Mining Induced and Embedded Subtrees in Ordered, Unordered, and Partially-Ordered Trees

Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero

Dept. Computer Science and Artificial Intelligence,
ETSIIT - University of Granada, 18071, Granada, Spain
{aidajm,jc.cubero,fberzal}@decsai.ugr.es

Abstract. Many data mining problems can be represented with non-linear data structures like trees. In this paper, we introduce a scalable algorithm to mine partially-ordered trees. Our algorithm, POTMiner, is able to identify both induced and embedded subtrees and, as special cases, it can handle both completely ordered and completely unordered trees (i.e. the particular situations existing algorithms address).

1 Introduction

Non-linear data structures are becoming more and more common in data mining problems. Graphs, for instance, are commonly used to represent data and their relationships in different problem domains, ranging from web mining and XML documents to bioinformatics and computer networks. Trees, in particular, are amenable to efficient mining techniques and they have recently attracted the attention of the research community.

The aim of this paper is to present a new algorithm, POTMiner, to identify frequent patterns in partially-ordered trees, a particular kind of trees that is present in several problems domains. However, existing tree mining algorithms cannot be directly applied to this important kind of trees.

Our paper is organized as follows. We introduce the idea of partially-ordered trees as well as some standard terms in Section 2. Section 3 describes the state of the art in tree mining algorithms. Our algorithm is presented in Section 4. Section 5 shows some experimental results. Finally, in Section 6, we present some conclusions and pointers to future work in this area.

2 Background

We will first review some basic concepts related to labeled trees using the notation from [1].

A **tree** is a connected and acyclic graph. A tree is rooted if its edges are directed and a special node, called root, can be identified. The root is the node from which it is possible to reach all the other vertices in the tree. In contrast, a

A. An et al. (Eds.): ISMIS 2008, LNAI 4994, pp. 111–120, 2008.

tree is free if its edges have no direction, that is, when it is an undirected graph. Therefore, a free tree has no predefined root.

Rooted trees can be classified as:

- **Ordered trees**, when there is a predefined order within each set of siblings.
- **Unordered trees**, when there is not such a predefined order among siblings.

In this paper, we consider **partially-ordered trees**, witch can be defined as trees that have both ordered and unordered sets of siblings. They can be useful when the order within some sets of siblings is important but it is not necessary to establish an order relationship among all the tree nodes.

In Figure 1, we show a dataset example with different kinds of rooted trees. In this figure, circled nodes represent ordered nodes, while squared nodes represent unordered ones.

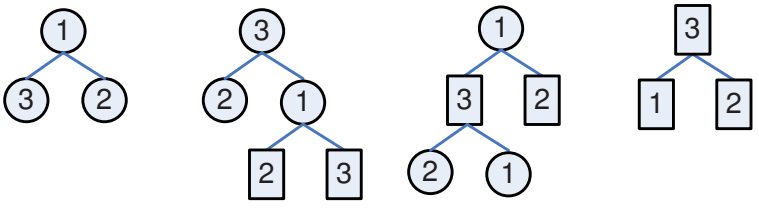

Fig. 1. Example dataset with different kinds of rooted trees (from left to right): *(a)* completely ordered tree, *(b) and (c)* partially-ordered trees, *(d)* completely unordered tree

Different kinds of subtrees can be mined from a set of trees depending on the way we define the matching function between a pattern and a tree. We can obtain **induced subtrees** from a tree T by repeatedly removing leaf nodes from a bottom-up subtree of T (i.e. the subtree obtained by taking one vertex from T and all its descendants). We can also obtain **embedded subtrees** by removing nodes from a bottom-up subtree provided that we do not to break the ancestor relationship between the vertices of T.

Figure 2 shows the induced subtrees of size 3 that appear at least 3 times in the example dataset from Figure 1. In this example, this would also be the set of frequent embedded subtrees of size 3.

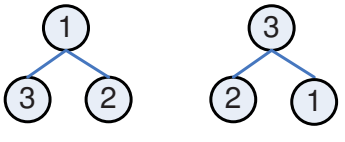

Fig. 2. Frequent subtrees of size 3 found in the dataset from Figure 1

3 Tree Pattern Mining

Once we have introduced some basic terminology, we will survey the state of the art in tree pattern mining.

The goal of frequent tree mining is the discovery of all the frequent subtrees in a large database of trees D, also referred to as *forest*, or in an unique large tree.

Let $\delta_T(\mathrm{S})$ be the occurrence count of a subtree S in a tree T and d_T a variable such that $d_T(\mathrm{S})=0$ if $\delta_T(S) = 0$ and $d_T(\mathrm{S})=1$ if $\delta_T(S) > 0$. We define the **support** of a subtree as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e., the number of trees in D that include at least one occurrence of the subtree S. Analogously, the **weighted support** of a subtree is $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., the total number of occurrences of S within all the trees in D.

We say that a subtree S is **frequent** if its support is greater than or equal to a predefined minimum support threshold. We define F_k as the set of all frequent subtrees of size k.

A frequent tree t is **maximal** if is not a subtree of other frequent tree in T and is **closed** if is not a subtree of another tree with the same support in D.

Before we proceed to introduce the algorithms proposed in the literature, we will describe the most common tree representations used by such algorithms.

3.1 Tree Representation

A canonical tree representation is an unique way of representing a labeled tree. This representation makes the problems of tree comparison and subtree enumeration easier.

Three alternatives have been proposed in the literature to represent rooted ordered trees as strings:

- **Depth-first codification:** The string representing the tree is built by adding the label of the tree nodes in a depth-first order. A special symbol ↑, which is not in the label alphabet, is used when the sequence comes back from a child to his parent. In Figure 1, the depth-first codification of tree *(b)* is 32↑12↑3↑↑ while the codification of tree *(c)* is 132↑1↑↑2↑.
- **Breadth-first codification:** Using this codification scheme, the string is obtained by traversing the tree in a breadth-first order, i.e., level by level. Again, we need an additional symbol \$, which is not in the label alphabet, to separate sibling families. The breadth-first codification for trees (b) and (c) in the previous example is 3\$21\$\$23 and 1\$32\$21, respectively.
- **Depth-sequence-based codification:** This codification scheme is also based on a depth-first traversal of the tree, but it explicitly stores the depth of each node within the tree. The resulting string is built with pairs (l, d) where the first element (l) is the node label and the second one (d) is the depth of the node. The depth sequence codifications of the previous examples is (3,0)(2,1) (1,1)(2,2)(3,2) for tree (b) and (1,0)(3,1)(2,2)(1,2)(2,1) for tree (c).

The canonical representation for unordered trees can be defined as the minimum codification, in lexicographical order, of all the ordered trees that can be derived from it. You can use any of the codification schemes described above.

Free trees have no predefined root, but it is possible to select one node as the root in order to get an unique canonical representation of the tree, as described in [2].

3.2 Tree Mining Algorithms

Several frequent tree pattern mining algorithms have been proposed in the literature. These algorithms are mainly derived from two well-known frequent pattern mining algorithms: Apriori [3] and FP-Growth [4].

Many algorithms follow the well-known Apriori [3] iterative pattern mining strategy, where each iteration is broken up into two distinct phases:

- *Candidate Generation*: Potentially frequent candidates are generated from the frequent patterns discovered in the previous iteration. Most algorithms generate candidates of size $k+1$ by merging two patterns of size k having $k-1$ elements in common. There are several strategies for candidate subtree generation:
  - The **rightmost expansion** strategy generates subtrees of size $k+1$ from frequent subtrees of size k by adding nodes only to the rightmost branch of the tree. This technique is used in algorithms like FREQT [5], uFreqt [6], and UNOT [7].
  - The **equivalence class-based extension** technique is based on the depth-first canonical representation of trees. It generates a candidate $(k+1)$-subtree through joining two frequent k-subtrees with $(k-1)$ nodes in common. Zaki used this extension method in his TreeMiner [8] and SLEUTH [9] algorithms.
  - The **right-and-left tree join** method was proposed with the AMIOT algorithm [10]. In candidate generation, this algorithm considers the rightmost and the leftmost leaves of a tree.
  - The **extension and join** technique is based on the breadth-first codification of trees and defines two extension mechanisms and a join operation to generate candidates. This method is used by HybridTreeMiner [2].
- *Support Counting*: Given the set of potentially frequent candidates, this phase consists of determining their support and keeping only those candidates that are actually frequent.

Other algorithms are derived from the FP-Growth [4] algorithm. For instance, the PathJoin algorithm [11] uses compacted structures called FP-Trees to encode input data, while CHOPPER and XSpanner [12] use a sequence codification for trees and extract subtrees using frequent subsequences.

Table 1 summarizes some frequent tree mining algorithms that have been proposed in the literature, pointing out the kind of input trees they can be applied to (ordered, unordered, or free) and the kind of subtrees they are able to identify (induced, embedded, or maximal).

Table 1. Some frequent tree mining algorithms

Algorithm	Input trees			Discovered patterns		
	Ordered	Unordered	Free	Induced	Embedded	Maximal
FreqT[5]	•			•		
AMIOT [10]	•			•		
uFreqT [6]		•		•		
TreeMiner [8]	•				•	
CHOPPER [12]	•				•	
XSpanner [12]	•				•	
SLEUTH [9]		•			•	
Unot [7]		•			•	
TreeFinder [13]		•			•	•
PathJoin [11]		•		•		•
CMTreeMiner [14]	•	•		•		•
FreeTreeMiner [15]			•	•		
HybridTreeMiner [2]		•	•	•		

4 Mining Partially-Ordered Trees

The algorithms described in the previous section can extract frequents patterns from trees that are completely ordered or completely unordered, but none of them works with partially-ordered trees.

In this section, we describe how Zaki's TreeMiner [8] and SLEUTH [9] algorithms can be adapted to mine partially ordered trees. The algorithm we have devised, POTMiner (Partially-Ordered Tree Miner), is able to identify frequent subtrees, both induced and embedded, in ordered, unordered, and partially-ordered trees.

Our algorithm is based on Apriori [3]. Therefore, it has two phases: candidate generation and support counting.

4.1 Candidate Generation for Partially-Ordered Trees

We use a depth-first codification to represent trees and generate $(k+1)$-subtree candidates by joining two frequent k-subtrees with $k-1$ elements in common.

We use Zaki's class extension method to generate candidates. Two k-subtrees are in the same equivalence class $[P]$ if they share the same codification string until the node $k-1$. Each element of the class can then be represented by a single pair (x, p) where x is the k-th node label and p specifies the depth-first position of its parent.

TreeMiner [8] and SLEUTH [9] use the class extension method to mine ordered and unordered embedded subtrees, respectively. The main different between TreeMiner and SLEUTH is that, in SLEUTH, which works with unordered trees, only those extensions that produce canonical subtrees are allowed in order to avoid the duplicate generation of candidates.

In POTMiner, as in TreeMiner, all extensions are allowed because tree nodes might be ordered or unordered. The candidate generation method is defined as follows:

Let (x, i) and (y, j) denote two elements in the same class $[P]$, and $[P_x^i]$ be the set of candidate trees derived from the tree that is obtained by adding the element (x, i) to P. The join procedure considers two scenarios [9]:

1. *Cousin extension*, when the element (y,j) is a right relative of the element (x,i): If $j \leq i$ and $|P| = k - 1 \geq 1$, then $(y, j) \in [P_x^i]$.
2. *Child extension*, when the element (y,j) is a descendant of the element (x,i): If $j = i$ then $(y, k - 1) \in [P_x^i]$.

4.2 Support Counting for Induced and Embedded Subtrees

For the support counting phase, we define the scope of a node [9] as a pair [l,u] where l is the position of the node in depth-first order and u is the position of its rightmost descendant.

Our algorithm preserves each occurrence of a pattern X in each database tree using a tuple (t, m, s, d) where t is the tree identifier, m stores which nodes of the tree match those of the (k-1) prefix of the pattern X in depth-first order, s is the scope of the last node in the pattern X, and d is a depth-based parameter used for mining induced subtrees (it is not needed when mining embedded subtrees). The scope list of a pattern X is the list of all the tuples (t, m, s, d) representing the occurrences of X in the database.

For patterns of size 1, m is an empty list and the element d is initialized with the depth of the pattern only node in the database tree.

The scope list for a new candidate is built by joining the scope lists of the subtrees involved in the generation of the candidate. Let [t_x, m_x, s_x, d_x] and [t_y, m_y, s_y, d_y] be the scope lists of the subtrees involved in the generation of the candidate. The scope list for this candidate is built by a join operation that depends on the candidate extension method used, i.e., whether it has been generated by cousin extension or child extension.

1. *In-scope test* (used in conjunction with child extension):
   If
   (a) $t_x = t_y = t$ and
   (b) $m_x = m_y = m$ and
   (c) $s_y \subset s_x$, (i.e., $l_x \leq l_y$ and $u_x \geq u_y$)
   then add [t, m $\bigcup \{l_x\}$, s_y, $d_y - d_x$] to the scope list of the generated candidate.
2. *Out-scope test* (used in conjunction with cousin extension):
   If
   (a) $t_x = t_y = t$ and
   (b) $m_x = m_y = m$ and
   (c) If the node is ordered and $s_x < s_y$ (i.e. $u_x < l_y$) or the node is unordered and either $s_x < s_y$ or $s_y < s_x$ (i.e. either $u_x < l_y$ or $u_y < l_x$)
   then add [t, m $\bigcup \{l_x\}$, s_y ,d_y] to the scope list of the generated candidate.

The weighted support of an embedded pattern is the number of elements in its scope list. The weighted support of an induced pattern is the number of elements within its scope list whose d parameter equals 1. Intuitively, d represents the distance between the last node in the pattern and its prefix m. This parameter is needed to perform the support counting phase for induced patterns, since only the occurrences with $d_x = 1$ have to be considered when joining scope lists.

5 Experimental Results

All the experiments described in this section have been performed on a 2GHz Core 2 Duo processor with 2GB of main memory running on Windows Vista. POTMiner has been implemented in Java using Sun Microsystems JDK 5.0, while Zaki's TreeMiner and SLEUTH C++ implementations were obtained from *http://www.cs.rpi.edu/∼zaki/*.

The experiments were performed with 5 synthetic datasets generated by the tree generator available at *http://www.cs.rpi.edu/∼zaki/software/TreeGen.tgz*. The datasets were obtained using the generator default values and varying the number of trees from 10 to 100000.

In our first experiments, we have compared POTMiner and TreeMiner / SLEUTH execution times. POTMiner and TreeMiner [8] are compared for (completely) ordered trees, while POTMiner and SLEUTH [9] are compared for unordered trees. The results we have obtained are summarized in Figure 3.

Looking at the charts in Figure 3, which use a logarithmic scale for their y axis, we can see that TreeMiner, SLEUTH, and POTMiner are efficient, scalable algorithms for mining induced and embedded subtrees in ordered (TreeMiner and POTMiner) and unordered (SLEUTH and POTMiner) trees. The observed differences are probably due to the different programming platforms used in their implementation (Java for POTMiner, C++ for TreeMiner and SLEUTH).

Figure 3 also shows that there are no significant differences between the POTMiner's execution times for ordered and unordered trees in the experiments performed with the aforementioned synthetic datasets.

Execution times of POTMiner and TreeMiner algorithm (*left*)for extracting induced (*top*) and embedded (*down*) subtrees in completely ordered trees. POTMiner and SLEUTH behaviour when mining completely unordered trees, induced (*top*) and embedded (*down*)is shown in right images.

We have also performed some experiments with partially-ordered trees. In this case, since TreeMiner [8] and SLEUTH [9] cannot be applied to partially-ordered trees, we have studied the behavior of POTMiner when dealing with this kind of trees. Starting from the same datasets used in the previous experiments, we have varied the percentage of ordered nodes in the datasets. The results we have obtained are shown in Figure 4.

As expected, we have found that execution times slightly decrease when the percentage of ordered nodes is increased, since ordered trees are easier to mine than unordered trees.

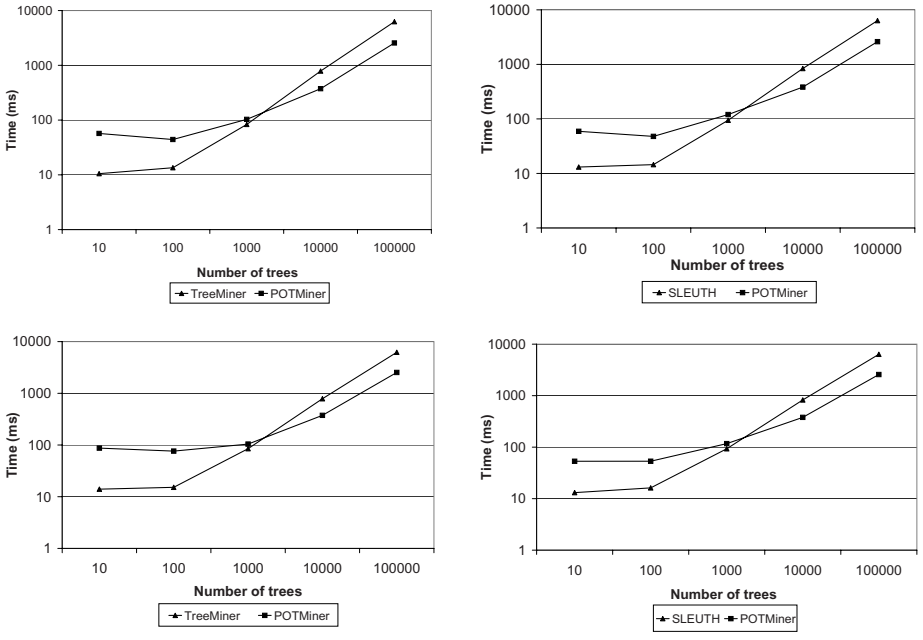

Fig. 3. Execution times of POTMiner and TreeMiner algorithms(*left images*) for extracting induced (*top*) and embedded (*down*) subtrees in completely ordered trees. POTMiner and SLEUTH execution times and completely unordered (*right*) trees.

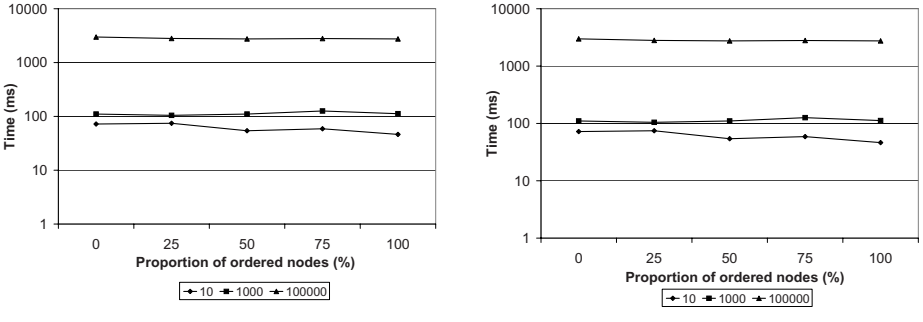

Fig. 4. POTMiner execution times varying the percentage of ordered nodes. The charts display POTMiner behavior when mining induced *(left)* and embedded *(right)* subtrees. The series show execution times for different dataset sizes (10, 1000, and 100000 trees).

6 Conclusions and Future Work

There are many different tree mining algorithms that work either on ordered or unordered trees, but none of them, to our knowledge, works with partially-ordered trees, that is, trees that have both ordered and unordered nodes. We have

devised a new algorithm to address this situation that is as efficient and scalable as existing algorithms that exclusively work on either ordered or unordered trees.

Partially-ordered trees are important because they appear in different application domains. In the future, we expect to apply our tree mining algorithm to some of these domains. In particular, we believe that our algorithm for identifying frequent subtrees in partially-ordered trees can be useful in the following contexts:

- XML documents [16], due to their hierarchical structure, are directly amenable to tree mining techniques. Since XML documents can contain both ordered and unordered sets of nodes, partially-ordered trees provide a better representation model for them and they are better suited for knowledge discovery than existing ordered (or unordered) tree mining techniques.
- Multi-relational data mining [17] is another emerging research area where tree mining techniques can be useful. They might help improve existing multi-relational classification [18] and clustering [19] algorithms.
- In Software Engineering, it is usually acknowledged that mining the wealth of information stored in software repositories can "support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques" [20]. For instance, there are hierarchical program representations, such as dependence higraphs [21], which can be viewed as partially-ordered trees, hence the potential of tree mining techniques in software mining.

We also have the goal of making some refinements to our POTMiner algorithm, such as extending it for dealing with partial or approximate tree matching, a feature that would be invaluable for many real-world problems, from entity resolution in XML documents to program element matching in software mining.

Acknowledgements

Work partially supported by research project TIN2006-07262.

References

1. Chi, Y., Muntz, R.R., Nijssen, S., Kok, J.N.: Frequent subtree mining - an overview. Fundamenta Informaticae 66(1-2), 161–198 (2005)
2. Chi, Y., Yang, Y., Muntz, R.R.: HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: SSDBM 2004, pp. 11–20 (2004)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB 1994, pp. 487–499 (1994)
4. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: SIGMOD 2000, pp. 1–12 (2000)
5. Abe, K., Kawasoe, S., Asai, T., Arimura, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data. In: SDM 2002 (2002)

6. Nijssen, S., Kok, J.N.: Efficient discovery of frequent unordered trees. In: First International Workshop on Mining Graphs, Trees and Sequences (MGTS 2003), in conjunction with ECML/PKDD 2003, pp. 55–64 (2003)
7. Asai, T., Arimura, H., Uno, T., ichi Nakano, S.: Discovering frequent substructures in large unordered trees. In: Grieser, G., Tanaka, Y., Yamamoto, A. (eds.) DS 2003. LNCS (LNAI), vol. 2843, pp. 47–61. Springer, Heidelberg (2003)
8. Zaki, M.J.: Efficiently mining frequent trees in a forest: Algorithms and applications. IEEE Transactions on Knowledge and Data Engineering 17(8), 1021–1035 (2005)
9. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae 66(1-2), 33–52 (2005)
10. Hido, S., Kawano, H.: AMIOT: induced ordered tree mining in tree-structured databases. In: ICDM 2005, pp. 170–177 (2005)
11. Xiao, Y., Yao, J.F., Li, Z., Dunham, M.H.: Efficient data mining for maximal frequent subtrees. In: ICDM 2003, pp. 379–386 (2003)
12. Wang, C., Hong, M., Pei, J., Zhou, H., Wang, W., Shi., B.: Efficient pattern-growth methods for frequent tree pattern mining. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 441–451. Springer, Heidelberg (2004)
13. Termier, A., Rousset, M.C., Sebag, M.: TreeFinder: a first step towards xml data mining. In: ICDM 2002, pp. 450–457 (2002)
14. Chi, Y., Xia, Y., Yang, Y., Muntz, R.R.: Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Transactions on Knowledge and Data Engineering 17(2), 190–202 (2005)
15. Chi, Y., Yang, Y., Muntz, R.R.: Indexing and mining free trees. In: ICDM 2003, pp. 509–512 (2003)
16. Nayak, R., Zaki, M.J. (eds.): Knowledge discovery from xml documents. In: Nayak, R., Zaki, M.J. (eds.) KDXD 2006. LNCS, vol. 3915, Springer, Heidelberg (2006)
17. Džeroski, S.: Multi-relational data mining: An introduction. SIGKDD Explorations Newsletter 5(1), 1–16 (2003)
18. Yin, X., Han, J., Yang, J., Yu, P.S.: CrossMine: efficient classification across multiple database relations. In: ICDE 2004, pp. 399–410 (2004)
19. Yin, X., Han, J., Yu, P.S.: Cross-relational clustering with user's guidance. In: KDD 2005, pp. 344–353 (2005)
20. Gall, H., Lanza, M., Zimmermann, T.: 4th International Workshop on Mining Software Repositories (MSR 2007). In: ICSE COMPANION 2007, pp. 107–108 (2007)
21. Berzal, F., Cubero, J.C., Jimenez, A.: Hierarchical program representation for program element matching. In: Yin, H., Tino, P., Corchado, E., Byrne, W., Yao, X. (eds.) IDEAL 2007. LNCS, vol. 4881, pp. 467–476. Springer, Heidelberg (2007)