite relation induced by* P *on* t, $\rightarrow_{P,t}$. Here

$$\rightarrow_{P,t}: \mathcal{T}^{\Delta}(LC(t)^*) \rightarrow \mathcal{T}^{\Delta}(LC(t)^*).$$

For $\xi, \zeta \in \mathcal{T}^{\Delta}(LC(t)^*)$, first we explain the simpler case, when a leaf labelled with a single local configuration is considered. Then $\xi \rightarrow_{P,t} \zeta$, if ξ has a leaf edge e with leaf label (e, m, ρ), where $lab(e) = \sigma$, and the parameters of (m, σ)-rule T are all included in the domain of ρ. ζ is constructed by substituting e with f_o, the result of T instantiated on e with variable assignment ρ.

To understand the construction of f_o, remember the syntax of T. First we evaluate r_i-s taking e as the actual edge to get the possible values of y_i-s ($1 \leq i \leq s$). Denote E_i the result of r_i. Then, assuming that c_j is the first condition becoming true, $z_j \in F^{\Delta}(\mathcal{AT})$ is transformed into f_o, where we substitute each at-expression leaf label with a sequence of local configurations. Namely, a leaf label $m'(p, \bar{z}, \rho')$ is substituted with sequence $(e_1, m', \rho'), \ldots, (e_l, m', \rho')$. Here $\bar{z} \subseteq \{x_1, \ldots x_n, y_1, \ldots y_m\}$, $\rho'(x_i) = \rho(x_i)$ and $\rho'(y_j) = E_j$, $p(e) = \{e_1, \ldots, e_l\}$ (p is the XPath$_0$ expression of the at-expression in question, while e is the actual edge on which T is instantiated). Furthermore, e_r precedes e_o in document order, if $r < o$ ($1 \leq i \leq n, 1 \leq j \leq s, 1 \leq r, o \leq l$).

If e is labelled with a sequence local configurations, then ζ should be obtained by applying the former method to each local configuration one after the other.

The initial local configuration is defined to be t_0. With this the transformation realized by P, is the (partial) function $\tau_P : \mathcal{T}^{\Sigma} \rightarrow \mathcal{T}^{\Delta}$, with $\tau_P(t) = s$, if $t \in T^{\Sigma}$, $s \in T^{\Delta}$, and $t_0 \rightarrow^*_{P,t} s$.

3 Structural recursions

Syntax

A structural recursion f is constituted by *structural functions*, in notation $f = (f_1, \ldots, f_n)$, which can call each other. In the definition of a structural function we consider inputs given with ssd-expressions and specify what should happen for the different constructors. For the syntax of a row of a structural

SR: structural recursion, Reg: registers, $r(f_i)$: a row of f_i, C: condition, C^d: condition for the default, $Sf = \{f_1(t), \ldots, f_n(t)\}$, $a \in \Sigma, \alpha \in \mathcal{D}, y \in \mathcal{V}, t \in \mathcal{T}^\Sigma$, n.i. stands for *not isempty*	
$SR ::=$	S, Reg
$S ::=$	$f_i : r(f_i)$, $(1 \leq i \leq n)$
$r(f_i) ::=$	$(t_1@t_2) = f_i(t_1)@f_i(t_2) \mid (\{\}) = \{\} \mid (\{a : t\}) = R \mid$ $(\{* : t\}) = R^d \mid$ $(\{a : t\}) =$ if C_1 then R_1 else $R_2 \mid$ $(\{* : t\}) =$ if C_1^d then R_1^d; else R_2^d
$R ::=$	fo, where $fo \in \mathcal{F}^{\Delta}(Sf)$
$R^d ::=$	fo, where $fo \in \mathcal{F}^{\Delta \cup \{*\}}(Sf)$
$C ::=$	$\text{n.i.}(f_j(t)) \mid val(a) = \alpha \mid val(a) = y \mid (C_1 \wedge C_2) \mid$ $(C_1 \vee C_2) \mid \neg(C)$
$C^d ::=$	$\text{n.i.}(f_j(t)) \mid val(*) = \alpha \mid val(*) = y \mid (C_1 \wedge C_2) \mid$ $(C_1 \vee C_2) \mid \neg(C)$
$\mathsf{Reg} ::=$	$X^b_{f_i} = X^b_{f_j} \mid (\mathsf{Reg}_1 \wedge \mathsf{Reg}_2) \mid (\mathsf{Reg}_1 \vee \mathsf{Reg}_2) \mid \neg(\mathsf{Reg})$

$\varepsilon^C : \mathsf{condition} \to (E.t \to \{\mathsf{true}, \mathsf{false}\})$, $\theta \in \{a, *\}$, $\rho : \mathcal{V} \to 2^{\mathcal{D}}$	
$\varepsilon^p[\![val(\theta) = \alpha]\!](e) =$	true iff $lab(e) = \theta, val(e) = \alpha$
$\varepsilon^p[\![val(\theta) = y]\!](e) =$	true iff $lab(e) = \theta, \exists e' \in \rho(y)$ s.t. $val(e) = val(e')$

Figure 6: The syntax rules of structural recursion $f = (f_1, \ldots, f_n)$ (first table). The semantics of a subset of conditions (second table).

function, consider the $r(f_i)$ row of the first table of Fig. 6. Here, it turns out that for constructors $t_1@t_2, \{\}$ structural functions always work in the same manner, hence they are not given in the definition. For $t_1@t_2$ they call themselves both on t_1 and t_2, and at the end append the results. For $\{\}$, they construct a single node.

Example 2 *As an example, we give structural recursion* $f = (f_1, f_2, f_3)$*, which copies a subtree* $\{a : t\}$ *if the* a *edge has an* Ann *child.*

f_1: $(\{a : t\}) =$ *if n.i.*$(f_2(t))$ *then* $\{a : f_3(t)\}$
$(\{l : t\}) = f_1(t)$

f_2: $(\{Ann : t\}) = \{\psi : \{\}\}$
$(\{l : t\}) = \{\}$

f_3: $(\{* : t\}) = \{* : f_3(t)\}$

Example 3 *As another example we give the rewriting of XPath$_0$ expression:*

$$q_1 = \texttt{self::a[par::c=5]/child::b/desc::d/par::a},$$

where `desc, par` *are abbreviations of axes* `descendant, parent` *respectively.*

$$\begin{array}{lll}
f_{a_1}: & (\{a:t\}) = f_b(t) & f_b: \quad (\{b:t\}) = f_d^1(t) \\
& (\{*:t\}) = f_{a_1}(t) & \qquad (\{*:t\}) = f_{a_1}(t)
\end{array}$$

$$\begin{array}{ll}
f_d^1: & (\{d:t\}) = \{\psi:\{\}\}@\, f_d^1(t) \\
& (\{*:t\}) = f_d^1(t)
\end{array}$$

$$\begin{array}{lll}
f_{a_2}: & (\{a:t\}) = \textit{if } n.i.(f_d^2(t)) \textit{ then } \{\psi:\{\}\}@\, f_{a_2}(t) & f_d^2: (\{d:t\}) = \{\psi:\{\}\} \\
& \qquad\qquad\qquad\quad \textit{else } f_{a_2}(t) & \qquad\quad (\{*:t\}) = \{\} \\
& (\{*:t\}) = f_{a_2}(t) &
\end{array}$$

$$\begin{array}{lll}
f_c^{pr_1}: & (\{c:t\}) = \textit{if } val(c) = 5 \textit{ then } f_{a_1}^{pr_1}(t) & f_{a_1}^{pr_1}: (\{a:t\}) = \{\psi:\{\}\}@\, f_c^{pr_1}(t) \\
& \qquad\qquad\qquad\quad \textit{else } f_c^{pr_1}(t) & \qquad\quad (\{*:t\}) = f_c^{pr_1}(t) \\
& (\{*:t\}) = f_c^{pr_1}(t) &
\end{array}$$

$$X_{fd^1}^d = X_{fd^2}^d,\ X_{f_{a_1}^{pr_1}}^a = X_{f_{a_1}}^a$$

Here, $pr_1 =$ `self::a/par::c=5`. $f_c^{pr_1}$ checks whether the value of the actual c edge is equal to 5, then calls $f_{a_1}^{pr_1}$, which constructs a ψ edge as a result of an a edge, if it immediately follows the aforementioned c edge.

`self::a/child::b/desc::d`, `desc::d/par::a`

are simulated with f_{a_1}, f_b, f_d^1, f_{a_1}, f_d^2, respectively. The connection is given with the register restriction $X_{f_d^1}^d = X_{f_d^2}^d$, whose intended meaning is that the d edges processed by f_d^1 should also be processed by f_d^2 and vice versa. Thus these d edges both have a b ancestor with an a parent, and an a parent. One may consider $X_{f_d^i}^d$-s as *registers* containing the id-s of the d edges processed by f_d^i $(i = 1, 2)$. Note the similar role of $X_{f_{a_1}^{pr_1}}^a = X_{f_{a_1}}^a$. In f_{a_2} n.i. stands for the *not-isempty* condition. The $\{*:t\}$ rows represent the default cases.

Note that only ψ $(\psi \in \Delta)$ edges are constructed here. This is understandable, since we simulate an XPath$_0$ expression. In the output, only ψ edges constructed by f_{a_2} representing the last location step `parent::a` will be considered. Thus we get a star as a result, i.e., a single node with outgoing ψ

edges. If for document tree t and e_0 as context edge q_1 selects $e_1, \ldots, e_s$, then the star consists of s ψ edges, the first constructed as a result of processing e_1 according to the a row of f_{a_2}, the second as processing e_2, etc.

Now, consider the syntax rules in the first table of Fig. 6. Note that one may construct forests in $F^{\Delta}(Sf)$, i.e., the leaves of these forest may be labelled with $f_i(t)$-s. The intended meaning of such a label is that the result of f_i called on t is to be connected to the leaf in question. In the previous example with $f_b(t)$ on the right hand side of a row we denoted a node labelled with $f_b(t)$.

Semantics

Operational graphs. As in [2], in order to define the semantics, we introduce *operational graphs*, which will represent the "relationships" among structural functions. For structural recursion $f = (f_1, \ldots, f_n)$, we denote its operational graph with U_f.

In the construction, for each f_i we assign a node with name f_i ($1 \leq i \leq n$). The edges of U_f are given with respect to the rows of f_i-s. As a warm-up we consider a simple row:

$$(\{a : t\}) = \{b : f_j(t)\}.$$

Here, we add an $a(x)$ edge from f_i to f_j. The intended meaning is obvious, we represent that as a result of singleton $\{a : t\}$ f_i calls f_j. For

$$(\{a : t\}) = \text{ if n.i.}(f_j(t)) \text{ then } f_k(t) \text{ else } f_l(t),$$

we add an $a(x)$ edge with an additional pr (predicate) label to f_j and two other $a(x)$ edges with labels th, el (then, else) to f_k and f_l respectively. If no structural function is called, we use an additional node w. Formally, for row:

- $(\{\theta : t\}) = fo$ ($fo \in \mathcal{F}^{\Delta \cup \{*\}}(Sf), \theta \in \{a, *\}$):

  $(f_i, p, f_j) \in E.U_f$, if $f_j(t)$ is among the labels of leaves of fo, i.e., f_j is called by f_i as a result of a θ edge. If $f_j(t)$ appears more than once among the labels, still only one (f_i, p, f_j) edge is added. Here, if $\theta = a$, then $p = a(x)$. Otherwise, if $\theta = *$, $p = \neg a_1(x) \wedge \ldots \wedge \neg a_l(x)$, where $a_1, \ldots, a_l$ are the symbols appearing in the singleton sets on the left side before the default case in the definition of f_i. If there are no such rows in the definition, then $p = \top(x)$. $a(x)$ is a predicate symbol, which we always interpret over Σ s.t. $a(x)$ becomes true iff $x = a$, while $\top(x)$ is satisfied by all constants of Σ. As an example, consider the operational graph of (f_{a_1}, f_b, f_d^1) of Example 3 in Fig. 7(b) (the leftmost graph). Here and in the rest of the examples we abbreviate $a(x)$ with a.

If f_o has no leaf labels, i.e., no structural function is called, then an (f_i, p, w) edge is added instead of an (f_i, p, f_j) edge.

- $(\{\theta : t\}) =$ if C then fo_1 else fo_2:

  (f_i, p, f_j) is in $E.U_f$ with an additional label pr, if n.i.$(f_j(t))$ appears in C. Such an edge will be called *premise*. The meaning of θ and p is the same as in the previous case.

  Furthermore, (f_i, p, f_k) is in $E.U_f$ with label th if $f_k(t)$ appears among the labels of leaves of fo_1. If $f_k(t)$ appears as a leaf label in fo_2, an el label is added. These edges are called *then-, else-edges* respectively. The premise, then- and else-edges together will be called *conditional edges*. If $f_k(t)$ appears more than once in fo_s, still only one (f_i, p, f_k) edge is added $(s = 1, 2)$. However, if it appears in fo_1 and also in fo_2, then this edge is labelled with both th and el. Again, an (f_i, p, w) edge is added with the appropriate th, el label, if the leaves of fo_s are not labelled. As an example, consider the outgoing a edges of f_{a_2} and the outgoing c edges of $f_c^{pr_1}$ of Example 3 in Fig. 7(b).

An edge of U_f is called *constructor edge*, if it results a construction, i.e., the forest on the right side of the corresponding row is not a single node.

The corresponding operational graph of f_{q_1} in Example 3 can be found in Fig. 7(b). Note that operational graphs are not necessarily connected.

Process of an input. Next, we show how a document tree is processed by an operational graph. For this we introduce a new operation, which is very similar to the intersection operation in automata theory.

Definition 1 *Let* U_f, t *be an operational graph and a document tree* $(f = f_1, \ldots, f_n)$. *Then the* intersection *of* U_f *and* t, *in notation* $U_f \sqcap t$, *is defined as follows:* $V.U_f \sqcap t := \{(\varphi, u) \mid \varphi \in V.U_f, u \in V.t\}$, $(\varphi \in \{f_1, \ldots, f_n, w\})$. $E.U_f \sqcap t := \{((f_i, u), p(x) \wedge a(x), (\varphi, v)) \mid (f_i, p(x), \varphi) \in E.U_f, (u, a, v) \in E.t, \Sigma \models p(a)\}$.

The intuition behind this definition is clear. For instance an $((f_i, u), p \wedge a, (f_j, v))$ edge means that (u, a, v) is processed by f_i, and then f_j is called. Note that we have slightly blurred the distinction between predicates and constants. An edge-label a $(a \in \Sigma)$ of a document tree is considered as predicate $a(x)$. Also note that if $p(x) \wedge a(x)$ is satisfiable, then only $a \in \Sigma$ satisfies it, hence sometimes we shall write a instead of $p(x) \wedge a(x)$ as edge label.

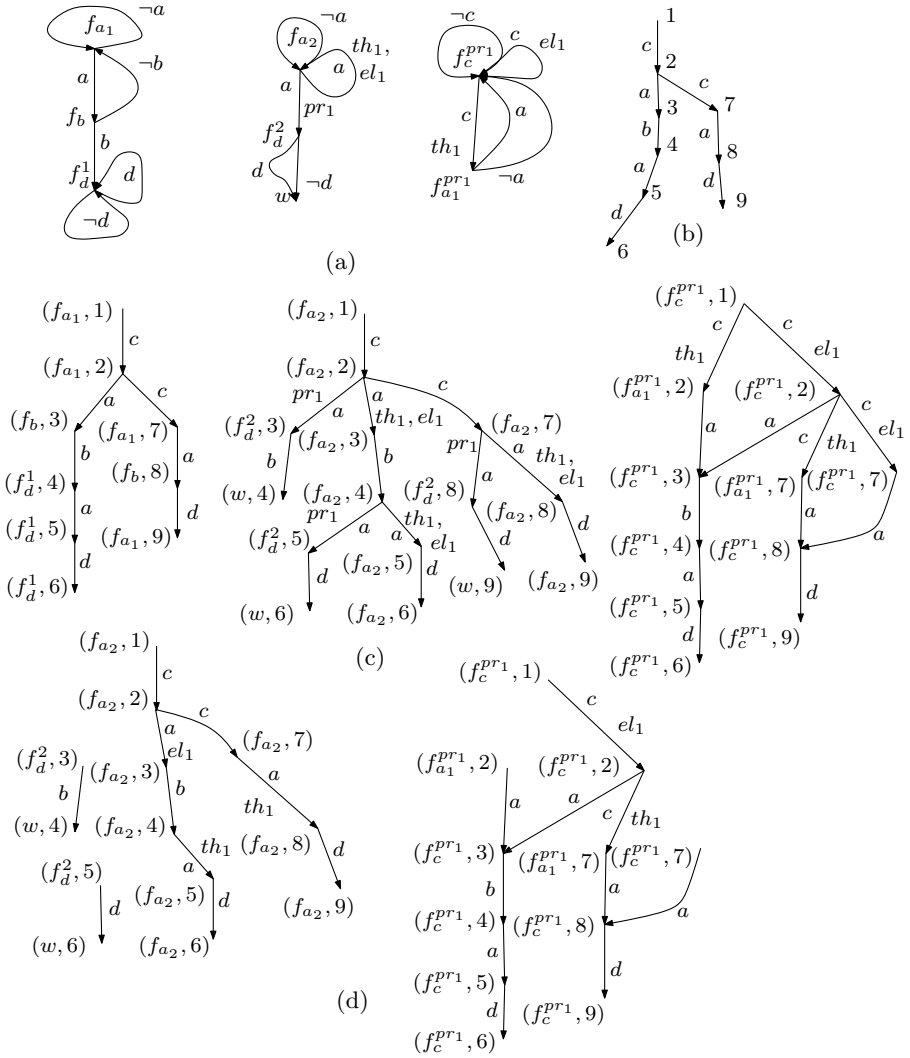


Figure 7: (a) $\mathsf{U}_{\mathsf{f}_{\mathsf{q}_1}}$. (b) An input tree t. (c) $\mathsf{U}_{\mathsf{f}_{\mathsf{q}_1}} \sqcap \mathsf{t}$. (d) A fragment of $\mathsf{G}_{\mathsf{f},\mathsf{t}}$.

In most cases only a subset of structural functions $f_{i_1}, \ldots, f_{i_k}$ is allowed to be called on the document edge. In this case we only consider (f_{i_j}, u_0) as elements of $V.U_f \sqcap t$, where u_0 is the root of t ($1 \leq j \leq k$). Such structural functions will be called *upper structural functions*. In f_{q_1} of Example 3 the upper structural functions are f_{a_1}, f_{a_2} and $f_c^{pr_1}$. An example for intersection can be seen in Fig. 7(c)-(d).

In the sequel, sometimes (e_{U_f}, e_t) will denote an edge in $E.U_f \sqcap t$. Here, e_{U_f}, e_t are called *ancestor images*. (e_{U_f}, e_t) is a *premise* (*constructor*, *then-*, *else-edge*), if its ancestor image in U_f is also a premise (constructor, then-, else-edge).

Deletion of the unnecessary conditional edges. In the next step, in $U_f \sqcap t$ we delete the premises and those then- and else-edges, whose condition is not satisfied. First, we have to note that for

$$Cond = \text{ if } C \text{ then } fo_1 \text{ else } fo_2$$

and for an edge e_t of t, if (e_{U_f}, e_t) is in $E.U_f \sqcap t$ s.t. e_{U_f} is a conditional edge of $Cond$, then e_t form pairs with the rest of the conditional edges of $Cond$. The set of these edges of $U_f \sqcap t$ will be referred as $Cond^{e_t}$. In order to be able to decide whether C is satisfied by e_t and the subtree under e_t, we take a copy of C, which we denote C^{e_t}. The algorithm eliminating the unnecessary conditional edges consists of three steps.

Equality conditions. First, the $val(\theta) = \alpha$ conditions are considered ($\theta \in \Sigma \cup \{*\}$). In Fig. 7(e) we have supposed that $val((1, c, 2)) \neq 5$, hence then-edge $((f_c^{pr_1}, 1), c, (f_{a_1}^{pr_1}, 2))$ is deleted. On the other hand, for $(2, c, 7)$ we have assumed that its value is 5, thus we delete else-edge $((f_c^{pr_1}, 2), c, (f_c^{pr_1}, 7))$.

Formally, in this step, the $val(\theta) = \alpha, val(\theta) = x$ conditions of C^{e_t} are considered. We assume that there is a given variable assignment ρ. With this and the value of e_t we substitute the preceding equality conditions with their truth values (cf. the second table of Fig. 6).

If C^{e_t} becomes true, then clearly, the then-branch should be executed, hence except for the then-edges, we delete all other conditional edges belonging to $Cond^{e_t}$. If an edge is a then- and else-edge at the same time, it is also kept. These and the remaining then-edges are considered as normal (non-conditional) edges in further steps of the algorithm. We refer to this method in the sequel as *deletion with respect to the then-branch.*

On the other hand, if C^{e_t} becomes false, then, for obvious reasons, we delete with respect to the else branch, i.e. edges with label el are kept, and considered as normal edges further on. Note that owing to the presence of n.i. conditions, we may not be able to determine the truth value of C^{e_t}.

The result of this step is denoted $\hat{G}_{f,t}$.

Registers. Second, we consider the constraints given by the registers. For an edge $((f_i, u), b, (\varphi_1, v)) \in \hat{G}_{f,t}$ and a restriction of registers $\mathtt{Reg}$, an atomic condition, $X^b_{f_i} = X^b_{f_j}$, becomes true, if there exists another edge $((f_j, u), b, (\varphi_2, v)) \in \hat{G}_{f,t}$ $(\varphi_1, \varphi_2 \in \{f_1, \ldots, f_n, w\})$. In other words, the b rows of f_i and f_j should be called on (u, b, v). Otherwise, $X^b_{f_i} = X^b_{f_j}$ becomes false. If at the end, $\mathtt{Reg}$ becomes false, then $((f_i, u), b, (\varphi_i, v))$ should be deleted from $\hat{G}_{f,t}$. The result after the evaluation of register restrictions is denoted $\check{G}_{f,t}$. Again, an example can be found in Fig. 7.(e). Here, $X^d_{f^1_d} = X^d_{f^2_d}$ is not satisfied by $((f^2_d, 8), d, (w, 9))$, since $(8, d, 9)$ has not got any b ancestor, f^1_d is not called on it. On the other hand, $((f^1_d, 5), d, (f^1_d, 6))$ and $((f^2_d, 5), d, (w, 6))$ satisfies this register restriction.

Not-isempty conditions. Third, the n.i. conditions are evaluated. (i) If from a premise of $\check{G}_{f,t}$ a constructor edge is reachable through a path not containing any conditional edges, then the n.i. condition in question, n.i.$(f_j(t))$, is satisfied, thus we substitute n.i.$(f_j(t))$ in C^{e_t} with a $\mathtt{true}$ constant. If C^{e_t} becomes true, then we delete with respect to the then-branch. In Fig. 7(e) the n.i. condition of $((f_{a_2}, 4), a, (f^2_d, 5))$ is satisfied.

(ii) If there are neither constructor, nor conditional edges reachable from the premise in question, then there is no further possibility to satisfy the corresponding n.i.$(f_j(t))$ condition, hence we substitute it with constant $\mathtt{false}$. If C^{e_t} becomes false, then we delete with respect to the else-branch. Again, in Fig. 7(e) the n.i. condition represented by premise $((f_{a_2}, 2), a, (f^2_d, 3))$ is not satisfied.

The algorithm stops, when there are no premises left. Suppose now that there are still premises, nevertheless steps (i)-(ii) cannot be applied. This means that each path from a premise to a constructor edge contains at least one conditional edge e beside the premise in question. If e is a then-, or an else-edge, since the equality conditions are all checked, we know that there is also a premise belonging to e. Hence, at the end we conclude that some of the premises form cycles, whereas, clearly, $U_f \sqcap t$ is a tree. Thus, the algorithm surely stops. Denote $G_{f,t}$ the result.

Construction of the result.

Example 4 *To describe the construction of the output we use another structural recursion* $f = (f_1, f_2, f_3)$ *as an example, where* f_1 *is the upper structural function. (The operational graph of* f *can be found in Fig. 8(a).)*

$f_1\colon (\{a : t\}) = \{b : \{\{a : f_2(t)\}@\{c : f_3(t)\}\}\}$ $\quad$ $f_2\colon (\{* : t\}) = \{* : f_2(t)\}$

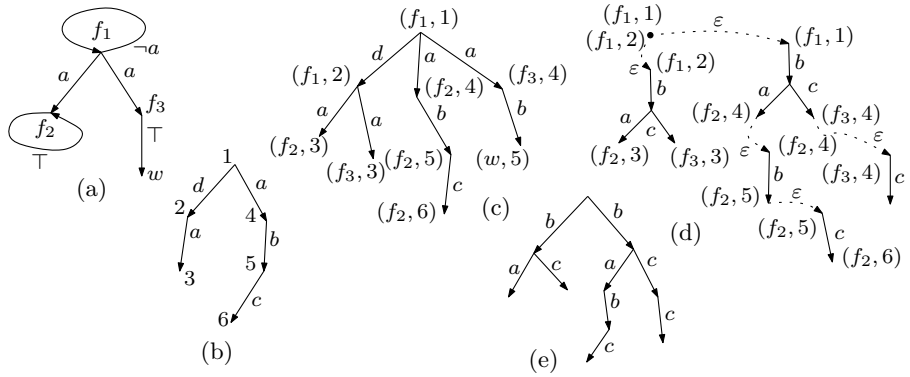


Figure 8: (a) The operational graph of $\mathsf{f} = (\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3)$. (b) An input. (c) The intersection of the operational graph of (a) and the input of (b). (d) Construction and connection of the basic forests. (e) The final result.

$$(\{* : \mathsf{t}\}) = \mathsf{f}_1(\mathsf{t}) \qquad\qquad \mathsf{f}_3\colon (\{* : \mathsf{t}\}) = \{\mathsf{c} : \{\}\}$$

If an edge $\mathsf{e}_\mathsf{t} = (\mathsf{u}, \mathsf{a}, \mathsf{v})$ of a document tree t is processed according to the a row of f_1, i.e., $((\mathsf{f}_1, \mathsf{u}), \mathsf{a}, (\mathsf{f}_j, \mathsf{v}))$ is in $\mathsf{E}.\mathsf{U}_\mathsf{f} \sqcap \mathsf{t}$ $(\mathsf{j} = 1, 2)$, then we take a tree $\hat{\mathsf{t}} = \{\mathsf{b} : \{\{\mathsf{a} : \{\}\}@\{\mathsf{c} : \{\}\}\}\}$, and we label its root with $(\mathsf{f}_1, \mathsf{u})$ and the leaves of the b and c edges with $(\mathsf{f}_2, \mathsf{v})$, $(\mathsf{f}_3, \mathsf{v})$, respectively. The labels of the leaves indicate that the results of f_2 and f_3 applied on the subtrees under e_t should be "connected" to $\hat{\mathsf{t}}$. Similarly, the label of the root shows that which fragment of the result $\hat{\mathsf{t}}$ should be connected to. The connection is accomplished through ε edges. As an example, consider Fig. 8(a)-(e). Note that here, when f_1 is applied to $(1, \mathsf{d}, 2)$, only a node is constructed with two labels (cf. Fig. 8(d)).

Formally, denote $\mathsf{E}(\mathsf{f}_\mathsf{i}, \mathsf{e}_\mathsf{t})$ the set of neighbouring edges in $\mathsf{G}_{\mathsf{f},\mathsf{t}}$, whose labels are the same, and whose ancestor image is $\mathsf{e}_\mathsf{t} = (\mathsf{u}, \mathsf{a}, \mathsf{v})$ in t. Such a set may consists of only one edge. The ancestor images in U_f are all of the form $(\mathsf{f}_\mathsf{i}, \mathsf{p}, \varphi)$, $\varphi \in \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{n}, \mathsf{w}\}$, i.e., they belong to the same condition. In accordance with the previous observation, the edges of $\mathsf{E}(\mathsf{f}_\mathsf{i}, \mathsf{e}_\mathsf{t})$ represent that in the process of t, f_i is called on e_t, as a consequence, forest fo should be constructed, and the structural functions appearing as leaf labels should be called. To represent this, for $\mathsf{E}(\mathsf{f}_\mathsf{i}, \mathsf{e}_\mathsf{t})$ we take a copy of fo, we substitute the $*$ labels with a (the label of e_t), we label the root with $(\mathsf{f}_\mathsf{i}, \mathsf{u})$, and if a leaf has label $\mathsf{f}_\mathsf{j}(\mathsf{t})$, then we change it to $(\mathsf{f}_\mathsf{j}, \mathsf{v})$. These new labels will be used to establish the connection between these forests. Denote $\mathsf{fo}(\mathsf{f}_\mathsf{i}, \mathsf{e}_\mathsf{t})$ the result,

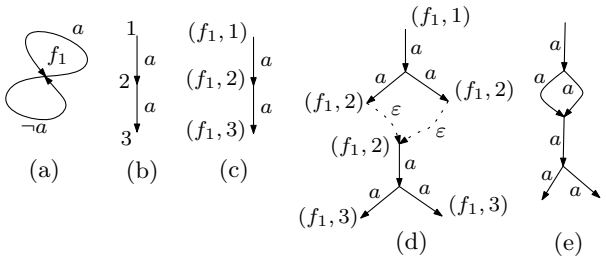


Figure 9: (a) The operational graph of $\mathsf{f} = (\mathsf{f}_1)$. (b) An input. (c) The intersection of the operational graph of (a) and the input of (b). (d) Construction and connection of the basic forests. (e) The final result.

which we call *basic forest* in the sequel.

The output should be constructed from the basic forests. For this, first, add ε edges from the (f_j, ν) leaves to the (f_j, ν) roots. Second, if there are more than one (f_j, ν) roots, then choose one, and add ε edges to the rest of these roots. At the last step the ε edges should be eliminated.

In most cases we get forests as results instead of trees. Hence, one may choose a root structural function f_k among the upper structural functions $(1 \leq k \leq n)$. This means that in the output only the tree reachable from the root of $\mathsf{fo}(\mathsf{f}_k, /)$ (f_k called on the document edge) should be considered.

Example 5 *The next example in Fig. 9(a)-(e) with structural recursion* $\mathsf{f} = (\mathsf{f}_1)$ *shows how our semantics avoids outputs of exponential size.*

$$\begin{aligned}\mathsf{f}_1\colon (\{a : t\}) &= \{a : \{\{a : \mathsf{f}_1(t)\}@\{a : \mathsf{f}_1(t)\}\}\}\\ (\{* : t\}) &= \mathsf{f}_1(t)\end{aligned}$$

Note that the result in Fig. 9(e) does not fit our data model, since it cannot be constructed by using constructors: $@, \{l : t\}, \{\}$. However, it "encodes" all necessary information, and it is easy to unfold it to a tree in our data model.

Order of the result. As a consequence of rule

$$\mathsf{f}_i : (\{t_1@t_2\}) = \mathsf{f}_i(t_1)@\mathsf{f}_i(t_2),$$

for a structural recursion f and document tree t, if e^1 precedes e^2 in document order $(e^1, e^2 \in \mathsf{E}.t)$, then the edges of the fragment constructed by calling f on e^1 and the subtree under e^1 precedes the edges of the fragment

resulted from the process of e^2 and the subtree under e^2. In the simulation of XPath_0 expressions the aforementioned fragments are single ψ edges, hence the document order straightforwardly defines the order of the result.

When XSLT_0 is considered, for each input, the simulated program will define the order among the basic forests.

Number of steps. If we consider the size of f, $|f|$, as the number of equality conditions, structural functions appearing on the right sides of rows (each appearance of f_j counts one) and atomic conditions of registers, and the size of t as $|V.t| + |E.t|$, then the size of $U_f \sqcap t$ is $O(|f||t|)$. The equality constraints can be checked in $O(K|f||t|)$ steps, where $K = \max\{|\rho(x)| \mid x \text{ is used in } f\}$. Clearly, $K \leq |t|$. Note that if there are no variables, then the equality conditions can be checked in $O(|f||t|)$. For the register restrictions, we suppose that for each edge $(u, a, v) = e \in E.t$, we have an array that stores f_i, if $((f_i, u), a, (\varphi, v)) \in \hat{G}_{f,t}$ ($\varphi \in \{f_1, \dots f_n, w\}, 1 \leq i \leq n$). With this, the register restrictions, satisfied by e, can be found in $O|f|$ steps. Hence register restrictions can be checked in $O(|t||f|)$ steps. Since a path from a premise to a constructor edge contains at most $|f|$ different conditional edges, $G_{f,t}$ can be constructed in $O(|f|^2|t|)$ steps. All in all, supposing that $|f| < |t|$, we get that $f(t)$ can be constructed in $O(|t|^2|f|)$ time. If variables are not used, then $f(t)$ can be constructed in $O(|t||f|^2)$ time. Furthermore, if the number of embedded n.i. conditions is limited with a constant, as in the case of the following simulations of XPath_0 expressions and XSLT_0 programs, then $f(t)$ can be computed in $O(|t||f|)$ time.

4 Rewriting of XPath_0

First of all, we have to define formally the equivalence of an XPath_0 expression and a structural recursion. For this, let

$$q = \chi_1 :: \tau_1[p_{1_1}] \dots [p_{m_1}] / \dots / \chi_n :: \tau_n[p_{1_n}] \dots [p_{m_n}]$$

be an XPath_0 expression. Then for a document tree t, $e_0 \in E.t$, a sequence of edges $e_0, e_1, \dots, e_n$ is called a *result-chain (of q)*, if $e_i \in E.t$, $e_i \chi_{i+1} e_{i+1}$, $val(e_{i+1}) = \tau_{i+1}$, and $p_{j_{i+1}}(e_{i+1})$ is true ($0 \leq i \leq n-1, 1 \leq j \leq m$). If we denote

$$\chi_1 :: \tau_1[p_{1_1}] \dots [p_{m_1}] / \dots / \chi_j :: \tau_j[p_{1_j}] \dots [p_{m_j}]$$

with q_{τ_j}, then $e_j \in q_{\tau_j}(e_0)$, i.e., e_j is selected by q_{τ_j} initialized on e_0 ($1 \leq j \leq n$). Furthermore, we say that an edge e is *touched*, if there exists a j s.t. $e \in q_j(e_0)$, or e is touched in one of the pr_{i_j}-s. A touched edge e is *uppermost*,

if there is not any edge $e' \in E.t$ s.t. e' is also touched, and e' is an ancestor of e.

Lemma 1 *Let* q *be an XPath*$_0$ *expression without predicates,* t *a document tree,* $e_0 \in E.t$. *Then, there exists at most one uppermost node for* t *and* e_0.

Proof. Suppose that $q(e_0)$ is not empty, i.e., there exists at least one result chain, and there are two uppermost nodes e and e'. Then neither e is the ancestor of e', nor e' is the ancestor of e. However, since t is a tree, they have at most one common ancestor e''. As we only use axes `self, child, desc, par, anc`, it is easy to see, that e'' is touched, thus we get a contradiction.

If $q(e_0)$ is empty, then let i be that maximal number, to which $q_{\tau_i}(e_0)$ is not empty. Then, the previous reasoning can be applied to q_{τ_i}. If $q_{\tau_i}(e_0)$ is empty for all i ($1 \leq i \leq n$), then e_0 is defined to be the uppermost node. ■

Now, denote $t_q^{e_0}$ the subtree of t containing all of the touched edges. From the lemma straightforwardly follows that $t_q^{e_0}$ is rooted. Now, for simulating XPath$_0$ expressions we shall use special structural recursions constructing only ψ edges.

Definition 2 *Let* $f = f_1, \ldots, f_n$ *be a structural recursion constructing only* ψ *edges and* t *a document tree. For an edge* $(u, a, v) \in E.t$ *we say that* f stops *on* (u, a, v), *if there is an edge* $(f_i, p, \varphi) \in E.U_f$, *s.t.* $((f_i, u), p \wedge a, (\varphi, v)) \in G_{f,t}$, *and as a result of this edge a* ψ *edge is constructed* ($\varphi \in \{f_1, \ldots, f_n, w\}$).

Definition 3 *Let* q *be an XPath*$_0$ *expression and* f *a structural recursion. Then* f *is* equivalent with q, *in notation* $f \simeq q$, *if for all document tree* t *and* $e_0, e \in E.t$, $e \in q(e_0)$ *iff* f *called on* $t_q^{e_0}$ *stops on* e.

Note that the definition still makes sense, when $q(e_0)$ is empty.

XPath$_0$ without predicates

In this subsection, when we talk about an XPath$_0$ expression q, we always assume that it is without predicates. Consider the following expression

$$q_2 := \texttt{self::a/child::b/desc::d/anc::c},$$

where `anc` is an abbreviation of `ancestor`. Here we do not know whether a or b is an ancestor or a descendant of c. Hence when we are to simulate q_2, we have to construct structural recursions checking only whether an a edge has a b child, having a descendant d edge etc.

Furthermore, structural recursions run in a top-down manner, thus if a structural recursion simulating q_2 stops on appropriate c edges, it should have already checked whether there are d edges below these c edges, not to mention other relations among other edges.

Finally, we should keep in mind that if for $x, y, z \in E.t$, $lab(y) = lab(z)$, $x\chi y, x\chi z$ both holds, where χ, again, denotes an axis, then the corresponding simulation should both return y and z. That is, in the document

```
<a><b><b>...</b></b></a>
```

there are two descendant b elements, and both should be returned in a simulation of `self::a/desc::b`.

From now on, we assume that axis `self` only appears as the first axis of an expression. Since subexpression `a/self::b` is unsatisfiable, while in subexpressions `a/self::a`, `a/self::*`, `self::a`, `self::*` can be omitted, our assumption is justified.

XPgraphs. Let q be an $XPath_0$ expression. An edge test a precedes edge test b in q, if a is written first in left-to-right order. The last edge test is called *aim*. If there are more than one a edge test in q, then the first occurrence is indexed a_1, the second a_2 etc. Thus, in the rest of this section we shall assume that all edge tests are different.

For representing the relations between edge tests we construct an auxiliary graph, *XPgraph*. We denote the XPgraph of q with XP_q. The nodes of XP_q are labelled with edge tests of q. There is an edge from a to b, if either `a/desc::b`, or `b/anc::a` is a subexpression of q. In case of `a/child::b`, `b/par::a` this edge is labelled with ch, par respectively. Clearly, XP_q is without cycles. The representation of the aim is called *aim node*. In XP_q node a precedes node b, if edge test a precedes edge test b. A node a in XP_q is an *upper node*, if it has no ingoing edges.

Example 6 *For* $q_3 =$ *`self::a/child::b/desc::d/par::a`,* XP_{q_3} *is given in Fig. 10(a).*

XPgraphs with one upper node. For the simulation of q we use XP_q. First we suppose that it has only one upper node. First we illustrate the method with an example.

Example 7 *The representation of* $q_4 :=$ *`self::a/anc::b/child::d/child::c`,* $f_{q_4} = (f_a, f_b, f_c, f_d)$*, is the following (*$XP_{q_4}$ *can be found in Fig. 10(b)):*

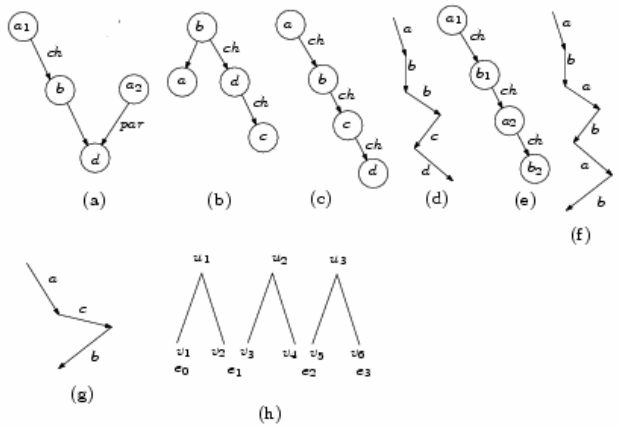


Figure 10: (a) XP_{q_3}. (b) XP_{q_4}. (c) The XPgraph of Example 8 (d) The document tree of Example 8. (e) The XPgraph of Example 9. (f) The document tree of Example 9. (g) A document tree.

$\mathsf{f_b}$: $(\{\mathsf{b} : \mathsf{t}\}) =$ *if* *n.i.*$(\mathsf{f_a}(\mathsf{t}))$ *then* $\mathsf{f_d}(\mathsf{t})$ *else* $\mathsf{f_b}(\mathsf{t})$
$(\{* : \mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$

$\mathsf{f_c}$: $(\{\mathsf{c} : \mathsf{t}\}) = \{\psi : \{\}\}$
$(\{* : \mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$

$\mathsf{f_a}$: $(\{\mathsf{a} : \mathsf{t}\}) = \{\psi : \{\}\}$
$(\{* : \mathsf{t}\}) = \mathsf{f_a}(\mathsf{t})$

$\mathsf{f_d}$: $(\{\mathsf{d} : \mathsf{t}\}) = \mathsf{f_c}(\mathsf{t})$
$(\{* : \mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$

In the simulation to each $\mathsf{a} \in \mathsf{V.XP_q}$ we assign a structural function $\mathsf{f_a}$. Each such function contains two rows: the default case and another row corresponding to its label. With the latter we specify what should happen, when the "desired label" is reached. For instance, the function representing the aim node, in accordance with the definition of the equivalence of XPath_0 expressions and structural recursions, should construct a ψ edge. In the default case, we can control whether we are to simulate a `desc`, `anc` axis, or a `child`, `par`. In our example, $\mathsf{f_c}$ calls $\mathsf{f_b}$ in the default case, since the b edge should be followed by a c edge immediately. For similar reasons, $\mathsf{f_d}$ also calls $\mathsf{f_b}$ in its default case. On the other hand, $\mathsf{f_a}$ should call itself, since there may be arbitrary number of edges between b and a.

The upper node of XP_{q_4} is b, hence the process of an input should start

with f_b, thus it is defined to be the upper structural function. With the n.i.$(f_a(t))$ condition we check whether the examined b edges have an a descendant. Note that in functions representing a leaf different from the aim node, in the non-default row we should also construct a ψ edge, since the check of a n.i. condition ends there.

Formally, for edges $(a, b) \in E.XP_q$ in the default case f_a calls itself. For edges $(a, par, b) \in E.XP_q$, in the default case the process should stop with $\{\}$. On the other hand, in case of $(b, ch, a) \in E.XP_q$, first we have to find the maximal child-chain, $u_1, \ldots, u_m$ s.t. $u_i \in V.XP_q, (u_i, ch, u_{i+1}) \in E.XP_q, u_m = a, (u_{m-1} = b)$. Then, in the default case f_a should call f_{u_1}.

In the a row $(\{a : t\}) = \ldots)$, (i) if a is a leaf, or the aim node, a ψ edge should be constructed. If there is a condition (see below), then this ψ edge should be constructed in the then-branch.

(ii) If (a, b) or (a, ch, b) or (a, par, b) is the only outgoing edge of a, then, if a is the aim node, then n.i.$(f_b(t))$ is called, and a ψ edge should be constructed in the then-branch. In the else branch f_a should call itself. In case of edge (a, par, b) and for edge (a, b), if a is the ancestor of b, a is surely the aim node. Otherwise, if a is not the aim node, f_b should be called. Again, in case of edge (a, ch, b) and for edge (a, b), if b is the descendant of a, a is surely not the aim node.

(iii) If a has another outgoing edge beside $(a, b), (a, par, b)$ or (a, ch, b), then if the aim node is not reachable through b on a directed path, then f_b should be called in a n.i. condition. Otherwise, it should be called in the then-branch. In the else-branch, again, f_a should call itself. Note that in this case a is an upper node.

Denote $\tilde{f}_q$ the result structural recursion. Here, the only upper structural function is the structural function of the upper node. In order to describe the connection between q and $\tilde{f}_q$, we introduce two notions.

Definition 4 *For an XPath$_0$ expression $q = \chi_1 :: \tau_1 / \ldots / \chi_n :: \tau_n$, document tree t, $e_0 \in E.t$, $e \in E.t$ is an* uppermost result element of $q_{\tau_i}(e_0)$*, if (i) $e \in q_{\tau_i}(e_0)$, e is in a result chain, (ii) there does not exist any $e' \in E.t$ s.t. e' is an ancestor of e, $e' \in q_{\tau_i}(e_0)$, and e' is also in a result chain.*

Definition 5 *Let q be an XPath$_0$ expression without predicates, t a document tree, $(u, a, v) = e \in E.t$. Then, we say that* the a row of f_a in $\tilde{f}_q$ is called successfully on e*, if (i) $((f_a, u), a, (\theta, v)) \in E.U_f \sqcap t$, (ii) if there is a condition in the a row, it is satisfied $(\theta \in \{f_\sigma, w\}, \sigma \in \Sigma)$.*

With $e \in R^t(a, f_a)$, or shortly $e \in R(a, f_a)$, if the input is clear from the

context, we denote that the a row of f_a is called successfully on e.

Lemma 2 *Let* $q = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$ *be an XPath*$_0$ *expression s.t.* XP_q *has only one upper node. Let* t *be a document tree,* $e_0 \in E.t$. *Consider* τ_j, *and* $e \in E.t$ *s.t.* $lab(e) = \tau_j$, *and* e *does not have any ancestor* e' *s.t.* e' *is an uppermost result element of* $q_{\tau_j}(e_0)$. *Then* $e \in q_{\tau_j}(e_0)$ *iff* $e \in R(\tau_j, f_{\tau_j})$ (f_{τ_j} *is in* $\tilde{f}_q, 1 \leq j \leq n$).

Proof. Suppose that $(u, \tau_i, v) = e^1$ is the uppermost element of $t_q^{e_0}$. From this it follows that τ_i is the upper node of XP_q, and $((f_{\tau_i}, u), \tau_i, (\theta, v)) \in E.U_f \sqcap t$ ($\theta \in \{f_\sigma, w\}, \sigma \in \Sigma$).

(a) If $e^1 \in q_{\tau_i}(e_0)$, then there exists a sequence of edges $e_0, \ldots, e_i$ s.t. $e_i = e^1$, $e_j \chi_{j+1}, e_{j+1}$ ($0 \leq j \leq i-1$). Since τ_i is the upper node, f_{τ_i} calls $f_{\tau_{i-1}}$ on e in a n.i. condition. Now, suppose that χ_{i-1} is `anc`. Then the τ_{i-1} row of $f_{\tau_{i-1}}$ is only instantiated when the first edge under e^1 with label τ_{i-1} is reached. If τ_{i-1} is a leaf in XP_q, a ψ edge is constructed, and the n.i.condition is satisfied, consequently, e^1 is in $R(\tau_i, f_{\tau_i})$.

If χ_{i-1} is `par`, the reasoning is similar. The proof can be continued inductively on the rest of the edge tests before τ_i. At the end we get that $e^1 \in R(\tau_i, f_{\tau_i})$

For the other direction, if $e^1 \in R(\tau_i, f_{\tau_i})$, and there is no condition in the τ_i row, then τ_i is not preceded by any edge test in q, hence $i = 1$. Obviously, $e^1 \in q_{\tau_i}(e_0)$. Otherwise, the condition is satisfied. Again, if τ_{i-1}`/anc::`τ_i (τ_{i-1}`/par::`τ_i) is a subexpression of q, then a trivial analysis of $f_{\tau_{i-1}}$ shows that it is guaranteed that e^1 has a descendant (child) with label τ_{i-1}. The proof can be continued inductively. At the end we get that $e^1 \in q_{\tau_i}(e_0)$. Hence, the statement of the lemma holds for e^1.

Beside the supposition that $e^1 \in q_{\tau_i}(e_0)$, assume further that τ_{i-1}`/anc::`τ_i is a subexpression of q. Let e^2 be a descendant of e^1 s.t. $lab(e^2) = \tau_{i-1}$, and there is not any ancestor e^3 of e^2 s.t. e^3 is a descendant of e^1, $lab(e^3) = \tau_{i-1}$. Such edges will be called *first* τ_{i-1} *descendants*. The previous reasoning shows that the τ_{i-1} row of $f_{\tau_{i-1}}$ is called on e^2. Furthermore, from the rewriting rules we also know that this τ_{i-1} row does not contain any condition. Consequently, $e^2 \in R(\tau_{i-1}, f_{\tau_{i-1}})$. Additionally, since $e^1 \in q_{\tau_i}(e_0)$, $e^2 \in q_{\tau_{i-1}}(e_0)$. Hence, the statement also holds for e^2. The proof is similar, when τ_{i-1}`/par::`τ_i is a subexpression of q. Thus, it can be shown inductively that the lemma holds for all edge tests before τ_i in q.

If τ_i`/desc::`τ_{i+1} is a subexpression, then f_{τ_i} calls $f_{\tau_{i+1}}$, which again finds the first τ_{i+1} descendants of e^1. Again in its τ_{i+1} row $f_{\tau_{i+1}}$, even if τ_{i+1} is the

aim node, does not have any condition. Consequently, the lemma also holds for these edges.

In case of τ_i/child::τ_{i+1}, if e^2 is a child of e^1 with label τ_{i+1}, then the statement is true for e^2. Otherwise, $f_{\tau_{i+1}}$ calls f_{τ_i} in its default case, and the whole previous reasoning can be applied to the first τ_i descendants of e^2.

(b) If $e^1 \notin q_{\tau_i}(e_0)$, then the condition of the τ_i row of f_{τ_i} is not satisfied, hence f_{τ_i} is called again in the else branch. Again, the reasoning is similar for the first τ_i descendants.

Furthermore, if edge e is an uppermost result element of $q_{\tau_j}(e_0)$, then it is easy to show, that f_{τ_j} is not instantiated on any descendants of e. ■

In other words the lemma says that $\tilde{f}_q$, for an arbitrary input tree and a fixed edge e_0, finds all of those "uppermost" edges that may be touched, or may be elements of a result chain of q called on e_0.

Corollary 1 *Keeping the assumptions of Lemma 2, for document tree* t*, edge test* τ_j*,* $e \in E.t$*, where* $lab(e) = \tau_j$*, and for an arbitrary edge* $\bar{e}$ *of* t*, if* e *does not have any ancestor* e' *s.t.* e' *is an uppermost result element of* $q_{\tau_j}(\bar{e})$*. Then* $e \in q_{\tau_j}(\bar{e})$ *iff* $e \in R(\tau_j, f_{\tau_j})$ (f_{τ_j} *is in* $\tilde{f}_q, 1 \le j \le n$)*.*

Note that the only difference between Lemma 2 and Corollary 1 is that in the latter case we do not fix edge e_0, where the evaluation of q should start.

A document tree t is called *simple*, if for an arbitrary edge e, where $lab(e) = \sigma$, e does not have any σ ancestor in t.

Corollary 2 *Keeping the above assumptions and notations, if* t *is simple, then* q *is equivalent with* $\tilde{f}_q$*.*

Corollary 2 shows that $\tilde{f}_q$ simulates q correctly for a large and practically the most important class of document trees.

To establish the equivalence for all document trees, we have to develop a modified version of function f_σ to be able to process σ edges that are in a descendant-ancestor relationship. For this we introduce a method called the *repetition of* σ. Informally, in the σ row we call f_σ or the appropriate structural function again. As an example of repetition, consider $f = (f_{a_1}, f_b, f^1_d)$ of Example 3 that simulates `self::a/child::b/desc::d`. There, we have repeated d in f^1_d.

Formally, in the repetition of τ_i, if neither the ingoing, nor the outgoing edge of τ_i is labelled with ch or par, then in its τ_i row f_{τ_i} should also call itself. This means that on the right side (in the then branch, if it exists) instead of $\{\psi : \{\}\}$ or $f_{\tau_{i+1}}(t)$, $\{\psi : \{\}\}@f_{\tau_i}(t)$ or $f_{\tau_{i+1}}(t)@f_{\tau_i}(t)$ should be written.

If τ_i is in a path $pa = u_1 \dots u_m$ in XP_q s.t. pa has either solely ch, or solely par edges, then (suppose that pa is maximal in terms subsumption with respect to this property), then in the repetition of $\tau_i\ f_{u_1}$ should call itself in its u_1 row.

In order to describe why we have chosen this construction we show two examples.

Example 8 *First, consider the XPgraph of Fig. 10(c), and suppose that we are to repeat* b*, but here* f_b *calls itself in its* b *row. (According to our definition* f_a *should call itself in its* a *row.)*

$$\begin{array}{ll} f_a\colon (\{a:t\}) = f_b(t) & f_b\colon (\{b:t\}) = f_c(t)@f_b(t) \\ \quad (\{*:t\}) = f_a(t) & \quad (\{*:t\}) = f_a(t) \\ \\ f_c\colon (\{c:t\}) = f_d(t) & f_d\colon (\{d:t\}) = \{\psi:\{\}\} \\ \quad (\{*:t\}) = f_a(t) & \quad (\{*:t\}) = f_a(t) \end{array}$$

It is not difficult to see that for the document tree of Fig. 10(d) this structural recursion constructs a ψ-edge, though it should not. Here the problem is that it is not checked whether the second b edge has an a parent. This shows that in the repetition the whole check should start from the beginning, hence f_{u_1} should be called.

Example 9 *Secondly, for the XPgraph of Fig. 10(e) we repeat* a_2*, but* f_{a_1} *is called in the* a *row of* f_{a_2} *instead of the* a *row of* f_{a_1}*.*

$$\begin{array}{ll} f_{a_1}\colon (\{a:t\}) = f_{b_1}(t) & f_{b_1}\colon (\{b:t\}) = f_{a_2}(t) \\ \quad (\{*:t\}) = f_{a_1}(t) & \quad (\{*:t\}) = f_{a_1}(t) \\ \\ f_{a_2}\colon (\{a:t\}) = f_{b_2}(t)@f_{a_1}(t) & f_{b_2}\colon (\{b:t\}) = \{\psi:\{\}\} \\ \quad (\{*:t\}) = f_{a_1}(t) & \quad (\{*:t\}) = f_{a_1}(t) \end{array}$$

Again, for the document tree of Fig. 10(f) only one ψ edge is constructed as a result of the second b edge, though as a result of the third b edge another ψ edge should also be constructed. This is because f_{a_1} is not called on the second a edge. This shows that in the repetition f_{u_1} should be called in its own u_1 row.

Suppose now that in Example 9 we call f_{a_1} in its a row. Then for the document tree of Fig. 10(g) after the c edge two "instances" of f_{a_1} are called on b, although only one should process this edge. Hence, we should slightly change the rewriting rules, when a node of a child-chain is to be repeated. In this case, in the default cases $f_{u_2}, \ldots, f_{u_m}$ instead of calling f_{u_1} the empty graph should be constructed. With this the XPgraph of Fig. 10(e) should be rewritten as follows.

$f_{a_1} : (\{a : t\}) = f_{b_1}(t)@f_{a_1}(t)$
$(\{* : t\}) = f_{a_1}(t)$

$f_{b_1} : (\{b : t\}) = f_{a_2}(t)$
$(\{* : t\}) = \{\}$

$f_{a_2} : (\{a : t\}) = f_{b_2}(t)$
$(\{* : t\}) = \{\}$

$f_{b_2} : (\{b : t\}) = \{\psi : \{\}\}$
$(\{* : t\}) = \{\}$

Finally, if we are to repeat $\tau_i = u_k$ and $\tau_j = u_r$, $(1 \leq k, l \leq m)$, then it is enough to call f_{u_1} in its u_1 row once.

Lemma 3 *For a given XPath$_0$ expression $q = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$, where XP_q has only one upper node, document tree t and $e_0 \in E.t$, $e \in E.t$ is in $q_{\tau_i}(e_0)$ iff $e \in R(\tau_i, f_{\tau_i})$, where we have repeated τ_i in $\tilde{f}_q$.*

Proof. The statement straightforwardly follows from the proof of Lemma 2 and from the previous consideration. ■

Now, repeat τ_n in $\tilde{f}_q$. We denote the result with f_q.

Corollary 3 *Keeping the above notations, $q \simeq f_q$.*

Note that from Lemma 2 and Lemma 3 it also turns out that, when we simulate q with f_q, and traverse t top-down, in most cases we only look for the first elements that are in a result chain. When we find one, the corresponding part of the process stops. It seems hard to find such an implementation strategy that would not process these elements in a top-down traverse. Thus, we can say that we only process those edges that are necessary to process.

XPgraphs with several upper nodes. Unfortunately, till now, we have not found any rewriting technique with which we could simulate an XPath$_0$ expression, whose XPgraph has several upper nodes, with a structural recursion not using registers. However, with registers, the simulation is almost straightforward.

Let $u_1, \dots u_m$ be the upper nodes of XP_q. For u_i denote $XP_q^{u_i}$ the subtree reachable from u_i in XP_q ($1 \leq i \leq m$). Clearly, each $XP_q^{u_i}$ represents a "subquery" of q. For

$$q_3 = \texttt{self::a/child::b/desc::d/par::a}$$

these subqueries are `self::a/child::b/desc::d` and `desc::d/par::a` (cf. XP_{q_3} in Fig. 10(a)). With the results of the previous paragraphs, the corresponding structural recursion, $f_q^{u_i}$, can be given. Note that a leaf a of XP_q may appear in both $XP_q^{u_i}$ and $XP_q^{u_{i+1}}$ ($1 \leq i \leq m-1$). To differentiate between the appropriate structural functions, we denote them f_a^i, f_a^{i+1}, respectively. With this for such leaves we require $X_{f_a^i}^a = X_{f_a^{i+1}}^a$. The intuition is clear, simply, we are to connect structural recursions $f_q^{u_i}$ and $f_q^{u_{i+1}}$.

However, here, in the structural recursion we have to repeat the preceding a leaves as well, otherwise it is not difficult to see, we may lose elements of the result. We denote these new structural recursions $f_q^{u_i}$ and $f_q^{u_{i+1}}$ again. f_q is constituted by the structural functions of $f_q^{u_i}$-s ($1 \leq i \leq n$). $f_{u_1}, \dots, f_{u_m}$ are the upper structural functions and, since the aim node is reachable from u_m, we designate f_{u_m} to be the root structural function of f_q. It is important to note that the size of f_q is linear in the size of q.

As an example, consider $f_{q_3} = (f_{a_1}, f_b, f_d^1, f_{a_2}, f_d^2)$ of Example 3 with register restriction $X_{f_d^1}^d = X_{f_d^2}^d$. Here, the root structural function is f_{a_2}.

Theorem 1 *Let q be an XPath$_0$ expression without predicates. Then $f_q \simeq q$.*

Proof. For $e \in E.t$ with $e \in R_{reg}^t(a, f_a)$, or shortly $e \in R_{reg}(a, f_a)$, we are to indicate that (i) $e \in R^t(a, f_a)$ (ii) e satisfies all register restrictions $X_{f_a}^a = X_{f_\sigma}^a$ $\sigma \in \Sigma$. (For f_a^i, register restrictions $X_{f_a^i}^a = X_{f_a^{i+1}}^a$ should also be satisfied.)

Assume now that $q = \chi_1 :: \tau_1/\dots/\chi_n :: \tau_n$, XP_q has two upper nodes, u_1, u_2, and the common leaf represents τ_i ($1 \leq i \leq n-1$). $q' := \chi_{i+1} :: \tau_{i+1}/\dots/\chi_n :: \tau_n$. Clearly, for an arbitrary document tree t, $e_0 \in E.t$, $e \in q(e_0)$ iff there exists $\bar{e} \in E.t$, $\bar{e} \in q_{\tau_i}(e_0)$ s.t. $e \in q'(\bar{e})$. Hence, using the results of Corollary 1 and Lemma 3 $e \in q(e_0)$ iff $e \in R_{reg}^t(\tau_n, f_{\tau_n})$. Since the root structural function is f_{u_2}, f_q may only stop on edges in $R_{reg}^t(\tau_n, f_{\tau_n})$. (If there was not any root structural function, since τ_i is the aim node of $XP_q^{u_1}$, f_q would also stop on edges in $R_{reg}^t(\tau_i, f_{\tau_i}^1)$, thus q and f_q would not be equivalent (see Definition 3)). All in all, we get that $e \in q(e_0)$ iff f_q stops on e. The proof can be continued inductively on the number of upper nodes of XP_q in the same way. ■

Now, one may find too costly that for example in f_{q_3} we traverse the whole input with (f_{a_2}, f^1_d), with which, remember, we seek for a edges with d children. There may be too many of such a edges that are not included in the result, and another implementation would not take care of them, since in the previous steps it has already found the appropriate d edges with a b ancestor, whose parent is a. On the reverse side of the coin, in most cases elements do not have descendants with the same name, in these scenarios it is enough to check the first a elements. This information can be obtained when the structure of the input is given by an XML Schema or a DTD. In [4] we show how DTDs and extended DTDs work together with our methods and handle, among other things, the preceding problem.

XPath$_0$ with predicates

In order to ease the notation, we shall assume that

$$q = \chi_1 :: \tau_1[p_1]/\ldots/\chi_n :: \tau_n[p_n].$$

Here, τ_i is called the *base* of pr_i-s ($1 \leq i \leq n$), while

$$q^{sk} = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$$

is called the *skeleton (of* q*)*. First, we suppose that each p_i consists of a single atomic condition, $q^{p_i} = c$ or $q^{p_i} = x$. Furthermore, for a moment, we also assume that all q^{p_i}-s are of the form `self::*` or `self::`τ_i.

Then, when we construct f_q, the τ_i row of f_{τ_i} should be completed with $val(\tau_i) = c$ or $val(\tau_i) = x$ condition in accordance with the conditions of p_i-s. (If there is already a condition, then the conjunction of the conditions should be taken.) In the else branch (if it has not given previously) f_{τ_i} should call itself.

Lemma 4 *Keeping the previous notation* $q \simeq f_q$.

Proof. The statement trivially follows from Theorem 1 and of the additional rule of construction. ■

XPath$_0$ without embedded predicates. Loosening our restrictions, we only assume now that the predicates consist of a single atomic condition, whose XPath$_0$ expression does not contain predicates. Using the indexing of edge tests of the previous subsection, again, we shall assume that the edge tests of q are all different from each other. Straightforwardly, the XPath$_0$

expression of a predicate $q^p = \chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k$ can be rewritten as $\texttt{self::}\tau_0/\chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k$, where τ_0 is the base of q^p. Similarly, condition $q^p = \theta$ means the same as

$$\texttt{self::}\tau_0/\chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k\texttt{[self::*=}\theta\texttt{]}\ (\theta \in \{c, x\}),$$

consequently q^p can be simulated using the technique of the previous paragraph. In what follows, we shall always use this rewriting.

Next, by means of register restrictions, we are to "join" the structural recursions of predicates with the structural recursion of the skeleton using bases as connection points. Hence, when we rewrite q^p, we should take the base as the aim node, which was the starting point of q^p originally. This means that we move in the opposite direction. Consequently, if in q^p χ_j is `ch` or `par`, then in the XPgraph of q^p we should change the `ch` labels to `par` labels and vice versa ($1 \leq j \leq k$). With this $f_{q^{p_i}}$, or shortly f_{p_i}, can be constructed. The structural function corresponding for the predicate of base τ_i will be denoted $f^{p_i}_{\tau_i}$.

In the structural recursion of the skeleton, f_{sk}, we repeat each base (remember that in a child- or parent-chain $u_1 \ldots u_m$, if we are to repeat u_k, u_s, then f_{u_1} is called in its u_1 row only once ($1 \leq k, s \leq m$)). Again, if we did not do so, we may lose elements of the result.

To use bases as connection points, we add the restrictions $X^{\tau_i}_{f_{\tau_i}} = X^{\tau_i}_{f^{p_i}_{\tau_i}}$. f_q is constituted by the structural functions of f_{p_i}-s, f_{sk} and the preceding register restrictions. Its upper structural functions are the structural functions of upper nodes. Moreover, suppose that XP^u is the XPgraph containing the aim node of the skeleton. Then f_u is designated to be the root structural function. As an example consider the rewriting of q_1, $f = (f_{a_1}, f_b, f^1_d, f_{a_2}, f^2_d, f^{pr_1}_c, f^{pr_1}_{a_1})$ of Example 3.

Theorem 2 *Let* q *be an XPath*$_0$ *expression s.t. each edge test has at most one predicate, and the XPath*$_0$ *expressions of these predicates do not contain predicates, then* $q \simeq f_q$.

Proof. The theorem straightforwardly follows from the previous considerations. ■

Corollary 4 *Let* q *be an XPath*$_0$ *expression without variables. Then for an arbitrary document tree* t, $f_q(t)$ *can be constructed in* $O(|f||t|)$ *time. If the size of variables, i.e., the size of the list of edges which the variable is equal to, is restricted with a constant, the evaluation of* $f_q(t)$ *still works in linear time. In worst case scenarios the construction can be accomplished in* $O(|f||t|^2)$ *time.*

General case. Firstly, we allow Boolean combinations of atomic conditions in the predicates. Without loss of generality, we may suppose that base τ_r has predicate $(\neg)(q^{p_1} = b_1)\theta_1 \dots \theta_k(\neg)(q^{p_k} = b_k)$; with $(\neg)$ we indicate that there may be a negation ($\theta_i \in \{\wedge, \vee\}, 1 \leq i \leq k$). Then the corresponding register restriction is of the form: $\hat{R}eg_1\theta_1 \dots \theta_k\hat{R}eg_k$, where $\hat{R}eg_i \in \{Reg_i, \neg Reg_i\}$, and Reg_i is $X^{\tau_r}_{f_{\tau_r}} = X^{\tau_r}_{f^{p_{r_i}}_{\tau_r}}$. In $\hat{R}eg_i\ Reg_i$ is negated if the corresponding $q^{p_i} = b_i$ condition is negated.

Secondly, if we allow predicates in the $XPath_0$ expressions of predicates, then the rewriting algorithm can be continued recursively. Again, the size of the result structural recursion is linear in the size of the simulated $XPath_0$ expression.

5 Rewriting of $XSLT_0$

As in the previous section, first we have to define the equivalence of an $XSLT_0$ program and a structural recursion.

Definition 6 *Let* t, t' *be document trees. Then* t *is equivalent with* t'*, if there is a one-to-one mapping* $\phi : V.t \to V.t'$ *s.t. (i)* $\phi(u_0) = u'_0$*, where* u_0, u'_0 *denote the roots of* t *and* t'*, respectively. (ii) For* $e = (u, a, v) \in E.t$*,* $e' = (\phi(u), a, \phi(v))$ *is also in* $E.t'$*, and* $val(e) = val(e')$*. (iii) If for* $e^1 = (u^1, a^1, v^1), e^2 = (u^2, a^2, v^2) \in E.t$*,* e^1 *precedes* e^2*, then* $(\phi(u^1), a^1, \phi(v^1))$ *also precedes* $(\phi(u^2), a^2, \phi(v^2))$*.*

Definition 7 *For an arbitrary XSLT*$_0$ *program* P *and structural recursion* f*,* P *is equivalent with* f*, if for all document tree* t*,* $\tau_P(t)$ *is equivalent with* $f(t)$*.*

Rewriting of (m, σ)-rules

First, we are to simulate a single (m, σ)-rule, T, with a given variable assignment. For this we assume that T does not construct anything just selects a list of edges for further processing in mode r. An instantiated template then simply constructs a ψ edge and stops. Furthermore, remember that each program contains a special $(st, /)$-rule, which is called on the document edge and constructs a `result` edge as the document edge of the output. Here, we also suppose that this rule invokes the next template in σ descendants in mode m, i.e., T is called. The corresponding program is denoted P_T. With these suppositions we get that P_T constructs a star of ψ edges.

Example 10 *Consider this rather artificial example, where* T *is called on* $\mathtt{a}$ *edges, and if the edge in question has a* $\mathtt{c}$ *ancestor with value* 5*, the next template is invoked on parent* $\mathtt{b}$*, otherwise on children* $\mathtt{c}$*.*

template $\mathtt{st}(/, \epsilon)$ $\quad(\mathsf{T}_{\mathtt{st}})$
  return
    if true *then* $\{\mathtt{result} : \{\}\}$ *at-expr:* $\mathtt{m}(\mathtt{desc::a}, \epsilon, \epsilon)$;
end.

template $\mathtt{m}(\mathtt{a}, \epsilon)$ $\quad(\mathsf{T})$
  return
    if $\mathtt{anc:c} = 5$ *then* $\{\}$ *at-expr:* $\mathtt{r}(\mathtt{par::b}, \epsilon, \epsilon)$;
    if true *then* $\{\}$ *at-expr:* $\mathtt{r}(\mathtt{child::c}, \epsilon, \epsilon)$;
end.

template $\mathtt{r}(\mathtt{b}, \epsilon)$ $\quad(\mathsf{T}_1)$
  return
    if true *then* $\{\psi : \{\}\}$
end.

template $\mathtt{r}(\mathtt{c}, \epsilon)$ $\quad(\mathsf{T}_2)$
  return
    if true *then* $\{\psi : \{\}\}$
end.

Remember that an $(\mathtt{m}, \sigma)$-rule contains conditions of the form: if $\mathtt{c}_i$ then z_i; where $z_i \in \mathcal{F}^{\Delta}(\mathcal{AT})$ (forests with edge labels from Δ and with possible at-expression leaf labels) $(1 \leq i \leq k)$. Since T constructs nothing, each z_i is a node with at-expression $\mathtt{r}(\mathtt{p}, \epsilon, \epsilon)$, where, remember, $\mathtt{p}$ is an XPath_0 expression, the first ϵ means T has no parameters, the second ϵ indicates that there is not any variable assignment. In the sequel $\mathtt{p}$ will be referred as $\mathtt{xp}(z_i)$.

It is easy to see now that such a condition works in the same way as $\mathtt{q}_i :=$ $\mathtt{self::}\sigma\mathtt{[}c_i\mathtt{]/xp}(z_i)$. Denote XP_i the XPgraph of $\mathtt{q}_i$. Then we may speak of the aim node of the i^{th} condition. Clearly, the corresponding structural recursion of $\mathtt{q}_i$, ${}_{\mathsf{T}}\mathtt{f}_{\mathtt{q}_i}$ can be constructed (here T in the subindex implies that the structural recursions simulates a condition in template T). Remember that ${}_{\mathsf{T}}\mathtt{f}_{\mathtt{q}_i}$ is divided into two structural recursions representing

$$\mathtt{self::}\sigma/\mathtt{c}_i \text{ (predicate) and } \mathtt{self::}\sigma/\mathtt{xp}(z_i) \text{ (skeleton).}$$

Denote them ${}_{\mathsf{T}}\mathtt{f}_{\mathtt{c}_i}$ and ${}_{\mathsf{T}}\mathtt{f}_{z_i}$, respectively.

Owing to its distinguished role, the representation of σ will be called *matching node*. Since we are to call structural recursion ${}_{\mathsf{T}}\mathtt{f}_{\mathtt{q}_i}$ on every possible subtree $\{\sigma : t\}$ of the input, σ should also be repeated. If $\mathtt{c}_i$ is a real condition,

then σ is also a base, and according to the rewriting rules of XPath_0 expression with predicates, every base should be repeated. However, if c_i is constant $\mathtt{true}$, then we should repeat σ explicitly. In the rewriting of the program of Example 10, in order to be able to simulate P_T, we introduce two new constructions.

$${}_{T_{st}}f_{/}:(\{/:t\}) = \{\mathtt{result} : \{{}^{z_1}_{T}f_b(t), {}^{z_2}_{T}f_a(t)\}\}$$

$${}^{c_1}_{T}f_c:\ (\{c:t\}) = \text{if } val(c) = 5 \text{ then } {}^{c_1}_{T}f_a(t) \text{ else } {}^{c_1}_{T}f_c(t)$$
$$(\{*:t\}) = {}^{c_1}_{T}f_c(t)$$

$${}^{c_1}_{T}f_a:\ (\{a:t\}) = \{\psi:\{\}\}@{}^{c_1}_{T}f_a(t)$$
$$(\{*:t\}) = {}^{c_1}_{T}f_a(t)$$

$${}^{z_1}_{T}f_b:\ (\{b:t\}) = \text{if n.i.}({}^{z_1}_{T}f_a)(t)) \text{ then } \{\psi:\{\}\}@{}^{z_1}_{T}f_b(t) \text{ else } {}^{z_1}_{T}f_b(t)$$
$$(\{*:t\}) = {}^{z_1}_{T}f_b(t)$$

$${}^{z_1}_{T}f_a:\ (\{a:t\}) = \{\psi:\{\}\}$$
$$(\{*:t\}) = \{\},\qquad X^{a}_{{}^{c_1}_{T}f_a} = X^{a}_{{}^{z_1}_{T}f_a}$$

$${}^{z_2}_{T}f_a:\ (\{a:t\}) = {}^{z_2}_{T}f_c(t)@{}^{z_2}_{T}f_a(t) \qquad (\{*:t\}) = {}^{z_2}_{T}f_a(t)$$

$${}^{z_2}_{T}f_c:\ (\{c:t\}) = \{\psi:\{\}\} \qquad (\{*:t\}) = \{\}$$

if $e \in \tilde{X}^{a}_{{}^{c_1}_{T}f_a}$ then delete ${}^{z_2}_{T}f_a(e)$
else if $\mathtt{true}$ then delete ${}^{z_1}_{T}f_a(e)$

The first of these new constructions is in the right side of the / row of ${}_{T_{st}}f_{/}$. Namely, the leaf of $\{\mathtt{result} : \{\}\}$ is labelled with a set of structural functions ${}^{z_1}_{T}f_b(t), {}^{z_2}_{T}f_a(t)$ instead of a single structural function (cf. Fig. 6), in notation $\{\mathtt{result} : \{{}^{z_1}_{T}f_b(t), {}^{z_2}_{T}f_a(t)\}\}$. With this we indicate that both the results of ${}^{z_1}_{T}f_b, {}^{z_2}_{T}f_a$ called on t should be appended to the $\mathtt{result}$ edge. Here, ${}^{z_1}_{T}f_b, {}^{z_2}_{T}f_a(t)$ are the root structural functions of ${}_{T}f_{z_1}, {}_{T}f_{z_2}$. Note that the root structural function ${}^{c_1}_{T}f_c$ of ${}_{T}f_{c_1}$ is not called anywhere, however, in order to be able to construct the result of ${}_{T}f_{z_1}$, ${}_{T}f_{c_1}$ should also be evaluated, thus when we call ${}^{z_1}_{T}f_b$, implicitly we also call ${}^{c_1}_{T}f_c$.

The second condition is introduced so as to be able to simulate conditions if c_i then z_i;. The meaning of

$$\begin{array}{l}\text{if } e \in \tilde{X}^{a}_{\substack{c_1 \\ T}\, f_a} \text{ then delete } {}^{z_2}_{T} f_a(e)\\ \text{else if } \mathtt{true} \text{ then delete } {}^{z_1}_{T} f_a(e),\end{array}$$

is that if edge $e = (u, a, v)$ of document tree t is processed by the a row of ${}^{c_1}_{T} f_a$ ($e \in X^{a}_{\substack{c_1 \\ T}\, f_a}$) and this edge does not become unreachable in $U_f \sqcap t$ after the evaluation of n.i. conditions (this is denoted with ($e \in \tilde{X}^{a}_{\substack{c_1 \\ T}\, f_a}$)), then edge $(({}^{z_2}_{T} f_a, u), a, ({}^{z_2}_{T} f_c, v))$ (${}^{z_2}_{T} f_a$ called on e) should be deleted from $U_f \sqcap t$. Otherwise edge $(({}^{z_1}_{T} f_a, u), a, (w, v))$ (${}^{z_1}_{T} f_a$ called on e) should be deleted. Since ${}^{c_1}_{T} f_a$ represents the aim of the XPath$_0$ expression of c_1, if e is in $X^{a}_{\substack{c_1 \\ T}\, f_a}$, and the corresponding edge is kept, then c_1 is satisfied by the appropriate subgraph of e, hence this branch of the condition is to be executed. (Note that as a result of an $\{a : t\}$ singleton ${}^{z_2}_{T} f_a$ calls ${}^{z_2}_{T} f_c$, while ${}^{z_1}_{T} f_a$ calls no other structural function, thus the corresponding node is linked to w in U_{f_T}. Consequently, edges $(({}^{z_2}_{T} f_a, u), a, ({}^{z_2}_{T} f_c, v))$, $(({}^{z_1}_{T} f_a, u), a, (w, v))$ are surely in $U_f \sqcap t$.)

As we have already indicated the matching nodes of XP_1, XP_2 should be repeated, which is the a node in both XPgraphs. In the first case a is a child of b, hence according to the rules of repetition ${}^{z_1}_{T} f_b$ should call itself in its b row. Note that b is also the aim node of XP_1, and the aim node should also be repeated. However, according to the rules of repetition, it is enough to call ${}^{z_1}_{T} f_b$ only once.

In the second case ${}^{z_2}_{T} f_a$ calls itself in the a row. Note that according to the rewriting rules of XPath$_0$ expressions, since it is the aim node, again, we should also repeat c.

In the general case in its / row ${}_{T_{st}} f$ calls all of the root structural functions of ${}_{T} f_{q_i}$-s, i.e., $\{\mathtt{result} : \{{}^{z_1}_{T} f_{\sigma_1}, \ldots, {}^{z_k}_{T} f_{\sigma_k}\}\}$ should be constructed on the right side of the / row, where ${}^{z_i}_{T} f_{\sigma_i}$ denotes the root structural function of ${}_{T} f_{q_i}$ ($1 \le i \le k$). (Remember that ${}_{T} f_{q_i}$ represents condition *if* c_i *then* z_i of T.)

Furthermore, when we are to simulate conditions:

$$\text{if } c_1 \text{ then } z_1; \ldots \text{ if } c_k \text{ then } z_k;$$

then the following should be written:

$$\begin{array}{l}\text{if } e \in \tilde{X}^{}_{\substack{c_1 \\ T}\, f_a} \text{then delete } {}^{z_2}_{T} f_a(e), \ldots, {}^{z_k}_{T} f_a(e)\\ \text{if } e \in \tilde{X}^{}_{\substack{c_2 \\ T}\, f_a} \text{ then delete } {}^{z_1}_{T} f_a(e), {}^{z_3}_{T} f_a(e) \ldots, {}^{z_k}_{T} f_a(e)\\ \qquad\qquad \vdots \qquad\qquad \vdots\\ \text{if } e \in \tilde{X}^{}_{\substack{c_k \\ T}\, f_a} \text{ then delete } {}^{z_1}_{T} f_a(e), {}^{z_1}_{T} f_a(e) \ldots, {}^{z_{k-1}}_{T} f_a(e)\end{array}$$

Here, the evaluation ${}^{c_i}_{T}f_a$ is independent from the evaluation ${}^{z_1}_{T}f_a, \ldots, {}^{z_k}_{T}f_a$. Thus, first ${}^{c_i}_{T}f_a$-s should be evaluated and then only the remaining ${}^{z_1}_{T}f_a$ should be called on e ($1 \leq i \leq k$).

Theorem 3 *Keeping the above notation and suppositions,* P_T *is equivalent with* f_T.

Proof. Let t be a document tree. Suppose that T is called on edges $\hat{e}_1, \ldots, \hat{e}_r$, for $\hat{e}_i$ the s^{th} condition is satisfied, and the (r, σ_s)-rule is called on edges $e_1, \ldots, e_k$. Remember that this rule constructs a single ψ edge. Hence as a result of $\hat{e}_i$ P constructs a node with k outgoing ψ edges. Denote ψ_{e_i} the ψ edge constructed on e_i ($1 \leq i \leq k$). From Theorem 2 we know that when ${}_{T}f_{z_s}$ starts on $\hat{e}_i$ and it stops on $e_1, \ldots, e_k$. Furthermore, for each e_i it constructs a ψ edge ($1 \leq i \leq k$). Hence, as a result $\hat{e}_i$ ${}_{T}f_{q_s}$ constructs a node with k outgoing ψ edges. Additionally, in both cases ψ_{e_i} precedes ψ_{e_j}, if e_i precedes e_j in document order. With this, the construction of the corresponding mapping ϕ (Definition 6) is trivial. ■

Rewriting of programs with several rules

Programs with a given variable assignment. Now, we consider an $XSLT_0$ program P with rules $T_1, \ldots, T_m$. Again, we assume that there is a given variable assignment ρ. We do not assume that z_i-s are single at-expressions, but for the sake of transparency, we suppose that they contain only one at-expression as a leaf label. This leaf will be referred as *at-leaf*. This means that in its i^{th} condition T_j may call at most one T_k ($1 \leq j, k \leq m$). The $XPath_0$ expression of this at-expression is still denoted $xp_j(z_i)$ or $xp(z_i)$, if j is clear from the context. We also assume that the edge tests of aim nodes of $xp(z_i)$-s are all different from $*$-s (except for $(st, /)$-rule). From this, it follows that in all cases T_j in its i^{th} condition calls the same T_k.

To delineate the relationships among the templates, we define an auxiliary graph, the *precedence graph (of rules of* P*)*. Its nodes are labelled with T_i-s. There is an edge from T_i to T_j labelled with k, if the k^{th} condition of T_i calls T_j ($1 \leq i, j \leq m$).

Example 11 *As an example of simulation, consider the following program* $P = (T_1, T_2, T_3)$ *and its rewriting. In* P T_2 *is called on* b *descendants of the document edge. It constructs a* b *edge and calls* T_3 *on* c *parents.* T_3 *constructs a tree* $\{a : \{b : \{\}\}@\{c : \{\}\}\}$ *and stops.*

$\mathtt{template}\ \mathsf{st}(/, \epsilon) \qquad (\mathsf{T}_1)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{result} : \{\}\}$ *at-expr:* $\mathfrak{m}(\mathtt{desc::b}, \epsilon, \epsilon)$;
$\mathtt{end}.$

$\mathtt{template}\ \mathfrak{m}(\mathsf{b}, \epsilon) \qquad (\mathsf{T}_2)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{b} : \{\}\}$ *at-expr:* $\mathfrak{m}(\mathtt{par::c}, \epsilon, \epsilon)$;
$\mathtt{end}.$
$\mathtt{template}\ \mathfrak{m}(\mathsf{c}, \epsilon) \qquad (\mathsf{T}_3)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{a} : \{\mathsf{b} : \{\}\}@\{\mathsf{c} : \{\}\}\}$
$\mathtt{end}.$

$$
\begin{array}{lll}
{}_{\mathsf{T}_1}\mathsf{f}_{/}: & (\{/ : \mathsf{t}\}) & = \{\mathsf{result} : {}^{z_1}_{\mathsf{T}_1}\ \mathsf{f}_{\mathsf{b}}(\mathsf{t})\} \\
 & (\{* : \mathsf{t}\}) & = \{\} \\
 & & \\
{}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}: & (\{\mathsf{b} : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}(\mathsf{t}) \\
 & & \\
{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}: & (\{\mathsf{c} : \mathsf{t}\}) & = \textit{if } \textit{n.i.}({}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}(\mathsf{t})) \textit{ then } \{\mathsf{b} : \{\}\}@{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & & \\
{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}: & (\{\mathsf{b} : \mathsf{t}\}) & = \{\psi : \{\}\} \\
 & (\{* : \mathsf{t}\}) & = \{\}, \qquad \mathsf{X}^{\mathsf{b}}_{{}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}} = \mathsf{X}^{\mathsf{b}}_{{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}} \\
 & & \\
{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}: & (\{\mathsf{c} : \mathsf{t}\}) & = \{\mathsf{a} : \{\mathsf{b} : \{\}\}@\{\mathsf{c} : \{\}\}\}@{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}(\mathsf{t}), \qquad \mathsf{X}^{\mathsf{c}}_{{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}} = \mathsf{X}^{\mathsf{c}}_{{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}}
\end{array}
$$

The rewriting works similarly as in the previous subsection. The main difference is that here, instead of a star of ψ edges an arbitrary document tree is constructed. In order to properly simulate this construction, first, we have to note that constructions should be accomplished, when the matching node is reached. In other words this means that in most cases the construction should take place in the μ_i row of ${}^{z_j}_{\mathsf{T}_i}\mathsf{f}_{\mu_i}$, where μ_i denotes the matching node of template T_i ($1 \leq i \leq m$) (here, we have also assumed that the j^{th} condition

is satisfied first). This is the case in T_1 and T_3 in our example, where the matching nodes are b and c.

It may happen, however, that ${}^{z_j}_{T_i}f_{\mu_i}$ is called in the check of a not-isempty condition, thus its construction does not appear in the result. This is the case in T_2. It is easy to see now that in this case the first axis in $xp(z_j)$ is either par, or anc. (Remember that the j^{th} condition, *if* c_j *then* z_j, is considered as $self::\mu_i[c_j]/xp(z_j)$.) Denote u_1 the upper node of the first XPgraph of $self::\mu_i/xp(z_j)$. (This means that there is a directed path $w_1 \dots w_s$ in this XPgraph, where $w_1 = u_1$, $w_s = \mu_i$ and the edges are not labelled or they are labelled with par.) Construction should take place in the u_1 row of ${}^{z_j}_{T_i}f_{u_1}$. More accurately, in the then-branch of ${}^{z_j}_{T_i}f_{u_1}$.

Now, suppose that the construction takes place in the φ row of ${}^{z_j}_{T_i}f_{\varphi}$ and let ${}^{z_j}_{T_i}f_{\sigma}(t)$ be the structural function to be called there. Then (in the then-branch) we construct z_j changing its at-expression label to ${}^{z_j}_{T_i}f_{\sigma}(t)$. In our example in T_1, this new forest is $\{result :^{z_1}_{T_1} f_b(t)\}$. (In T_3 z_1 does not have any at-expression leaf label, hence it should be used without changes.)

Furthermore, in f_P we have to connect the structural recursions of (m, σ)-rules using the precedence graph. For edge (T_i, s, T_j) we add restriction, $X^{\alpha^i_s}_{f_{\alpha^i_s}} = X^{\mu_j}_{f_{\mu_j}}$ $(1 \le i, j \le m)$, here α^i_s denotes the aim node of $xp_i(z_s)$.

Note that f_P does not specify how the basic forests constructed by structural recursions of templates should be connected. Consequently, it does not guarantee any order among the basic forests. In what follows, we show how the output should be constructed and how the order of the basic forests should be defined. Meanwhile, we also establish the equivalence of P and f_P.

Instantiation of an (m, σ)-rule by another. As a first step, recall the semantics of $XSLT_0$. Remember that if (e, m, ρ) is a local configuration, then $e \in E.t$, m is a mode, ρ is a given variable assignment, and it shows that (m, σ)-rule T is to be applied on e (here $lab(e) = \sigma$ and the parameters of T are in the domain of ρ). Now, let T_j, T_k be (m_j, σ_j)-, (m_k, σ_k)-rules and $e^1, e^2 \in E.t$. We say that (T_j, e^1) *instantiates* (T_k, e^2) *in its* s^{th} *condition*, if there is a $\xi \in \mathcal{T}^{\Delta}(LC^*(t))$ ($LC^*(t)$ denotes sequences of local configurations) s.t. in a former step, location configuration (e^1, m_j, ρ) was substituted with the result of T_j called on e^1, fo, where the s^{th} condition of T_j was satisfied, and as a result we get ξ. Furthermore, in fo there is a leaf label (e^2, m_k, ρ), which is to be substituted with the result of T_k called on e^2 $(1 \le j, k \le m)$. Here, $lab(e^1) = \sigma_j$ and $lab(e^2) = \sigma_k$.

Belonging to the same calling of f_{T_j}. To catch this notion with struc-

tural functions we introduce two notions. Suppose that $G_{f_P,t}$ has already been constructed. For a moment, however, suppose that we rewrite the premises that have already been deleted. For edges $e^1, e^2 \in E.t$, $e^1 \in R_{reg}(\mu_j, f_{\mu_j})$, $e^2 \in R_{reg}(\alpha_s^j, f\alpha_s^j)$, we are to define and check when e^1, e^2 belong to the same call of f_{T_j}. Here, remember that $e^1 \in R_{reg}(\mu_j, f_{\mu_j})$, $e^2 \in R_{reg}(\alpha_s^j, f\alpha_s^j)$ mean that structural functions respectively representing the matching node of T_j and the aim node of its s^{th} condition are called successfully on e^1 and e^2. Intuitively, e^1 and e^2 belong to the same of f_{T_j}, if there is a path from e^1 to e^2 specified by $xp_j(z_s)$ in $G_{f_P,t}$ with the rewritten premises, which shows that T_j was instantiated on e^1, and e^2 was selected for further processing.

Formally, consider the XPgraph of the skeleton of the s^{th} condition of T_j. Suppose that $u_1, \ldots, u_r$ are the upper nodes. Suppose also that each XP_i^u has two leaves v_{2i-1}, v_{2i} $(1 \leq i \leq r)$ (Fig. 10(d)). Denote $e_{v_i} \in E.U_{f_P}$ the edge corresponding to the then-branch of the v_i row of f_{v_i} $(1 \leq i \leq 2r)$. e^1 and e^2 *belong to the same call of* f_{T_j}, in notation $(f_{T_i}(e^1, e^2))$, if there exist edges in E.t $e_0, \ldots, e_r$ s.t. $e_0 = e^1$, $e_r = e^2$, (i) $e_i \in R_{reg}(v_{2i}, f_{2i})$, and $e_i \in R_{reg}(v_{2i+1}, f_{2i+1})$ $(1 \leq i \leq r-1)$. (ii) for $(e_{v_{2j+1}}, e_j), (e_{v_{2j+2}}, e_{j+1}) \in E.U_{f_P} \sqcap t$, $(e_{v_{2j+2}}, e_{j+1})$ is reachable from $(e_{v_{2j+1}}, e_j)$ through a path containing exactly one neighbouring premise and then-edge pairs (the ancestor images of these conditional edges correspond to the u_{2j+1} row of $f_{u_{2j+1}}$) $(0 \leq j \leq r)$. (Consider Fig. 10(d) again.) Here, (i) means that e_i satisfies the corresponding register restriction $X^{v_{2i}}_{f_{v_{2i}}} = X^{v_{2i+1}}_{f_{v_{2i+1}}}$. On the other hand, condition (ii) says that e_j, e_{j+1} "correspond" to the two leaves of XP^{u_j+1}.

Note that, an algorithm that takes T_i and edges $e^1 \in R(\mu_i, f_{\mu_i})$ as input and finds all those edges e^2 to which $(f_{T_i}(e^1, e^2))$ holds, uses only the edges of paths from $(e_{v_{2j+1}}, e_j)$ to $(e_{v_{2j+2}}, e_{j+1})$ of condition (ii). Thus, it is possible to develop such an algorithm working in $O(|t||f|)$ time.

Instantiation of structural functions. With $Inst(f_{T_i}, e^1, s, f_{T_j}, e^2)$ we denote that $e^1 \in R_{reg}(\mu_i, f_{\mu_i})$, $e^2 \in R(\alpha_s^i, f\alpha_s^i)$, $e^2 \in R_{reg}(\mu_j, f_{\mu_j})$, and $(f_{T_i}(e^1, e^2))$. Clearly, with this definition we are to simulate the instantiation of T_j by T_i in the s^{th} condition. Note that, here e^2 also satisfies restriction $X^{\alpha_s^i}_{f_{\alpha_s^i}} = X^{\mu_j}_{f_{\mu_j}}$.

Note also that, when for e^1 we have found an edge e^2 s.t. $f_{T_i}(e^1, e^2)$, then the appropriate T_j, to which $Inst(f_{T_i}, e^1, s, f_{T_j}, e^2)$ holds, can be found using the precedence graph.

In what follows, the template to be called on the document edge is denoted T_{st}.

Definition 8 *Keeping the above notations, we say that* f_{T_r} is called on e via the document edge (in the k^{th} step)*, if there exists a sequence of edges of* E.t $e_1, \ldots, e_k$*, and* $f_{T_{i_1}}, \ldots, f_{T_{i_k}}$*, s.t.* $f_{T_{i_1}}$*,* $f_{T_{i_k}}$ *are respectively the same as* $f_{T_{st}}, f_{T_r}$*, and* $\mathtt{Inst}(f_{T_{i_j}}, e^j, s_j, f_{T_{i_{j+1}}}, e^{j+1})$ $(1 \leq j \leq k-1)$.

Note that register restrictions connecting matching nodes and aim nodes of different templates guarantee that if a subgraph of the input is processed by f_{T_i}, then the result is only included in the output, if f_{T_i} is called via the document edge.

Lemma 5 *For given XSLT$_0$ program* $P = (T_1, \ldots, T_m)$*, document tree* t*,* $e^1, e^2 \in$ E.t*,* (T_i, e^1) *instantiates* (T_j, e^2) *in its* s^{th} *condition iff* f_{T_i} *is called on* e^1 *via the document edge, and* $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$.

Proof. We use induction on the number of steps k in which f_{T_i} has been called on e^1 via the document edge. First suppose that $k = 0$, i.e., f_{T_i} is $f_{T_{st}}$ and e^1 is the document edge.

$\Rightarrow$: Clearly, in this case f_{T_i} is called on e^1 via the document edge. Suppose now that T_i, T_j are (m_i, σ_i)-, (m_j, σ_j)-rules. Then (T_i, e^1) instantiates (T_j, e^2) in its s^{th} condition, if there is a $\xi \in \mathcal{T}^{\Delta}(LC^*(t))$ s.t. in a former step, location configuration (e^1, m_i, ρ) was substituted with the result, fo, of the s^{th} condition of T_i called on e^1. Furthermore, in fo there is a leaf label (e^2, m_j, ρ), which is to be substituted with the result of T_j called on e^2 $(1 \leq j, k \leq m)$.

The fact that T_i has been called on e^1 means that $e^1 \in R(\mu_i, f_{\mu_i})$. Note that, since this instantiation of T_i does not depend on any other instantiation of T_j-s, here e^1 should not satisfy any rule register restrictions $(1 \leq j \leq m)$. Hence $e^1 \in R_{reg}(\mu_i, f_{\mu_i})$. Since the s^{th} condition is of the form:

$$q_s = \texttt{self::}\sigma\texttt{[}c_s\texttt{]/}xp_i(z_s),$$

the fact that the s^{th} condition of T_i has been satisfied guarantees that $e^2 \in q_s(e^1)$ (Theorem 2), which means that $e^2 \in R(\alpha_s^i, f_{\alpha_s^i})$. Since T_j is called on e^2, we know that $e^2 \in R(\mu_j, f_{\mu_j})$. It is also obvious that $(f_{T_i}(e^1, e^2))$. Consequently $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$. Furthermore, it has also turned out that f_{τ_j} is called on e^2 via the root.

The general step of this direction is similar.

$\Leftarrow$: The proof is similar to the proof of the other direction. In this case, we only have to change the "direction" of the reasoning. ■

Connection of the basic forests. Finally, we should connect the basic forests in an order corresponding to the order given by the simulated XSLT$_0$

program. Here, the fact that $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$ holds means that a basic forest fo belonging to the s^{th} condition of T_i is constructed. Label its root and at-leaf with $(f_{T_i}, e^1), (f_{T_j}, e^2)$ respectively. (Remember that the at-leaf is the leaf previously labelled by an at-expression.) The result forest is denoted $fo(f_{T_i}, e^1)$. Afterwards, we connect the basic forests with ε edges. Namely, we add ε edges from leaf labels of (f_{T_j}, e^2) to root labels (f_{T_j}, e^2). At the end these ε edges should be eliminated. Note that, with the connection of the at-leaf (f_{T_j}, e^2) of $fo(f_{T_i}, e^1)$ and the root of $fo(f_{T_j}, e^2)$, we simulate that moment, when in the instantiation of T_j by T_i, the leaf with location configuration label (e^2, m_j, ρ) is substituted with the result of T_j called on e^2.

Order of the result. In order to define an order in the result, we first give an order among the basic forests. We construct an auxiliary graph, $\mathtt{Inst}_{P,t}$, whose nodes are labelled with (f_{T_i}, e)-s. We add an edge from (f_{T_i}, e^1) to (f_{T_j}, e^2) with label s, if $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$.

Now, $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, (i) if (f_{T_i}, e^1) and (f_{T_j}, e^2) have a common parent (f_{T_s}, e) in $\mathtt{Inst}_{P,t}$, and e^1 precedes e^2 in the document order $(1 \leq i, j \leq m)$. Clearly, this case represents that, when T_s has been instantiated on e and both e^1 and e^2 have been chosen for further processing. Thus e^1 and e^2 have been selected by the same XPath_0 expression and in the same mode, consequently $i = j$ holds.

(ii) Denote (f_{T_s}, e) the first common ancestor of (f_{T_i}, e^1) and (f_{T_j}, e^2) in $\mathtt{Inst}_{P,t}$. Suppose that (f_{T_k}, e^3), (f_{T_k}, e^4) are children of (f_{T_s}, e), and (f_{T_i}, e^1), (f_{T_j}, e^2) are reachable through (f_{T_k}, e^3), (f_{T_k}, e^4) respectively. Then $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, if $fo(f_{T_k}, e^3)$ precedes $fo(f_{T_k}, e^4)$.

Now, if $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, then the edges of $fo(f_{T_i}, e^1)$ precedes the edges of $fo(f_{T_j}, e^2)$. The order of the edges in $fo(f_{T_i}, e^1)$ is given by the corresponding ssd-expression.

Theorem 4 *Let P an XSLT$_0$ program without variables, then f_P is equivalent with P.*

Proof. Let t be a document tree. Clearly, the construction of $\tau_P(t)$ can be described as a sequence of (m, σ)-rules $T_{i_1}, \ldots, T_{i_k}$ s.t. in the j^{th} step T_{i_j} instantiates $T_{i_{j+1}}$ $(1 \leq j \leq k-1)$. Here, T_{i_1} is the $(st, /)$-rule, and there may exist several such sequences for the same construction of $\tau_P(t)$, but they are all of the same length.

Now, we prove the theorem using induction on this length. If $k = 2$, the statement follows from Lemma 5.

Next, suppose that the statement holds for all document trees, where the

aforementioned sequences have length $\leq k$. Let t be such a document tree, where this length is $k+1$. Suppose that, in the last step $T_{i_{k+1}}$ is called on location configuration (e, m, ρ). Trivially, if the corresponding leaf did not have any location configuration label, i.e., the construction stopped before the call of $T_{i_{k+1}}$, then, according to our assumptions, the appropriate mapping ϕ^1 (Definition 6) would exist between $V.\tau_P(t)$ and $V.f_P(t)$. We also know that there is an edge e' and number s s.t. $Inst(f_{T_{i_k}}, e', s, f_{T_{i_{k+1}}}, e)$. Furthermore, using again Lemma 5, the result, t_T, of $T_{i_{k+1}}$ called on e is equivalent with the result, t_f, of $f_{T_{i_{k+1}}}$ called also on e ($e \in R(\tau_m^{i_{k+1}}, f_{\tau_m^{i_{k+1}}})$). Hence, the appropriate mapping ϕ^2 can be given between $V.t_T$ and $V.t_f$.

Now, we only have to extend ϕ^1 with ϕ^2. Denote ϕ this new mapping. Obviously, condition (i)-(ii) of Definition 6 holds for ϕ. The truthfulness of condition (iii) can be proven easily with the use of the rules of defining an order among the connected basic forests. ■

Infinite loops. It is not difficult to see that how our method avoids infinite loops. Consider the following program P and its rewriting f_P:

$template\ st(/, \epsilon)$ (T_1)
  return
    if $c_1 = true$ then $\{c : \{\}\}$; at-expr: $m(\texttt{child::*}, \epsilon, \epsilon)$
end.

$template\ m(a, \epsilon)$ (T_2)
  return
    if $c_1 = true$ then $\{a : \{\}\}$; at-expr: $m(\texttt{desc::b}, \epsilon, \epsilon)$
end.

$template\ m(b, \epsilon)$ (T_3)
  return
    if $c_1 = true$ then $\{b : \{\}\}$; at-expr: $m(\texttt{anc::a}, \epsilon, \epsilon)$
end.

${}^{z1}_{T_2}f_a$: $(\{a : t\}) = \{a :^{z1}_{T_2} f_b(t)\}@^{z1}_{T_2}f_a(t)$ ${}^{z1}_{T_2}f_b$: $(\{b : t\}) = {}^{z1}_{T_2}f_b(t)$
$(\{* : t\}) = {}^{z1}_{T_2}f_a(t)$ $(\{* : t\}) = {}^{z1}_{T_2}f_b(t)$

${}^{z1}_{T_3}f_a$: $(\{a : t\}) =$ if n.i.$({}^{z1}_{T_3}f_b(t))$ then $\{b : \{\}\}@^{z1}_{T_3}f_a$
else ${}^{z1}_{T_3}f_a(t)$
$(\{* : t\}) = {}^{z1}_{T_3}f_a(t)$

$\overset{z1}{T_3} f_b$: $(\{b : t\}) = \{\psi : \{\}\}$
$(\{* : t\}) = \overset{z1}{T_3} f_b(t)$

$T_2 f_s$: $(\{/ : t\}) =$ if true then $\overset{z1}{T_2} f_a(t)$ $\quad$ $T_3 f_s$: $(\{/ : t\}) =$ if true then $\overset{z1}{T_3} f_a(t)$

$P f_s$: $(\{/ : t\}) = \{/ : (T_2 f_s(\{/ : t\}), T_3 f_s(\{/ : t\}))\}$,

$X^a_{\overset{z1}{T_2} f_a} = X^a_{\overset{z1}{T_3} f_a}$, $X^b_{\overset{z1}{T_2} f_b} = X^b_{\overset{z1}{T_3} f_b}$

Clearly, for document trees t with root edge a having a b descendant, P enters into an infinite loop. On the other hand, as f_P traverses t top-down, every edge is "considered" at most once. If an a edge has a b descendant, then $\overset{z1}{T_2} f_a$ constructs an a edge, to which a b edge is connected constructed by $\overset{z1}{T_3} f_a$.

How variable assignment can be obtained. Till now, we have always assumed the existence of a given variable assignment. In what follows, we are to show, how this assignment, ρ, can be given. Remember that variable definitions are only allowed to appear in the $(st, /)$-rule. Consider now variable definition $x = r$. Here r is an $XPath_0$ expression. Denote f^{ro}_x the root structural function of the structural recursion f_x representing r. Then, for an $XSLT_0$ program P with templates $T_1, \ldots, T_m$, variables $x_1, \ldots, x_n$, the $\{/ : t\}$ row of $P f_s$ should extended with these root structural functions.

$$f_s : (\{/ : t\}) = f^{ro}_{x_1}, \ldots, f^{ro}_{x_n}, \ldots, \{/ : (T_1 f_s(\{/ : t\}), \ldots, T_1 f_s(\{/ : t\}))\}$$

First, for a given document tree t, the results of f_{x_1} should be constructed. The edge-set on which f_{x_1} stops gives $\rho(x_1)$. Next, $\rho(x_2)$ is calculated. Here, we may have to use the result of $\rho(x_1)$. After the construction of $\rho(x_n)$ we get ρ, and the construction of $f_P(t)$ should be continued with this variable assignment. As an example, we give the rewriting of Example 1.

$$\begin{array}{lll} T_{st} f_/ : & (\{/ : t\}) & = \overset{x}{T_{st}} f_{gr}(t), \{\text{result} : \overset{z1}{T_{st}} f_{gr}(t)\} \\ & (\{* : t\}) & = \{\} \end{array}$$

$$\begin{array}{lll} \overset{z1}{T_{st}} f_{gr} : & (\{\text{group} : t\}) & = \overset{z1}{T_{st}} f_{gr}(t) \\ & (\{* : t\}) & = \overset{z1}{T_{st}} f_{gr}(t) \end{array}$$

$$
\begin{array}{lll}
{}^{x}_{T_{st}}f_{gr}: & (\{group : t\}) & = {}^{x}_{T_{st}}f_{id}(t) \quad {}^{x}_{T_{st}}f_{id}(t) : (\{id : t\}) = {}^{x}_{T_{st}}f_{id}(t) \\
 & (\{* : t\}) & = {}^{x}_{T_{st}}f_{gr}(t) \qquad\qquad (\{* : t\}) = {}^{x}_{T_{st}}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{x}_{T_{st}}f^{pr}_{gr}: & (\{group : t\}) & = \text{if } n.i.({}^{x}_{T_{st}}f^{pr}_{emp}(t)) \text{ then } {}^{x}f^{pr}_{gr}(t) \\
 & & \qquad\qquad \text{else } {}^{x}_{T_{st}}f^{pr}_{gr}(t) \\
 & (\{* : t\}) & = {}^{x}_{T_{st}}f^{pr}_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{x}_{T_{st}}f^{pr}_{emp}: & (\{emp : t\}) & = {}^{x}_{T_{st}}f^{pr}_{na}(t) \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{x}_{T_{st}}f^{pr}_{na}: & (\{name : t\}) & = \text{if } val(name) = Ann \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}, \quad X^{gr}_{{}^{x}_{T_{st}}f_{gr}} = X^{gr}_{{}^{x}_{T_{st}}f^{pr}_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{gr}: & (\{group : t\}) & = \text{if } n.i.({}^{c_1}_{T_1}f_{to}(t)) \text{ then } {}^{c_1}_{T_1}f_{gr}(t) \\
 & & \qquad\qquad \text{else } {}^{c_1}_{T_1}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{c_1}_{T_1}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{to}: & (\{topMgr : t\}) & = {}^{c_1}_{T_1}f_{na}(t) \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{na}: & (\{name : t\}) & = \text{if } val(name) = John \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_1}f_{gr}: & (\{group : t\}) & = \{topGroup : \{id : {}^{z_1}_{T_1}f^{val}_{id}(t)\}\}@{}^{z_1}_{T_1}f^{Ann}_{gr}(t)@{}^{z_1}_{T_1}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{z_1}_{T_1}f_{gr}(t), \quad X^{gr}_{{}^{c_1}_{T_1}f_{gr}} = X^{gr}_{{}^{z_1}_{T_1}f_{gr}}, X^{gr}_{{}^{z_1}_{T_{st}}f_{gr}} = X^{gr}_{{}^{z_1}_{T_1}f_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_1}f^{val}_{id}: & (\{id : t\}) & = \{\} \quad {}^{z_1}_{T_1}f^{Ann}_{gr} : (\{group : t\}) = {}^{z_1}_{T_1}f^{Ann}_{gr}(t) \\
 & (\{* : t\}) & = \{\} \qquad\qquad (\{* : t\}) \quad = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_2}f_{gr}: & (\{group : t\}) & = \text{if } n.i.({}^{c_1}_{T_2}f_{id}(t)) \text{ then } {}^{c_1}_{T_2}f_{gr}(t) \\
 & & \qquad\qquad \text{else } {}^{c_1}_{T_2}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{c_1}_{T_2}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_2}f_{id}: & (\{id : t\}) & = \text{if } val(id) = x \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_2}f_{gr}: & (\{group:t\}) & = \{id :^{z_1}_{T_2} f^{val}_{id}(t)\} \\
 & (\{*:t\}) & = {}^{z_1}_{T_1}f_{gr}(t), \quad X^{gr}_{{}^{z_1}_{T_2}f_{gr}} = X^{gr}_{{}^{z_1}_{T_2}f_{gr}}, X^{gr}_{{}^{z_1}_{T_{st}}f_{gr}} = X^{gr}_{{}^{z_1}_{T_2}f_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_2}f^{val}_{id}: & (\{id:t\}) = \{\} & {}^{z_1}_{T_3}f_{id}:(\{id:t\}) = \{val(id):\{\}\}@{}^{z_1}_{T_3}f_{id}(t) \\
 & (\{*:t\}) = \{\} & (\{*:t\}) = {}^{z_1}_{T_3}f_{id}
\end{array}
$$

6 Conclusions

In this paper we have introduced a new version of structural recursions, where we have added registers to be able to connect the results of structural functions called on the same XML document. To underpin the usefulness of this extension, we have showed how a practically important fragment of XPath and XSLT can be implemented with these structural recursions. As it has turned out, our technique has the same efficiency as the fastest implementation algorithm [8] known by the authors of this paper.

In the near future we shall work out how schema information given in the form of extended DTD-s [12] can be incorporated into our model. Looking for further optimization possibilities, we also plan to implement our techniques in a software and compare its speed with the existing XPath and XSLT implementations.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, 2000.

[2] A. Benczúr, B. Kósa, Static Analysis of Structural Recursion in Semistructured Databases and Its Consequences. *Advances in Databases and Information Systems, 8th East European Conference Proceedings*, 2004, pp. 189–203.

[3] A. Benczúr, B. Kósa, Satisfiability and Containment Problem of Structural Recursions with Conditions with Respect to XML. *To appear.*

[4] A. Benczúr, A. Kiss B. Kósa, Implementation of XPath Using Structural Recursions, *Advances in Databases and Information Systems, 13th East European Conference. To appear.*

[5] G. J. Bex, S. Maneth, F. Neven, A Formal Model for an Expressive Fragment of XSLT. 2000. *Manuscript.*

[6] V. Breazu-Tannen, P. S. Buneman, Structural Recursion as a Query Language, *Proceedings of the 3rd International Workshop on Database Programming Languages*, 1991, pp. 9–19.

[7] P. Buneman, M. Fernandez, D. Suciu, UnQL: a query language and algebra for semistructured data based on structured recursion. *The VLDB Journal*, (2000) 76–110.

[8] G. Gottlob, C. Koch, R. Pichler, Efficient Algorithms for Processing XPath Queries, *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[9] G. Gottlob, C. Koch, R. Pichler, XPath Processing in a Nutshell. *ACM SIGMOD Record*, **32**, 2 (2003) 21–27.

[10] G. Gottlob, C. Koch, R. Pichler, XPath Query Evaluation: Improving Time and Space Efficiency, *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003.

[11] W. Martens, F. Neven: On the complexity of typechecking top-down XML transformations, *Theoret. Comput. Science*, **336**, 1 (2005) 153–180.

[12] W. Martens, F. Neven, T. Schwentick, G. J. Bex: Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, **31**, 3 (2006) 770–813.

[13] T. Milo, D. Suciu, V. Vianu. Type checking for XML transformers, *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, 2000, pp. 11–22.

[14] F. Neven. Automata, Logic and XML. *Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, 2002, pp. 2–26.

[15] Word Wide Web Consortium. Extensible Markup Language (XML) 1.1, 2004. `http://www.w3.org/TR/2004/REC-xml11-20040204/`

[16] Word Wide Web Consortium. XML Path Language (XPath) Version 1.0, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116/`

[17] Word Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116/

[18] Word Wide Web Consortium. XML Path Language (XPath) 2.0, 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/

[19] Word Wide Web Consortium. XSL Transformations (XSLT) Version 2.0, 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/

Received: April 26, 2009

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 215–233

Towards a framework for near-duplicate detection in document collections based on closed sets of attributes

Dmitry I. Ignatov
Higher School of Economics,
Department of Applied Mathematics
and Information Science
Moscow, Russia
email: dignatov@hse.ru

Katalin Tünde Jánosi-Rancz
Sapientia Hungarian University of
Transylvania, Cluj, Department of
Mathematics and Informatics
Tg. Mureş, Romania
email: tsuto@ms.sapientia.ro

Sergei O. Kuznetzov
Higher School of Economics, Department of
Applied Mathematics and Information Science,
Moscow, Russia
email: skuznetsov@hse.ru

Abstract. Around 30% of documents on the web have duplicates. Near-duplicate documents bear high similarity to each other, yet they are not bitwise identical. They are identical in terms of content but differ in a small portion of the document. Thus, algorithms for detecting these pages are needed. In the course of developing a near-duplicate detection system in this article we present an approach based on frequent closed sets of attributes for constructing clusters of duplicate documents, documents being represented by both syntactic and lexical methods. We provide a prototype of software environment for those who want to utilize such methods for finding near-duplicate documents in large text collections. This software includes two syntactic methods of finding near duplicate documents, a clustering technique based on frequent closed itemsets, means of evaluation of results and a tool for generating test collections of near-duplicate documents.

AMS 2000 subject classifications: 69P99, 62H30
CR Categories and Descriptors: H.3.3 [Information Search and Retrieval]: Search Process, Clustering
Key words and phrases: near-duplicate detection, content duplication, web document

1 Introduction

The Web makes it easy for words to be copied and spread from one page to another, and the same content may be found at more than one web site, regardless of whether its author intended it to be or not. Duplicate and near-duplicate web pages are creating large problems for web search engines: they increase the space needed to store the index, either slow down or increase the cost of serving results, and annoy the users. This requires the creation of efficient algorithms for computing clusters of duplicates [4, 6, 7, 10, 11, 16, 21, 22, 30, 29].

A naive solution is to compare all pairs to documents. The first algorithms for detecting near-duplicate documents with a reduced number of comparisons were proposed by Manber [25] and Heintze [17]. Both algorithms work on sequences of adjacent characters. Brin [3] started to use word sequences to detect copyright violations. Shivakumar and Garcia-Molina [31] continued this research and focused on scaling it up to multi-gigabyte databases. Broder [5] also used word sequences to efficiently find near-duplicate web pages. Charikar [9] developed an approach based on random projections of the words in a document. Hoad and Zobel [19] developed and compared methods for identifying versioned and plagiarised documents. Henzinger [18] tests and explores how some different existing methods (Broder's [5] and Charikar's [9]) for detecting near-duplicate content could be used together to try to identify near-duplicates on the Web. A good overview of approaches to detect exact duplicates and near-duplicates of web pages can be found in [15].

We define duplicates in terms of similarity. We say that two documents are duplicates, if a numerical measure of their similarity exceeds a given threshold [7]. This can be represented by a graph, where nodes correspond to documents and the edges of the graph represent the pairs of the similarity relation. From this similarity graph we can compute the clusters of similar documents by counting the number of connected components of the graph. The main steps in finding clusters of duplicates are: representing documents by sets of attributes, making solid document images and computing clusters of similar documents. First of all, we have to remove the HTML markup and punctuation marks of the web documents. After this, as the first step, we turn these documents into strings of words, which are represented by sets of attributes. We have two options of doing this: from a syntactical approach or from a lexical approach.

In the syntactical approach we define binary attributes that correspond to each fixed length substring of words (or characters). These substrings are

called shingles. We can say that a shingle is a sequence of words. A shingle has two parameters: the length and the offset. The length of the shingle is the number of the words in a shingle and the offset is the distance between the beginnings of the shingles. We assign a hash code to each shingle, so equal shingles have the same hash code and it is improbable that different shingles would have the same hash codes (this depends on the hashing algorithm we use). After this we randomly choose a subset of shingles for a concise image of the document [4, 6, 7]. An approach like this is used in AltaVista search engine [29]. There are several methods for selecting the shingles for the image: a fixed number of shingles, a logarithmic number of shingles, a linear number of shingle (every n^{th} shingle), etc. In lexical methods, representative words are chosen according to their significance. Usually these values are based on frequencies: those words whose frequencies are in an interval (except for stop-words from a special list of about 30 stop-words with articles, prepositions and pronouns) are taken: words with high frequency can be non informative and words with low frequencies can be misprints or occasional words.

In lexical methods, like I-Match [11], a large text corpus is used for generating the lexicon. The words that appear in the lexicon represent the document. When the lexicon is generated the words with the lowest and highest frequencies are deleted. I-Match generates a signature and a hash code of the document. If two documents get the same hash code it is likely that the similarity measures of these documents are equal as well. I-Match is sometimes instable to changes in texts [22]. In lexical method [21] the focus is towards the construction of a lexicon, a set of descriptive words, which should be concise, but cover well the collection. The occurrence of a word in a document image is robust with respect to small changes in the document. When we define document images, we define a similarity relation on documents starting from a similarity measure, which takes to two documents to a number into the [0,1] interval, depending on the amount of their common description units. Then we choose a threshold. If this threshold is exceeded, it means that there is a large similarity between the documents (the two documents are very close to being duplicates). The metrics and the threshold define similarity relation on document pairs. The similarity relation on document pairs determines clusters of near-duplicates. There are several possible definitions for a cluster, but one of them often used in practice is as follows: Consider a graph, in which nodes represent the Internet documents and edges correspond to similarity relations. Then a cluster of near-duplicates is a connected component of this graph. The advantage of this definition is in the efficiency of computation: a connected component of a graph can be computed in linear time in the number of edges.

A drawback of the definition is also obvious: the relation to be near-duplicates is not transitive, so absolutely different documents can occur in a cluster. The strongest definition of a cluster is based on a graph clique, but it is much harder computationally, because generation of maximal cliques is a classical problem. We can use an intermediate formulation, which is between these two extreme definitions, and this way make a trade-off between the precision and the complexity of the cluster computation.

In this paper we consider similarity as an operation taking two documents to the set of all common elements of their concise description. Description elements can be syntactical units (shingles) or lexical units (representative words). A cluster of similar documents is defined as a set of all documents with a certain set of common description units. A cluster of duplicates is defined as a set of documents, where the number of common description units exceeds a given threshold. In this article we compare results of its application with the list of duplicates obtained by applying other methods to the same collection of documents. We examined the impact of the following parameters on the result:

- The use of the syntactical or lexical methods for representing documents
- the use of method "$\mathfrak{n}$ *minimal elements in a permutation*" or "*minimal elements in* $\mathfrak{n}$ *permutations*" [4, 6, 7] (the second method, having better probability-theoretical properties, has worse computational complexity)
- shingling parameter
- threshold value of similarity of document images.

We used a definition based on formal concepts for a cluster: clusters of documents are given by formal concepts of the context where objects correspond to description units and attributes are document names. So a cluster of very similar documents corresponds to a formal concept so that the size of the extent exceeds the threshold given by a parameter. In this approach, the problem of generating very similar documents is reduced to the problem of data mining, known as generating frequent closed item sets.

There are many web services, such as web search engines, which use near-duplicate detection techniques. These techniques are also useful for plagiarism detection in R&D reports and scientific articles [20]. To the best of our knowledge, there is no freely available framework with implementation of basic methods of near-duplicate detection. We made a first attempt to develop such a system with taking into account researcher's needs. Potthast and Stein [28]

note that there are no public collections suited for the analysis and evaluation of near-duplicate detection algorithms. Then they propose to use the Wikipedia Revision Corpus for the task. Our solution of this problem is a tool for generating near-duplicate collections based on one's own corpus of texts. We give a detailed description of our system in section 3.

2 Computational model

2.1 Document image

We used standard syntactical and lexical approaches with different parameters, for creating document images. Within syntactical approach we realized the shingling scheme and computing document image (sketch) with the method "$\mathfrak{n}$ *minimal elements in a permutation*" and the method "*minimal elements in* $\mathfrak{n}$ *permutations*", a detailed description of which can be found in [4, 6, 7]. For each text the program **shingle** with two parameters (*length* and *offset*) generates contiguous subsequences of size *length* so that the distance between the beginnings of two subsequent substrings is *offset*. The set of sequences obtained in this way is hashed so that each sequence receives its own hash code. From the set of hash codes that corresponds to the document a fixed size (given by parameter) subset is chosen by means of random permutations described in [4, 6, 7]. The probability of the fact that minimal elements in permutations on hash code sets of shingles of documents A and B (these sets are denoted by F_A and F_B, respectively) coincide, equals to the similarity measure of these documents $\mathrm{sim}(\mathsf{A},\mathsf{B})$:

$$\mathrm{sim}(\mathsf{A},\mathsf{B}) = \mathsf{P}[\mathfrak{min}\{\pi(\mathsf{F}_\mathsf{A})\} = \min\{\pi(\mathsf{F}_\mathsf{B})\}] = \frac{|\mathsf{F}_\mathsf{A} \cap \mathsf{F}_\mathsf{B}|}{|\mathsf{F}_\mathsf{A} \cup \mathsf{F}_\mathsf{B}|}$$

Permutations (that can be represented by renumbering of shingles) are realized by multiplying binary vectors that represent document images (each component of such a vector corresponds to the hash code of a particular shingle from the image) on random binary matrices. For each hash code from the set of hash codes of a document its number in each random permutation is computed as a product of the hash code given in the form of binary vector on the randomly generated binary matrix that corresponds to the permutation. The number of permutations is also a parameter. For each permutation (given by a binary matrix) the minimal element (i.e., hash code of a shingle that became the first after the permutation) is chosen. The image of a document in the method "*n minimal elements in a permutation*" is the set of n minimal

(first) hash codes in a permutation. The image of a document in the method "*minimal elements in n permutations*" is the set consisting of minimal (first) hash codes in n independent permutations. In both methods the images of all documents have fixed length n. The second approach has better randomization properties (see [4, 6, 7] for details), although it needs more time for computations (n times more than in the first approach).

2.2 Definition of similarity and similarity clusters by means of frequent concepts

First, we briefly recall the main definitions of Formal Concept Analysis (FCA) [12]. Let G and M be sets, called the set of objects and the set of attributes, respectively. Let I be a relation $I \subseteq G \times M$ between objects and attributes: for $g \in G$, $m \in M$, gIm holds iff the object g has the attribute m. The triple $K = (G, M, I)$ is called a *(formal) context*. Formal contexts are naturally given by cross tables, where a cross for a pair (g, m) means that this pair belongs to the relation I. If $A \subseteq G$, $B \subseteq M$ are arbitrary subsets, then *derivation operators* are given as follows:

$$A' := \{m \in M \mid gIm \text{ for all } g \in A\},$$
$$B' := \{g \in G \mid gIm \text{ for all } m \in B\}.$$

The pair (A, B), where $A \subseteq G$, $B \subseteq M$, $A' = B$, and $B' = A$ is called a *(formal) concept (of the context* K*)* with *extent* A and *intent* B.

The operation $(\cdot)''$ is a closure operator, i.e., it is idempotent ($X''' = X''$), extensive ($X \subseteq X''$), and monotone ($X \subseteq Y \Rightarrow X'' \subseteq Y''$). Sets $A \subseteq G$, $B \subseteq M$ are called *closed* if $A'' = A$ and $B'' = B$. Obviously, extents and intents are closed sets. Formal concepts of context are ordered as follows: $(A_1, B_1) \leq (A_2, B_2)$ iff $A_1 \subseteq A_2 (\Leftrightarrow B_1 \supseteq B_2)$. With respect to this order the set of all formal concepts of the context K makes a lattice, called a *concept lattice* $\underline{\mathfrak{B}}(K)$ [12].

Now we recall some definitions related to association rules in data mining. For $B \subseteq M$ the value $|B'| = |\{g \in G \mid \forall m \in B(gIm)\}|$ is called *support* of B and denoted by $\sup(B)$. It is easily seen that set B is closed if and only if for any $D \supset B$ one has $\sup(D) < \sup(B)$. This property is used for the definition of a closed itemset in data mining. A set $B \in M$ is called k-*frequent* if $|B'| \leq k$ (i.e., the set of attributes B occurs in more than k objects), where k is parameter. Computing frequent closed sets of attributes (or itemsets) became important in data mining since these sets give the set of all association rules [27]. For our implementation where contexts are given by set G of description units (e.g.,

shingles), set M of documents and incidence (occurrence) relation I on them, we define a cluster of k-similar documents as intent B of a concept (A, B) where $|A| \geq k$. Although the set of all closed sets of attributes (intents) may be exponential with respect to the number of attributes, in practice contexts are *sparse* (i.e., the average number of attributes per object is fairly small). For such cases there are efficient algorithms for constructing all most frequent closed sets of attributes (see also survey [23] on algorithms for constructing all concepts). Recently, competitions in time efficiency for such algorithms were organized in a series of workshops on Frequent Itemset Mining Implementations (FIMI). By now, a leader in time efficiency is the algorithm FPmax* [14]. We used this algorithm in order to find similarities of documents and generate clusters of *very similar documents*. As mentioned before, objects are description units (shingles or words) and attributes are documents. For representations of this type *frequent closed itemsets* are closed sets of documents, for which the number of common description units in document images exceeds a given threshold. Actually, FPmax* generates *frequent itemsets* (which are not necessarily closed) and *maximal frequent itemsets*, i.e., frequent itemsets that are maximal by set inclusion. Obviously, maximal frequent sets of attributes are closed.

3 Program implementation

Software for experiments with syntactical representation comprise the units that perform the following operations:

1. XML Parser (provided by Yandex): it parses XML packed collections of web documents,
2. removing html-markup of the documents,
3. generating shingles with given parameters length-of-shingle, offset,
4. hashing shingles,
5. composition of document image by selecting subsets (of hash codes) of shingles by means of n *minimal elements in a permutation* and *minimal elements in* n *permutations* methods,
6. composition of the inverted table, the list of identifiers of documents shingle, thus preparing data to the format of programs for computing closed itemsets,

7. computation of clusters of k-*similar documents* with FPmax* algorithm: the output consists of strings, where the first elements are names (ids) of documents and the last element is the number of common shingles for these documents,

8. comparing results with the existing list of duplicates (in our experiments with the ROMIP collection of web documents, we were supplied by a precomputed list of duplicate pairs),

9. generation of test collections of near-duplicate documents.

Unit 8 (for evaluation of results) outputs five values: 1) the number of duplicate pairs in the ROMIP collection, 2) the number of duplicate pairs for our realization, 3) the number of unique duplicate pairs in the ROMIP collection, 4) the number of unique duplicate pairs in our results, 5) the number of common pairs for the ROMIP collection and our results. For the lexical method, the description units are words (not occurring in the stop list) the frequencies of which lie in a certain interval. The amount of words in the dictionary is controlled by placing closer the extreme points of the interval.

3.1 GUI

The application has a Graphical User Interface similar to a setup application.

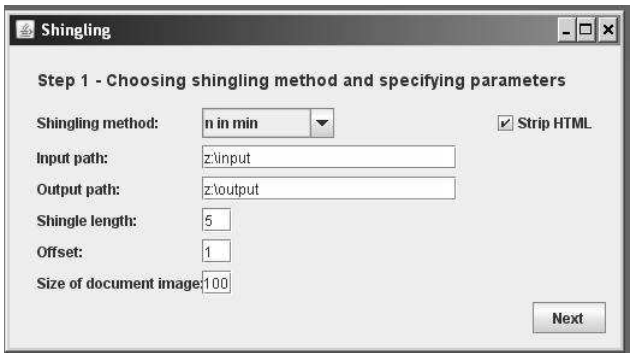


Figure 1: Input form for a duplicate search

In Fig. 1 the user specifies the shingling method and the following parameters: input path, output path, shingle length, offset, size of document image

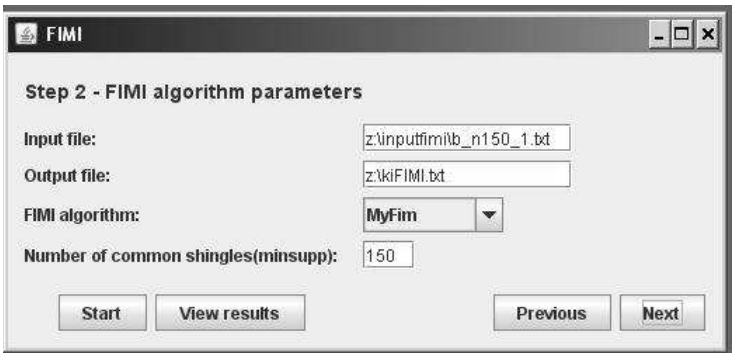


Figure 2: Input form for FIMI algorithm.

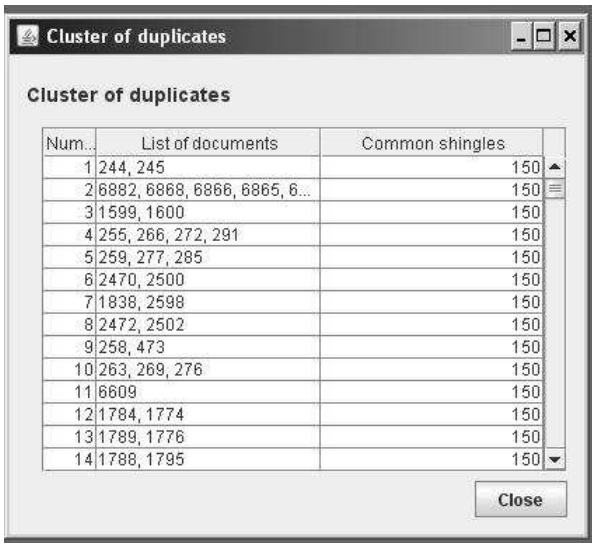


Figure 3: The output of the duplicates

and an option to strip the HTML or not. With the 'Next' button we can advance to the next form (Fig. 2).

On the next form the user has to choose the FIMI algorithm to be used and specify the parameters of the chosen algorithm: the input and output path of the algorithm and the number of common shingles. The 'Start' button starts the algorithm, the 'View results' button shows the results, see Fig. 3. With

the 'Previous' and 'Next' buttons we can either return to the previous form or advance to the next form. On the 'Clusters of duplicates' form we can see the results of the FIMI algorithm, shown in a grid view. On the last form the user can compare the results. The user has to specify the path of the list of duplicate pairs and the path of the log file. With the 'Start' button the user can start the comparison. With the 'Previous' button the user can return to the previous form. With the 'Close' button the application will exit. The screenshots of the GUI can be seen in the Fig. 1, 2, 3.

3.2 Tool for construction of near-duplicate test collection

The technique of generating test collection for near-duplicates proposed below uses the editing styles of near-duplicates: Block Edit (add or delete several paragraphs), Key Block (contains one or more well-known paragraphs), Minor Change (small editing changes), Block Reordering (reorder known paragraphs) and their combinations. Short description of these methods with indication of parameters is given in Table 1.

	Operation	Parameter(s)
1	Reordering of existing paragraphs	Percentage of reordering paragraphs
2	Deletion of existing paragraphs	Percentage of deleting paragraphs
3	Addition of existing paragraphs	Percentage of deleting paragraphs
4	Replacement of existing words	Percentage of replacing words
5	Addition of repeated paragraphs	Amount of paragraphs and number of paragraph repeats
6	replacement of characters	Set of character pairs: (initial character, new character)

Table 1: Operations for generation of test near-duplicate collection

We note that any of these operations use random number generator to construct the set of editable elements. For example, any reordering of paragraphs is random. This gives us more precise results than the use of manual edition of documents. By the way, stemming and finding synonyms of words require significant computational resources. We use simple replacement of exciting words chosen randomly from a dictionary; this makes no difference to find near-duplicate by non semantic methods. The user can apply a sequence of editing operations choosing an item from Table 1. During the loading process each document splits into paragraphs, each paragraph splits into sentences, and each sentence splits into words. As a result the user can see the statistics for each document (number of paragraphs, sentences and words). The tool

produces a log file in an output folder with names of input and generated files, number of sentences and words of input file, and parameters of the changes made. For example, for paragraph deletion the operation system saves the number of deleted words and characters. This information will be used to compare the results with those of the tested methods.

4 Experiments

As experimental data we used ROMIP collection of URLs (see www.romip.ru) consisting of 52 files of 4.04 GB general size. For experiments the collection was partitioned into several parts consisting of three to 24 files (from 5% to 50% percent of the whole collection). Shingling parameters used in experiments were as follows: the number of words in shingles was 10 and 20, the offset was always taken to be 1 (which means that the initial set of shingles contained all possible contiguous word sequences of a given length). Two methods of composing document image described in Section 2.1 were studied: $\mathtt{n}$ *minimal elements in a permutation* and *minimal elements in* $\mathtt{n}$ *permutations*.

The sizes of resulting document images were taken in the interval of 100 to 200 shingles. In case of the lexical representation described in Section 2.1, only words from the resulting dictionary were taken in the document image (the set of descriptive words). As thresholds defining *frequent closed sets* (i.e., the numbers of common shingles in document images from one cluster) we experimentally studied different values in intervals, where the maximal value is equal to the number of shingles in the document image, e.g., [85, 100] for document images with 100 shingles, the interval [135, 150] for document images of size 150, etc. Obviously, choosing the maximal value in the interval, we obtain clusters where document images coincide completely. For parameters taking values in these intervals we studied the relation between resulting clusters of duplicates and ROMIP collection of duplicates (computed by other methods). The ROMIP collection of duplicates consists of pairs of web documents that are considered to be duplicates. For each such pair we sought an intent, which contains both elements of the pair, and vice versa, for each cluster of *very similar documents* (i.e., for each corresponding closed set of documents with more than $\mathtt{k}$ common description units) we took each pair of documents in the cluster and looked for the corresponding pair in the ROMIP collection. The output of this unit is the table with the number of common number of duplicate pairs found by our method (denoted by HSE) and those in the ROMIP collection, and the number of unique pairs of HSE duplicates (document pairs

occurring in a cluster of "very similar documents" and not occurring in the ROMIP collection). The results of our experiments showed that the ROMIP collection of duplicates, considered to be a bench-mark, is far from being perfect. First, we detected that there is a large number of false duplicate pairs in this list due to similar framing of documents. For example the pages with the following information about historical personalities Garibald II, Duke of Bavaria and Giovanni, Duke of Milan were declared to be duplicates.

However these pages, as well as many other analogous false duplicate pairs in ROMIP collection do not belong to concept-based (maximal frequent) clusters generated in our approach.

In our study we also looked for *false duplicate clusters* in the ROMIP collection, caused by transitive closure of the binary relation "X is a duplicate of Y" (as in the typical definition of a document cluster in [7]). Since the similarity relation is generally not transitive, the clusters formed by transitive closure of the relation may contain absolutely nonsimilar documents. Note that if clusters are defined via maximal frequent itemsets there cannot be effects like this, because documents in these clusters share necessarily large itemsets.

4.1 Performance of algorithms and their comparison

We measured the elapsed time on the shingling stage, composing document images and generating clusters of similar documents (by algorithms for computing frequent closed itemsets). In the last stage we used and compared various algorithms: several well-known algorithms from data mining [13] and AddIntent, an algorithm which proved to be one of the most efficient algorithms for constructing the set of all formal concept and concept lattices [26]

Experiments were carried out on a PC P-IV HT with 3.0 MHz frequency, 1024 MB RAM under Windows XP Professional. Experimental results and the elapsed time are partially represented in Tables 2, 3, and 4.

In our experiments the best performance is attained by Fpmax* algorithm, followed by the AFOPT algorithm [24]. These two algorithms proved to be the fastest in FIMI competitions [13]. AddIntent* (AddIntent modified for maximal frequent itemsets) lags behind these two, although it performs much better than MAFIA [8]. Optimized implenations of APRIORI and ECLAT [2] failed to compute the output even in the case of small subcollections of documents (about 10% of the whole collection). This relative behavior of algorithms is similar to that observed in [13] in experiments with low support. In the following table we present running times in a typical experiment with different algorithms on a subcollection of about 10% of the whole collection.

FPmax		All Pairs of Duplicates		Unique pairs of duplicates		Common pairs
Input	Threshold	ROMIP	HSE	ROMIP	HSE	
b_1_20_s_100_n1-12.txt	100	105570	15072	97055	6557	8515
b_1_20_s_100_n1-12.txt	95	105570	20434	93982	8846	11588
b_1_20_s_100_n1-12.txt	90	105570	30858	87863	13151	17707
b_1_20_s_100_n1-12.txt	85	105570	41158	83150	18738	22420
b_1_20_s_100_n1-24.txt	100	191834	41938	175876	25980	15958
b_1_20_s_100_n1-24.txt	95	191834	55643	169024	32833	22810
b_1_20_s_100_n1-24.txt	90	191834	84012	155138	47316	36696
b_1_20_s_100_n1-24.txt	85	191834	113100	136534	57800	55300
b_1_10_s_150_n1-6.txt	150	33267	6905	28813	2451	4454
b_1_10_s_150_n1-6.txt	145	33267	9543	27153	3429	6114
b_1_10_s_150_n1-6.txt	140	33267	13827	24579	5139	8688
b_1_10_s_150_n1-6.txt	135	33267	17958	21744	6435	11523
b_1_10_s_150_n1-6.txt	130	33267	21384	19927	8044	13340
b_1_10_s_150_n1-6.txt	125	33267	24490	19236	10459	14031

Table 2: Results of the method $\mathfrak{n}$ *minimal elements in a permutation.*

FPmax		All Pairs of Duplicates		Unique pairs of duplicates		Common pairs
Input	Threshold	ROMIP	HSE	ROMIP	HSE	
m_1_20_s_100_n1-3.txt	100	16666	4409	14616	2359	2050
m_1_20_s_100_n1-3.txt	95	16666	5764	13887	2985	2779
m_1_20_s_100_n1-3.txt	90	16666	7601	12790	3725	3876
m_1_20_s_100_n1-3.txt	85	16666	9802	11763	4899	4903
m_1_20_s_100_n1-6.txt	100	33267	13266	28089	8088	5178
m_1_20_s_100_n1-6.txt	95	33267	15439	26802	8974	6465
m_1_20_s_100_n1-6.txt	90	33267	19393	24216	10342	9051
m_1_20_s_100_n1-12.txt	100	105570	21866	95223	11519	10347
m_1_20_s_100_n1-12.txt	95	105570	25457	93000	12887	12570

Table 3: Results for the method*minimal elements in* $\mathfrak{n}$ *permutations.*

Algorithm	Dataset	Threshold	Time elapsed sec
Fpmax*	b_1_20_s_100_n1-6.txt	95	2,0
	b_1_20_s_100_n1-6.txt	90	3,1
	b_1_20_s_100_n1-6.txt	85	5,3
	b_1_20_s_100_n1-12.txt	100	3,0
	b_1_20_s_100_n1-12.txt	95	9,0
	b_1_20_s_100_n1-12.txt	90	14,2
	b_1_20_s_100_n1-12.txt	85	25,7
	b_1_20_s_100_n1-24.txt	100	16,1
	b_1_20_s_100_n1-24.txt	95	120,0
	b_1_20_s_100_n1-24.txt	90	590,4
	b_1_20_s_100_n1-24.txt	85	1710,6
Afopt	b_1_20_s_100_n1-6.txt	100	1,39
	b_1_20_s_100_n1-6.txt	95	1,984
	b_1_20_s_100_n1-6.txt	90	2,359
	b_1_20_s_100_n1-6.txt	80	3,078
Mafia	b_1_20_s_100_n1-6.txt	100	123
	b_1_20_s_100_n1-6.txt	95	584
	b_1_20_s_100_n1-6.txt	90	1160
	b_1_20_s_100_n1-6.txt	80	2186
	b_1_20_s_100_n1-12.txt	100	1157
apriori_borgelt	b_1_20_s_100_n1-6.txt	100 - 85	failed
eclat_apriori	b_1_20_s_100_n1-6.txt	100 - 85	failed
AddIntent*	b_1_20_s_100_n1-6.txt	100	177,64
	b_1_20_s_100_n1-6.txt	95	186,765
	b_1_20_s_100_n1-6.txt	90	192,765
	b_1_20_s_100_n1-6.txt	85	204,031

Table 4: Performance of FIMI algorithms

In the contexts corresponding to these subcollections, the number of objects is relatively large compared to the threshold minsup value defined by parameters in the definition of duplicates. Thus, these are typical problems of generating frequent itemsets in low-support data and relative performance of data mining algorithms in our experiments is similar to that in survey [13].

5 Conclusions and further work

We propose a framework to detect near-duplicate documents in large text collections. Analyzing the results of our experiments with concept-based def-

inition of clusters of similar documents with ROMIP data collection we can draw the following conclusions:

- ROMIP collection of URLs is a good testbed for comparing performance of FCA and data mining algorithms in generating (maximal frequent) closed sets of attributes.
- The list of ROMIP duplicates contains many false duplicates, which are not detected as such by the methods based on closed itemsets.
- Approaches based on closed sets of attributes propose adequate and efficient techniques for both determining similarity of document images and generating clusters of very similar documents. They can be efficiently used on the stage of outputting documents relevant to a query, when the number of all found relevant documents does not exceed several thousands (around 10,000 documents). However, this algorithm may encounter major difficulties in treating larger collections of documents due to intrinsic exponential worst-case complexity of the problem of computing maximal frequent itemsets.
- For our datasets (which are very "column-sparse"), the best data mining algorithms for computing frequent closed itemsets, FPmax* and Afopt, outperform AddIntent, one of the best algorithm for constructing concept lattice, adapted for computing maximal frequent itemset.
- The results of syntactical methods essentially depend on the *shingle length* parameter. Thus, in our experiments, for the shingle length 10 the results (pairs of duplicates) were much closer to those in the ROMIP list as for the lengths of shingles equal to 20, 15, and 5.
- In our experiments the results obtained by different methods of document representation – π *minimal elements in a permutation* and *minimal elements in n permutations* – did not differ much, which testifies in favor of the first, faster method.

We would also like to create a site of our project on SourceForge.net with freely available sources of the framework. In further developments, we are going to release implementation of other methods for near-duplicate detection (NDD) like I-match, super shingling, and so on. Development of complex techniques for testing NDD methods and creation of testing collections seems to be quite of interest for computer scientists in this field.

Acknowledgements

This work was supported by the Scientific Foundation of Russian State University Higher School of Economics as a part of project 08-04-0022. The authors would like to thank Sergei A. Obiedkov, Igor A. Selitsky, and Mikhail V. Samokhin for helpful discussions and participating in the software realization of the approach.

References

[1] A. M. Hasnah, A new filtering algorithm for duplicate document based on concept analysis, *Journal of Computer Science*, **2**, 5 (2006) 434–440.

[2] C. Borgelt, Efficient implementations of Apriori and Eclat, *Proc. Workshop on Frequent Itemset Mining Implementations Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*, 2003, Melbourne, FL, USA.

[3] S. Brin, J. Davis, H. Garcia-Molina, Copy detection mechanisms for digital documents, *1995 ACM SIGMOD International Conference on Management of Data* 1995, pp. 398–409. http://borgelt.net/apriori.html

[4] A. Broder, On the resemblance and containment of documents, *Proc. Compression and Complexity of Sequences (SEQS: Sequences97)*. pp. 21–29.

[5] A. Broder, S. Glassman, M. Manasse, G. Zweig, Syntactic clustering of the web, *6th International World Wide Web Conference*, Apr. 1997, pp. 393–404.

[6] A. Broder, M. Charikar, A.M. Frieze, M. Mitzenmacher, Min-wise independent permutations, *Proc. STOC*, 1998, pp. 327–336.

[7] A. Broder, Identifying and filtering near-duplicate documents, *Proc. Annual Symposium on Combinatorial Pattern Matching*, *Lecture Notes in Computer Science* (eds. R. Giancarlo and D. Sankoff) **1848**, 2000, pp. 1–10.

[8] D. Burdick et al., MAFIA: A performance study of mining maximal frequent itemsets, *Proc. Workshop on Frequent Itemset Mining Implementations Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*, 2003, Melbourne, FL, USA.

[9] M. S. Charikar, Similarity estimation techniques from rounding algorithms, *34th Annual ACM Symposium on Theory of Computing* (2002) 380–388.

[10] J. Cho, N. Shivakumar, H. Garcia-Molina, Finding replicated web collections, *Proc. SIGMOD Conference*, (2000) 355–366.

[11] A. Chowdhury, O. Frieder, D.A. Grossman, M.C. McCabe, Collection statistics for fast duplicate document detection, *ACM Transactions on Information Systems*, **20**, 2 (2002) 171–191.

[12] B. Ganter, R. Wille, *Formal concept analysis: mathematical foundations*, Springer, Berlin, 1999.

[13] B. Goethals, M. Zaki, Advances in Frequent Itemset Mining Implementations: Introduction to FIMI03, *Proc. Workshop on Frequent Itemset Mining Implementations Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*, 2003.

[14] G. Grahne, J. Zhu, Efficiently using prefix-trees in mining frequent itemsets, in *Proc. FIMI03 Workshop*, 2003.

[15] G. S. Manku, A. Jain, A. Das Sarma, Detecting near-duplicates for web crawling, *Proceedings of the 16th international conference on World Wide Web*, May 8-12, 2007, Banff, Alberta, Canada

[16] T. H. Haveliwala, A. Gionis, D. Klein, P. Indyk, Evaluating strategies for similarity search on the web, *Proc. WWW'2002*, Honolulu, 2002, pp. 432–442.

[17] N. Heintze, Scalable document fingerprinting, *Proc. of the 2nd USENIX Workshop on Electronic Commerce*, 1996, pp. 191–200.

[18] M. Henzinger, Finding near-duplicate web pages: a largeccale evaluation of algorithms, *Annual ACM Conference on Research and Development in Information Retrieval*, 2006, pp. 284–291.

[19] T. C. Hoad, J. Zobel, Methods for identifying versioned and plagiarised documents, *Journal of the American Society for Information Science and Technology*, **54**, 3 (2003) 203–215.

[20] D. I. Ignatov, S. O. Kuznetsov, V. B. Lopatmikova, I. A. Selitsky, Development and testing of system for detection of near-duplicates in collection of R&D documents, *Theoretical and Practical Journal of State University*

Higher School of Economics for Business Informatics **4**, 6 (2008) 21–28 (in Russian).

[21] S. Ilyinsky, M. Kuzmin, A. Melkov, I. Segalovich, An efficient method to detect duplicates of web documents with the use of inverted index, *Proc. 11th Int. World Wide Web Conference (WWW'2002)*, Honolulu, Hawaii, USA, 7-11 May 2002, *ACM*, 2002

[22] A. Kolcz, A. Chowdhury, J. Alspector, Improved robustness of signature-based near-replica detection via lexicon randomization, *Proc. KDD'04* (eds. W. Kim, R. Kohavi, J. Gehrke, W. DuMouchel) Seattle, 2004, pp. 605–610.

[23] S.O. Kuznetsov, S.A. Obiedkov, Comparing performance of algorithms for generating concept lattices, *Journal of Experimental and Theoretical Artificial Intelligence*, **14** (2002) 189–216.

[24] G. Liu, H. Lu, J. Xu Yu, W. Wei, X. Xiao, AFOPT: An efficient implementation of pattern growth approach, Proc. Workshop on Frequent Itemset Mining Implementations Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03), 2003.

[25] U. Manber. Finding similar files in a large file system, *Proc. of the USENIX Winter 1994 Technical Conference*, 1994, pp. 1–10.

[26] D. van der Merwe, S.A. Obiedkov, D. Kourie, AddIntent: a new incremental algorithm for constructing concept lattices, *Proc. International Conference on Formal Concept Analysis (ICFCA'04)*, *Lecture Notes in Artificial Intelligence*, **2961**, 2004, pp. 372–385.

[27] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Efficient mining of association rules using closed itemset lattices, *Inform. Syst.*,**24**, 1 (1999) 25–46.

[28] M. Potthast, B. Stein, *New issues in near-duplicate detection* in *Data Analysis, Machine Learning and Applications*, Springer, 2007, pp. 601–609.

[29] W. Pugh, M. Henzinger, Detecting duplicate and near-duplicate files, *United States Patent 6658423* (December 2, 2003).

[30] N. Shivakumar, H. Garcia-Molina, Finding near-replicas of documents on the web, *Proc. The World Wide Web and Databases, International*

Workshop (WebDB'98), *Lecture Notes in Computer Science*, **1590**, 1999, pp. 204–212.

[31] N. Shivakumar, H. Garcia-Molina. Building a scalable and accurate copy detection mechanism, *Proc. ACM Conference on Digital Libraries*, March 1996, pp. 160–168.

[32] C. Xiao, W. Wang, X. Lin, J. X. Yu, Efficient similarity joins for near duplicate detection, *Proceedings of the 17th International Conference on World Wide Web*, Beijing, China, 2008, pp. 131–140.

[33] Y. Zhao, G. Karypis, Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, **55** (2004) 311–331.

Received: May 11, 2009.

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 235–258

Systolic multiplication – comparing two automatic systolic array design methods

Laura Ruff
Babeş-Bolyai University, Cluj-Napoca
Faculty of Mathematics and Computer Science
email: laura@cs.ubbcluj.ro

Abstract. This paper provides a comparison between two automatic systolic array design methods: the so called *space-time transformation* methodology (a unifying approach to the design of VLSI algorithms [14], [15]), and a *functional–based design method* (see [6], [9], [10]).

The advantages (and possible disadvantages) of each method are pointed out by representative case studies (variants of systolic arrays generated with both design methods).

Many algorithms were already parallelised using the efficient technique of space-time transformations. However, it also has some drawbacks. It may be hard to formulate the problem to be solved in the form of a *system of uniform recurrence equations*, which is the usual starting point for this method. On the other hand, the space-time transformation method depends heavily on finding an affine timing function, which can also lead to complex computations.

The functional-based method exploits the similarity between the inductive structure of a systolic array and the inductive decomposition of the argument by a functional program. Although it is less general in the sense that it generates systolic arrays with certain properties, its most significant advantage is that it needs to investigate the behaviour of only the first processor of the systolic array, while other methods (as the space-time transformation method, too) must work with an array of processors. Moreover, the method is based on rewriting of terms (according to certain equations, which are general for function definitions

AMS 2000 subject classifications: 68M07, 65Y05, 68W35
CR Categories and Descriptors: B.6.3 [Design Aids]: Subtopic - Automatic synthesis B.6.1 [Design Styles]: Subtopic - Parallel circuits
Key words and phrases: systolic array, automatic systolic array design, space-time transformations, functional approach

and systolic arrays), thus the resulting systolic algorithm is certified to be correct, and the method itself is relatively easy to automatize.

1 Introduction

The first examples of systolic arrays (very efficient special purpose parallel computing devices) and the systolic algorithms running on them were conceived in an ad–hoc manner, requiring a great amount of intuition and creativity from their inventors.

Later several [semi]automatic systolic array design methods were proposed (a short survey can be found in [16]). Most of these systematic methods use an *iterative view* of systolic arrays: the arrays (and the computations) are represented as multidimensional matrices of a certain size (in fact some methods only work for a fixed size, the problem cannot be parametrized). This kind of representation leads to complex operations over the multidimensional index space, on the other hand, due to the symmetric organisation of systolic structures, there are many repetitions in the design process.

The most widespread and also the most general method is referred to as the *space-time transformation method*. This is in fact a unifying approach to the design of systolic arrays, which incorporates the main ideas used in several automatic synthesis methods. The work of many researchers like Quinton, Robert, Van Dongen [12, 14, 13], Delosme and Ipsen [1], Nelis and Deprettere [8] relies on it. A review of the main ideas involved in the space-time transformation method is presented by Song in [15].

Many algorithms were already parallelised with this efficient method, however it also has its drawbacks.

The problem to be solved should be formulated as a *uniform recurrence equation system*, which is sometimes not an easy task. The uniformisation of linear recurrence equations was tackled by Quinton and Dongen [13], Fortes and Moldovan [3] and others but it is still not definitely solved.

The space-time transformation method heavily depends on finding an adequate affine timing function. The problem with finding such a function is that one needs to solve a linear equation system, which is usually a tedious and difficult task, on the other hand it is only possible for systems having certain properties.

As we already mentioned, the most design methods, thus the space-time transformation method, too, uses an *iterative* approach to the problem. In contrast, our design method presented in [6] follows a *functional view*: a linear

systolic array is composed of a *head processor* and an identical *tail array*. Similarly, functional programs for list operations describe how to compute the head and the tail of the result in function of the head and the tail of the arguments.

Our design method exploits this similarity, thus the synthesis problem can be solved by [essentially] rewriting of the functional programs.

In this paper we compare this functional-based method with the space-time transformation method using some representative case studies. Our purpose is not the detailed presentation of the two methods (one can find such descriptions in [14, 15] –about the space-time transformation method–, respectively in [6, 9, 10] –about the functional/based method–). However, we would like to point out the advantages (or disadvantages) of the two distinct methods through some practical examples.

2 Systolic array design for polynomial multiplication

We start with a simple problem, the polynomial multiplication.
Let A and B two univariate polynomials of degree $n-1$ and $m-1$, respectively:

$$A = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_{n-1} * x^{n-1}$$
$$B = b_0 + b_1 * x + b_2 * x^2 + \ldots + b_{m-1} * x^{m-1}$$

We denote the product of A and B with C (polynomial of degree $n + m - 2$):

$$C = A * B = c_0 + c_1 * x + c_2 * x^2 + \ldots + c_{m+n-2} * x^{n+m-2},$$

where

$$c_k = \sum_{i+j=k} a_i * b_j, \quad \forall k,\ 0 \le k \le m + n - 2; 0 \le i \le n - 1, 0 \le j \le m - 1 \tag{1}$$

2.1 Solutions to the problem using the space-time transformation method

In order to be able to apply the space-time transformation methodology to the problem, the coefficients of C should be given in a recursive way, thus (1) is not an adequate formulation to start up with. We need a *uniform recurrence equation system* which is a subclass of linear recurrence equation systems. Only such systems are suitable for being directly mapped onto systolic architectures,

as they require local data and local interconnections between the processing elements (PEs).

Definition 1 ***Uniform recurrence equation system***
*A system of uniform recurrence equations (*SURE*) is a collection of* $s \in \mathbb{N}$ *equations of the form (2) and input equation of the form (3):*

$$V_i(z) = f_i(V_1(z-\theta_{i_1}),\ldots,V_k(z-\theta_{i_k})) \quad (2)$$

$$V_i(z_i^j) = v_i^j, j \in \{1,...,l_i\} \quad (3)$$

where

- $V_i : D \to R$. $V_i, i \in \{1,\ldots,s\}$ *are variable names belonging to a finite set* V. *Each variable is indexed with an integral index, whose dimension,* n *(called the index dimension), is constant for a given SURE (in practice this is usually 2 or 3).*
- $z \in D$, *where* $D \subseteq \mathcal{Z}^n$ *is the domain of the SURE.*
- v_i^j *is a scalar constant (input),* $z_i^j \in D_{inp}$, *where* $D_{inp} \subseteq \mathcal{Z}^n$ *is the domain of the inputs.*
- $\theta_{i_1},\ldots,\theta_{i_k}$ *are vectors of* Z^n *and are called* dependence vectors *of the* SURE.
- $V_i(z)$ *does not appear on the right-hand side of the equation.*
- $f_i : R^s \to R$.

Informally, a SURE (as well as the associated dependence graph) can be seen as a multidimensional systolic array, where the points of the domain D are the PEs of the array and the communication channels are determined by the dependencies. In this context, a transformation applied to the system which preserves the number of domain points and the dependencies leads to a computationally equivalent system. The goal of such a transformation is to obtain a system where one of the indices can be interpreted as the *time index* and the others as *space-indices*.

The form of the SURE, used as a starting point, has a considerable impact on the result of the design.

If we start for example with the following algorithm:

$c_j = 0, \forall j, 0 \le j \le m+n-2$
for $i = 0$ to $n-1$
    for $j = i$ to $i+m-1$
        $c_j = c_j + a_i * b_{j-i}$,

then we can formulate the uniform recurrence equation system (4)-(5). We mention that there are well-known uniformisation techniques (see [13]) to deduce the SURE from the given algorithm, but unfortunately they cannot be applied in a fully automatic way. Let us therefore consider the system (4)-(5) as starting point.

Equations:

$$\begin{cases} C_{i,j} &= C_{i-1,j} + B_{i-1,j-1} * A_{i,j-1} \\ B_{i,j} &= B_{i-1,j-1} \\ A_{i,j} &= A_{i,j-1}, \end{cases} \tag{4}$$

where $0 \leq i \leq n-1$, $i \leq j \leq i+m-1$.

Input equations:

$$\begin{cases} B_{-1,i} &= b_{i+1}, & -1 \leq i \leq m-2 \\ C_{-1,i} &= 0, & 0 \leq i \leq m-1 \\ C_{i-1,i+m-1} &= 0, & 1 \leq i \leq n-1 \\ A_{i,i-1} &= a_i, & 0 \leq i \leq n-1. \end{cases} \tag{5}$$

The results are considered to be the values of the following variables:

$$c_i = \begin{cases} C_{i,i}, & 0 \leq i \leq n-2 \\ C_{n-1,i}, & n-1 \leq i \leq n+m-2. \end{cases}$$

Fig. 1 presents the dependence graph associated to the SURE (4)-(5), when $n = 3, m = 4$.

Each of the points of the domain $D = \{(i,j) | 0 \leq i \leq 2, i \leq j \leq i+3\}$ corresponds to a computation, while the arrows represent the data dependencies. The placement of the input values can also be read from the figure, although this was determined after the computation of the timing function. The small dots between the points (i,j), $(i+1,j+1)$ of the domain D indicate a delay, that is, the b values need two time steps to move from point (i,j) to $(i+1,j+1)$. Now we have the dependencies shown in Table 1.

The space-time transformation method consists basically of four main steps:

1. the formulation of the problem as a system of uniform recurrence equations,
2. finding one (ore more) adequate *timing function(s)*
   – the timing function determines the time instant when the computation takes place,
3. finding one (ore more) adequate *allocation function(s)* corresponding to a certain timing function

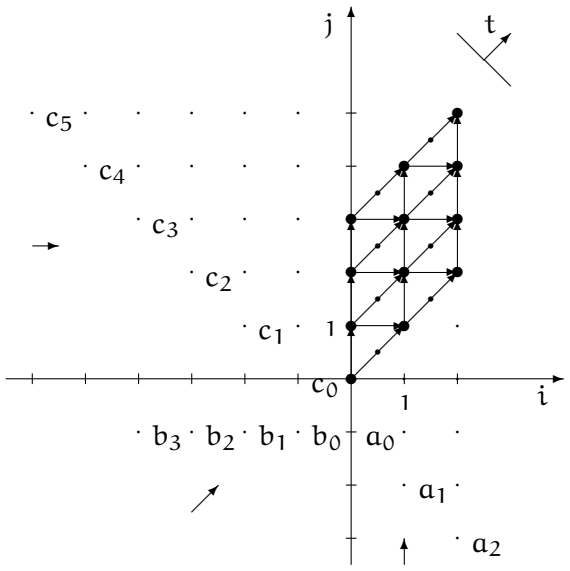


Figure 1: Dependence graph for polynomial multiplication ($n = 3, m = 4$): data dependencies, placement of input data

– the allocation function determines the place (that is the processor) where the computation is performed,

4. application of the space-time transformation.

Linear (and affine) transformations are most commonly used for both, the timing and the allocation function, because thus we obtain a linear space-time transformation, which preserves the dependencies between the computations. Moreover, if the transformation is *unimodular*, then it has the advantage that it preserves the number of points in the domain, and in addition it admits an integral inverse. However, it is not mandatory to use unimodular transformations.

Given a SURE, the next step is to obtain a possible linear (or affine) timing function, which should be positive and should preserve the data dependencies. A natural requirement is that in order to be able to perform the computations of $V_i(z)$, its arguments should have been computed before. If such a function exists, then we say that the SURE is *computable*.

Equation	lhs	rhs	Dependence vector
(14)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1,1)$
(16)	$A1_{i,j}$	$A1_{i,j-1}$	$(0,1)$
(18)	$B1_{i,j}$	$B1_{i,j-1}$	$(0,1)$
(20)	$A2_{i,j}$	$A2_{i-1,j}$	$(1,0)$
(22)	$B2_{i,j}$	$B2_{i-1,j}$	$(1,0)$

Table 1: Dependence vectors

The previously mentioned constraints build a system of inequalities. Any of its solutions gives an adequate timing function.

We might also want to minimise the computation time of the system. In this case, we obtain the timing function:

$$t(i,j) = i + j.$$

The timing function was determined according to the method described in [14, 2] (we avoid to detail the computations here).

In order to get an adequate allocation function for a given timing function, the condition that should hold (we also call it *general constraint*) can be intuitively expressed in the following way: two different computations performed at the same time-step should not be mapped onto the same processor. This means that the linear part P of the allocation function should not be parallel to the direction T corresponding to the timing function (in our case $T = (1,1)$).

For the previously obtained timing function, we get the following allocation functions, which satisfy the above-mentioned condition, moreover, the resulting space-time transformation is unimodular:

$$\begin{aligned} p(i,j) &= j - i \\ p(i,j) &= i \\ p(i,j) &= j. \end{aligned}$$

If we choose the allocation function $p(i,j) = j - i$, after the application of the space-time transformation we obtain the linear systolic array depicted in Fig. 2.

With the allocation function $p(i,j) = i$, we get the systolic array from Fig. 3, while with $p(i,j) = j$ the array from Fig. 4 is obtained. The placement of the inputs is also depicted in the figures. In case of Fig. 3, the structure of the array respectively the transition function is also shown.

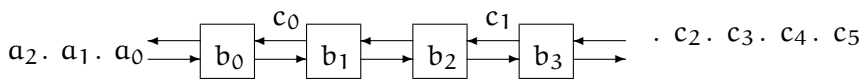


Figure 2: Systolic array for polynomial multiplication (the allocation function $p(i,j) = j - i$ was used)

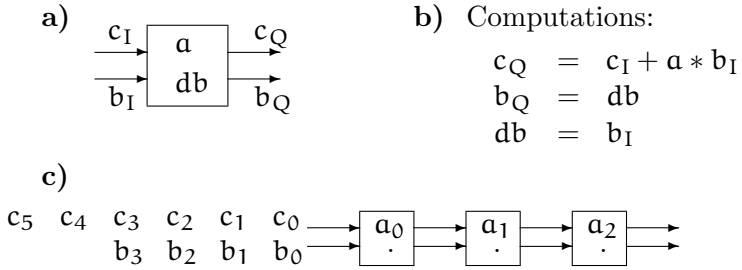


Figure 3: Unidirectional systolic array for polynomial multiplication (the allocation function $p(i,j) = i$ was used for the projection) a) structure of a PE, b) transition function, c) structure of the array and placement of the input values

The data-flow in the arrays of Fig. 3 and Fig. 4 is unidirectional. In the case of the array of Fig. 3 the elements of the result appear after n time steps (where n is the number of PEs) as the output of the PE on the right edge of the array, while in the case of the array from Fig. 4 the results are computed in the local memories of the PEs.

The systolic array depicted in Fig. 2 is bidirectional, but the PEs work alternately and they only perform useful computation at each second time step. There are some well-known techniques to transform such arrays into a more efficient one. Some ideas are presented in [11].

2.2 Functional approach

We have seen that the space-time transformation methodology works with the whole index space. Due to the symmetric structure of the systolic array, this leads to many repetitions in the design process. In case of our functional-based approach to the systolic array design, however, we only have to analyse the

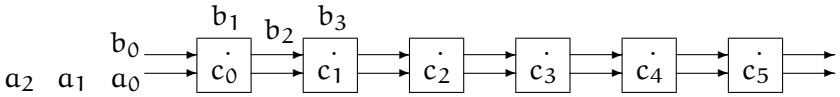


Figure 4: Systolic Array for Polynomial Multiplication (the allocation function $p(i,j) = j$ was used)

behaviour of the first processor, exploiting the idea that the tail-array works in the same way as the original one, solving actually the same kind of problem of a smaller size.

The *functional view* (or inductive view) of systolic arrays is shown in Fig. 5: informally, a linear systolic array with n PEs can be seen as a device, that is composed of a head processor (PE_0), connected to a tail-array, which is an identical array of size $n-1$.

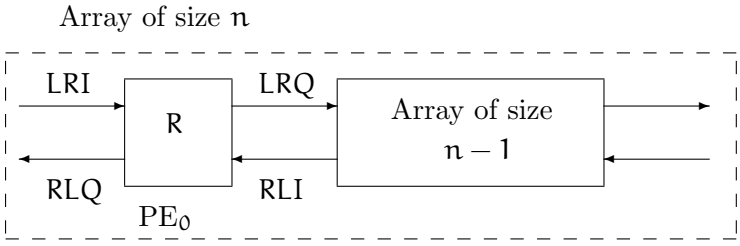


Figure 5: Informal view of a linear systolic array – functional approach

The arrows indicate the direction of the data-flow, from left to right (LR) or from right to left (RL). The letter I stands for *input channels*, Q indicates the *output channels* and R stands for the *internal state registers* (also called local memory).

At each time step the PEs update their internal state (the values of the output channels, respectively that of the internal registers) in function of the input, respectively the value of the internal state registers in the previous time step. The computations performed by a PE are given by the so called *transition function*.

The global input is fed step by step into the array through the input channels of the PEs on the edge, while the result appears at one or more output channels of the marginal PEs (in some cases the result may be computed in the internal state registers as in the case of the systolic array in Fig. 4).

A fundamental step of the design method is the formal analysis of the different systolic array types. In case of each class of systolic arrays, we characterise by a recursive description the class of functions which can be realised by such type of arrays.

Then, by equational rewriting, the expression of the list function which must be realised is transformed into an expression having the required structure. The resulting expression reveals the scalar function, which must be implemented by each individual processor.

The linear systolic arrays can be one- or bidirectional, depending on the direction of the data-flow. A typical subclass of systolic arrays is that, where the input data passes through the array unchanged.

The input or the output data-flow can be delayed or not, the arrays may have more simple building blocks, that is PEs without internal state (also called *combinatorial PEs*), or PEs having constant or variable internal state registers (*local memory*).

Let us consider for example unidirectional systolic arrays with constant internal state registers and delayed input. An example for such an array is depicted in Fig. 3.

We use the following notations (same as in [6, 9]):

- We denote by X_i (where $i \in \mathbb{Z}$) the *infinite list* $\langle x_i, x_{i+1}, x_{i+2} \ldots \rangle$. X stands for X_0. $X_{n,n+m}$ (where $n \in \mathbb{Z}$ and $m \in \mathbb{N}$) denotes the finite list having $m+1$ elements: $\langle x_n, x_{n+1}, \ldots x_{n+m} \rangle$.
- We will denote by a^n the list of n elements all equal to a and by a^∞ the infinite constant list with all elements equal to a.
- For any list $X = \langle x_0, x_1, \ldots, x_n, \ldots \rangle$, we denote by $H[X] = x_0$ the *head* of it, and by $T[X] = \langle x_1, \ldots, x_n, \ldots \rangle$ the *tail* of it.
- The k^{th} *tail* respectively *head* of X:
  $T_k[X] = \langle x_k, x_{k+1}, \ldots, x_n, \ldots \rangle$, for $k \geq 0$ is obtained by iterating T k times. Note that $T_1 = T$. By convention $T_0[X] = X$.
  T_k, for $k < 0$ is obtained by iterating T_{-1} $|k|$ times,
  where $T_{-1}[X_i] = X_{i-1}$
  $H_k[X] = H[T_k[X]]$ gives the $(k+1)^{\text{th}}$ element of X (thus $H_0 = H$).
- The *prefix* of order n of a list is $P_n[X] = \langle x_0, \ldots, x_{n-1} \rangle = X_{0,n-1}$.
- The *concatenation* of two lists is denoted by "$\frown$":
  $\langle a_0, a_1, \ldots, a_k \rangle \frown X = \langle a_0, a_1, \ldots, a_k, x_0, x_1, \ldots \rangle$.
  The first operand must be finite, but the second may also be infinite.
  We also use " $\overset{\cdot}{\frown}$ " for *prepending* a scalar to a (finite or infinite) list:
  $a \overset{\cdot}{\frown} X = \langle a \rangle \frown X$.

- We use (as in the theory of cellular automata) a special *quiescent symbol* "$" in order to encode the "blank" values.

The list function $\vec{f}$ in (6) is the *list extension* of the scalar function f (we obtain $\vec{f}[X]$ by applying the function f onto the elements of X). In the same time the expression (6) characterizes the transition function of one PE [10].

$$\vec{f}[x \dot{\smile} X] = f[x] \dot{\smile} \vec{f}[X] \tag{6}$$

Note that the syntactic restriction to one argument (and one value) is not essential. X could also represent a multiple list (composed of a finite k number of lists):

$$\begin{aligned} X &= \langle w^1 \dot{\smile} W^1,\ w^2 \dot{\smile} W^2, \ldots w^k \dot{\smile} W^k \rangle^T = \\ &= \langle w^1, w^2, \ldots w^k \rangle^T \dot{\smile} \langle W^1, W^2, \ldots W^k \rangle^T \ . \end{aligned}$$

Thus a function having multiple list arguments can be seen as a function with one single list-argument (even if we do not mention it explicitly).

The functioning of unidirectional systolic arrays with constant local memory and delayed input is characterised by (7)-(8) (see [10]), where n is the number of PEs (the size-parameter of the problem), X is the input list, the values of $U_{0,n-1}$ correspond to the constant values of the local memory variables and can be considered as parameters of the problem. Y denotes the global output-list which collects the (partial) results, while $Y^0 = (y^0)^\infty$ gives the list of initial values, which contribute to the computation of the results (usually the same y^0 value is introduced repeatedly).

$$F_{Q_{0,n-1}}[n, X] = \vec{f}_{q_{n-1}}[X_{-(n-1)}, X_{-n}, F_{Q_{0,n-2}}[n-1, X]] \tag{7}$$

$$F_{q_0}[1, X] = \vec{f}_{q_0}[X, X_{-1}, Y^0], \tag{8}$$

where the list function $\vec{f}_q[\langle X, X', Y \rangle]$ satisfies property (6).
Given $F_{U_{0,n-1}}[n, X]$, our task is to find $Q_{0,n-1}$, which is a permutation of $U_{0,n-1}$, y^0 (such that $Y^0 = (y^0)^\infty$) and the transition function, denoted by f such that (7)-(8) should hold.

Let us consider again the problem of polynomial multiplication. The coefficients of one polynomial will be matched with the finite list of parameters (let us choose for this purpose the n coefficients of polynomial A, that is the list $A_{0,n-1}$), while the coefficients of the other polynomial will form the input list. We get an infinite input list by adding an infinite number of 0 elements

to the list of coefficients ($B_{0,m-1} \frown 0^\infty$). We will also insert a number of $n-1$ elements of 0 in front of the list in order to describe the problem according to the idea depicted in Fig. 6.

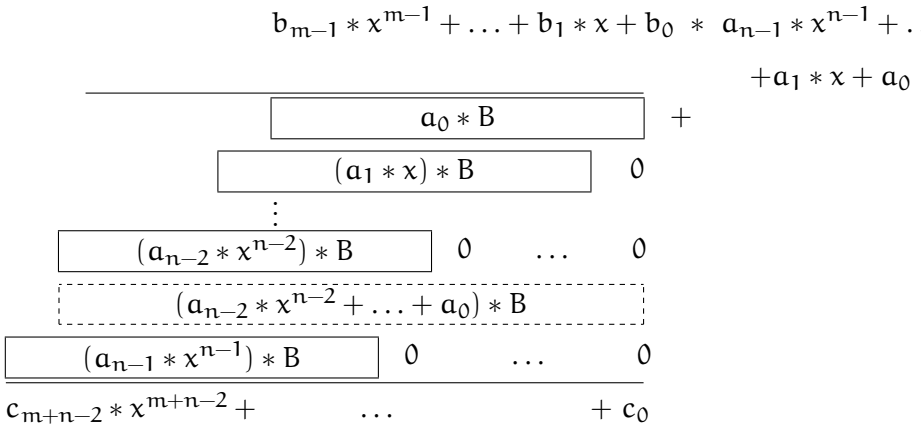


Figure 6: Polynomial multiplication

The coefficients of the product will be the first $n + m - 1$ elements of the list C:
$C = \langle c_0, c_1, \ldots \rangle$, where

$$c_i = \sum_{j=0}^{n-1} a_j * b_{i-j}, \forall i, i = 0, 1, \ldots n + m - 2, c_i = \$, \forall i, i \geq n + m - 1$$

Using the more concise list notation this means:

$$C = \sum_{j=0}^{n-1} a_j * B_{-j} \tag{9}$$

(where $a*(b \dot\frown B) = \langle a*b \rangle \dot\frown (a*B)$)

We can write that

$$\begin{aligned} F_{A_{0,n-1}}[n, B] &= \textstyle\sum_{j=0}^{n-1} a_j * B_{-j} = a_{n-1} * B_{-(n-1)} + \sum_{j=0}^{n-2} a_j * B_{-j} = \\ &= a_{n-1} * B_{-(n-1)} + F_{A_{0,n-2}}[n-1, B], \end{aligned}$$

which is of the form (7). We get by simple projection the part of the transition function corresponding to the computation of the c (partial) results: $f[b, db, a, c] = a*b + c$. The rest of the transition function is already known.

By analysing condition (8) we get: $F_{a_0}[1, B] = \vec{f}[a_0, X, X_{-1}, C^0]$, that is $a_0{*}B = a_0{*}B + C^0 \Rightarrow c^0 = 0$ and $C^0 = 0^\infty$.

We got the solution of Fig. 3 with much less effort than in the case of the space-time transformation method. The systolic array obtained is also appropriate for integer multiplication, only a local memory register of variable value should be added to each PE, in order to preserve the carry [10].

The automatic synthesis of the arrays shown in Fig. 2 and Fig. 4 is similarly simple. However, because these have a different structure, we have to start from another description of the problem, while in the case of the space-time transformation method we obtained the three different solutions starting from the same recurrence equation. However, if we would like to design a systolic array with predefined properties, this is not a drawback at all.

3 Online systolic multiplication

In this section we describe an online systolic array, the functional-based design of which was detailed in a former paper [6]. After outlining the results, we present how such an array could have been synthesized using the space-time transformation methodology.

Online arrays are an important special subclass of bidirectional arrays. They are characterised by the fact that they begin to provide the first result after a constant number of time steps (regardless of the number of PEs). This feature make them very useful for solving real time problems, where the response time is a critical factor.

The array receives the input data through the first PE and the elements of the result leave the array through the same PE.

3.1 Solutions obtained using the functional approach

We have presented the design of such systolic arrays in [6], and we used as case study the design of online arrays for polynomial multiplication, respectively the multiplication of multiple precision integers. That is why we do not detail the design process here; in the sequel, we will only outline its main steps.

Step 0: formal analysis of the systolic array with the given properties
We have analysed the behaviour of specific online systolic arrays with input list X, where the input X' of the tail-array is $T_k[X]$ for some fixed k, thus if the array computes the function $F[X]$, then the tail-array will compute $F[T_k[X]]$.

Such a behaviour of the input can be achieved by including into the internal state a "state variable" s with values from $\{0 = \$, 1, 2, \ldots, k + 2\}$, and the

following assignments for s:

$$s := \begin{cases} s, & x = \$ \ \vee \ s = k+2 \\ s+1, & x \neq \$ \ \wedge \ s < k+2 \end{cases}$$

The PE will send the x values to the next PE if $s \geq k$, otherwise a \$ value will be passed.

The functioning of such an array (for $k = 2$) is characterised by (10), where G denotes the function which computes the internal state of the array (the internal state includes besides the computation of the output values the values of the local memory variables, too), and f_y is the part of the transition function which computes the (partial) results:

$$T_4[F[X]] = \vec{f}_y[T_4[X],\ T_3[G[X]],\ F[T_2[X]]]. \tag{10}$$

Step 1: formulation of the problem as a functional program
The two polynomials are represented by the list of coefficients completed with an infinite number of redundant zeroes. The input list is the multiple list compound of these two lists.

We assume as known the scalar operations " $\dot{\dot{+}}$ " and " $\dot{\dot{*}}$ " in the ring of the coefficients. We will use the functional definition of the simple operations to *unfold* the expression "$A * B$", until we get an equation of the form (10).

Some definitions (we transformed the notations used in mathematics to our list-notation in a very simple, natural way):

- addition of a scalar with a polynomial: $a \dot{+} (b \dot{\smile} B) = (a \dot{\dot{+}} b) \dot{\smile} B$
- addition of polynomials: $(a \dot{\smile} A) + (b \dot{\smile} B) = (a \dot{\dot{+}} b) \dot{\smile} (A + B)$
- multiplication of a scalar with a polynomial:
  $a \dot{*} (b \dot{\smile} B) = (a \dot{\dot{*}} b) \dot{\smile} (a \dot{*} B)$
- multiplication of polynomials:
  $(a \dot{\smile} A) * (b \dot{\smile} B) = (a \dot{\dot{*}} b) \dot{\smile} ((a \dot{*} B) + (b \dot{*} A) + (0 \dot{\smile} (A * B)))$.

Step 2: unfolding
Unfolding consists in extracting repetitively the elements of the result list, beginning with the first one, by using the functional definitions of the list functions and a few simple unfolding rules, presented in [6].

After the unfolding of the first four elements of the expression $A * B$, we get

the following result:

$$
\begin{aligned}
&A * B = \\
&= \ldots \\
&= \langle \quad a_0 \mathbin{\dot{*}} b_0, \\
&\qquad a_0 \mathbin{\dot{*}} b_1 \mathbin{\dot{+}} b_0 \mathbin{\dot{*}} a_1, \\
&\qquad a_2 \mathbin{\dot{*}} b_0 \mathbin{\dot{+}} a_1 \mathbin{\dot{*}} b_1 \mathbin{\dot{+}} a_0 \mathbin{\dot{*}} b_2), \\
&\qquad a_3 \mathbin{\dot{*}} b_0 \mathbin{\dot{+}} a_2 \mathbin{\dot{*}} b_1 \mathbin{\dot{+}} a_1 \mathbin{\dot{*}} b_2 \mathbin{\dot{+}} a_0 \mathbin{\dot{*}} b_3 \rangle \smile \\
&\smile ((a_0 \mathbin{\dot{*}} B_4) + (b_0 \mathbin{\dot{*}} A_4) + \\
&\qquad + (a_1 \mathbin{\dot{*}} B_3) + (b_1 \mathbin{\dot{*}} A_3) + (A_2 * B_2))
\end{aligned}
$$

From here we can write the equality of the form (10):

$$
T_4[A * B] = + \begin{cases} H_0[A] \mathbin{\dot{*}} T_4[B] \\ H_0[B] \mathbin{\dot{*}} T_4[A] \\ H_1[A] \mathbin{\dot{*}} T_3[B] \\ H_1[B] \mathbin{\dot{*}} T_3[A] \\ T_2[A] * T_2[B] \ , \end{cases}
$$

respectively the first four elements of the output.

Step 3: the elements of the resulted expression are associated to the corresponding elements of the systolic array (using already specified rewrite rules). The head and tail functions H_i and T_i are realised by adding some suitable *static* respectively *transition* variables to the internal state. The list having (almost) all elements equal to H_i is realized by a "static" variable h_i having the assignment: $h_i := \begin{cases} x, \text{ if } s = i \\ h_i, \text{ if } s \neq i \end{cases}$

Let us also consider the "transition" variables z_0, z_1, z_2, z_3 having the assignments: $z_0 = z_1,\ z_1 = z_2,\ z_2 = z_3,\ z_3 = x$. In the expression of $T_4[F[X]]$, the subexpression $T_4[X]$ will be realized by the expression x, and each $T_i[X]$ will be realized by the expression z_i (for $0 \leq i \leq 3$).

We denote the input channels by xa and xb, and the corresponding static and transition variables by ha_i, hb_i, respectively za_i, zb_i, as shown in Fig. 7.

According to the rules mentioned above, the expression on the right–hand side is projected into:

$$
\begin{aligned}
&(ha_0 \mathbin{\dot{*}} xb) \mathbin{\dot{+}} (hb_0 \mathbin{\dot{*}} xa) \mathbin{\dot{+}} \\
&\mathbin{\dot{+}} (ha_1 \mathbin{\dot{*}} zb_3) \mathbin{\dot{+}} (hb_1 \mathbin{\dot{*}} za_3) \mathbin{\dot{+}} y'
\end{aligned}
$$

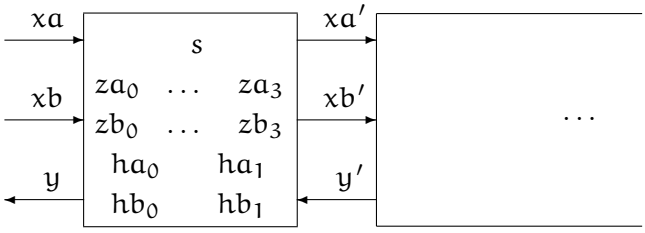


Figure 7: Online systolic array for polynomial multiplication

The expressions representing the computation of the first 4 elements are found in the same way.

Thus, the part of the transition function describing the assignment for the output channel y (that is, the computation of the result) is:

$$\begin{cases} \$ & s = \$ = xa \\ xa \mathbin{\dot{*}} xb & s = \$ \neq xa \\ hb_0 \mathbin{\dot{*}} xa \mathbin{\dot{+}} ha_0 \mathbin{\dot{*}} xb & s = 1 \\ ha_1 \mathbin{\dot{*}} hb_1 \mathbin{\dot{+}} & \\ \quad \mathbin{\dot{+}} hb_0 \mathbin{\dot{*}} xa \mathbin{\dot{+}} ha_0 \mathbin{\dot{*}} xb & s = 2 \\ hb_1 \mathbin{\dot{*}} za_3 \mathbin{\dot{+}} ha_1 \mathbin{\dot{*}} zb_3 \mathbin{\dot{+}} & \\ \quad \mathbin{\dot{+}} hb_0 \mathbin{\dot{*}} xa \mathbin{\dot{+}} ha_0 \mathbin{\dot{*}} xb & s = 3 \\ hb_1 \mathbin{\dot{*}} za_3 \mathbin{\dot{+}} ha_1 \mathbin{\dot{*}} zb_3 \mathbin{\dot{+}} & \\ \quad \mathbin{\dot{+}} hb_0 \mathbin{\dot{*}} xa \mathbin{\dot{+}} ha_0 \mathbin{\dot{*}} xb \mathbin{\dot{+}} y' & s = 4 \end{cases}$$

The rest of the transition function, containing the computation of variables $s, ha_i, hb_i, za_i, zb_i, xa', xb'$ is known.

We can use the same kind of array for the multiplication of arbitrary large integers again, by adding a register of variable value to each PE [6].

3.2 Solution using the space-time transformation method

None of the systolic arrays obtained from the SURE (4)–(5) is an online one. As we already mentioned, the result is significantly influenced by the form of the SURE used as starting point. Consequently, we need another formulation

of the problem, which again requires some intuition. We use the following notation:
In the sequel let the values of A_i be equal to a_i if $0 \leq i \leq n-1$, otherwise 0. In the same way $B_j = b_j$, if $0 \leq j \leq m-1$, otherwise 0.

$$A*B = \underbrace{A_0*B_0}_{C_{0,0}} + \underbrace{(A_0*B_1 + A_1*B_0)}_{C_{0,1}} * x + \underbrace{(A_0*B_2 + \overbrace{A_1*B_1}^{C1,1} + A_2*B_0)}_{C_{0,2}} * x^2 +$$

$$+ \underbrace{(A_0*B_3 + \overbrace{A_1*B_2 + A_2*B_1}^{C_{1,2}} + A_3*B_0)}_{C_{0,3}} * x^3 + \ldots$$

Generally:
$\forall i,j:\ 0 \leq i \leq j;\ i+j \leq m+n-2$

$$C_{i,j} \quad = \quad \begin{cases} A_i * B_i & i = j \\ A_i * B_j + A_j * B_i & j = i+1 \\ A_i * B_j + A_j * B_i + C_{i+1,j-1} & j > i+1 \end{cases} \tag{11}$$

The result: $c_k = C_{0,k}$, $\forall k,\ 0 \leq k \leq m+n-2$.

3.2.1 Uniformisation of the recurrence equation

In equation (11), A_i is needed in the computation of $C_{i,j}$ for all values of j, $i \leq j \leq m+n-2-i$, this means a broadcast of A_i. Similarly, A_j is needed in the computation of $C_{i,j}$, $\forall i, 0 \leq i \leq m+n-2-j$. A common method to eliminate broadcast is to pipeline the given value through the nodes where it is needed (see [13]). Thus, we replace A_i with a new variable $A1_{i,i}$, and pipeline it in the direction $(i,j) \rightarrow (i,j+1)$. A_j will be replaced by the variable $A2_{0,j}$ and pipelined through the direction $(i,j) \rightarrow (i+1,j)$. B_i and B_j will be replaced in the same way with B1 and B2, respectively.

We obtain the following uniform recurrence equation:

$$\forall i,j:\ 0 \leq i \leq j;\ i+j \leq m+n-2$$

$$C_{i,j} = \begin{cases} A2_{i,j} * B2_{i,j} & j = i \qquad (12) \\ A1_{i,j} * B2_{i,j} + A2_{i,j} * B1_{i,j} & j = i+1 \qquad (13) \\ A1_{i,j} * B2_{i,j} + A2_{i,j} * B1_{i,j} + C_{i+1,j-1} & j > i+1 \qquad (14) \end{cases}$$

$$A1_{i,j} = \begin{cases} A_i & j = i \qquad (15) \\ A1_{i,j-1} & j > i \qquad (16) \end{cases}$$

$$B1_{i,j} = \begin{cases} B_i & j = i \quad (17) \\ B1_{i,j-1} & j > i \quad (18) \end{cases}$$

$$A2_{i,j} = \begin{cases} A_j & i = 0 \quad (19) \\ A2_{i-1,j} & i > 0 \quad (20) \end{cases}$$

$$B2_{i,j} = \begin{cases} B_j & i = 0 \quad (21) \\ B2_{i-1,j} & i > 0 \quad (22) \end{cases}$$

Note that equations (15), (17), (19), (21) are input equations of the form (3).

Now the input A_i appears in input equation (15) and (19), too. B_i also appears in two input equation. This would mean that we have to input the coefficients of the polynomials A and B twice.

This can be avoided by changing input equation (15) with

$$A1_{i,j} = A2_{i,j} \quad j = i. \tag{23}$$

In the same way, we change (17) by:

$$B1_{i,j} = B2_{i,j} \quad j = i. \tag{24}$$

Table 2. shows the dependencies of the SURE.

Equation	lhs	rhs	Dependence vector
(14)	$C_{i,j}$	$C_{i+1,j-1}$	$(-1,1)$
(16)	$A1_{i,j}$	$A1_{i,j-1}$	$(0,1)$
(18)	$B1_{i,j}$	$B1_{i,j-1}$	$(0,1)$
(20)	$A2_{i,j}$	$A2_{i-1,j}$	$(1,0)$
(22)	$B2_{i,j}$	$B2_{i-1,j}$	$(1,0)$
(14)	$C_{i,j}$	$A1_{i,j}, A2_{i,j}, B1_{i,j}, B2_{i,j}$	$(0,0)$
(23)	$A1_{i,j}$	$A2_{i,j}$	$(0,0)$
(24)	$B1_{i,j}$	$B2_{i,j}$	$(0,0)$

Table 2: Dependence vectors

Note that the dependencies for $A1$ and $B1$ respectively $A2$ and $B2$ are the same. In the following we will only reason about $A1$ and $A2$; $B1$, respectively $B2$, can be handled similarly.

3.2.2 Finding an adequate timing function

According to the method presented in [2], we are looking for affine timing functions with the same linear part for each variable V of the SURE (12)–(22) of the form $t_V = x * i + y * j + z_V$.

For each dependence of Table 2 of the form $V_i(z) \leftarrow V_j(z')$, we are writing the dependency constraint of the form $t_{V_i}(z) > t_{V_j}(z')$. We get:

$$\begin{array}{lcll} C_{i,j} & \leftarrow & A1_{i,j} & \Rightarrow \; t_C(i,j) > t_{A1}(i,j) \\ C_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow \; t_C(i,j) > t_{A2}(i,j) \\ C_{i,j} & \leftarrow & C_{i+1,j-1} & \Rightarrow \; t_C(i,j) > t_C(i+1,j-1) \\ A1_{i,j} & \leftarrow & A1_{i,j-1} & \Rightarrow \; t_{A1}(i,j) > t_{A1}(i,j-1) \\ A2_{i,j} & \leftarrow & A2_{i-1,j} & \Rightarrow \; t_{A2}(i,j) > t_{A2}(i-1,j) \\ A1_{i,j} & \leftarrow & A2_{i,j} & \Rightarrow \; t_{A1}(i,j) > t_{A2}(i,j) \end{array} \tag{25}$$

From the conditions marked with (25) and the computation time minimization condition we get the following system of inequalities:

$$\begin{cases} z_c > z_{A1} \\ z_c > z_{A2} \\ y - x > 0 \\ y > 0 \\ x > 0 \\ z_{A1} > z_{A2} \\ x + y + z_C + z_{A1} + z_{A2} \rightarrow \text{minimal} \end{cases} \tag{26}$$

We also need the constraint that the time function is positive on the domain. Then from (26) we get the solution:

$$\begin{cases} x = 1 \\ y = 2 \\ z_{A2} = 0 \\ z_{A1} = 1 \\ z_C = 2 \end{cases}$$

The time functions are the following:

$$\begin{array}{lcl} t_C(i,j) & = & i + 2j + 2 \\ t_{A1}(i,j) & = & t_{B1}(i,j) = i + 2j + 1 \\ t_{A2}(i,j) & = & t_{B2}(i,j) = i + 2j \end{array} \tag{27}$$

The common linear part of the time functions is $T = (1, 2)$.

3.2.3 Possible allocation functions

Given the timing functions found in section 3.2.2, we are looking for affine allocation functions with the same linear part for each variable V of the SURE (12)–(22) of the form $p_V = \alpha * i + \beta * j + \gamma_V$. The common linear part of the allocation functions is $P = (\alpha, \beta)$. The *general constraint* in our case is:

$$\frac{\alpha}{\beta} \neq \frac{1}{2}. \tag{28}$$

In Table 2 one can look for the dependence vector corresponding to a certain variable. That is $(-1,1),(0,1)$ and $(1,0)$ for variables C, $A1$ and $A2$, respectively.

For a variable V and a corresponding dependence vector θ_V, the dataflow-direction is $(T * \theta_V, P * \theta_V) = ((1,2) * \theta_V, (\alpha,\beta) * \theta_V)$, where the component $T * \theta_V$ indicates the "speed" of variable V, while the component $P * \theta_V$ shows the direction of the V values.

According to the *weak conditions* (only for dependencies of the form $V(z) \leftarrow V(z')$), the node z' should be "close enough" to z, such that $V(z')$ can arrive to the required place in $t_V(z) - t_V(z')$ steps. The conditions are:

$$\begin{array}{lll} C_{i,j} \leftarrow C_{i+1,j-1} & \Rightarrow |p_C(i,j) - p_C(i+1,j-1)| & \leq t_C(i,j) - t_C(i+1,j-1) \\ A1_{i,j} \leftarrow A1_{i,j-1} & \Rightarrow |p_{A1}(i,j) - p_{A1}(i,j-1)| & \leq t_{A1}(i,j) - t_{A1}(i,j-1) \\ A2_{i,j} \leftarrow A2_{i-1,j} & \Rightarrow |p_{A2}(i,j) - p_{A2}(i-1,j)| & \leq t_{A2}(i,j) - t_{A2}(i-1,j) \end{array} \tag{29}$$

In the case of dependencies of the form $V_i(z) \leftarrow V_j(z'), i \neq j$, we can write the so-called *strong dependencies* of the form (30).

$$\|p_{V_i}(z) - p_{V_j}(z')\| = \left\lfloor \frac{t_{V_i}(z) - t_{V_j}(z')}{T * \theta_{V_j}} \right\rfloor P * \theta_{V_j} \tag{30}$$

In our case, these are:

$$\begin{array}{lllll} C_{i,j} \leftarrow A1_{i,j} & \Rightarrow & p_C(i,j) - p_{A1}(i,j) & = & \left\lfloor \frac{1}{2}(t_C(i,j) - t_{A1}(i,j)) \right\rfloor \beta \\ C_{i,j} \leftarrow A2_{i,j} & \Rightarrow & p_C(i,j) - p_{A2}(i,j) & = & (t_C(i,j) - t_{A2}(i,j))\alpha \\ A1_{i,j} \leftarrow A2_{i,j} & \Rightarrow & p_{A1}(i,j) - p_{A2}(i,j) & = & (t_{A1}(i,j) - t_{A2}(i,j))\alpha \end{array} \tag{31}$$

From (29) we get:

$$\begin{cases} |\beta - \alpha| & \leq & 1 \\ |\beta| & \leq & 2 \\ |\alpha| & \leq & 1 \end{cases} \tag{32}$$

From (31) we get:

$$\begin{cases} \gamma_C - \gamma_{A1} &= 0 \\ \gamma_C - \gamma_{A2} &= \alpha \\ \gamma_{A1} - \gamma_{A2} &= \alpha \end{cases} \tag{33}$$

From conditions (28) and (32) we get the set of solutions for α and β:

$$(\alpha, \beta) \in \{(-1,-1), (-1,0), (0,-1), (0,0), (0,1), (1,0), (1,1)\} \tag{34}$$

In (34) the first and the last three solutions are symmetric and the solution $(\alpha, \beta) = (0,0)$ can be excluded because the transformation matrix $\begin{pmatrix} T \\ P \end{pmatrix}$ would be then singular (that means that it would transform some points of D lying on a line into a single point, which is not admitted). Thus we have only three different results:

$$P \in \{(0,1), (1,0), (1,1)\} \tag{35}$$

From (35) and (33) we get three different solutions for adequate allocation functions corresponding to the given timing functions:

$$p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = p_{A2}(i,j) = p_{B2}(i,j) = j \tag{36}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i \\ p_{A2}(i,j) = p_{B2}(i,j) = i-1 \end{cases} \tag{37}$$

$$\begin{cases} p_C(i,j) = p_{A1}(i,j) = p_{B1}(i,j) = i+j \\ p_{A2}(i,j) = p_{B2}(i,j) = i+j-1 \end{cases} \tag{38}$$

3.2.4 Mappings to different systolic arrays

We apply the space-time transformation onto the SURE (12)–(22) according to the timing functions from (27) and allocation functions from (37). That is:

$$\begin{array}{lllll} t_C(i,j) &= i+2j+2 & & p_C(i,j) &= i \\ t_{A1}(i,j) &= t_{B1}(i,j) &= i+2j+1 & p_{A1}(i,j) &= p_{B1}(i,j) &= i \\ t_{A2}(i,j) &= t_{B2}(i,j) &= i+2j & p_{A2}(i,j) &= p_{B2}(i,j) &= i-1 \end{array}$$

We have chosen this transformation, because this is the one the application of which results in an online array. The transformed SURE:

$$\forall t, p: \; p \geq 0; 3p+2 \leq t \leq -p+2(m+n)-2; \frac{t-p}{2} \in \mathcal{Z}$$

$$\overline{C}_{t,p} = \begin{cases} \overline{A2}_{t-2,p-1} * \overline{B2}_{t-2,p-1} & t = 3p+2 \quad (39) \\ \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} & t = 3p+4 \quad (40) \\ \overline{A1}_{t-1,p} * \overline{B2}_{t-2,p-1} + \overline{A2}_{t-2,p-1} * \overline{B1}_{t-1,p} + & \\ \quad + \overline{C}_{t-1,p+1} & t > 3p+4 \quad (41) \end{cases}$$

$$\overline{A1}_{t,p} = \begin{cases} \overline{A2}_{t-1,p-1} & t = 3p+2 \quad (42) \\ \overline{A1}_{t-2,p} & t > 3p+2 \quad (43) \end{cases}$$

$$\overline{B1}_{t,p} = \begin{cases} \overline{B2}_{t-1,p-1} & t = 3p+2 \quad (44) \\ \overline{B1}_{t-2,p} & t > 3p+2 \quad (45) \end{cases}$$

$$\overline{A2}_{t,p} = \begin{cases} A_{\frac{t}{2}} & p = 0 \quad (46) \\ \overline{A2}_{t-1,p-1} & p > 0 \quad (47) \end{cases}$$

$$\overline{B2}_{t,p} = \begin{cases} B_{\frac{t}{2}} & p = 0 \quad (48) \\ \overline{B2}_{t-1,p-1} & p > 0 \quad (49) \end{cases}$$

Note that this transformation is not unimodular, for this reason the domain of the system (39)-(49) is sparse (see the $(t-p)/2 \in \mathcal{Z}$ condition). The resulted array can be optimised: by merging two neighbouring PEs, we get the online array presented in Section 3.1.

As a conclusion, in this case it is obvious that we have succeeded to design the same systolic array in a more "elegant" and efficient way using the functional approach.

4 Conclusions

In this paper we have compared two automatic systolic array design methods: the space-time transformation methodology and the functional-based method.

We presented different solutions of a representative problem, using both methods, in order to demonstrate the main characteristics, differences, advantages and eventual disadvantages of the two design methods.

The space-time transformation method is obviously the most widespread methodology, and also the most complex one. However, besides its numerous advantages it also has some drawbacks, too: the formulation of the problem as a SURE may be sometimes of serious difficulty, complex computations on the whole index space (repetitions), in order to find an adequate timing function, a complex linear programming problem has to be solved.

Other methods that imply more simple computations (for example concerning the computation of the timing function [7]) only work for a fixed size.

The most relevant advantage of our functional-based method is, that exploiting the symmetric structure of the systolic array, in fact we only have to analyse the behaviour of the first PE. The method also works for parametrized problems (the size of the problem does not have to be fixed in advance). Moreover, the design process consists basically in the application of rewrite rules, thus its implementation is relatively simple.

For the moment, the method is applicable only to linear systolic arrays; this is, however, for practical reasons (efficiency, reliability and ease of implementation) the most popular class of systolic arrays. It relies on a formal analysis performed in advance, thus it is less general than the space-time transformation method.

The considerations above make us believe that it is worth working on the improvement of the functional-based design method by analysing other classes of systolic arrays, too.

References

[1] J.-M. Delosme, I. C. F. Ipsen. Systolic array synthesis: computability and time cones, in *Parallel algorithms & architectures (Luminy, 1986)*, pp. 295–312, North-Holland, 1986.

[2] P. Feautrier, Some efficient solutions to the affine scheduling problem, Part I : One-dimensional Time, *Int. J. of Parallel Programming*, **21**, 5 (1992) 313–348.

[3] J. A. B. Fortes, D. I. Moldovan, Data broadcasting in linearly scheduled array processors, *11th Annual Symp. on Computer Architecture*, 1984, pp. 224–231.

[4] P. Gribomont, V. Van Dongen, Generic systolic arrays: a methodology for systolic design, *Lecture Notes in Computer Science*, **668**, 1992, pp. 746–761.

[5] T. Jebelean, *Systolic multiprecision arithmetic*, PhD Thesis, RISC-Linz Report 94-37, April 1994.

[6] T. Jebelean, L. Szakács. Functional-Based Synthesis of Systolic Online Multipliers, *Proceedings of SYNASC-05 (International Symposium on*

Symbolic and Numeric Scientific Computing), (eds. D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Ciortaş, A. Abraham, M. Paprzycki), IEEE Computer Society, 2005, pp. 267–275.

[7] L. Kazerouni, B. Rajan, R. K. Shyamasundar, Mapping linear recurrence equations onto systolic architectures, *International Journal of High Speed Computing (IJHSC)*, **8**, 3 (1996) 229–270.

[8] H. W. Nelis, E. F. Deprettere, Automatic design and partitioning of systolic/wavefront arrays for VLSI, *Circuits, systems, and signal processing*, **7**, 2 (1988) 235–251.

[9] L. Ruff, T. Jebelean, Functional–based synthesis of a systolic array for gcd computation, *Lecture Notes in Computer Science*, **4449**, 2007, pp. 37–54.

[10] L. Ruff, Functional–based synthesis of unidirectional linear systolic arrays, *Pure Math. Appl.*, **17**, 3–4 (2006) 419–443.

[11] L. Ruff, Optimisation of bidirectional systolic arrays with sparse input by "folding", *10th Symposium on Programming Languages and Software Tools, SPLST07*, Dobogókő, Hungary, Eötvös University Press (eds. Zoltán Horváth, László Kozma, Viktória Zsók), 2007, pp. 420–432.

[12] P. Quinton, *The systematic design of systolic arrays* in *Automata Networks in Computer Science*, (eds. F. F. Soulie, Y. Robert, M. Tchuente), Manchester University Press, 1987, pp. 229–260.

[13] P. Quinton, V. Van Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays, *Journal of VLSI Signal Processing* **1**, 2 (1989) 95–113.

[14] P. Quinton, Y. Robert. *Systolic algorithms and architectures*, Prentice-Hall, 1990.

[15] S. W. Song. *Systolic algorithms: concepts, synthesis, and evolution*, Temuco, CIMPA School of Parallel Computing, Chile, 1994.

[16] L. Szakács. *Automatic design of systolic arrays: a short survey.* Technical report no. 02-27 in RISC Report Series, University of Linz, Austria, December 2002.

[17] S. Wolfram. *The Mathematica Book*, 5th edition, Wolfram Media, 2003.

Received: April 28, 2009

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 259–266

Solving routing in telecommunication problems using sensitive ants

Alexandra-Roxana Tănase
Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Kogălniceanu 1, RO-400084 Cluj-Napoca, Romania
email: alexandra.roxana.tanase@gmail.com

D. Dumitrescu
Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Kogălniceanu 1, RO-400084 Cluj-Napoca, Romania
email: ddumitr@cs.ubbcluj.ro

Abstract. Ants, which have a sensitive reaction to pheromone, are considered to be agents for the metaheuristic called *Sensitive ACS (SACS)*. Within *SACS* model, each ant is endowed with a pheromone sensitivity level, which allows certain types of responses to pheromone trails. Such an artificial system, based on emergent behavior promise to generate engineering solutions to distributed systems management problems, for example, in telecommunication networks. A Sensitive Ants Algorithm for Routing (SAR), based on SACS model is developed for solving networks communication problems. The aim of this is to provide a comparison between AntNet Algorithm, based on ACO model and SAR, by giving a formal and comprehensive systematization of the subject.

1 Introduction

Ants are social insects, which have captured the attention of many scientists because of the high structuration level from their colonies. Ant as a single

AMS 2000 subject classifications: 90C32, 90C59, 90B18
CR Categories and Descriptors: I.2.8 [Problem Solving Control Methods and Search]: Subtopic - Heuristic methods, I.1.12 [Distributed Artificial Intelligence]: Subtopic - Multiagent systems
Key words and phrases: sensitivity, metaheuristics, communication networks, multiagent systems

individual has a very limited effectiveness. But as a part of a well-organised colony, it becomes a powerful agent. Taking into account that a human brain has about 10 billion neurons and ants only have $250,000$, one may ask how they can perform such amazing tasks when they are in a collective body. Their real power resides in their colony brain. Even though the individuals are limited in number of neurons, a collective of $40,000$ would have approximately the number of neurons that a human brain has [9].

Stigmergy is defined as a particular form of indirect communication in a self-organizing emergent system used by social insects to coordinate their activities [8]. Stigmergic information is local: it can only be accessed by those insects that visit the place in which it was released. In many ant species, ants walking from or to a food source, deposit on the ground a substance called *pheromone*. Generally, insects are known to make more use of pheromones for diverse tasks such as reproduction, alert, identification, navigation and aggregation [8,9].

Sensitive ants have different degrees of perceiving the presence of pheromone. This is suggested by the Pheromone Sensitivity Level (PSL), whose value is between 0 and 1 [2,3,4]. The idea of using ants for solving routing problems is not new: for instance, the ACO metaheuristic provides good results in this area [5]. The paper aims to provide an algorithm where sensitive ants can be used for routing. Numerical experiments indicate the potential of the proposed algorithm.

2 Routing information

Routing can be characterized by the following general way. Let the network be represented in terms of directed, weighted graph: $G = (V, E)$, where each node in the set V represents a processing and forwarding unit and each edge in E is a transmission system with some capacity/bandwidth and propagation characteristics.

Data traffic originates from one node and can be directed to another node (unicast traffic) or to a set of nodes (multicast traffic) and/or to all the other nodes (broadcast traffic). The nodes between sources and destinations are called intermediate or relay nodes. The node, from where the traffic flow originates is also called source, while the nodes to which traffic is directed are the final end-points, or destinations [1,5,6].

The characteristics of the routing problem make it well suited to be solved by a mobile multi-agent approach. The ideea of using ants in routing problems is not new, i.e. M. Dorigo and G. Di Caro originally proposed four ACO

algorithms for adaptive agent-based routing. They are the following: AntNet, and some improvements of it: AntNet-FA, AntNetSELA and AntHoc Net [5,6,7].

3 Sensitive ants model used for routing in telecommunication networks

The Sensitive ACS metaheuristic (SACS) [4] uses different reactions of sensitive ants to the pheromone trail. The model implements both exploration and exploitation search for solving problems with a high degree of complexity.

Within the proposed model, each agent is endowed with a *pheromone sensitivity level* (*PSL*), which is expresed by a real number in the unit interval $[0, 1]$. Agents with low PSL values will normally choose very high pheromone levels moves. They are more independent and they are very good environment explorers [4,5]. Agents with high PSL value will follow any pheromone trail. They are able to exploit the already indentified paths.

The ACS and SACS models were implemented for solving the *Generalized Traveling Salesman Problem (GTSP)*. The search space was an n-node, undirected graph, $G = (V, E)$ [4]. To favour the selection of an edge, (i, j) with a high pheromone value, τ, and a high visibility value, $\eta = \frac{1}{c_{ij}}$, the transition probability, p_{iu}^k is considered:

$$p_{iu}^k = \frac{[\tau_{iu}(t)] \cdot [\eta_{iu}(t)]^\beta}{\sum_{o \in J_i^k} [\tau_{io}(t)] \cdot [\eta_{io}(t)]^\beta}, \tag{1}$$

where β is a parameter used for tuning the relative importance of edge cost (c_{ij}) in selecting the next node.

The main purpose of *Sensitive Ants Algorithm for Routing (SAR)* is to point out a comparison between multi-agents with random PSL and multi-agents with a fixed (global) PSL. The algorithm refers to finding a minimum-cost path from a certain source to a randomly chosen destination, by using sensitive ants.

The model's main purpose is to improve routing in telecommunication networks. The network is represented by a weighted graph $G = (V, E)$, where each node represents a processing and forwarding unit for every ant passing by, and each edge in E is a transmission system with propagation characteristics. Ants are used to explore the search space so that data packets can reach the destination taking into account the improvements made by the ants. When all the routing tables are updated with the minimum costs, the algorithm stops.

Two classes of ants are used for this purpose: the first one is the management ants. Here, three types of ants are considered: exploration ants, which have the role to explore the unoriented graph and find more candidate routes between the nodes. Another type of ants are the message ants or response ants (they work as backward ants). Error ants appear if one node or edge is deleted. The second class are the exploitation ants, which only take into account the improvements made by management ants, and because of this, they only choose the edges with the low cost. They are also called data packets [5,6].

At iteration $t+1$ every ant moves to a new node and the parameters controlling the algorithm are updated. Each edge is labelled by a trail intensity; let $\tau_{ij}(t)$ be the trail intensity for the edge (i, j) at iteration t. At each time unit evaporation takes place and its value is between 0 and 1. Every ant decides which node is the next move with a probability, which is based on the distance to that node and on the amount of trail intensity on the edge connecting the nodes. The inverse of distance from a node to the next node is called visibility and is denoted by the formula: $\eta = \frac{1}{c_{ij}}$, where c_{ij} is the cost on the edge (i, j). To favour the selection of an edge that has a high pheromone value, τ, and high visibility value, η, a transition probability is proposed:

$$p_{iu}^{k} = \frac{[\tau_{iu}(t)] \cdot \eta_{iu}(t)^{PSL}}{\sum_{o \in J_i^k} [\tau_{io}(t)] \cdot \eta_{io}(t)^{PSL}}. \tag{2}$$

(2) expresses the probability of ant k from the node i to choose the next node u; PSL represents the pheromone sensitivity level for an ant. It is used for tuning the relative importance of the quantity of pheromone on the edge. J_i^k are the unvisited neighbors of node i.

The algorithm can be resumed as follows: from each network node, ants are randomly launched towards specific destination nodes. The agent generation processes happen concurrently and asyncronously; the best (minimum) path is searched from a certain source to a randomly chosen destination, using ants. Destination is chosen randomly. The agents moving from their source to destination node are called *forward ants*. Every forward ant has a taboo list: where it has the source node, the destination node, the PSL value and the intermediate nodes, between its source and destination; the pheromone trail is updated on every edge taking into account the evaporation rate and the number of ants which pass on the edge in that moment of time. When an ant arrives at destination it is deleted and a *backward ant* (response ant) is created, which goes back following the same path as before, but in the opposite

direction. If an ant does not reach the destination and the maximum Time-To-Live (TTL) has expired, then it is also destroyed. The pheromone on the trail is updated as follows:

$$\tau_{i,j}(t+1) = (1-\rho)\cdot\tau_{i,j}(t) + \ln(N_{i,j}(t)+1), \tag{3}$$

where $N_{i,j}$ is the number of ants which pass on the edge (i,j) at iteration t; ρ represents the evaporation rate; if an ant arrives at destination, it is deleted and a backward ant (response ant), $B_{d\to s}$, is created and goes back to the same path $P_{s\to d} = [s, v_1 v_2 \ldots, d]$ as before, but in the opposite direction.

In AntNet technique [6], the routing decision policy is adopted by forward ants in choosing the next node (hop). Therefore, the ant's decision is influenced by the entries in the pheromone table, the status of the local link queues (heuristic values), and it depends on the memory of the already visited nodes. At each intermediate node k, the forward ant $F_{s\to d}$ heading to its destination d must select the neighbor node $n \in N_k$ to move to. The probability p_{nd} assigned of each neighbor n of being selected as next hop is:

$$y = \begin{cases} p_{nd} = \frac{\tau_{nd}+\alpha\cdot l_n}{1+\alpha(|N_k|-1)} & \text{if } \forall n \in N_k \wedge n \notin V_{s\to k} \\ p_{nd} = 0 & \text{otherwise,} \end{cases} \tag{4}$$

where τ_{nd} are the values of the pheromone stochastic matrix τ_k corresponding to the estimated goodness of choosing n as the next hop for destination d; l_n is a $[0,1]$ normalised value proportional to the length $q_{n'}$; $V_{s\to d}$ is the list of the nodes visited so far and $\alpha \in [0,1]$ weighs the relative importance of the heuristic correction with respect to the pheromone values stored in the pheromone matrix [5].

4 Numerical experiments

SAR paradigm presents a simulation on a routing network, which is represented by a connection-less graph $G = (V, E)$. The number of nodes is denoted by N, where $N \subseteq V$. NFSNet, one of the graphs used for the simulation, is a WAN composed of 14 nodes and 21 bi-directional links with a bandwidth of 1.5 Mbit/s [5].

SAR showed good performance under the NSFNet graph. All reported data are averaged over 10 trials. The best results were rather obtained using sensitive ants, with a random PSL, than ants with a global PSL. Costs on

PSL	No of steps	No of ants	No of packets	No of delivered packets
random	**2524.8**	**274.5**	**277.8**	**255.4**
0.2	2653.4	291	290.7	266.9
0.5	2795.8	304.6	306.8	283.8
0.7	2781.5	302.8	306	282.3
0.9	2912.2	316.8	323.1	300.1
1	3024.8	335.3	327.5	304.9

Table 1: NFSNet: Comparative results for different PSL values in SAR

the edges were computed taking into account only the propagation delays: costs range from 4 to 20 msec. TTL was set at 255 sec. As it can be seen from Table 1, better results were obtained using a random PSL value, so sensitive ants (with PSL values between 0 and 1) are better than ants which have a global PSL value, i.e., 0.2, 0.5, 0.7, 0.9 or 1. In ACO metaheuristic, every ant has the same global value for the PSL. Time of the simulation is proportional with the number of steps; a step occurs at every 15 msec. As it can be seen from Table 3 the percentage of delivered packets obtained using the SAR paradigm is bigger than the percentage of delivered packets from the AntNet results. From this point of view, SAR obtained better results. However, AntNet was always, within the statistical fluctuations, among the best performing algorithms. AntNet showed a robust behaviour, being able to rapidly reach a good stable level in performance. Moreover, the proposed algorithm has a negligible impact on network resources and a small set of robustly tuneable parameters. Its features make it an interesting alternative to classical shortest path algorithms.

PSL	No of Steps	Time (sec)
random	**2524.8**	**37.8**
0.2	2653.2	39.7
0.5	2795.8	41.9
0.7	2781.5	41.7
0.9	2912.2	43.6
1	3024.8	45.3

Table 2: Time results of SAR on NSFNet WAN

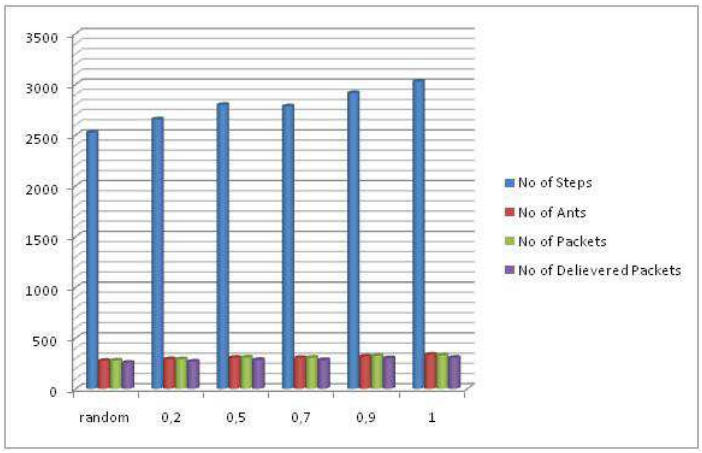


Figure 1: NFSNet: A comparison between different PSL values

Application	Percentage of delivered packets
SAR	92.25 %
AntNet	90 %

Table 3: Delivered packets on SAR vs AntNet

5 Conclusions and future work

The Sensitive Ants Paradigm for Routing (SAR), based on the model SACS is presented. SAR obtained good results in routing taking into account the NSFNet graph which was also used by M. Dorigo et al. in AntNet paradigm. From AntNet statistics some numerical results can be pointed out: the number of generated ants is $567,000$, the number of received ants is $107,000$ and number of dropped ants is $429,000$ [9]. The computational results concerning the SAR model show that sensitive ants achieve better results than ants with global PSL because ants with a random PSL value are able to make a more efficient exploration of the proposed WAN. These results may be improved by considering different parameter settings.

Future work focuses on the improvement of the proposed SAR model, by quantifying specific roles of the stigmergetic communication in ant colonies in order to bring better results in routing research.

References

[1] B. Baran, R. Sosa, AntNet: *Routing algorithm for data networks based on mobile agents*, A DIPRI research grant of the National University of Asuncion-Paraguay, 2000.

[2] C. Chira, C.-M. Pintea, D. Dumitrescu, Sensitive stigmergic agent systems, adaptive and learning agents and multi-agent systems (ALAMAS), *MICC Technical Report Series*, Maastricht, The Netherlands, number 07-04 (2007) 51–57.

[3] C. Chira, C.-M. Pintea, D. Dumitrescu, Sensitive ants: Inducing diversity in the colony, *International Workshop on Nature Inspired Cooperated Strategies for Optimization, NICSO*, Puerto de la Cruz, Tenerife, 2008.

[4] C. Chira, C.-M. Pintea, D. Dumitrescu, Sensitive ant systems in combinatorial optimization, *Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT2007*, Cluj-Napoca, Romania, 2007, pp. 185–192.

[5] G. Di Caro, *Ant colony optimization and its application to adaptive routing in telecommunication networks*, PhD Dissertation, Bruxelles, 2004.

[6] G. Di Caro, M. Dorigo, AntNet: distributed stigmergetic control for communications networks, *IRIDIA- Journal of Artificial Intelligence Research* **9** (1998) 317–365.

[7] M. Dorigo, M. Birattari, T. Stützle, Ant colony optimization artificial ants as a computational intelligence technique, *IRIDIA-Technical Report Series*, 2006.

[8] M. Dorigo, E. Bonabeaub, G. Theraulaz, *Ant algorithms and stigmergy*, IRIDIA Elsevier Science BV, 2000.

[9] P. E. Merloti, *Optimization algorithms inspired by biological ants and swarm behavior*, 2295 Cabo Bahia, Chula Vista, CA 91914 United States of America, 2004.

[10] `http://www.aco-metaheuristic.org/aco-code/public-software.html`

Received: May 2, 2009

Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 267–274

Solution concepts for coevolutionary 2-period cumulated games

Radu M. Berciu
Babeş-Bolyai University of Cluj-Napoca, Romania
Department of Computer Science
email: raduberciu@yahoo.com

Abstract. Based on strategic games, a new type of dynamic game has been introduced, the n-period cumulated game (n-PCG), where players engage in a repetitive play of a constituent strategic game for n number of times (an accumulation period), without receiving their payoff after each stage of the game, but only the cumulated payoffs of all the stages of the game at the end of the accumulation period. Some solution concepts for n-PCGs are discussed, namely subgame perfect equilibria and Nash equilibria. Then a two-population based genetic algorithm is introduced in order to find these equilibria in 2-period cumulated games.

1 Introduction

Genetic algorithms are a well-known optimization method introduced by John Holland in the early 1970s [3]. Nash strategy [4, 5] is the most commonly encountered solution concept in game theory. The idea to use genetic algorithms together with the Nash strategy concept, such that the algorithm searches for the Nash equilibrium, belongs to Sefrioui [10]. As described in [7] at each generation a player improves its strategy with respect to the other players' best strategies of the previous generation: Nash equilibrium is reached when no player can improve its strategy. Based on this, NCA (Nash Coevolution

AMS 2000 subject classifications: 68T20, 91A25, 91A20, 91A18, 91A10, 91A05
CR Categories and Descriptors: F.1.0 [Computation by Abstract Devices]: General, F.2.0 [Analysis of Algorithms and Problem Complexity]: General, I.2.8 [Problem Solving, Control Methods, and Search]: Subtopic - Heuristic methods.
Key words and phrases: n-period cumulated game, subgame perfect equilibrium, Nash equilibrium, coevolution, game theory

Algorithm), a two population based genetic algorithm, finds subgame perfect equilibria in 2-period cumulated games interpreted as extensive games with imperfect information.

2 $\mathfrak{n}$-period cumulated game ($\mathfrak{n}$-PCG)

The notion of cumulated game basically means that, for instance, having two individuals playing a number of strategic games, there exists a mechanism that sums and withholds the benefits until the players have completed $\mathfrak{n}$ plays of the strategic game; $\mathfrak{n}$ is called the length of the accumulation period. Their accumulated benefits are reported only after $\mathfrak{n}$ stage games. Our goal is to model the situations where players engage in games, in which they have different knowledge of the previous plays, and in which they receive their payoff after different accumulation periods. For example, consider the model of the relation between an employer and an employee where the employer agrees to pay the employee a small amount of money every two weeks even though he may gain benefits from the employee's work at every two months. The accumulation period of the employee is two weeks and the accumulation period of the employer is two months. However, here the focus is on the notion of $\mathfrak{n}$-period cumulated games (i.e. equal accumulation periods), and on the analysis of four models of 2-PCGs distinct by their elementary game.

The notion of cumulative benefit game already appeared in [11]. There the author argued that in the context of repetitive games and a strong temporal discounting, accumulation can promote a cooperative strategy. However, an $\mathfrak{n}$-PCG is a not a repetitive game even though it is a dynamic one.

2.1 $\mathfrak{n}$-period cumulated games as strategic games

An $\mathfrak{n}$-PCG can be described as a strategic game. Take for example the strategic game in Fig. 1(a) and an accumulation period of two, i.e., after the first stage of the game no payoff is reported, and after the second stage of the game the players receive their cumulated payoffs over the two plays.

This 2-PCG can be viewed as a strategic game, as depicted in Table 1. In this game the pure strategies for Player 1 *(AA, AB, BA , BB)* and Player 2 *(CC, CD, DC, DD)* are composed by the pure strategies of the initial strategic game. The payoffs are the cumulated payoffs over a period of the cumulated game. For example, if the first player (the row player) chooses AB and the second player chooses DD, then the payoff is 2 for the first player and 6 for the second player, i.e., in the first stage of the game they play AD, which has an

unreported payoff of $(1, 5)$, and in the second stage they play *BD*, which has an unreported payoff of $(1, 1)$, thus having an accumulated reported payoff of $(2, 6)$.

	CC	CD	DC	DD
AA	6,6	4,8	4,8	2,10
AB	8,4	4,4	6,6	*2,6*
BA	8,4	6,6	4,4	2,6
BB	10,2	6,2	6,2	2,2

Table 1: A 2-PCG based on the strategic game from Fig. 1(a)

2.2 n-period cumulated games as dynamic games

An image of the described game is that of a dynamic game. A dynamic game captures the idea that players act sequentially and can incorporate information about earlier moves in the game in choosing their next move. Even though a stage game (called constituent game) is being repeatedly played, the difference between an n-Period Cumulated Game and a finitely repeated game is that the benefit/payoff is reported only after an accumulation period.

The n-PCG can be represented as an extensive game. For instance, the static game in Fig. 1(a) is equivalent to the dynamic game represented in Fig. 1(b). The dashed line between some nodes means that the current decision maker does not know in which state she is at. Therefore, the extensive game can be viewed as a 2×2 strategic game where players act simultaneously. This is the case of extensive games with imperfect information, i.e., each player, when making a decision, is not perfectly informed about the events that have previously occurred [9, 6].

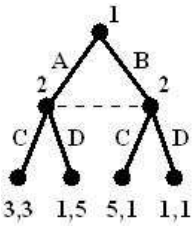
3 Solution concepts for n-period cumulated games

Interpreting the n-period cumulated game as a new strategic game, the apropriate solution concept is that of Nash equilibrium [4, 5], the most commonly encountered solution concept in game theory. The Nash equilibrium and most of its variants express the idea that each player individually maximizes its utility [1].

Interpreting the n-period cumulated game as an extensive game, the adequate solution concept is that of subgame perfect equilibrium. A subgame is

	C	D
A	3,3	1,5
B	5,1	1,1

(a) the game in its normal form



(b) the game in its extensive form

Figure 1: Views of the same 2×2 strategic game

a subtree from a game's directed tree that has the properties: it begins at a decision node, it gives the initial player all the decisions that have been made until that time, and it contains all the decision nodes that follow the initial node. The subgame perfect equilibrium (or subgame perfect Nash equilibrium) is a refinement of Nash equilibrium (NE) that induces a NE in every subgame (subtree) of that game. In an $\mathfrak{n}$-PCG there are subgames with simultaneous decisions, therefore all possible Nash equilibria of that subgames may appear in a subgame perfect Nash equilibrium.

3.1 Numerical experiments

After excluding the games that are strategically trivial in the sense of having equilibrium points that are uniquely Pareto-efficient, there remain four archetypal 2×2 games: prisoner's dilemma, chicken (Hawk-Dove), battle of the sexes, and leader [8]. These games where taken as constituent games for 2-PCGs. The theoretical solving of these 2-PCGs is described by solving the 2-PCG based on the Hawk-Dove Game.

3.1.1 Hawk-Dove game

The strategic form of this game depicted in Table 2 has two Nash equilibria: *DH* and *HD*. The game tree from Fig. 2 represents the 2-PCG based on this Hawk-Dove game. It has five subgames, four of them are the extensive form of the strategic constituent game (Table 2), and one is the whole tree. Working through backwards induction, the four subgame perfect equilibria are found. These are the dotted paths from root to leaves in Fig. 2.

An $\mathfrak{n}$-PCG can be described as a strategic game, hence the 2-PCG depicted in Fig. 2 can be represented as the strategic game in Table 3. Using the

best response strategy, the Nash equilibria of this strategic game are found. They are the same with those of the subgame perfect equilibria found before. Certainly, this is due to the manner in which the game was constructed, but nevertheless, this gives us a static insight of a dynamic process.

	D	H
D	3,3	*1,5*
H	*5,1*	0,0

Table 2: The Hawk-Dove game

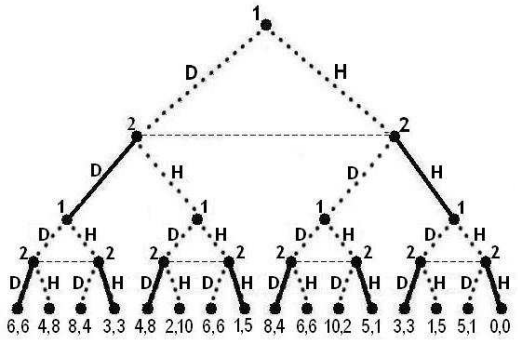


Figure 2: Game tree for a 2-PCG based on the Hawk-Dove game from Table 2. The Nash equilibria of every subgame are marked with dots. The dotted paths from root to leaves are the subgame perfect equilibria

	CC	CD	DC	DD
AA	6,6	4,8	4,8	*2,10*
AB	8,4	3,3	*6,6*	1,5
BA	8,4	*6,6*	3,3	1,5
BB	*10,2*	5,1	5,1	0,0

Table 3: A 2-PCG based on the strategic game from Table 2

Similar with the Hawk-Dove game based 2-PCG the solutions for the 2-PCGs with the constituent strategic games prisoner's dilemma, battle of the sexes and leader are found.

4 NCA (Nash coevolution algorithm)

A coevolution algorithm for finding subgame perfect equilibria in 2-period cumulated games is proposed. This algorithm is called Nash coevolution algorithm (NCA).

4.1 Encoding a strategy

With NCA, a chromosome represents a strategy that is a function of the state of the game: every possible state (history) has one slot $\mathtt{i}$,$0 \leq \mathtt{i} < \mathtt{s}$, in the $\mathtt{s}^{\mathtt{m}}$ string that codes the move the player will take if she uses that strategy and she encounters the history with index $\mathtt{i}$; $\mathtt{s}$ is the number of states of the constituent strategic game (stage game), $\mathtt{m}$ is the length of the history (the number of stage games that the player recalls) and the hypothetical game is there to induce the first actions that the player will take. The difference from [2] is that here there is no restriction for $\mathtt{m}$.

4.2 Fitness assignment

Every player is represented by a population, which maximizes the player's payoff at every generation. The populations evolve by optimizing at each step their chromosomes using the other population's best $\mathtt{q}$ chromosomes from the previous generation [7], i.e. each player has a population that tries to maximize its payoff using the best solutions found by the other player's population one generation before. The fitness of each chromosome will be the mean payoff it receives after playing with each of the other player's best chromosomes from the previous generation.

In games with imperfect information the notion of subgame perfect equilibrium may require mixed strategies, for instance the game of matching pennies. This implementation does not support these types of strategies.

5 Convergence, stability and more

NCA was tested using fixed and variable parameters. The fixed one where:

- *population size = 40* strategies for each player;
- a number of *700 to 1000 generations* for each run;
- *recombination probability = 0.2%*;
- *accumulation period = 2* for both players.

The other parameters were:

- *exchange* $\in \{5, 20, 40\}$, where *exchange* is the number of individuals used to evaluate a strategy;
- *probability of mutation*, *p.m.* $\in \{0.1\%, 0.2\%, 0.3\%\}$;
- *history length* $\in \{1, 2\}$.

They varied one at a time, thus allowing an objective impact analysis.

Different parameters configuration gave an insight into the stability of NCA and proved empirically that

$$\mathtt{stability} \approx \frac{1}{\mathtt{exchange} \times \mathtt{p.m.}}.$$

The convergence time to an equilibrium is between 40 and 60 generations for all the configurations of the Leader game. For the Battle of sexes, for the test configurations 1 to 9 (*history length of 1*) an equilibrium was found in less than 70 generations in more than 90% of the cases; for the last three test configurations (*history length of 2*), the convergence time to an equilibrium is under 40 generations. For the Prisoner's dilemma game, it needs less than 30 generations to achieve its equilibrium in all the configurations. For the Hawk-Dove game the algorithm achieves an equilibrium under 70 generations with a probability of 87%.

6 Conclusion and future work

Based on strategic and repetitive games, a new dynamic game called n-period cumulated game has been introduced. Solution concepts for these n-PCGs, namely subgame perfect equilibria and Nash equilibria, were analyzed. Then 2-PCGs were tested in a coevolutionary environment (provided by the developed NCA) based on archetypal 2×2 games. The main result was that players eventually play a Nash equilibrium in every subgame, thus a subgame perfect equilibrium, even though they are not aware of their payoff until a certain number of stage games are played.

An interesting focus for future work is that of working with different levels of accumulation periods, testing the outcome of having the players engage in these basically different games. Another challenging issue can be that of letting two players play a number of games between themselves, thus transforming the n-PCG into a repetitive one; for instance, preliminary tests show that in the repetitive 2-PCG based on the prisoner's dilemma the players start by cooperating.

References

[1] R. Aumann, What is game theory trying to accomplish? *Frontiers in Economics* (1985) 909–924.

[2] R. Axelrod, The evolution of strategies in the iterated prisoner's dilemma, *Genetic Algorithms and Simulated Annealing* (1985) 32–41.

[3] J. H. Holland, *Adaptation in natural and artificial systems*, The MIT Press, Cambridge, MA, USA, 1992.

[4] J. F. Nash, Equilibrium points in n-person games, *Proc. Nat. Acad. Sci.* **36** (1950) 48–49.

[5] J. F. Nash, *Noncooperative games*, PhD Thesis, Princeton University, 1951.

[6] N. Nisan, T. Roughgarden, E. Tardos, V. V. Vazirani, *Algorithmic Game Theory*, Cambridge University Press, 2007.

[7] J. Periaux, M. Sefrioui, Nash genetic algorithms: examples and applications, *Proc. of the 2000 Congress on Evolutionary Computation*, 1999, pp. 509–516.

[8] A. Rapoport, Exploiter, leader, hero, and martyr: the four archetypes of the 2x2 game, *Behavioral Science* **12**, 2 (1967) 81–84.

[9] A. Rubinstein, M. J. Osborne, *A course in game theory*, The MIT Press, Cambridge, MA, USA, 1994.

[10] M. Sefrioui, *Algorithmes evolutionnaires pour le calcul scientifique: Application a l'electromagnetisme et a la mecanique des fluides*, PhD Thesis, Université Pierre et Marie Curie, Paris, 1998.

[11] D. W. Stephens, Cumulative benefit games: Achieving cooperation when players discount the future *J. Theoretical Biology*, **205**, 1 (2000) 1–16.

Received: May 2, 2009

DYNAMIC MODELING OF THE HUMAN HEART

Sándor Miklós Szilágyi
Scientia Publishing House (http://www.scientiakiado.ro), Cluj-Napoca, 2009.
ISBN 978-973-1970-11-0

Sudden cardiac death, mostly caused by ventricular fibrillation, is responsible for at least five million deaths in the world each year. Despite several years of research, the responsible mechanisms for ventricular fibrillation are not yet well understood.

As most simulation studies are limited to planar simulations, the responsible mechanisms for the spatial phenomenon of ventricular fibrillations are not elucidated by far. It would be important to know how the most important heart parameters, such as the heart's size, geometry, mechanical and electrical state, tissue homogeneity and fiber structure, affect the development of ventricular fibrillation. The main difficulty in the development of a quantitatively accurate simulation of an entire three-dimensional human heart consists in the limited number of heart models, and the rapidly varying, highly localized fronts produced within the human heart muscle. Moreover, in pathological cases, the most relevant parameters of the conduction properties are significantly altered and they can produce spiral, self-inducing depolarization waves, which often transform into ventricular fibrillation. These regional alterations of the conduction properties are mostly patient-specific. To approach towards the solution of these problems, a complex modeling of the heart is necessary.

This book focuses on the adaptive ECG analysis and heart modeling. In the first chapter, a detailed ECG processing method is presented, which uses an iterative filtering and parameter estimation technique to obtain the aimed results. This algorithm is capable of properly adapting itself to patient-specific demands. Instead of the direct or transformation based processing methods, which cannot cover the uncommon waveforms even if using large sample databases, this feature-specific ECG estimation method can handle almost all perturbed waveforms. The signal estimation and efficient compression processes are highly correlated by the a priori determined medical parameters. The advanced distortion analysis allows to adaptively modify the compression rate, assuring a predefined quality of the biological parameters.

In the second chapter a dynamic heart model is presented, which is capable OF simulate almost all important pathological cases. The depolarization waveform is simulated at a dynamically, locally and temporally variable resolution that yields a fast simulation, keeping the estimation error at a reasonable level. The adaptive mesh refinement algorithm establishes the proper local res-

olution based on the first derivative of the intracellular potential. The whole method is highly parallelized, so video cards can efficiently perform the bulk of the calculation.

In the third chapter the simultaneous processing of the ECG signal and echocardiography image sequence determines the latent connection between the heart's electrical and mechanical properties. This connection stands at the basis of the electro-mechanical model of the heart. The massive amount of a priori medical information can be used to determine the spatial coordinates of the heart walls. Using a time-dependent surface model, the 4D model of the heart can be determined. This spatio-temporal model was determined for normal and ectopic beats.

In the fourth chapter an advanced accessory pathway localization method is presented using the standard 12-lead ECG record. Although the published localization methods yield an almost 90% recognition rate, the weak points of the Arruda localization method can be exploited partially by the replacement of some decision criteria using a heart model simulation. The obtained clinical data evaluation supported the author's heart model based considerations.

Contents
Volume 1, 2009

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

This volume contains two issues

Sapientia University

Scientia Publishing House

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

Information for authors

Acta Universitatis Sapientiae, Informatica publishes original papers and 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, AMS Subject Classification codes (`http://www.ams.org/msc/`) and CR Categories and Description codes (`http://oldwww.acm.org/class/1998`). References should be listed alphabetically using the following examples:

For papers in journals:

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

For books:

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

For papers in contributed volumes:

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

For papers in conference volumes:

C. Xiao, W. Wang, X. Lin, J. X. Yu, Efficient similarity joins for near duplicate detection, *Proceedings of the 17th International Conference on World Wide Web*, Beijing, China, 2008, pp. 131–140.

For internet sources:

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

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

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

Publication supported by