A Note on Trees, Tables and Algorithms

Wayne Goddard, Stephen T. Hedetniemi
School of Computing, Clemson University
{goddard,hedet}@cs.clemson.edu

Abstract

Several algorithms for optimal vertex subsets in trees are given by simple tables. In this paper we investigate the properties of and operations on these tables. We use known techniques for combining tables to correct a table in the literature; give a necessary and sufficient condition for a table to correspond to some tree property; discuss the question of dividing one table by another; explain how to derive a table from a set of representatives; and apply this to finding the table for the parameter external redundance. All of this is facilitated by computer software.

Key words: Trees, algorithms, tables, external redundance

1 Introduction

Dynamic programming has been used for many algorithms on trees and other recursively decomposable structures. Many of these algorithms involve a state space, which can sometimes be expressed as a table. In this paper we explore the properties of these tables and operations on them. We restrict attention to tables for algorithms on trees, though in principle most of the ideas can be generalized.

The idea behind a table is that it can be used to determine the minimum or maximum cardinality of a valid vertex subset. This approach can be applied to other recursive structures, provided the recursive decomposition of the structure is known or can be computed. For example, Borie et al. [4] and others showed how a table can be generated from a description of the parameter, such as one written in logic [8]. Bern et al. [2] showed how to obtain one table from another. Related ideas were explored by Arnborg and Proskurowski [1], Bodlaender [3], Courcelle [5], and many others.

1.1 Tables

We define a ***marked tree*** as a tree in which some subset of the vertices is marked. We define a ***tree set-property*** $\mathcal{P}$, or simply a ***property***, as a subset of the marked trees. Alternatively, one could define a property as a boolean predicate. For example, independence is the property where no two adjacent vertices are marked.

We next consider a way of building a tree. A rooted tree (T, r) is a tree T in which some vertex r is labeled the root. Given two rooted marked trees (T_1, r_1) and (T_2, r_2), one forms the rooted marked tree $(T_1, r_1) \odot (T_2, r_2)$ by adding an edge between r_1 in T_1 and r_2 in T_2 and making r_1 the root. We call T_1 the parent and T_2 the child. This is illustrated in Figure 1.

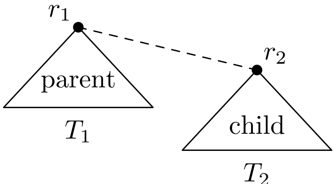


Figure 1: The combination of two trees.

Let $\mathcal{M}$ be the set of all rooted marked trees. We define a ***tree congruence of index*** k as a partition of $\mathcal{M}$ into k classes numbered 0 to $k-1$, such that the corresponding equivalence relation $\sim$ is a congruence under the above binary operation. That is, if $(T_1, r_1) \sim (T_2, r_2)$, and (U, s) is any rooted tree, then $(T_1, r_1) \odot (U, s) \sim (T_2, r_2) \odot (U, s)$ and $(U, s) \odot (T_1, r_1) \sim (U, s) \odot (T_2, r_2)$. Thus $\odot$ can also be thought of as an operator on $\{0, \ldots, k-1\}$, and can be represented by a $k \times k$ matrix, where the (i, j)-entry gives the congruence class of $i \odot j$. This we call a ***table***. If one knows the classes of the marked and the unmarked singleton trees, then the table can be used to calculate the class of any tree. A table can also be thought of as a tree automaton (see [7]).

Now, given a tree congruence we can label some of the classes as ***valid***. The tree set-property $\mathcal{P}$ associated with the congruence is the union of all valid classes of the congruence. Courcelle [5] showed that a tree set-property is given by some table if and only if the property is recognizable by a tree automaton.

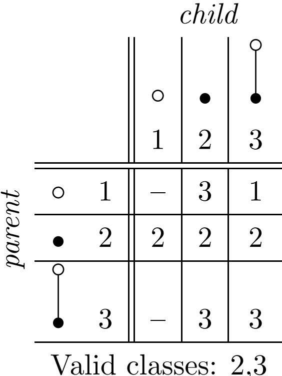
1.2 An example

In our examples, the class 0 is always ***dead***: this means that no extension of a tree in class 0 can have the property. In particular, it follows that $0 \odot x = x \odot 0 = 0$ for every class x. In our tables, the class 0 is suppressed and replaced by –. Further, the unmarked singleton tree is always in class 1 and the marked singleton tree is always in class 2.

> EXAMPLE 1. *Recall that a **dominating set** in a graph is a set S of vertices such that every vertex not in S is adjacent to a vertex in S. The known table for dominating set is given in Figure 2. In the table, class 1 corresponds to sets that dominate all vertices except the root; class 2 to dominating sets that include the root; and class 3 to dominating sets that do not include the root. Class 0 contains all other marked trees. Associated with each class we draw a representative tree of smallest order: the dark vertices are the marked ones.*
>
> *For the associated algorithm, one performs a postorder traversal of the tree. At each vertex one considers the subtree rooted at that vertex and calculates the triple (f_1, f_2, f_3), where f_i is the minimum cardinality of a set in class i. The table provides a recursive formula. At the end, one looks at the triple for the overall root and the domination number is the minimum of f_2 and f_3.*

1.3 Contribution

In this paper we proceed as follows. In Section 2 we review the techniques of Bern et al. [2], and use them to correct a table in the literature. In Section 3, we give a necessary and sufficient condition for a table to correspond to some tree set-property. In Section 4, we introduce the question of dividing one table by another, which sheds light on the so-called domination chain. In Section 5 we explain how to construct a table from a set of representatives and apply this technique to finding the table for the parameter external redundance.



parent \ child	1	2	3
1	–	3	1
2	2	2	2
3	–	3	3

Valid classes: 2,3

Figure 2: The table for domination.

2 New Tables From Old

Some of the questions we consider involve operations on tables. So we now review the known techniques to produce tables from existing tables. Almost all of the machinery in this section was given in Bern et al. [2], and mirrors techniques in finite automata. As an example, we correct a mistake in the literature.

2.1 Shrinking tables

As Bern et al. [2] pointed out, one can apply operations to try to shrink a table. These include:

- *Discard empty classes.* The nonempty classes can be found by a search from the base classes, the two classes containing singleton trees.
- *Merge equivalent classes.* Two classes a and b are ***equivalent*** if no matter what sequence of operations $\odot$ one performs (on the right or left), the result starting from a is valid iff the result starting from b is valid. These can be identified using ideas similar to finite-state machine minimization.

These algorithms run in time polynomial in the number of classes [2].

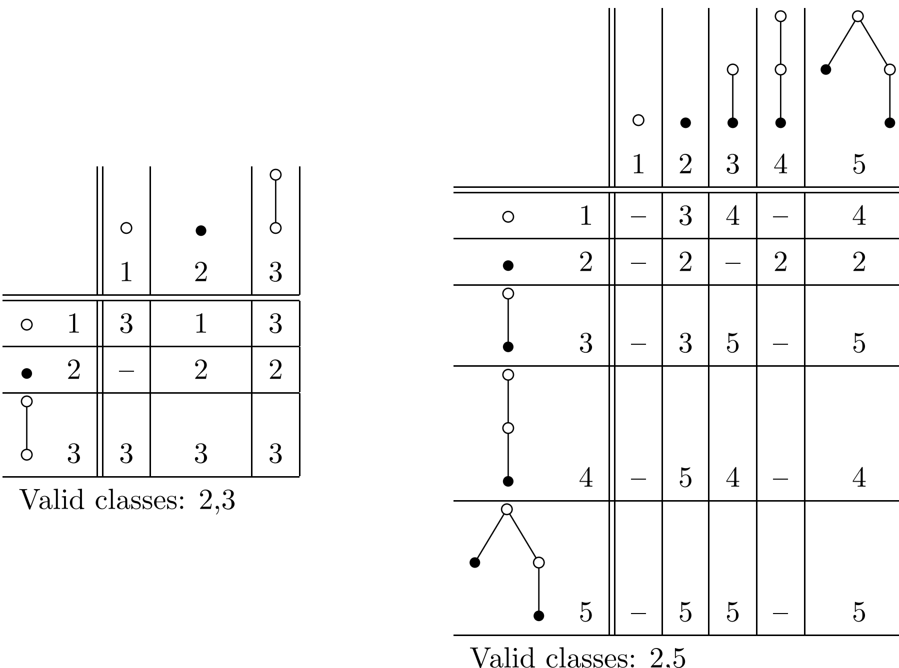
2.2 Product tables

Bern et al. [2] showed how to obtain new tables from old ones. For example, if one has two properties $\mathcal{P}_1$ and $\mathcal{P}_2$ with tables T_1 and T_2, one obtains the table for the intersection property $\mathcal{P}_1 \cap \mathcal{P}_2$ by taking the Cartesian product of the tables (equivalent to the product construction from finite-state machines). If the two tables are of sizes $a \times a$ and $b \times b$, then the new table starts out at size $ab \times ab$, but one may then shrink it as above.

We note here that one can also obtain the table for the union $P_1 \cup P_2$ if one wants.

Here is an example that corrects a mistake in the literature.

> EXAMPLE 2. *A dominating set S is a **restrained dominating set** if every vertex outside S has also a neighbor outside S. For example, the entire vertex set is always a restrained dominating set. This property was introduced in [6].*
>
> *That paper gives a table for the property. But, as its authors subsequently noticed, there is an error. It turns out that one class should actually be split into two. To obtain the correct table, it is sufficient to compute the table for restrained-ness, and take its product with the table for domination, given in Example 1. These tables are given in Figure 3.*
>
> *As in many tables, one can give each class an English description. Here, in all classes the condition of an unmarked vertex having both a marked and an unmarked neighbor is true at all vertices other than the root. The class describes the root:*
> *(1) Root is out, and has no neighbor;*
> *(2) Root is in;*
> *(3) Root is out and has only marked neighbors;*
> *(4) Root is out and has only unmarked neighbors;*
> *(5) Root is out and has both a marked and an unmarked neighbor.*



	1	2	3
1	3	1	3
2	–	2	2
3	3	3	3

Valid classes: 2,3

	1	2	3	4	5
1	–	3	4	–	4
2	–	2	–	2	2
3	–	3	5	–	5
4	–	5	4	–	4
5	–	5	5	–	5

Valid classes: 2,5

Figure 3: The tables for restrained-ness and restrained domination.

2.3 Minimal and maximal tables

Another table operation involves going from a given property to the 1-minimal or 1-maximal version of the property. Recall that a $\mathcal{P}$-set is ***1-minimal*** if removing any one vertex changes the set to not being a $\mathcal{P}$-set, and ***1-maximal*** if adding any one vertex changes the set to not being a $\mathcal{P}$-set. Bern et al. [2] observed that to determine if a given tree T is 1-maximal or 1-minimal, all one needs to calculate is: (a) what class T is in; and (b) what classes T can be in if a vertex is added or removed. It follows that in the table for 1-maximal or 1-minimal, each class is defined by an old class and a subset of the old classes. Thus the resulting table can be exponential in the size of the original table.

2.4 Testing a property for being hereditary

Recall that a set-property is ***hereditary*** if it is closed under subsets. Given a table for a property, one can quickly determine if a property is hereditary or not. What one actually has to check is whether changing a marked vertex to

an unmarked vertex can change the tree from being valid to invalid. To do this, one determines the complete list of all possible class changes that can result from unmarking a vertex. One starts with the change of class $2 \to 1$ (singleton marked to singleton unmarked tree); adds all changes $2 \odot i \to 1 \odot i$ etc.; and then takes the transitive closure of this list.

Another approach is based on the fact that the condition of being hereditary is equivalent to the empty set being the **only** 1-minimal set. So one can calculate the table for 1-minimality and answer the question. But the above algorithm is polynomial-time, whereas minimality finding takes potentially exponential time.

2.5 Comparing two properties

We say that a property $\mathcal{P}_1$ is ***more general*** than property $\mathcal{P}_2$ if every graph/tree having property $\mathcal{P}_2$ also has property $\mathcal{P}_1$. This test can be implemented by constructing the property that is the intersection of $\mathcal{P}_2$ and the complement of $\mathcal{P}_1$, and testing for nonemptiness. This can be automated: the intersection is given by the product construction described above, and the complement essentially involves swapping the valid and invalid classes. This approach can easily be extended to do equality testing.

3 Feasible Tables

In this section we resolve the following question: Given a matrix that purports to be the table for an undisclosed tree set-property, can one check that it is indeed a legitimate table? That is, we are presented with a table and a choice of which classes are valid, and want to know whether the matrix always assigns the same class to a rooted tree no matter how one builds it, and the same validity to a tree no matter where one roots it.

Theorem 1 *A table corresponds to some tree set-property if and only if the following two conditions are met:*

1. *The table is "right commutative": that is, for all classes x, y, z:* $(x \odot y) \odot z = (x \odot z) \odot y$.

2. *The table is "boolean symmetric": that is, for all classes x, y:* $x \odot y$ *is valid iff* $y \odot x$ *is valid.*

PROOF. We first observe that the two conditions are necessary for a tree set-property. Condition 1 says that if we attach two children to the root vertex, then the order of the attachment does not matter. Condition 2 says that if we combine two trees, then which one becomes the root does not matter as far as to the validity of the overall tree.

So we now show that it is sufficient to test the table for these properties.

Lemma 1 *Condition 1 ensures that no matter what sequence of operations one uses to produce a given rooted tree, one will get the same value of the operator* $\odot$.

PROOF. The proof is by induction on the size of the tree. Let r be the root of the tree, and $c_1, \ldots, c_t$ its children. Note that the combination operation of trees affects the child-set of only the root. That is, before a vertex c_i becomes the child of r, the subtree rooted at c_i must be completely built. Hence, we may assume that one produces all the child subtrees first. By the inductive hypothesis, the class of these child subtrees is uniquely determined.

Now, the remaining operations are to start with the singleton root, and combine in some order all its children. Say the root has class x, and the children have classes $y_1, \ldots, y_t$. We claim that the condition of right commutativity ensures that the product $x \odot y_{a_1} \odot \ldots \odot y_{a_t}$ evaluated from left to right is the same irrespective of the order of the y_i. The proof is that one can by repeated application of Condition 1, shuffle y_1 to the first multiplicand, and then y_2 and so on. Hence the product is uniquely determined. □

Lemma 2 *The two conditions ensure that no matter what root we choose and no matter what sequence of operations we use to produce the rooted tree, we will always get the same validity of the tree.*

PROOF. It suffices to show that the validity value for root r_1 and for root r_2 are the same provided r_1 and r_2 are adjacent. For one can then iterate through the tree moving the root to any desired vertex.

If we split the tree at the edge $e = r_1 r_2$, we get two rooted trees. By the previous lemma, the class of these two rooted trees is uniquely determined. The class of the overall tree is determined as the $\odot$ of the two values of the two subtrees. The overall validity is then the same, by Condition 2. $\square$

This completes the proof of Theorem 1. $\square$

This theorem clearly provides an $O(n^3)$ algorithm to test whether a given table is feasible or not.

4 Division Tables

In this section we consider the problem of division in tables. This we use to shed light on the so-called domination chain.

Recall that the domination chain is:

$$ir \leq \gamma \leq i \leq \beta \leq \Gamma \leq IR$$

where ir and IR are the smallest and largest cardinality of a maximal irredundant set; γ and Γ are the smallest and largest cardinality of a minimal dominating set; and i and β are the smallest and largest cardinality of a maximal independent set. A set S is ***irredundant*** if for every vertex $v \in S$ there is a ***private neighbor*** w_v; that is, w_v is dominated by v but by no other vertex of S (a vertex can be its own private neighbor). A set S is ***externally redundant*** if $\forall v \in V - S$ $pnc(S \cup \{v\}) \leq pnc(S)$. The parameter $pnc(S)$ (private neighbor count) gives the number of vertices in S that have private neighbors. For a discussion of this, see [9].

4.1 The definition of division

It has been said that domination can be defined as that property that makes a maximal independent set maximal. So one can ask the question: what properties $\mathcal{P}$ have the property that $\mathcal{P} \cap \mathcal{I} = \mathcal{MI}$, where $\mathcal{I}$ is the property of being an independent set and $\mathcal{MI}$ the property of being a maximal independent set.

Clearly, $\mathcal{P}$ equals domination will do. But so will $\mathcal{P}$ being $\mathcal{MI}$ itself. This suggests we should define $\mathcal{A}/\mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are table properties, as the most general property $\mathcal{C}$ such that $\mathcal{C} \cap \mathcal{B} = \mathcal{A}$.

But, the most general property $\mathcal{C}$ that combined with independence gives maximal independence is not domination. Indeed, the property $\mathcal{C}$ is simply $\overline{\mathcal{I}} \cup \mathcal{MI}$: the property of a set being either dominating or not independent or both. So maybe the correct definition of $\mathcal{A}/\mathcal{B}$ is the "simplest" property $\mathcal{C}$ such that $\mathcal{C} \cap \mathcal{B} = \mathcal{A}$. By ***simplest*** we mean that it has the fewest classes possible.

It is easy to show that any property $\mathcal{C}$ that turns independence into maximal independence has at least three live classes. And if it has three live classes, then it has the form given in Figure 4.

		○ 1	● 2	○–● 3
○	1	–	3	1
●	2	2	??	2
○–●	3	–	3	3

Valid classes: 2,3

Figure 4: The simplest table $\mathcal{P}$ such that $\mathcal{P} \cap \mathcal{I} = \mathcal{MI}$.

It is not hard to show that there are only two possible choices for the missing entry: – gives maximal independence itself, and 2 gives domination. This suggests the following definition:

> *For tables $\mathcal{A}$ and $\mathcal{B}$, we define $\mathcal{A}/\mathcal{B}$ as the property $\mathcal{C}$ with the fewest classes such that $\mathcal{C} \cap \mathcal{B} = \mathcal{A}$ and, subject to this, the property that is the most general.*

However, it turns out that $\mathcal{C}$ does not have to be unique, as we see next.

4.2 Application to the domination chain

Irredundance is said to be the added condition that makes a dominating set minimal. Again, we can ask: what is the simplest table whose intersection with domination gives minimal domination?

It turns out that the minimum size is the size of the minimal domination table, namely six live classes. We enlisted a computer to do the search. For the search, one needs the characterization of feasible tables given earlier in Theorem 1. The search showed that there are three tables of minimum size that work. One is minimal domination itself, one is irredundance, and one is new. (The new one does not appear to correspond to an interesting property.) Further, irredundance is more general than both of the other two properties. That is, minimal domination divided by domination gives irredundance.

Now, external redundance has been claimed [9] to be the added condition that makes an irredundant set maximal. Again using a computer search, we applied this procedure to determine the simplest table that makes an irredundant set maximal. Again the smallest table is the size of maximal irredundance (which is 20 live classes), and it turns out that there are 24 candidate simplest tables. However, when one asks for the most general property:

> *there are three simplest properties that are not subsets of other properties, and none of these is external redundance.*

One can check that these are not external redundance by exhibiting a tree that the table claims is valid but is not externally redundant, or vice versa. We discuss the actual table for external redundance below.

Although external redundance is not closed under taking supersets or subsets, unlike domination, independence and irredundance, one could then apply the same question, asking for that property which makes external redundance minimal, but the computations seem intractable.

5 Representatives and Tables

In this section we revisit the construction and verification of a table. We show that some parts of this can be automated if one has an oracle for the property. We apply this to support our claim of what the table for external redundance is.

5.1 Independent trees

Suppose we have an alleged table for some property $\mathcal{P}$ and an alleged representative T_i from each class. If we take the table and replace each class by its claimed truth value, we get a truth table. One can think of each row and column as a boolean vector. It is straightforward to see that:

Lemma 3 *Let $\vec{r}_k$ and $\vec{c}_k$ denote the row and column of the truth table corresponding to representative T_k. If these vectors are correct for each T_k according to the oracle for $\mathcal{P}$, and the pairs $(\vec{r}_k, \vec{c}_k)$ are distinct, then the representatives are in different classes for any tree congruence for $\mathcal{P}$.*

5.2 From representatives to a table

The same idea can be used to produce the table from a list of representatives. This is based on the fact that two rooted marked trees are in the same class iff they behave the same.

Suppose we have rooted marked trees $T_1, T_2, \ldots, T_r$ known to be in different classes. One can determine the entries of the table as follows. Start by calculating for each k the boolean vectors $\vec{r}_k$ and $\vec{c}_k$ as the validity of $\{T_k \odot T_s\}_s$ and $\{T_s \odot T_k\}_s$, as given by the oracle. Then to calculate the (i,j)-entry, one calculates $T_{ij} = T_i \odot T_j$, combines it with each representative, and tests the result for validity using the oracle. This yields two vectors $\vec{r}_{ij}$ and $\vec{c}_{ij}$ where $\vec{r}_{ij}$ gives the validity of $\{T_{ij} \odot T_s\}_s$ and $\vec{c}_{ij}$ the validity of $\{T_s \odot T_{ij}\}_s$. These are then compared with the set of vectors $\vec{r}_k$ and $\vec{c}_k$. If all is okay, this pair $(\vec{r}_{ij}, \vec{c}_{ij})$ will be the same as $(\vec{r}_k, \vec{c}_k)$ for a unique k. This k is the table entry.

EXAMPLE 3. *Recall that a set S is externally redundant iff for all $v \notin S$, either v will have no private neighbor if added to S, or adding*

v to S will destroy all private neighbors of some vertex in S. In Figure 5 is a list of 23 representatives for external redundance and in Figure 6 is a claimed table for the property.

We used computer code and Lemma 3 to show that the 23 representatives are independent with respect to the property external redundance. We also used computer code to produce this table from the set given in Figure 5. From the above discussion, the table is correct IF the 23 live classes are indeed all.

It is to be noted that this process was facilitated by the computer software, written in Java. Earlier lists were incomplete or sometimes had two equivalent representatives. Running the above table-generating algorithm showed that certain table-entries could not be uniquely specified or that the table did not satisfy the feasibility criterion. This helped us to identify the missing classes.

We believe that the table for external redundance is correct. For a proof of this, it would be sufficient to prove that none of the classes needs to be divided. This is a lengthy and tedious argument and is omitted.

Thus we have shown how to automate the process of going from representatives to a table, and how software can assist in obtaining the table in practice. Of course, the results of Borie et al. [4], Courcelle [5] and others say that the whole process is, at least in theory, automatable.

6 Further Questions

We have shown that the process of generating a table can be partially automated, and that the table can shed light on questions about the associated property. However, having such a table is rather restrictive. So it would be useful to consider equivalent structures for more general properties, or for more general graphs.

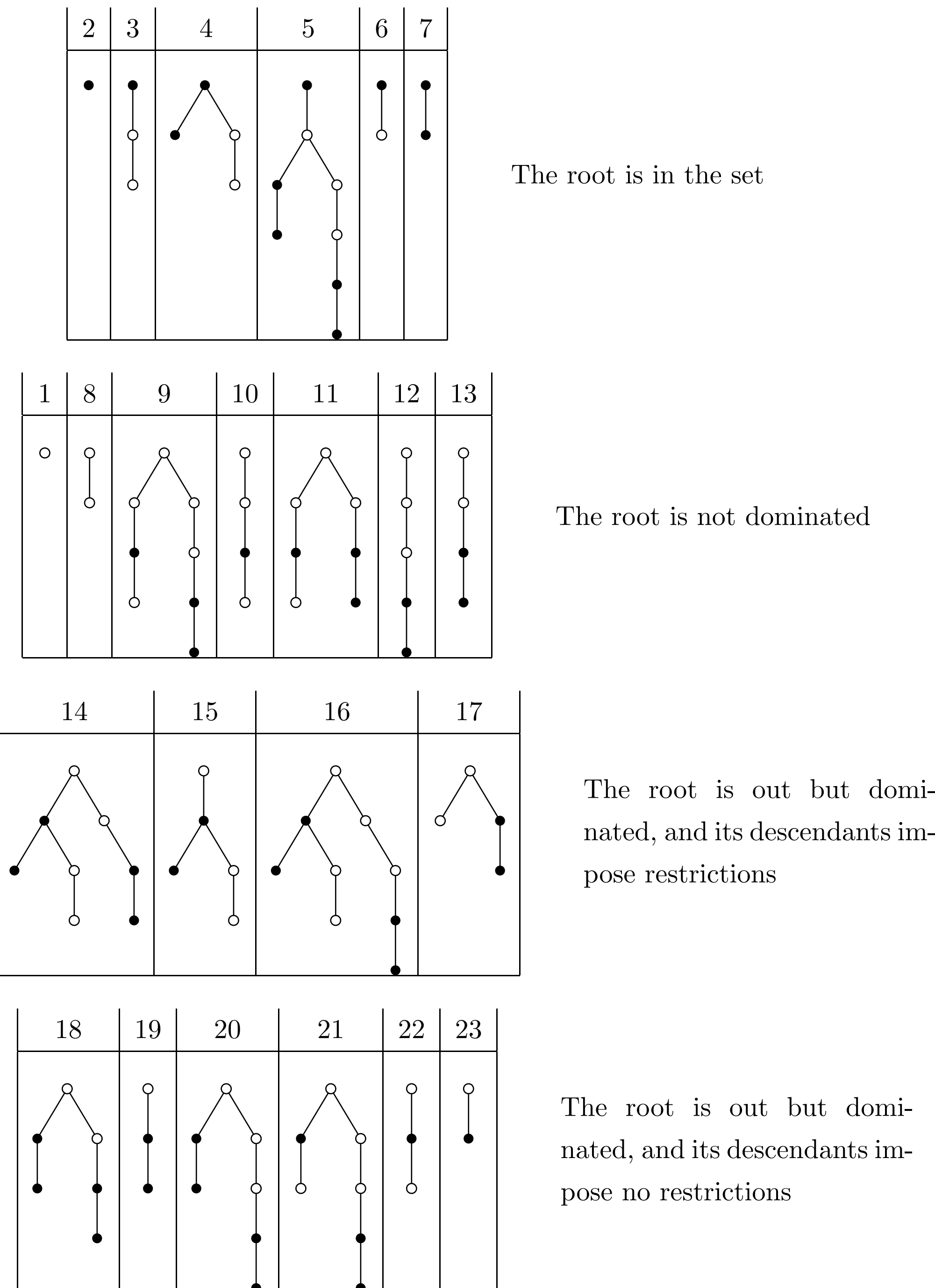


Figure 5: The claimed list of live representatives for external redundance.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	8	23	–	15	14	22	19	–	–	–	–	8	12	–	–	–	13	13	13	13	8	11	1
2	6	7	7	7	–	7	7	3	4	6	6	4	6	2	2	5	–	2	2	5	5	2	2
3	–	4	4	4	–	4	4	–	–	–	–	–	–	3	3	–	–	3	3	–	–	3	3
4	–	4	4	4	–	4	4	–	–	–	–	–	–	4	4	–	–	4	4	–	–	4	4
5	–	–	–	–	–	–	–	–	–	–	–	–	–	5	5	5	–	5	5	5	5	5	5
6	6	6	6	6	–	6	6	–	–	6	6	–	6	6	6	–	–	6	6	–	–	6	6
7	6	7	7	7	–	7	7	4	4	6	6	4	6	7	7	–	–	7	7	–	–	7	7
8	8	–	–	–	–	–	17	–	–	–	–	8	8	–	–	–	8	8	8	8	8	8	8
9	8	23	–	16	14	21	20	–	–	–	–	8	9	–	–	–	10	10	10	10	8	9	9
10	8	23	–	14	14	23	18	–	–	–	–	8	10	–	–	–	10	10	10	10	8	10	10
11	8	23	–	15	14	22	19	–	–	–	–	8	9	–	–	–	10	10	10	10	8	11	11
12	8	23	–	16	14	21	20	–	–	–	–	8	12	–	–	–	13	13	13	13	8	9	12
13	8	23	–	14	14	23	18	–	–	–	–	8	13	–	–	–	13	13	13	13	8	10	13
14	–	23	–	23	23	23	23	–	–	–	–	–	14	–	–	–	14	14	14	14	–	14	14
15	–	23	–	22	23	22	22	–	–	–	–	–	16	–	–	–	14	14	14	14	–	15	15
16	–	23	–	21	23	21	21	–	–	–	–	–	16	–	–	–	14	14	14	14	–	16	16
17	17	–	–	–	–	–	–	–	–	–	–	17	17	–	–	–	17	17	17	17	17	17	17
18	17	23	–	23	23	23	23	–	–	–	–	17	18	–	–	–	18	18	18	18	17	18	18
19	17	23	–	22	23	22	22	–	–	–	–	17	20	–	–	–	18	18	18	18	17	19	19
20	17	23	–	21	23	21	21	–	–	–	–	17	20	–	–	–	18	18	18	18	17	20	20
21	–	23	–	21	23	21	21	–	–	–	–	–	21	–	–	–	23	23	23	23	–	21	21
22	–	23	–	22	23	22	22	–	–	–	–	–	21	–	–	–	23	23	23	23	–	22	22
23	–	23	–	23	23	23	23	–	–	–	–	–	23	–	–	–	23	23	23	23	–	23	23

Valid classes: 2,4,5,6,7,13,17,18,19,20,22,23

Figure 6: The claimed table for external redundance.

References

[1] S. Arnborg and A. Proskurowski, Linear time algorithms for NP-hard problems restricted to partial k-trees, Discr Appl. Math. 23 (1989), 11–24.

[2] M. Bern, E. Lawler, and A. Wong, Linear-time computation of optimal subgraphs of decomposable graphs, J. Algorithms 8 (1987), 216–235.

[3] H.L. Bodlaender, "Dynamic programming on graphs with bounded treewidth," Automata, languages and programming (Tampere, 1988), Springer, Berlin, 1988, Vol. 317 of Lecture Notes in Comput. Sci., pp. 105–118.

[4] R. Borie, R. Parker, and C. Tovey, Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families, Algorithmica 7 (1992), 555–581.

[5] B. Courcelle, "Recognizable sets of unrooted trees," Tree automata and languages, North-Holland, Amsterdam, 1992, pp. 141–158.

[6] G. Domke, J. Hattingh, S. Hedetniemi, R. Laskar, and L. Markus, Restrained domination in graphs, Discr Math. 203 (1999), 61–69.

[7] F. Gécseg and M. Steinby, Tree automata, Akadémiai Kiadó (Publishing House of the Hungarian Academy of Sciences), Budapest, 1984.

[8] T. Hagerup, Dynamic algorithms for graphs of bounded treewidth, Algorithmica 27 (2000), 292–315.

[9] T. Haynes, S. Hedetniemi, and P. Slater, Fundamentals of domination in graphs Vol. 208 of *Monographs and Textbooks in Pure and Applied Mathematics*, Marcel Dekker, New York, 1998.