A General Framework for Enumerating Equivalence Classes of Solutions

Yishu Wang
Université de Lyon, Université Lyon 1, CNRS,
Laboratoire de Biométrie et Biologie Evolutive UMR 5558, F-69622 Villeurbanne, France
Inria Grenoble Rhône-Alpes, Villeurbanne, France

Arnaud Mary
Université de Lyon, Université Lyon 1, CNRS,
Laboratoire de Biométrie et Biologie Evolutive UMR 5558, F-69622 Villeurbanne, France
Inria Grenoble Rhône-Alpes, Villeurbanne, France

Marie-France Sagot
Inria Grenoble Rhône-Alpes, Villeurbanne, France
Université de Lyon, Université Lyon 1, CNRS,
Laboratoire de Biométrie et Biologie Evolutive UMR 5558, F-69622 Villeurbanne, France

Blerina Sinaimeri
Luiss University, Rome, Italy
ERABLE team, Inria Grenoble Rhône-Alpes, Villeurbanne, France

Abstract

When a problem has more than one solution, it is often important, depending on the underlying context, to enumerate (i.e., to list) them all. Even when the enumeration can be done in polynomial delay, that is, spending no more than polynomial time to go from one solution to the next, this can be costly as the number of solutions themselves may be huge, including sometimes exponential. Furthermore, depending on the application, many of these solutions can be considered equivalent. The problem of an efficient enumeration of the equivalence classes or of one representative per class (without generating all the solutions), although identified as a need in many areas, has been addressed only for very few specific cases. In this paper, we provide a general framework that solves this problem in polynomial delay for a wide variety of contexts, including optimization ones that can be addressed by dynamic programming algorithms, and for certain types of equivalence relations between solutions.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Mathematics of computing → Graph enumeration; Theory of computation → Backtracking

Keywords and phrases Enumeration algorithms, Equivalence relation, Dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.80

Related Version *Full Version*: `https://arxiv.org/abs/2004.12143`

1 Introduction

Enumerating the solutions of an optimization problem solved by a dynamic programming algorithm (DP-algorithm) is a classical and well-known question. However, many enumeration problems have a huge number of solutions in practice, which might be an issue. From a computational point of view, since the number of solutions is a lower bound on the time complexity of any enumeration algorithm, it might make the algorithm impractical on real instances. Furthermore, even if the number of solutions is reasonable enough to be enumerated, the purpose of some applications is to give the output of the algorithm to a human specialist (this is necessary, for example, when some of the constraints of the problem are subjective and cannot be modeled).

Indeed, one of the advantages of an enumeration algorithm compared to an optimization one which in general outputs only one optimal solution, is to be able to understand the space of solutions. While this is important in many cases, no human can understand an output composed of billions of solutions.

The approach generally used to address this consists of enumerating all solutions, and then applying some type of clustering (grouping) algorithm to the set of optimal solutions. The final output presented to the user would then be some "representative description" of the clusters (groups) themselves. However, since the number of solutions is a lower bound for the total execution time of any enumeration algorithm, the first step of such a strategy becomes impossible when the number of solutions is too big. A natural question is then whether it would be possible to enumerate directly what we just called a "representative description" of the clusters of solutions. This could be for instance an element per cluster. Sometimes a cluster can also be seen as a set of characteristics that the solutions within the cluster share. In such a case, the representative description of a cluster could then be such a set of characteristics. A particularly convenient situation is however when the clusters correspond to equivalence classes of an equivalence relation over the set of solutions that we could establish *a priori*. The output could be in this case the quotient space of the equivalence relation. Notice that the enumeration of equivalence classes of solutions is a combinatorial problem that could be solved exactly given a well-defined equivalence relation, and unlike data analysis methods such as incremental clustering, it does not require the definition of a similarity or dissimilarity measure between solutions which, depending on the mathematical nature of the solutions (numerical values, graphs, functions on graphs, etc.), can be difficult to define or costly to compute.

The problem this paper addresses is how to perform the task of enumerating equivalence classes of solutions with polynomial delay for a wide variety of problems (including optimization problems solved by dynamic programming algorithms), for certain types of equivalence relations between solutions.

The problem of enumerating equivalence classes, and particularly the generation of representative solutions is a challenge in the context of enumeration algorithms. It has been identified as a need in different areas, such as Genome Rearrangements [10], Artificial Intelligence [1] or Pattern Matching [7, 26]. It was listed as an important open problem in a recent Dagstuhl workshop on "Algorithmic Enumeration: Output-sensitive, Input-Sensitive, Parameterized, Approximative" (see, e.g., Sections 4.2 and 4.10 in [15]). To the best of our knowledge, this challenge has been addressed only for some few specific problems in the literature (e.g., [2, 10, 24, 25]).

To enumerate equivalence classes, we go through an intermediate problem, namely the enumeration of colored subtrees in acyclic decomposable AND/OR graphs (ad-AND/OR graph). The paper is organized as follows: Section 2 provides an algorithm to enumerate with polynomial delay colored subtrees in ad-AND/OR graphs; Section 3 details how this algorithm applies to the enumeration of equivalence classes in DP-problems. In that direction, we present some examples from well-known optimization problems in the literature. Finally, in Section 4 we conclude with some open problems.

2 Enumeration of colored subtrees in an ad-AND/OR graph

2.1 AND/OR graphs and solution subtrees

An *AND/OR graph* (see, for example [23, 27]) is a well-known structure in the field of Logic and Artificial Intelligence (AI) that represents problem solving and problem decomposition. In this paper, we consider a particular flavor of AND/OR graphs known as *explicit AND/OR graphs for trees* [12].

This is a directed acyclic graph (DAG) G which explicitly represents an *AND/OR state space* for solving a certain problem by decomposing it into subproblems. The set of nodes (or states) $S := V(G)$ contains OR and AND nodes (the OR nodes represent alternative ways for solving the problem while the AND nodes represent problem decomposition into subproblems, all of which need to be solved). There is a set of goal nodes $S_g \subseteq S$ and a set of start nodes $S_0 \subseteq S$ representing respectively the terminal states and the initial states. The children (out-neighbors) of an OR node are AND nodes, and the children of an AND node are OR nodes or goal nodes. We say that a node is an OR^+ node when it is either an OR node or a goal node. Furthermore, the AND/OR graphs that we consider must have the property of being *decomposable* (they can model a problem for which every decomposition yields disjoint subproblems that can be solved independently): for any AND node, the sets of nodes that are reachable from each one of its child nodes are pairwise disjoint. The example graph in Figure 1 is decomposable.

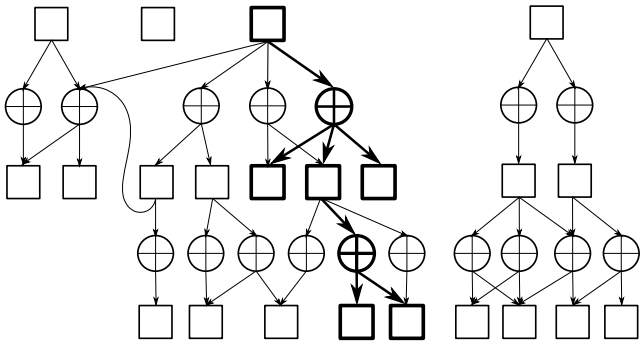

Figure 1 An acyclic decomposable AND/OR graph with four start nodes. Squares are OR^+ nodes (OR nodes or goal nodes); crossed circles are AND nodes. One solution subtree of size 8 is shown in bold.

Formally, in this paper, any graph that satisfies the properties in Definition 1 will be called an ad-AND/OR graph. Notice that this definition corresponds only to a particular case of the general AND/OR graphs in the AI literature; the latter may be neither acyclic nor decomposable.

▶ **Definition 1** (ad-AND/OR graph). *A directed graph G is an* acyclic decomposable AND/OR graph*, henceforth denoted by ad-AND/OR graph, if it satisfies the following:*

- *G is a DAG.*
- *G is bipartite: its node set $V(G)$ can be partitioned into $(\mathcal{A}, \mathcal{O})$ so that all arcs of G are between these two sets. Nodes in $\mathcal{A}$ are called* AND nodes*; nodes in $\mathcal{O}$ are called* OR^+ nodes*.*
- *Every AND node has in-degree at least one and out-degree at least one. The set of nodes with out-degree zero is then a subset of $\mathcal{O}$ and is called the set of* goal nodes*; the remaining OR^+ nodes are simply the* OR nodes*. The subset of OR nodes of in-degree zero is the set of* start nodes*.*
- *G is decomposable: for any AND node, the sets of nodes that are reachable from each one of its child nodes are pairwise disjoint.*

▶ **Definition 2** (solution subtree). *A* solution subtree *T of an ad-AND/OR graph G is a subgraph of G which: (1) contains exactly one start node; (2) for any OR node in T it contains one of its child nodes in G, and for any AND node in T it contains all its children in G.*

It is immediate to see that a solution subtree is indeed a subtree of G: it is a rooted tree, the root of which is a start node. If we would drop the requirement of G being decomposable, the object defined in Definition 2 would not be guaranteed to be a tree. One solution subtree of the example graph in Figure 1 is shown in bold.

The set of all solution subtrees of G is denoted by $\mathbb{T}(G)$. Given an ad-AND/OR graph G, counting the number of its solution subtrees and enumerating all solution subtrees can be solved by folklore approaches based on depth-first search (DFS).

Before going further, we recall that the motivation of this paper is concerned with solutions of dynamic programming problems. The correspondence between the solutions of DP-style recurrence equations and the solution subtrees of general AND/OR graphs has been formally proven in [16]. In the case where the underlying graph is acyclic, the recurrence equations can be solved efficiently by DP-algorithms. While we will now concentrate on the solution subtrees of an ad-AND/OR graph and on the equivalence classes of solution subtrees, we will demonstrate in Section 3 how to apply our algorithms to analyze equivalence classes of solutions of a very general class of problems solvable by DP. It is important to point out that solution subtrees of general AND/OR graphs are equivalent to various other well-known formalisms, e.g., acceptance trees of a nondeterministic tree automata, languages of regular tree grammars, complete subcircuits of tropical circuits. The reader who is more familiar with those may also find this paper interesting even outside of a dynamic programming context.

2.2 Equivalence classes

Let G be an ad-AND/OR graph. Let C be an **ordered** set of colors. We will consider equivalence relations on the set of solution subtrees of G which are based on a local comparison of the colors of the OR^+ nodes. Intuitively, two OR^+ nodes having the same color represent two alternative ways of solving the problem that can be considered equivalent.

▶ **Definition 3** (e-coloring). *An ad-AND/OR graph G is* e-colored *if its OR^+ nodes are colored in such a way that for any AND node all its children have distinct colors.*

Notations. If s is a OR^+ node of G, we denote by $c(s)$ its color. If s is an AND node, we denote by $\widetilde{C}(s)$ the tuple of colors of the children of s sorted in increasing order of the colors. If $T_1 \in \mathbb{T}(G)$ is a solution subtree of G, we use the notation $\pi(T_1)$ for the result of contracting the AND nodes in T_1: for each OR node s of T, contract the only child node of s in T (i.e., remove the child and connect s to each one of its "grandchildren").

▶ **Definition 4** (equivalence class). *A node-colored rooted tree T is an* equivalence class *of solution subtrees of an e-colored ad-AND/OR graph G (or shortly an equivalence class of G) if there exists a solution subtree T_1 of G such that $\pi(T_1)$ is equal to T. Such a T_1 is said to be a solution subtree* belonging to *the class T.*

More notations. We denote by $\mathbb{C}(G)$ the set of equivalence classes of G. The notation π can be seen as a function $\pi\colon \mathbb{T}(G) \to \mathbb{C}(G)$. We denote by $\pi^{-1}(T) := \{T_1 \in \mathbb{T}(G) \mid \pi(T_1) = T\}$ the subset of solution subtrees of G belonging to the class T. The notations $c(s)$ and $\widetilde{C}(s)$ are naturally extended to the case where s is a node in an equivalence class T. The root node of a rooted tree T is denoted by $r(T)$. The set of the children of a node s is denoted by $Ch(s)$.

An example of an e-colored ad-AND/OR graph with five equivalence classes is given in Figure 2.

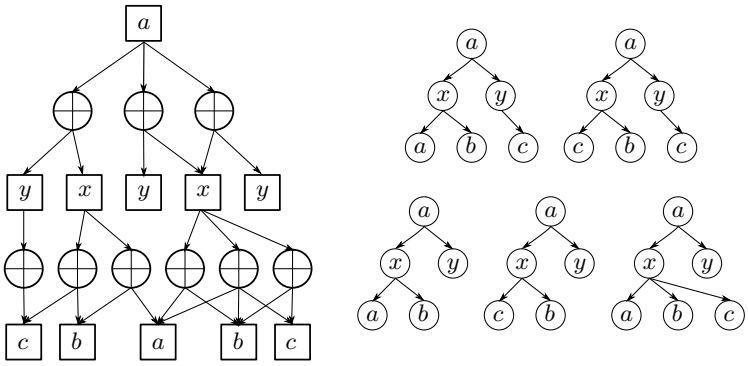

Figure 2 An e-colored ad-AND/OR graph and its five equivalence classes. The colors of the OR$^+$ nodes are written inside the squares.

2.3 Enumerating equivalence classes

Given an e-colored ad-AND/OR graph G, we propose a polynomial delay algorithm to enumerate all equivalence classes of G. Given a total ordering $c_1, \ldots, c_m$ of the colors of G, we define a total ordering $\prec$ over $\mathbb{C}(G)$, the set of equivalence classes of G. If T and T' have their roots colored differently, we say that T is smaller than T', denoted by $T \prec T'$, if the root color of T precedes the one of T'. If T and T' have the same root color, let $(T_1, \ldots, T_k)$ (resp. $(T'_1, \ldots, T'_\ell)$) be the child subtrees of $r(T)$ (resp. $r(T')$) sorted recursively with respect to $\prec$. We then say that T is smaller than T' if the tuple $(T_1, \ldots, T_k)$ is lexicographically smaller than $(T'_1, \ldots, T'_\ell)$, i.e., if $T_i \prec T'_i$ with i being the smallest index such that $T_i \neq T'_i$. We also assume that $\emptyset$ is smaller than any tree, and therefore a single node tree colored with color c comes before any other tree whose root is colored with c in $\prec$.

2.3.1 Definitions and notations

Recall that given an AND-node x, $\widetilde{C}(x)$ is the tuple of colors of the children of x sorted in increasing order. Given an OR-node o, we denote by $\mathcal{T}(o)$ the set of color tuples of its children, i.e., $\mathcal{T}(o) := \{\widetilde{C}(x) : x \in Ch(o)\}$. In other words, a color tuple $(c_1, \ldots, c_j)$ belongs to $\mathcal{T}(o)$ if o has an AND-child node whose children are colored with $(c_1, \ldots, c_j)$. If we consider an equivalence class T of $\mathbb{C}(G/\{o\})$ rooted at o, the tuples of $\mathcal{T}(o)$ are precisely the possible colorings of the children of $r(T)$. Indeed, if the AND-child node $x \in Ch(o)$ is chosen in a solution subtree, then $\widetilde{C}(x)$ will be the colors of the children of o in that solution. Notice that several AND-children nodes of o may have the same color tuple.

We extend this definition to a set $\mathcal{O}$ of OR$^+$ nodes with $\mathcal{T}(\mathcal{O}) = \bigcup_{o \in \mathcal{O}} \mathcal{T}(o)$. In the same way, $t = (c_1, \ldots, c_j)$ is a color tuple of $\mathcal{T}(\mathcal{O})$ if and only if there exists an equivalence class T of $\mathbb{C}(G/\mathcal{O})$ such that the children of $r(T)$ are colored with $(c_1, \ldots, c_j)$. Given a set $\mathcal{O}$ of OR$^+$ nodes, we denote by $t_1, \ldots, t_{|\mathcal{T}(\mathcal{O})|}$ the different color tuples of $\mathcal{T}(\mathcal{O})$ ordered lexicographically, and we denote by $Ch^\ell(\mathcal{O})$ the set of AND-nodes in $Ch(\mathcal{O})$ whose color tuple is t_ℓ, i.e., $Ch^\ell(\mathcal{O}) = \{x \in Ch(\mathcal{O}) : \widetilde{C}(x) = t_\ell\}$. The sets $Ch^1(\mathcal{O}), \ldots, Ch^{|\mathcal{T}(\mathcal{O})|}(\mathcal{O})$ form a partition of $Ch(\mathcal{O})$, each part corresponding to a color tuple $t_i \in \mathcal{T}(\mathcal{O})$.

Finally, given a color tuple $t_\ell := (c_1, \ldots, c_j) \in \mathcal{T}(\mathcal{O})$, for each $i \leq j$, by Definition 3, each node of $Ch^\ell(\mathcal{O})$ has exactly one child with color c_i. We denote by C_i^ℓ the set of children of $Ch^\ell(\mathcal{O})$ colored with c_i, i.e., $C_i^\ell = \{o \in Ch(x) : x \in Ch^\ell(\mathcal{O}),\ c(o) = c_i\}$ (it is a set of "grandchildren" of $\mathcal{O}$).

Algorithm 1 Next solution.

Input: A set $\mathcal{O}$ of OR^+ nodes having all the same color c and an equivalence class T of $G/\mathcal{O}$
Output: The equivalence class T' of $G/\mathcal{O}$ that follows T w.r.t the $\prec$ ordering.
Function `Next(`T`,`$\mathcal{O}$`)`**:**
  if $T = \emptyset$ *and* $\mathcal{O}$ *contains terminal nodes* **then**
    Return A tree with a single root node colored with c
  end
  $r \leftarrow 0$
  if $T = \emptyset$ *or* T *is a single node tree* **then**
    $r \leftarrow r + 1$
    if $r > |\mathcal{T}(\mathcal{O})|$ **then**
      Return $\perp$
    end
    Let $(c_1, \ldots, c_j)$ be the color tuple of $t_r \in \mathcal{T}(\mathcal{O})$ and let $T_1, \ldots, T_j \leftarrow \emptyset$
    $\mathcal{O}_1 \leftarrow C_1^r$
    $\ell \leftarrow 1$
  else
    Let r be such that $t_r \in \mathcal{T}(\mathcal{O})$ is the root color tuple $\widetilde{C}(r(T))$
    Let $(T_1, \ldots, T_j)$ be the child subtrees of $r(T)$, the roots of which are colored respectively with $(c_1, \ldots, c_j) := t_r$
    For all $i \leq j$, let $\mathcal{O}_i \subseteq C_i^r$ be the set of nodes in C_i^r compatible with $(T_1, \ldots, T_{i-1})$
    Let ℓ be the largest index $i \leq j$ such that `Next(`T_i`,`$\mathcal{O}_i$`)` $\neq \perp$ if such index exists. Otherwise, $T \leftarrow \emptyset$ and go to line 8
  end
  $T_\ell \leftarrow$ `Next(`T_ℓ`,`$\mathcal{O}_\ell$`)`
  for $\ell < i \leq j$ **do**
    Let $\mathcal{O}_i \subseteq C_i^r$ be the set of nodes in C_i^r compatible with $(T_1, \ldots, T_{i-1})$
    $T_i \leftarrow$ `Next(`$\emptyset$`,`$\mathcal{O}_i$`)`
  end
  Return A tree with root color c and root child subtrees $(T_1, \ldots, T_j)$

In the left panel of Figure 3, an example graph is shown where each node is labeled by an integer. The colors are, in increasing order, w, x, y, and z. For $\mathcal{O} = \{1, 2\}$, the set $\mathcal{T}(\mathcal{O})$ contains the three tuples $t_1 = (w, y)$, $t_2 = (x, y)$, and $t_3 = (x, y, z)$. We have $Ch^1(\mathcal{O}) = \{6\}$, $Ch^2(\mathcal{O}) = \{4, 5\}$, and $Ch^3(\mathcal{O}) = \{3\}$. The sets C_i^ℓ are $C_1^1 = \{12\}$, $C_2^1 = \{11\}$, $C_1^2 = \{9, 10\}$, $C_2^2 = \{8, 11\}$, $C_1^3 = \{9\}$, $C_2^3 = \{8\}$, and $C_2^3 = \{7\}$.

2.3.2 Algorithm description

Notice that by definition of $\prec$, given a set of OR^+ nodes $\mathcal{O}$, all having the same color c, and a color tuple $t_\ell \in \mathcal{T}(\mathcal{O})$, all equivalence classes T of $G/\mathcal{O}$ such that $\widetilde{C}(r(T)) = t_\ell$ are consecutive with respect to $\prec$.

The algorithm outputs the equivalence classes in ascending order with respect to $\prec$. Given an equivalence class T of $G/\mathcal{O}$ for a set of OR^+ nodes $\mathcal{O}$ of color c, it will output the equivalence class T' of $G/\mathcal{O}$ that succeeds T w.r.t. $\prec$ if it exists or output the symbol $\perp$ if T is the last solution.

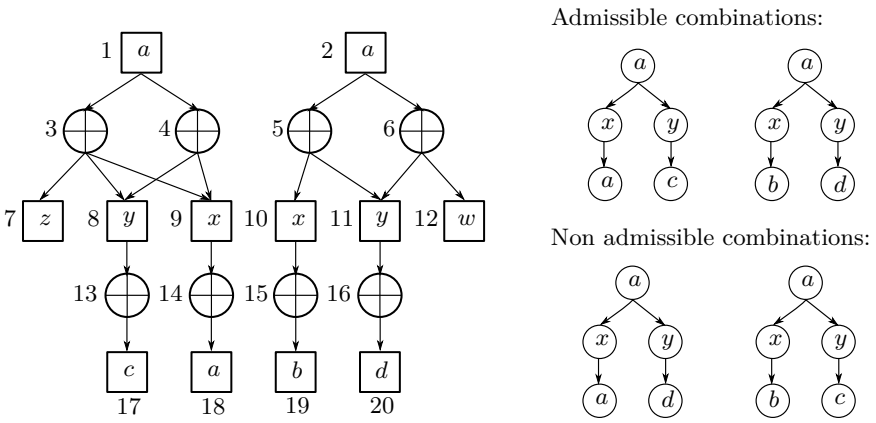

Figure 3 Left panel: An e-colored ad-AND/OR graph. Right panel: For $\mathcal{O} = \{1, 2\}$, there are four combinations between $\mathbb{C}(G/C_1^2)$ and $\mathbb{C}(G/C_2^2)$; only two of them are admissible.

Assume that the children of $r(T)$ are colored with the *root color tuple* $(c_1, \ldots, c_j) =: t_r \in \mathcal{T}(\mathcal{O})$ and let $(T_1, \ldots, T_j)$ be the child subtrees of T, the roots of which are colored with the tuple t_r. Notice that for all $i \leq j$, T_i is an equivalence class of G/C_i^r. The algorithm will output the next equivalence class T' such that $\widetilde{C}(r(T')) = t_r$ if there remains one (same color tuple at the root), or it will output the first solution such that $\widetilde{C}(r(T')) = t_{r+1} \in \mathcal{T}(\mathcal{O})$ otherwise (the next color tuple at the root).

To find the next solution corresponding to the root color tuple t_r, the algorithm will replace recursively T_j by its successor T'_j w.r.t. $\prec$ if there exists one. We obtain the solution T' whose subtrees are $(T_1, \ldots, T_{j-1}, T'_j)$ which is by definition the successor of T in $\prec$ whenever T'_j is the successor of T_j. If T_j has no successor (that is, if it is the last one), we replace if possible T_{j-1} by its successor T'_{j-1} and we replace T_j by the smallest admissible solution (i.e., the successor of $\emptyset$). In general, we select at each step the greatest index ℓ such that T_ℓ has a successor w.r.t. $\prec$, we replace it by its successor T'_ℓ and we take the smallest admissible solution for every $\ell < i \leq j$.

Without further care, the above described procedure would output solutions whose child subtrees $(T_1, \ldots, T_j)$ of the root correspond to the elements of the Cartesian product of $\mathbb{C}(G/C_i^r)$, $i \leq j$. However, while it is true that if T is a solution, its child subtree T_i is an equivalence class of G/C_i^r for all $i \leq j$, the converse is not true. Indeed, not all elements of $\mathbb{C}(G/C_1^r) \times \cdots \times \mathbb{C}(G/C_j^r)$ lead to an admissible solution (an example is given in the right panel of Figure 3). In order to find an admissible solution, we should guarantee that the choice of a given T_i is *compatible* with the previous choices $(T_1, \ldots, T_{i-1})$. This is done by selecting the subset of OR^+ nodes $\mathcal{O}_i \subseteq C_i^r$ that are compatible with $(T_1, \ldots, T_{i-1})$ (see Definition 5 below). An admissible choice of T_i will then be any equivalence class of $G/\mathcal{O}_i$. The two key properties are that the set $\mathcal{O}_i$ can be easily computed, and that it is never empty, i.e., there is always a choice for T_i that is compatible with the previous choices of $(T_1, \ldots, T_{i-1})$ (there is at least one choice that corresponds to the current solution). Notice that if the latter were not true, the algorithm would not have a polynomial delay complexity since we may spend exponential time without reaching a final solution. With this property, we are guaranteed that we can always extend a partial tuple $(T_1, \ldots, T_i)$ until we reach a complete tuple $(T_1, \ldots, T_j)$ that will form a solution.

Compatible nodes

Given a set of OR^+ nodes $\mathcal{O}$ all colored with the same color c and a tree T of $\mathbb{C}(G/\mathcal{O})$, we denote by $r(\pi^{-1}(T)) := \{r(S) : S \in \pi^{-1}(T)\}$ the subset of OR^+ nodes of $\mathcal{O}$, each one of which is the root of a solution subtree of class T. The following definition formalizes the notion of compatible nodes mentioned previously.

▶ **Definition 5.** *Let $\mathcal{O}$ be a set of OR^+ nodes of color c, $t_r =: (c_1, \ldots, c_j) \in \mathcal{T}(\mathcal{O})$, and let $T_1, \ldots, T_k$, with $k < j$, be respectively equivalence classes of $\mathbb{C}(G/C_i^r)$ for all $i \leq k$. We say that a node $o \in C_{k+1}^r$ is* compatible *with $(T_1, \ldots, T_k)$ if there exists an AND-node $x \in Ch^r(\mathcal{O})$ such that o is a child of x and such that $r(\pi^{-1}(T_i))$ contains a child of x for all $i \leq k$.*

2.3.3 Analysis

▶ **Proposition 6.** *Let T be an equivalence class of $G/\mathcal{O}$ for a set $\mathcal{O}$ of OR^+ nodes of G, all colored with the same color c. Then, the function* `Next` *of Algorithm 1 is such that:*

1. *`Next`$(\emptyset, \mathcal{O})$ returns the smallest equivalence class of $G/\mathcal{O}$ w.r.t. $\prec$.*
2. *`Next`$(T, \mathcal{O})$ returns the equivalence class of $G/\mathcal{O}$ that follows T w.r.t. $\prec$.*
3. *if T is the last equivalence class of $G/\mathcal{O}$, `Next`$(T, \mathcal{O})$ returns $\perp$.*

Due to space constraints, some proofs are omitted and can be found in the full version of this paper [36].

▶ **Theorem 7.** *Given an e-colored ad-AND/OR graph G, the set $\mathbb{C}(G)$ can be enumerated with delay $O(n \cdot s)$ where n is the number of nodes of G and s is the maximum size of a solution.*

Proof. To enumerate $\mathbb{C}(G)$, we first split the start nodes of G into sets $S_0, \ldots, S_k$ according to their colors. For each set S_i, starting with $T = \emptyset$, we repeatedly assign $\texttt{Next}(T, S_i)$ to T and output it until $T = \perp$. By Proposition 6, this guarantees that we output every solution of $\mathbb{C}(G/S_i)$ exactly once. Since any solution of $\mathbb{C}(G)$ belongs to $\mathbb{C}(G/S_i)$ for a given $i \leq k$, every solution of $\mathbb{C}(G)$ will be outputted exactly once.

For the complexity, notice that at most one recursive call is performed by the node of the next solution. More precisely, if $\texttt{Next}(T, \mathcal{O}) = T'$, there will be exactly one recursive call per node in T' that is not in T, and thus at most s recursive calls will be performed.

In each recursive call, both the set $\mathcal{T}(\mathcal{O})$ and the partition $\{C_i^r\}_{i \leq j}$ of grandchildren of $\mathcal{O}$ can be computed in $O(n)$ time. It remains to show that the sets of compatible nodes $\mathcal{O}_i$, $i \leq j$, can be computed in $O(n)$ time in total which will conclude the proof. To do this, we should be able to compute the sets $r(\pi^{-1}(T_i))$ for all $i \leq j$. If $\texttt{Next}(T, \mathcal{O}) = T'$, the easiest way is to return the set $r(\pi^{-1}(T'))$ together with T' when the call $\texttt{Next}(T, \mathcal{O})$ returns. This could be done by observing that if $T \in \mathbb{C}(G/\mathcal{O})$ where $\mathcal{O}$ is a set of goal nodes all having the same color c, then $r(\pi^{-1}(T)) = \mathcal{O}$, and if T has child subtrees $T_1, \ldots, T_j$ then $r(\pi^{-1}(T))$ is the set of nodes of $\mathcal{O}$ that has at least an AND-child x such that the children of x contain exactly one node in $r(\pi^{-1}(T_i))$ for each $i \leq j$, which can be found in $O(n)$ time. Thus only $O(n)$ time is necessary at each recursive call to return $r(\pi^{-1}(T'))$ in addition to T'. ◀

3 Application to dynamic programming

3.1 A formalism for tree-sequential dynamic programming

Since its introduction by Karp and Held [21], *monotone sequential decision processes* (mSDP) have been the classical model for problems solvable by dynamic programming (DP). This formalism is based on finite-state automata. The solutions of DP-problems are thus equivalent

to languages of regular expressions, or to paths in directed graphs. It is known that Bellman's *principle of optimality* [5] also applies to problems for which the solutions are not sequential but *tree-like* [9]. Various generalizations have been proposed to characterize broader classes of problems solvable by DP or DP-like techniques [11, 19]. In this paper, we consider a framework which is the immediate generalization of the mSDP model, i.e., generalizing finite automata (regular expressions, paths in DAGs) to finite tree automata (regular tree grammars, solution trees of general AND/OR graphs). Further generalizations exist (from trees to graphs of treewidth > 1); the collection of these methods is known as *Non-serial dynamic programming* [6].

In this model, a tree-sequential problem can be specified by a finite (bottom-up) tree automaton $A = (Q, \Sigma, \delta, q_0, Q_F)$, where Q is a finite set of states, Σ is a ranked alphabet, δ is a set of transition rules of the form $(q_1, \ldots, q_n, a, q)$ where $q_1, \ldots, q_n, q \in Q$ and $a \in \Sigma$, $q_0 \in Q$ is the initial state, $Q_F \subseteq Q$ is a set of final states. The problem specification also includes a cost function. The set $L(A)$ of trees accepted by the tree automaton A defines the set of feasible solutions. The minimization problem seeks to minimize the cost function over the set $L(A)$ of feasible solutions.

We will consider the simple case of a positive additive cost function that always equals zero in the initial state. An additive cost function can be defined via an incremental cost function $I\colon Q^* \times \Sigma \to \mathbb{R}$, where Q^* consists of tuples of states in Q of the form $(q_1, \ldots, q_n)$. $I(q_1, \ldots, q_n, a)$ can be viewed as the cost of attaching n child subtrees to a new root of symbol a. While it might seem restrictive to require an additive structure on the cost function, this simple case does cover many important problems admitting a DP-algorithm, for instance, Travelling Salesman [4, 18], Knapsack [22], or Levenshtein distance [34].

In this case, the answer of the minimization problem can be shown to be equal to $\min_{q \in Q_F} D(q)$, where $D : Q \to \mathbb{R}_{\geq 0}$ is defined by the following recurrence equations:

$$\begin{aligned} &D(q_0) = 0\,, \\ &\text{for } q \neq q_0\,, \quad D(q) = \min_{(q_1,\ldots,q_n,a,q)\in\delta} \sum_{1\leq i\leq n} D(q_i) + I(q_1, \ldots, q_n, a)\,. \end{aligned} \tag{1}$$

A *dynamic programming algorithm* for the minimization problem corresponds to an algorithm that computes D; the function D is commonly called a *dynamic programming table* (a DP-tabled, also called a DP-array, or a DP-matrix). Needless to say, such an algorithm does not exist in general for given arbitrary tree automata and cost functions [20].

Using an algebraic approach, Gnesi and Montanari [16] have shown that solving the functional Equation (1) corresponds to finding the solution subtrees of a general AND/OR graph. An important special case in which DP-algorithms exist is when the underlying AND/OR graph is acyclic.

When a fixed tree is given as an input to the problem, the underlying AND/OR graph is acyclic and decomposable (that is, it is an ad-AND/OR graph). Such problems are hence naturally solvable by DP-algorithms. These algorithms are known in folklore under the name *Dynamic programming on a tree*. Many graph-theoretical problems (e.g., maximum matching, longest path) can be solved optimally on trees by DP-algorithms. Numerous real-world applications also rely on DP-algorithms on trees; examples can be found, for instance, in Data Science [30], Computer Vision [14, 33], and Computational Biology [3, 13].

Explicit construction of the ad-AND/OR graph for DP on a fixed tree

Due to its usefulness for the examples that we will develop next, in the case of DP on a fixed tree, an explicit construction of the ad-AND/OR graph from Equation (1) is described below. The construction is done in two steps. In the first step, we build a graph in which every node

retains an additional attribute, its *value*, and every OR^+ node is labeled by a state $q \in Q$. In the second step, we *prune* the graph by removing nodes that do not yield optimal values.

1. For each $(q_0, a, q) \in \delta$, create a goal node of value 0 labeled by q. Then, for each $q \neq q_0$ in post-order,
   i. For each $(q_1, \ldots, q_n, a, q) \in \delta$, create an AND node, connect it to the n OR^+ nodes labeled by $q_1, \ldots, q_n$. Its value is equal to the sum of the values of its children, plus $I(q_1, \ldots, q_n, a)$.
   ii. Create a single OR node, connect it to every AND node created in the previous step. Its label is q, and its value is the minimum of the values of its children.
2. For each $q \in Q_F$, remove the OR node labeled by q unless its value is equal to $\min_{q \in Q_F} G(q)$. For each OR node s, remove the arc to its AND-child node s_i if the value of s_i is not equal to the value of s. Finally, remove recursively all AND nodes without incoming arcs.

3.2 Examples

3.2.1 Optimal tree coloring problem

Description

A prototypical problem that fits into the framework of DP on a tree is Optimal tree coloring, that is, finding an optimal node-coloring of the input tree. Many problems of practical interest reduce to Optimal tree coloring; three concrete examples are given later in this section.

If T is the input (rooted, ordered) tree and C is the set of colors, such a problem seeks a coloring $\phi\colon V(T) \to C$ that minimizes the cost function. There can be many constraints on the coloring function: some nodes of T may be forced to have a certain color, the possible colors of a node may depend on the colors of its descendants. In our tree-sequential dynamic programming formalism, a tree automaton and a cost function are given as part of the input. The tree automaton defines the set $L(A)$ of feasible coloring functions satisfying all those constraints. A state q can be interpreted as a colored subtree of T with a particular root color; the unique initial state is an empty coloring and transitions into a colored leaf of T; a final state corresponds to a fully colored T with a particular root color. A commonly used form of cost functions considers the (possibly weighted) sum over the edges of the tree of the cost of putting two colors on each end of an edge, that is, an incremental cost function I of the form $I(q_1, \ldots, q_n, a, p) = \sum_{1 \leq i \leq n} p(a_i, a)$ where a_i is the color of the root of the subtree in state q_i and $p\colon C^2 \to \mathbb{R}_{\geq 0}$ is a function that gives the cost of putting two colors at each end of an edge.

Equivalence relations on the set of solutions

A possible strategy to define equivalence classes on the solution space of the Optimal tree coloring problem is to consider some colors to be locally equivalent on a node. In practical applications (see the next section), the space of colors can be quite large. Even though the precise colors of each node are necessary for correctly computing the cost function, when the solutions are analyzed by a human expert, it can be desirable to omit the colors and just look at whether the color of a node belongs to some group of colors. Therefore, this kind of equivalence relations is natural in many situations. Our Definition 4 of equivalence classes of an e-colored ad-AND/OR graph deals exactly with equivalence relations of this type.

Let e be a function that maps a color c to its "color group" $e(c)$. Two solutions of the Optimal tree coloring problem $\phi_1, \phi_2 \colon V(T) \to C$ are said to be *equivalent* if $\forall u \in V(T)$, $e(\phi_1(u)) = e(\phi_2(u))$. Let G be the ad-AND/OR graph associated with this instance. For each OR$^+$ node s of G labeled with the state q, where q is interpreted as a colored subtree of T with a particular root color c, color the node s with $e(c)$. Then G is e-colored and $\mathbb{C}(G)$ corresponds to the set of equivalence classes of the solutions of the instance. Notice that the constraint we had on the e-coloring of an ad-AND/OR graph is naturally satisfied by any meaningful function e because in a DP setting we only consider ordered trees: the i-th and j-th children of a node of T cannot be in the same color group unless $i = j$.

Concrete examples of tree coloring problems

Example 1. The first example is related to the alignment of gene sequences on a phylogenetic tree [31]. The input is a tree T, a set Σ of letters (DNA alphabet or protein alphabet), a function that labels each leaf node of T with a letter, and a distance function $d \colon \Sigma^2 \to \mathbb{R}_{\geq 0}$ between two letters. The goal is to extend the leaf labeling to a full labeling $\phi \colon V(T) \to \Sigma$ such that the sum of the distances over the edges of T is minimized. Defining equivalence relations of the solutions based on a grouping of the letters is uncontrived in this problem: for the DNA or protein alphabet, the letters can be subdivided into structurally similar groups.

Example 2. The Frequency assignment problems are a family of problems that naturally arise in telecommunication networks, and that have been extensively studied in graph theory as a generalization of graph coloring known as the T-coloring problem [17, 29, 32]. In the variant called the list T-coloring, the input is a graph G representing the interference between radio stations, a set C of colors, a function S that gives for each vertex $v \in V(G)$ a set $S(v) \subseteq C$ of colors (possible frequencies for a station), and a set $T \subseteq C^2$ of forbidden pairs of colors (interfering frequencies). The goal is to find a coloring $\phi \colon V(G) \to C$ such that a $\forall v \in V(G)$, $\phi(v) \in S(v)$, and $\forall (u, v) \in E(G)$, $(\phi(u), \phi(v)) \notin T$. While this problem is hard in general, it can be solved by DP when the underlying graph is a tree. In this case, we can enumerate colorings of the input trees without any forbidden pair of colors on the edges. Defining equivalence relations by grouping some of the colors together (similar frequencies) can be a practical way of reducing the size of the output.

Example 3. The Tree Reconciliation problem is the main method for analyzing the co-evolution of two sets of species, the hosts and their parasites [28]. The input are two phylogenetic trees H, P (of the hosts and of the parasites, respectively), together with a mapping $\phi_0 \colon \mathsf{Leaves}(P) \to \mathsf{Leaves}(H)$ that reflects the present-day parasite infections. What needs to be enumerated are then all past associations, that is, all mappings of the non-leaf nodes of the parasite tree to the nodes of the host tree that optimize a function which overall represents the sum of the number of different possible "events" weighted by the inverse of their estimated probability. The number of optimal solutions is often huge and, by applying our Algorithm 1, the enumeration of biologically inspired equivalence classes have allowed a significant reduction (in some cases from 10^{42} to only 96 classes) of the size of the output while still preserving the important biological information (see [35]).

3.2.2 Dynamic programming on tree decomposition of a graph

Many graph problems can be solved in polynomial time with a dynamic programming algorithm when the input graph has bounded treewidth (see for example [8]). The underlying idea is that, given a tree decomposition of a graph, the dynamic programming algorithm

traverses the nodes (bags) of the decomposition and consecutively solves the respective sub-problems. For vertex subset optimization problems, given a bag X, a dynamic programming algorithm generally computes for each $Z \subseteq X$ the optimal solution of the sub-problem whose intersection with X is Z. In this context, we could define two solutions to be equivalent if they intersect each bag of the decomposition in an "equivalent" way. The equivalence relation on the solutions is then defined by an equivalence relation over the subsets of each bag, and two solutions S_1 and S_2 are equivalent if for all bags X of the decomposition, $S_1 \cap X$ is equivalent to $S_2 \cap X$.

One of the simplest examples is to consider that all the nonempty subsets of vertices of a bag are equivalent. Thus, what we are interested in is whether a solution "hits" a bag (i.e., whether it has a nonempty intersection with the vertices in the bag). Consequently, two solutions would be considered equivalent if they hit the same bags.

We can also consider two subsets of a bag to be equivalent if they have the same size. In this case, two solutions would be equivalent if each bag contains the same number of vertices in the two solutions.

We believe that considering solutions in the way they are distributed along the tree decomposition of a graph could give a good overview of the diversity of the solution space.

4 Conclusion and perspectives

In this paper, we provide a general framework for the enumeration of equivalence classes of solutions in polynomial delay for a wide variety of contexts. This work opens a door to different research directions.

It would be interesting to ask whether we can efficiently enumerate groups of solutions that result from classical clustering procedures, or one representative per group. Moreover, in this paper we heavily rely on the decomposability property of the structure of the solution space. It remains open whether the problem of enumerating equivalence classes is hard without this restriction.

References

1 Steen A. Andersson, David Madigan, and Michael D. Perlman. A characterization of markov equivalence classes for acyclic digraphs. *Annals of Statistics*, 25(2):505–541, April 1997. doi:10.7916/D8280JSB.

2 Albert Angel and Nick Koudas. Efficient diversity-aware search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 781–792, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1989323.1989405.

3 Mukul S. Bansal, Eric J. Alm, and Manolis Kellis. Efficient algorithms for the reconciliation problem with gene duplication, horizontal transfer and loss. *Bioinformatics*, 28(12):i283–i291, 2012. doi:10.1093/bioinformatics/bts225.

4 Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, January 1962. doi:10.1145/321105.321111.

5 Richard Bellman. *Dynamic Programming*. Dover Books on Computer Science. Dover Publications, 2013.

6 Umberto Bertele and Francesco Brioschi. *Nonserial Dynamic Programming*. Academic Press, Inc., USA, 1972.

7 Anselm Blumer, Janet A. Blumer, David H. Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987. doi:10.1145/28869.28873.

8 Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In Timo Lepistö and Arto Salomaa, editors, *Automata, Languages and Programming*, pages 105–118, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. doi:10.1007/3-540-19488-6_110.

9 Pierre E. Bonzon. Necessary and sufficient conditions for dynamic programming of combinatorial type. *Journal of the ACM*, 17:675–682, 1970. doi:10.1145/321607.321616.

10 Marília D.V. Braga, Marie-France Sagot, Celine Scornavacca, and Eric Tannier. Exploring the solution space of sorting by reversals, with experiments and an application to evolution. *IEEE/ACM transactions on computational biology and bioinformatics*, 5 3:348–56, 2008. doi:10.1109/TCBB.2008.16.

11 Joshua Buresh-Oppenheim, Sashka Davis, and Russell Impagliazzo. A stronger model of dynamic programming algorithms. *Algorithmica*, 60:938–968, August 2011. doi:10.1007/s00453-009-9385-1.

12 Rina Dechter and Robert Mateescu. And/or search spaces for graphical models. *Artificial Intelligence*, 171(2):73–106, 2007. doi:10.1016/j.artint.2006.11.003.

13 Beatrice Donati, Christian Baudet, Blerina Sinaimeri, Pierluigi Crescenzi, and Marie-France Sagot. Eucalypt: efficient tree reconciliation enumerator. *Algorithms for Molecular Biology*, 10(1):3, 2015. doi:10.1186/s13015-014-0031-3.

14 Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Pictorial structures for object recognition. *International Journal of Computer Vision*, 61(1):55–79, January 2005. doi:10.1023/B:VISI.0000042934.15159.49.

15 Henning Fernau, Petr A. Golovach, and Marie-France Sagot. Algorithmic Enumeration: Output-sensitive, Input-Sensitive, Parameterized, Approximative (Dagstuhl Seminar 18421). *Dagstuhl Reports*, 8(10):63–86, 2019. doi:10.4230/DagRep.8.10.63.

16 Stefania Gnesi, Ugo Montanari, and Alberto Martelli. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM*, 28(4):737–751, 1981. doi:10.1145/322276.322285.

17 William K. Hale. Frequency assignment: Theory and applications. *Proceedings of the IEEE*, 68(12):1497–1514, 1980. doi:10.1109/PROC.1980.11899.

18 Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. doi:10.1137/0110015.

19 Paul Helman. A common schema for dynamic programming and branch and bound algorithms. *Journal of the ACM*, 36(1):97–128, January 1989. doi:10.1145/58562.59304.

20 Toshihide Ibaraki. Classes of discrete optimization problems and their decision problems. *Journal of Computer and System Sciences*, 8(1):84–116, 1974. doi:10.1016/S0022-0000(74)80024-3.

21 Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):693–718, 1967. doi:10.1137/0115060.

22 Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Basic Algorithmic Concepts*, pages 15–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-24777-7_2.

23 Alberto Martelli and Ugo Montanari. Optimizing decision trees through heuristically guided search. *Communications of the ACM*, 21(12):1025–1039, 1978. doi:10.1145/359657.359664.

24 Cristian Molinaro, Amy Sliva, and Vs S. Subrahmanian. Super-solutions: Succinctly representing solutions in abductive annotated probabilistic temporal logic. *ACM Transactions on Computational Logic*, 15(3), July 2014. doi:10.1145/2627354.

25 Katherine Morrison. An enumeration of the equivalence classes of self-dual matrix codes. *Advances in Mathematics of Communications*, 9:415, 2015. doi:10.3934/amc.2015.9.415.

26 Kazuyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient computation of substring equivalence classes with suffix arrays. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching*, pages 340–351, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/s00453-016-0178-z.

27 Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag Berlin Heidelberg, Tioga, Palo Alto, CA, 1982.

28 Roderic D. M. Page. *Tangled trees: phylogeny, cospeciation, and coevolution*. The University of Chicago Press, 2003.

29 Fred S. Roberts. T-colorings of graphs: recent results and open problems. *Discrete Mathematics*, 93(2):229–245, 1991. doi:10.1016/0012-365X(91)90258-4.

30 Lior Rokach and Oded Z. Maimon. *Data Mining with Decision Trees: Theory and Applications*. Series in machine perception and artificial intelligence. World Scientific, 2008. doi:10.1142/9097.

31 David Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, 1975. doi:10.1137/0128004.

32 Barry A. Tesman. List t-colorings of graphs. *Discrete Applied Mathematics*, 45(3):277–289, 1993. doi:10.1016/0166-218X(93)90015-G.

33 Olga Veksler. Stereo correspondence by dynamic programming on a tree. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 384–390, 2005. doi:10.1109/CVPR.2005.334.

34 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974. doi:10.1145/321796.321811.

35 Yishu Wang, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. Capybara: equivalence class enumeration of cophylogeny event-based reconciliations. *Bioinformatics*, 36(14):4197–4199, 2020. doi:10.1093/bioinformatics/btaa498.

36 Yishu Wang, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. A general framework for enumerating equivalence classes of solutions, 2021. arXiv:2004.12143.