Chapter 2
Enumeration Algorithms

2.1 Introduction

The aim of enumeration is listing all the feasible solutions of a given problem. For instance, given a graph $G = (V, E)$, enumerating all the paths or the shortest paths from a vertex $s \in V$ to a vertex $t \in V$, enumerating cycles, or enumerating all the feasible solutions of a knapsack problem, are classical examples of *enumeration problems*. An enumeration algorithm solves an enumeration problem.

While an optimization problem aims to find just the best solution according to an objective function, i.e. an extreme case, an enumeration problem aims to find all the solutions satisfying some constraints, i.e. local extreme cases. This is particularly useful whenever the objective function is not clear: in these cases, the best solution should be chosen among the results of the enumeration.

Moreover, sometimes it can be interesting to capture local structures of the data, instead of the global one, so that enumerating all remarkable local structures becomes particularly helpful.

In such a context, a good model is the result of a tradeoff between the size and the number of the solutions: whenever the sizes of the solutions are huge, it is more desirable to have relatively few solutions. For these reasons, the models usually include some parameters (such as solution size, frequency, and weight) or unify similar solutions.

It is worth observing that the number of solutions increases with the size of the input. Whenever this size is small, brute force algorithms are helpful, and simple implementations can successfully solve the problem. On the other hand, for large-scale data more sophisticated approaches from algorithm theory are required in order to guarantee a bounded increase of computation time when the input size increases.

In this chapter, we will present an overview of the main computational issues related to enumeration problems and the main techniques to design algorithms and to prove their complexity. These are part of the lecture notes, written together with Gustavo A.T. Sacomoto, during the lectures given by Takeaki Uno at the school on Enumeration Algorithms and Exact Methods (ENUMEX) in Bertinoro, Italy, on September 25–26th, 2012.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_2

Algorithm 1: BruteForce(i, X)

```
Input: An integer $i \geq 1$, a sequence of values $X = \langle x_0, \ldots, x_{i-1} \rangle$, eventually empty
Output: All the feasible sequences of length $n$ whose prefix is $X$
1 if no solution includes $X$ then return;
2 if $i > n$ then
3     if $X$ is a solution then output $X$;
4 else
5     foreach feasible value $e$ of $x_i$ do
6         BruteForce($i + 1$, $\langle X, e \rangle$)
7     end
8 end
```

Structure of the Chapter
The chapter is structured as follows: in Sect. 2.2 we exploit the main algorithmic issues related to enumeration and we show some brute force approaches to solve them. In Sect. 2.3 we report the main technical framework to design efficient enumeration algorithms and in Sect. 2.4 we show the main amortization schema. In Sect. 2.5, we briefly discuss the tractability of enumeration problems in practice.

2.2 Algorithmic Issues and Brute Force Approaches

The design of enumeration algorithms involves several aspects that need to be taken into account in order to achieve correctness and effectiveness. Indeed, any enumeration algorithm has to guarantee that each solution is output exactly once, i.e. should avoid duplication. A straightforward way to achieve this is to store in memory all solutions already found, and whenever a new solution is encountered, test whether it has been already output or not. Clearly, this approach can be memory inefficient when the solutions are large with respect to the memory size, or there are too many of them. Dealing with this would require dynamic memory allocation mechanism and efficient search (*hash*). For these reasons, deciding whether a solution has been already output without storing the solutions already generated is a more suitable strategy that many enumeration algorithms try to apply.

Besides that, there are cases in which implicit forms of duplication should also be avoided, i.e. avoid outputting isomorphic solutions. To this aim, it is often useful to define a canonical form of encoding for the solutions allowing easy comparisons. The canonical form should provide a one-to-one mapping between the objects and their representation, without increasing drastically their size. In this way the problem of enumerating certain objects is turned into the enumeration of their canonical forms. However, in some cases, like graphs, sequence data and matrices, checking isomorphism is hard even by defining a canonical form. Nonetheless, in these cases the isomorphism can be still checked by using exponential algorithms that in practice turn out to be often efficient when the number of solutions is small.

Algorithm 2: BRUTEFORCE(X, D)

```
Input: A pattern X, a reference to a global database D
Output: All the patterns containing $X$ not isomorphic between them and to any pattern
        contained in $D$
1 $D \leftarrow D \cup \{X\}$
2 if no solution includes $X$ then return;
3 if $X$ is a solution then output $X$;
4 foreach $X'$ obtained by adding an element to $X$ do
5     if $\nexists Z \in D$ such that $Z$ isomorphic to $X'$ then
6         BRUTEFORCE($X'$, $D$)
7     end
8 end
```

Simple structures, such as cliques and paths are generally easy to enumerate, since cliques can be obtained by iteratively adding vertices, and the set of paths can be easily partitioned. More complex structures, such as maximal (nothing can be added to the solution without losing some required property) or minimal (nothing can be subtracted from the solution without losing some required property) structures, or constrained structures, are more difficult to enumerate. In these cases, even if a solution can be found in polynomial time, the main issue is designing a way to generate other solutions from a given one, i.e. *defining a solution neighbourhood*, in order to allow visiting all the solutions by moving iteratively through the neighbourhoods.

It should be noted that using an exponential time approach to find each neighbour or having an exponential number of neighbouring solutions, can lead to time inefficiency. When an exponential number of possible choices have to be applied to a solution in order to possibly obtain other solutions, the enumeration process can take an exponential time for each solution, since there is no guarantee that any choice leads to a solution. For example this is very often the case concerning maximal solutions: removing some elements and adding others to get maximality allows to move iteratively to any solution, but, when the number of these combinations is exponential, the final cost per solution is also exponential. In such a context, if possible, restricting the number of neighbours of a solution or applying some pruning strategy to avoid redundant computation, can lead to more efficiency.

More complex cases concern the problems in which even finding a solution is NP-complete, such as SAT or Hamiltonian cycle. Nonetheless, in these cases, heuristics often effectively apply, specially when the problem turn out to be *usually easy*, like SAT, the *solutions are not huge*, like maximal and minimal structure enumeration, and the *size of the solution space is bounded*.

When the instance sizes are small, another approach to these problems, is to use brute force algorithms. For example, using a divide and conquer approach to enumerate all the candidates and selecting all feasible solutions, or by enlarging the solutions one by one and removing the isomorphic ones. Two basic schemas for brute force algorithms are informally described in Algorithms 1 and 2. In Algorithm 1 every solution is seen as an ordered sequence of values: by invoking BRUTEFORCE(1,∅), the feasible values are recursively found by enlarging the current solution; in this case, just the test whether X is a solution or not is required. Also Algorithm 2 tries to enlarge the current solution, but at each step we check whether the current solution has been already considered in the past computation: the result of the past computation is stored in a database D.

Note that for both the algorithms, it is necessary to know how to transform a candidate X into another candidate X'. Moreover, it is worth observing that, in both cases, an accurate a priori checking whether X is contained in any solution or not could save a lot of useless computation.

2.3 Basic Algorithms

Since the number of solutions of many enumeration problems are usually exponential in the size of the instance, enumeration algorithms require often at least exponential time. On the other hand, it is quite natural to ask for a polynomial time algorithm whenever the number of solutions is polynomial. In such a context, the complexity classes of enumeration problems are defined depending on the number of solutions, so that if the number of solution is small, an efficient algorithm has to terminate after short (polynomial) time, otherwise it is allowed to spend more time. According to this idea, the following complexity classes have been defined [1].

Definition 2.1 An enumeration algorithm is *polynomial total time* if the time required to output all the solutions is bounded by a polynomial in the size of the input and the number of solutions.

Definition 2.2 An enumeration algorithm is *polynomial delay* if it generates the solutions, one after the other in some order, in such a way that the delay until the first is output, and thereafter the delay between any two consecutive solutions, is bounded by a polynomial in the input size.

Intuitively, the polynomial total time definition means that the delay between any two consecutive solutions has to be polynomial on the average, while the polynomial delay definition implies that the maximum delay has to be polynomial. Hence, Definition 2.2 implies Definition 2.1.

For a comprehensive catalogue of known enumeration algorithms and their classification we invite the reader to see [24].

The basic technique for designing enumeration algorithms are: backtracking (depth-first search with lexicographic ordering), binary partition (branch and bound like recursive partition algorithm), reverse search (search on traversal tree defined by parent-child relation). The rest of this section is devoted to exploit the features of these schemas. It is worth observing that this categorization is not strict, since very often these technique overlap each other.

2.3.1 Backtracking

A set $F \subseteq 2^U$ (of subsets of U) satisfies the *downward closure* if for any $X \in F$ and for any $X' \subseteq X$, we have $X' \in F$, in other words, for any X belonging to F we have that any subset of X also belongs to F. Given a set U and an oracle to decide whether $X \subset U$ belongs to F, an unknown set of 2^U satisfying the downward closure, we consider the problem of enumerating all (maximal) elements of F. The backtracking technique is mainly applied to these problems. In this approach by starting from an empty set, the elements are recursively added to a solution. The elements are usually indexed, so that in each iteration, in order to avoid duplication, only an element whose index is greater than the current maximum element is added. After all the examinations concerning one element, by backtracking, all the other possibilities are exploited. The basic schema of backtracking algorithms is shown by Algorithm 3. Note that whenever it is possible to apply this schema, we obtain a polynomial delay algorithm, whose space complexity is also polynomial. The technique proposed relies on a depth-first search approach. However, it is worth observing that in some cases of enumeration of families of subsets exhibiting the downward closure property, arising in the mining of frequent patterns (e.g., mining of frequent itemsets), besides the depth-first backtracking, a breadth-first approach can be also successfully used. For instance this is the case of the Apriori algorithm for discovering frequent itemsets [25].

Algorithm 3: BACKTRACK(S)

```
  Input: $S \subseteq U$ a set (eventually empty)
  Output: All the solutions containing $S$
1 output $S$
2 Let $\pi(x)$ be the index associated to an element $x \in U$
3 foreach $e > \max_{x \in S} \pi(x)$ do
4     if $S \cup \{e\}$ is a solution then
5         BACKTRACK($S \cup \{e\}$)
6     end
7 end
```

Algorithm 4: SUBSETSUM(S)

```
  Input: $S$ a set (eventually empty) of integers belonging to the collection $U = \{a_1, \ldots, a_n\}$
  Output: All the subsets of $U$ containing $S$ whose sum is less than $b$.
1 ouput $S$
2 Let $\pi(x)$ be the index associated to an element $x$
3 foreach $i > \max_{x \in S} \pi(x)$ do
4     if $a_i + \sum_{x \in S} x < b$ then
5         SUBSETSUM($S \cup \{a_i\}$)
6     end
7 end
```

2.3.1.1 Enumerating All the Subsets of a Collection $U = \{a_1, \ldots, a_n\}$ Whose Sum is Less Than b

By using the backtracking schema, it is possible to solve the problem as shown by Algorithm 4. Each iteration outputs a solution, and take $O(n)$ time, so that we have $O(n)$ time per solution. It is worth observing that if we sort the elements of U, then each recursive call can generate a solution in $O(1)$ time, so that we have $O(1)$ time per solution.

2.3.2 Binary Partition

Let X be a subset of F, the set of solutions, such that all elements of X satisfy a property P. The binary partition method outputs X only if the set is a singleton, otherwise, it partitions X into two sets X_1 and X_2, whose solutions are characterized by the disjoint properties P_1 and P_2 respectively. This procedure is repeated until the current set of solutions is a singleton. The bipartition schema can be successfully applied to the problem of enumeration of paths of a graph connecting two vertices s and t, of the perfect matchings of a bipartite graph [26], of the spanning trees of a graph [27]. If every partition is non-empty, i.e. all the internal nodes of the recursion tree are binary, we have that the number of internal nodes is bounded by the number of leaves. In addition, if we have that the partition oracle takes polynomial time, since every leaf outputs a solution, we have that the resulting algorithm is polynomial total time. On the other hand, even if there are empty partitions, i.e. internal unary nodes in the recursion tree, if the height of tree is bounded by a polynomial in the size of the input and the partition oracle takes polynomial time, then the resulting algorithm is polynomial delay.

2.3.2.1 Enumerating All the (s, t)-Paths in a Graph $G = (V, E)$

The partition schema chooses an arc $e = (s, r)$ incident to s, and partitions the set of all the (s, t)-paths into the ones including e and the ones not including e. The (s, t)-paths including e are obtained by removing all the arcs incident to s, and enumerating the (r, t)-paths in this new graph, denoted by $G - s$. The (s, t)-paths not including e are obtained by removing e and enumerating the (s, t)-paths in the new graph, denoted by $G - e$. The corresponding pseudocode is shown by Algorithm 5. It is worth observing that if the arc e is badly chosen, a subproblem could not generate any solution; in particular, the set of the (r, t)-paths in the graph $G - s$ is empty if t is not reachable from r, while the set of the (s, t)-paths in $G - e$ is empty if t is not reachable from s. Thus before performing the recursive call to the subproblems it could be useful to test the validity of e, by testing the reachability of t in these modified graphs. Notice that the height of the recursion tree is bounded by $O(|V| + |E|)$, since at every level the size of the graph is reduced by one vertex or arc. The cost per iteration is $O(|V| + |E|)$, the reachability test. Therefore, the algorithm has $O((|V| + |E|)^2)$ delay or $O(|E|^2)$ delay for connected graphs.

This problem has been studied in [9, 28, 29], and in [10], guaranteeing a linear delay. In Chap. 5 we will modify this latter algorithm in order to enumerate *bubbles*. In the particular case of undirected graphs, in Chap. 6 we will show an algorithm based on this bipartition approach having an output sensitive amortized complexity, as shown in [16]. In the particular case of shortest paths, the enumeration problem has been studied in [30]. It is worth observing that the problem of enumerating all the (s, t)-paths in a graph is equivalent to the problem of enumerating all the cycles passing through a vertex.

Algorithm 5: PATHS(G, s, t, S)

```
   Input: A graph $G$, the vertices $s$ and $t$, a sequence of vertices $S$ (eventually empty)
   Output: All the paths from $s$ to $t$ in $G$
1  if $s = t$ then
2      output S
3      return
4  end
5  choose an arc $e = (s, r)$
6  if no $(r, t)$-path in $G - s$ then
7      PATHS($G - e, s, t, S$)
8      return
9  end
10 if no $(s, t)$-path in $G - e$ then
11     PATHS($G - e, r, t, \langle S, s \rangle$)
12     return
13 end
14 PATHS($G - s, r, t, S$)
15 PATHS($G - e, s, t, S$)
```

2.3.3 Reverse Search

The reverse search schema defines for any solution a solution called *parent solution* [31], in a way that this parent-children relationship does not induce a cyclic graph or DAG, but induces a tree. In this way, in order to enumerate all the solutions, it is sufficient to traverse the tree by performing a depth first search, so that the number of iterations is equal to the number of solutions. It is worth observing that the tree induced by the parent child relationship does not need to be stored in memory, but it is sufficient to use an algorithm for finding all the children of a parent. Moreover it could be preferable to have an algorithm able to find the $(i + 1)$th child of a node, given the ith child.

Since the number of iterations is equal to the number of solutions, we have that the cost per solution is equal to the cost per iteration. Thus if finding the next child of a node costs $O(f(n))$ time, where n is the input size, the resulting computation time per iteration is $O(f(n))$. Hence the algorithm is polynomial total time whenever $f(n)$ is polynomial. The space complexity is given by the memory usage of an iteration and by the height of the depth first search tree. This latter cost is not required when we have an algorithm able to find the $(i+1)$th child of a node, given its ith child. The delay between two successive solutions is also $O(f(n))$ by using the alternative output technique [32].

Indeed alternative output technique aims to reduce the delay, by avoiding that the depth first search backtrack along long paths without outputting any solution. As shown by Algorithm 7 the solutions are outputted before the recursive calls when the current depth first search level is even, otherwise, i.e. in the odd levels, the solutions are output after the recursive calls. In this way for any two successive solutions we have a delay at most $2f(n)$, where $f(n)$ is the cost of an iteration. Indeed suppose that the parent child relationship induces a path of solutions $x_1, \ldots, x_{g(n)}$ and there is a solution $x_{g(n)+1}$ that is a child of x_1, where $g(n)$ is a function of n. If the cost per iteration is $O(f(n))$, by applying Algorithm 6, for any i with $1 \leq i \leq g(n)$, the delay is $O(f(n))$, and the delay between x_k and x_{k+1} is $O(g(n))$. By applying Algorithm 7, by supposing $g(n)$ odd, the solutions are generated in the following order $x_2, x_4, \ldots x_{g(n)-1}, x_{g(n)}, x_{g(n)-2}, x_{g(n)-4}, \ldots, x_3, x_1, x_{g(n)+1}$, so that the delay is $O(2 \cdot f(n)) = O(f(n))$.

In conclusion, by applying this technique, every time an enumeration algorithm takes $O(f(n))$ time in each iteration and also outputs a solution on each iteration, the delay $O(f(n))$ can be turned into a worst case delay $O(f(n))$.

Algorithm 6: REVERSESEARCH(S)

```
1 output S
2 foreach child S' of S do
3     ReverseSearch(S')
4 end
```

Algorithm 7: ALTERNATIVEOUTPUT(S, *depth*)

```
  Input: A solution S, an integer depth
  Output: All the solutions descendants of S in the tree induced by the parent-child
          relationship
1 if depth is even then output S;
2 foreach child S' of S do
3     AlternativeOutput(S, depth + 1)
4 end
5 if depth is odd then output S;
```

2.3.3.1 Maximal Clique Enumeration

A clique is a complete graph, i.e. a graph in which any two vertices are connected. Finding the clique of maximum size in a graph $G = (V, E)$ is NP-hard [33], while finding a maximal clique is an easy task that can be solved in $O(|E|)$ time: by starting with an arbitrary clique (for instance, a single vertex), grow the current clique one vertex at a time, adding it if connected to each vertex in the current clique, and discarding it otherwise. The clique enumeration problem is the problem of enumerating all the complete subgraph of a given graph in input. This problem has been widely studied by [34–36]. The bipartite clique enumeration problem is the problem of enumerating all the complete bipartite subgraphs of a bipartite graph and it can be efficiently reduced to a clique enumeration problem [34].

It is worth observing that the set of cliques is *monotone*, since any subset of the vertices of a clique is also a clique. This means that the backtracking technique can be successfully applied. Checking whether a recursive call is going to produce at least a clique costs $O(|E|)$ time, and has to be repeated for at most $|V|$ recursive calls, so that the final cost is $O(|V||E|)$ per clique.

When the number of solutions increase exponentially when the size of the instance input increases linearly, it seems hard post-processing the solutions found, so that often the simple enumeration problem is turned in enumeration of maximal structures. In this way, the solution set becomes not redundant. More formally, a solution X is maximal if for any $X \subset X'$, X' is not a solution. In general the problem of finding maximal solutions is more difficult, since it is often harder to find a *neighbourhood* relationship between them. However there are some exceptions, like enumerating maximal clique.

Also in real contexts it seems more promising enumerating all the maximal cliques instead of all the cliques: it has been estimated that in real world graphs, even if they are sparse and the size of their cliques is small, the number of maximal cliques is between 0.1 and 0.001 % the number of its cliques (see also [37]). Moreover, restricting the enumeration to maximal cliques does not lead to lose any information since any clique is included in at least one maximal clique.

Given a graph $G = (V, E)$, whose vertices are indexed, a set of vertices $X \subseteq V$ is said to be *lexicographically* greater than $Y \subseteq V$ if the vertex whose index is minimum in $(X \setminus Y) \cup (Y \setminus X)$ is contained in X. Moreover, for any $X, Y \subseteq V$, the trichotomy property holds, i.e. exactly one of the following holds: $X < Y$, $X = Y$, or $Y > X$. For any vertex set S, we define $S_{\leq i}$ as $S \cap \{v_1, \ldots, v_i\}$.

Let $C(K)$ be the lexicographically smallest maximal clique including a clique $K \subseteq V$, $C(K)$ can be computed by greedily adding vertices to K in lexicographic order of the indices. Observe that for any set K, $C(K)$ is not lexicographically smaller than K.

Given a maximal clique K we define the parent of K, $P(K)$, as $C(K_{\leq i-1})$, such that i is the maximum index satisfying $C(K_{\leq i-1}) \neq K$. Notice that $C(K_{\leq i-1})$ can be efficiently computed by removing the vertices from K by starting from the ones whose index is greater and computing C on the remaining vertices while $C(K) = K$ holds. The lexicographically smallest clique, denoted as K_0, has no parent. Since for

Algorithm 8: EnumMaximalCliques(G, K)

```
    Input: A graph $G = (V, E)$, a maximal clique $K \subseteq V$
    Output: All the maximal cliques descendants of $K$ in the tree induced by the parent-child
            relationship between maximal cliques
1   output K
2   foreach vertex $v \in V$ not in $K$ do
3       $K' \leftarrow K[v]$
4       if $P(K') = K$ then
5           EnumMaximalCliques($G$, $K$)
6       end
7   end
```

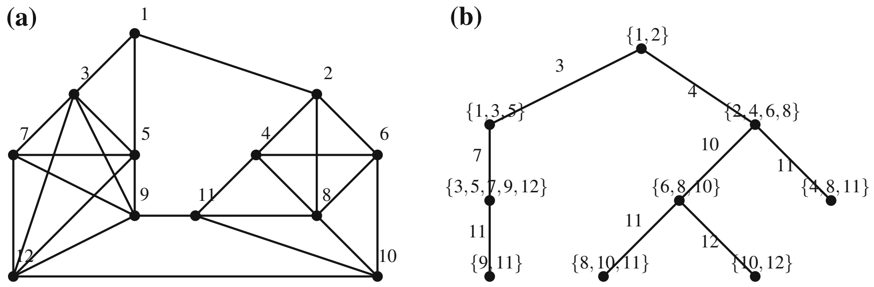

Fig. 2.1 A graph and the recursion tree induced by Algorithm 8

any K, $P(K)$ is lexicographically greater than K, and $P(K)$ is uniquely defined, the parent-child relationship induces an acyclic graph, that is a tree (Fig. 2.1).

For any maximal clique K and any vertex v_i, we define $K[v_i]$ as $C((K_{\leq i} \cap N(v_i)) \cup \{v_i\})$, where $N(v_i)$ is the neighbourhood of v_i. Thus a maximal clique K' is a child of the maximal clique K, if there exists v_i, with $v_i \notin K$, such that $K' = K[v_i]$. Hence in order to compute the children of a maximal clique K, it is sufficient to check for any v_i whether $P(K[v_i])$ is equal to K.

Observe that for any maximal clique K, $C(K)$ and $P(K)$ can be computed in $O(|E|)$ time. All children of K can be found by at most $|V|$ tests, so that the cost of each iteration is bounded by $O(|V||E|)$ time. Thus, since the number of iterations is equal to the number of solutions, the final cost is $O(|V||E|)$ per maximal clique.

2.3.3.2 Non-Isomorphic Ordered Tree Enumeration

Several enumeration problems aim to enumerate all the *substructures* of a given instance, like paths of a graph. However, applications sometimes require solutions satisfying certain constrains, like enumerating path or cycles of a fixed length or enumerating the cliques of a given size. Other problems instead aim to find all the *structures* of a given class, like enumerating the permutations of size n, enumerating trees, crossing lines in a plane, matroids, and binary matrices. Enumerating non trivial structures often implies enumerating non isomorphic structures. In general two structures are isomorphic whenever it is defined a one-to-one correspondence between their elements. For instance a circular sequence is isomorphic to another if and only if it can be transformed in it by using a rotation, a matrix is isomorphic to another matrix if and only if each one can be transformed in the other one by swapping rows and columns, a graph is isomorphic to another graph if and only if their adjacency matrices are isomorphic, i.e. there is a one to one mapping between their vertices that preserves the adjacency.

Let us consider the problem of enumerating ordered trees, trees in which the ordering of the children of each vertex is specified. The isomorphism between two ordered trees is inductively defined as follows: two leaves are isomorphic; two trees rooted on x and y, whose order lists of children are $\langle x_1, \ldots, x_p \rangle$ and $\langle y_1, \ldots, y_q \rangle$ respectively, are isomorphic if $p = q$ and for any i, with $1 \leq i \leq p = q$, the subtree rooted on x_i is isomorphic to the subtree rooted on y_i. This problem has been studied in [38], and by fixing the number of leaves in [39].

Given an ordered tree, we define the indexing of its vertices as the visiting order of a left-first DFS, i.e. a depth first search that visits the children of a vertex following their order. This indexing procedure is unique and isomorphism between two ordered trees, whose vertices are indexed as described, can be checked comparing the edge sets: the two indexed trees are isomorphic if and only if they have the same edge set.

Moreover, the left-first DFS can be used to encode the ordered trees. To this aim, we define the depth sequence as $\langle h_1, \ldots, h_n \rangle$, where h_i is the depth of vertex v_i in the left-first DFS tree, where v_i is the ith vertex visited by a left-first DFS. There is a one-to-one correspondence between the ordered trees and the depth sequences, so that isomorphism can be checked by comparing the depth sequences, as shown by Fig. 2.2.

By following the reverse search schema, we define the parent-child relationship between non-isomorphic trees. In particular the parent of an ordered tree is defined by the tree, obtained by removing the vertex having the largest index, i.e. by removing from a depth sequence its last element (the last element visited by a left-first DFS). Recall that the indexing induced by the left-first DFS is such that the largest index is the leaf of the rightmost branch of the tree. Observe that the size of the parent is

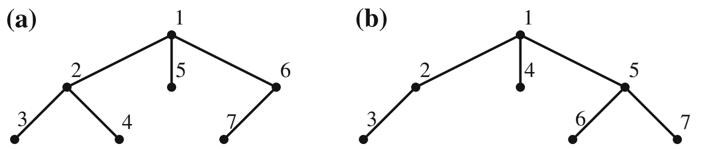

Fig. 2.2 Two non isomorphic ordered trees, labelled by using left-first DFS, and their depth sequences. **a** $\langle 0, 1, 2, 2, 1, 1, 2 \rangle$. **b** $\langle 0, 1, 2, 1, 1, 2, 2 \rangle$

smaller than the size of the children, any ordered tree have exact one parent, except the empty tree, so that the relationship induces an acyclic graph.

For any ordered tree T, whose depth-sequence is $\langle h_1, \dots, h_n \rangle$, the children of T according to the parent-child relationship defined before, are all the ordered trees obtained by adding a new vertex v_{n+1} as the rightmost child of a vertex belonging to the rightmost path. Let h_{n+1} be the depth of the new vertex v_{n+1}. Since h_n is the rightmost leaf of T, we have that it belongs to the rightmost path, to be precise, v_n is the last vertex of this path. Thus, the depths of the vertices of the rightmost path of T, from the root to v_n, are exactly the interval $[0, h_n]$. Since the new vertex v_{n+1} is a child of a vertex in this path, the depth h_{n+1} is in the interval $[1, h_n + 1]$. Thus the children of an ordered tree T, with depth-sequence $\langle h_1, \dots, h_n \rangle$, are all the ordered trees whose depth sequence is $\langle h_1, \dots, h_n, h_{n+1} \rangle$, with $1 \le h_{n+1} \le h_n + 1$. An example is given in Fig. 2.3.

By using these observations, we can enumerate all the ordered trees of size less than k, as shown by Algorithm 9. Notice that the inner loop takes constant time, so that the time complexity is $O(1)$ per solution.

```
Algorithm 9: EnumOrderedTree($T$, $k$)
  Input: A tree $T$ (eventually empty) and an integer $k$
  Output: All the non-isomorphic trees of size at most $k$, whose depth sequence contains as
          prefix the depth sequence of $T$
1 output T
2 if size of $T = k$ then return;
3 foreach vertex $v$ in the right most path do
4     Let $T'$ be the tree obtained from $T$ by adding a rightmost child to $v$
5     EnumOrderedTree($T'$, $k$)
6 end
```

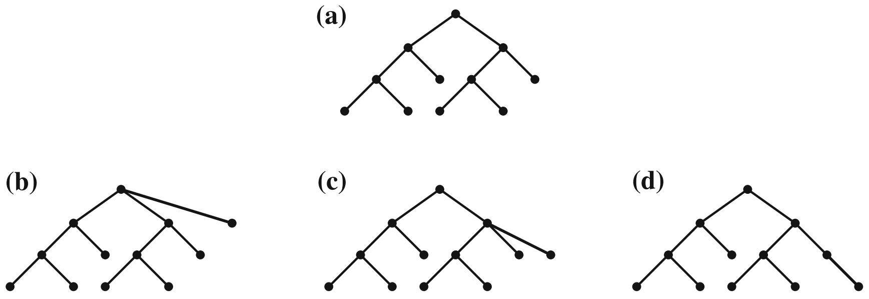

Fig. 2.3 An ordered tree, its depth sequence (**a**), and its children with their depth sequences (**b**), (**c**) and (**d**). **a** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**⟩. **b** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *1*⟩. **c** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *2*⟩. **d** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *3*⟩

2.3.3.3 Non-Isomorphic Tree Enumeration

We now consider the problem of enumerating non-ordered trees, i.e. trees in which the ordering of the children of each vertex is not specified. The isomorphism between two (non-ordered) trees is inductively defined as follows: two leaves are isomorphic; two trees rooted on x and y, whose children lists are X and Y respectively, are isomorphic if $|X| = |Y| = p$ and there exist two permutations of X and Y, $\langle x_1, \ldots, x_p \rangle$ and $\langle y_1, \ldots, y_p \rangle$ respectively, such that for any i, with $1 \leq i \leq p$, the subtree rooted on x_i is isomorphic to the subtree rooted on y_i. This problem has been studied in [40], by fixing the diameter in [41], and in the more general case of coloured rooted trees in [42].

The näive approach, to use the same algorithm for ordered tree enumeration to enumerate non-ordered trees, would produce many duplicate solutions, since each non-ordered tree may correspond to an exponential number of ordered trees. Which in turn, would be very inefficient.

In order to define the canonical form of representation of a rooted tree, we use its *left-heavy* embedding, defined as the lexicographically maximum depth sequence among all the ordered trees corresponding to T (Fig. 2.4). Therefore, two non-ordered rooted trees are isomorphic if and only if they have the same left-heavy embedding.

The parent child relationship between canonical forms is defined as follows: the parent of a left-heavy embedding is obtained by the removal of the rightmost leaf of the corresponding tree, the same for ordered trees. Observe that the parent t' of a left-heavy embedding t of T is a left-heavy embedding too, otherwise there would be another sequence greater than t' such that by adding back the rightmost leaf of T we would obtain a depth sequence for T that is lexicographically greater than t (Fig. 2.5).

Hence any child of a rooted tree T is obtained by adding a vertex as children of the vertices belonging of the rightmost path, like for ordered trees. However, some trees obtained by adding a vertex in this way are not children of T, since the resulting sequence does not coincide with their left-heavy embedding. This can happen if there exists a vertex x in the rightmost path of T, such that the depth sequence $t = \langle s_1, \ldots, s_p \rangle$ of $T(r)$, where r is the rightmost child of x, is a prefix of the depth sequence $t' = \langle s_1, \ldots, s_p, \ldots s_q \rangle$ of $T(r')$, where r' is the second rightmost child of x, so that the depth sequence of T ends with t concatenated with t'. Indeed, in this case, by adding a vertex at depth y to $T(r)$ and obtaining $t'' = \langle s_1, \ldots, s_p, y \rangle$

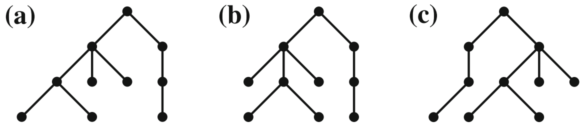

Fig. 2.4 Three isomorphic rooted tree and their depth sequences. The first one is the left heavy embedding. **a** $\langle 0, 1, 2, 3, 3, 2, 2, 1, 2, 3 \rangle$. **b** $\langle 0, 1, 2, 2, 3, 3, 2, 1, 2, 3 \rangle$. **c** $\langle 0, 1, 2, 3, 1, 2, 3, 3, 2, 2 \rangle$

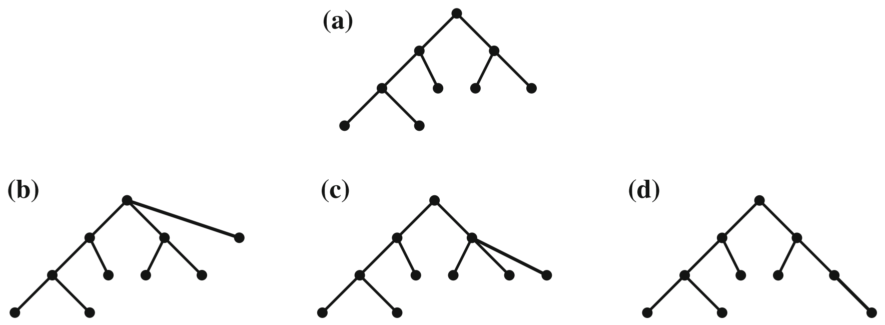

Fig. 2.5 An rooted tree, and its depth sequence (**a**). **b** and **c** are its children, while **d** is not a child of (**a**). **a** $\langle 0, 1, 2, 3, 3, 2, 1, 2, 3, 2\rangle$. **b** $\langle 0, 1, 2, 3, 3, 2, 1, 2, 2, \mathit{1}\rangle$. **c** $\langle 0, 1, 2, 3, 3, 2, 1, 2, 2, \mathit{2}\rangle$. **d** $\langle 0, 1, 2, 3, 3, 2, 1, 2, 2, \mathit{3}\rangle$

as depth sequence of $T(r)$, the depth sequence of T ends with t concatenated with t'': if s_{p+1} is lexicographically smaller than y, this is not a leaf-heavy embedding, since the depth sequence ending with t'' concatenated with t is lexicographically greater. Thus, since s_{p+1} is the depth of the rightmost leaf of $T(r')$, to get all and just the children of T, we have to consider all the possible ways to add a vertex as children of a vertex belonging to the rightmost path, so that its depth is smaller or equal to s_{p+1}.

The *copy vertex* is thus defined as the highest (lowest depth) vertex x in T with at least two[1] children, r and r' (the rightmost and the second rightmost child respectively), such that the depth sequence $\langle s_1, \ldots, s_p\rangle$ of $T(r)$ is a prefix of the depth sequence $\langle s_1, \ldots, s_p, \ldots s_q\rangle$ of $T(r')$. Given a tree T with copy vertex x, in order to generate the children of T, we have to consider two cases: the prefix of the depth sequences is proper or the depth sequences are equal. In the first case, there exists s_{p+1} and by attaching a new rightmost child to a vertex v, with depth $\leq s_{p+1}$, in the rightmost path of T we obtain a new tree T' that is also a left-heavy embedding. Moreover, the new copy vertex of T' is v, if the depth v is not equal to the depth of x; or x, otherwise. On the other case, the subtrees $T(r)$ and $T(r')$ are equal and by attaching a new rightmost child to a vertex v, with depth smaller or equal to the depth of x, in the rightmost path of T we obtain a new tree T' that is also a left-heavy embedding, and the new copy vertex of T' is v. In both cases, we are able to generate the new tree T' and update the copy vertex in constant time. The algorithm is shown by Algorithm 10. Each iteration of the loop costs $O(1)$, so that we have a final cost of $O(1)$ per solution.

[1] If T is a path, the copy vertex is defined as the root.

Algorithm 10: EnumRootedTree(T, x)

```
Input: A tree $T$ (eventually empty), an integer $k$, and a vertex $x$
Output: All the non-isomorphic rooted trees of size at most $k$, whose depth sequence
        contains as prefix the depth sequence of $T$
1  output T
2  if size of $T = k$ then return;
3  $r \leftarrow$ the rightmost child of $x$
4  $r' \leftarrow$ the second rightmost child of $x$
5  if depth sequence of $(T(r') \neq$ depth sequence of $T(r)$ then
6  |  $y \leftarrow$ the vertex of $T(r')$ after the prefix $T(r)$
7  else
8  |  $y \leftarrow x$
9  end
10 foreach vertex v in the rightmost path of T, in increasing depth order do
11    add a rightmost child to $v$
12    if depth of $v$ = depth of $y$ then
13       EnumRootedTree($T$, $x$)
14       break
15    end
16    EnumRootedTree($T$, $v$)
17    remove the rightmost child of $v$
18 end
```

2.4 Amortized Analysis

In this section, we explore techniques to analyse the running time of a certain kind of enumeration algorithms. Specifically, enumeration algorithms with a tree-shaped recursion structure.

Suppose a enumeration algorithm with a tree-shaped recursion structure takes $O(n)$ time per node. Based only on this, it is not possible to polynomially bound the time spent to output each solution. We can have exponentially many nodes and a small number of solutions as in, for example, the enumeration of feasible solutions of SAT using a branch-and-bound algorithm. However, if every node outputs a solution, then algorithm takes $O(n)$ per solution. Now, suppose that each leaf outputs a solution and each node takes $O(n)$ time. Again, this is not enough to polynomially bound time per solution, since we can have an exponential number of internal nodes and only few leaves. In addition, we need that either the height of the tree is bounded, in this case the number of nodes is bounded by the number of solutions (leaves) times the height; or each internal node has at least two children, the number of nodes is bounded by two times the number of solutions.

These three scenarios: every node outputs a solution, every leaf outputs a solution and the height of the tree is bounded, and every leaf outputs a solution and each internal nodes has at least two children, are the typical ones in which we can polynomially bound the time complexity. In each case, the time complexity per solution depends on the maximum time complexity $O(n)$ over all nodes. In order to do better, we have to use amortized analysis. The rest of the section is devoted to three amortized analysis techniques: basic amortization, amortization by children and push out amortization.

2.4.1 Basic Amortization

A recursive enumeration algorithm usually solves the problem by breaking it into subproblems, which are generally smaller, in both input and output size, than the original problem. The recursion tree for this case has many bottom level nodes taking a short time and a fewer nodes closer to the root taking a long time. We call this effect in the recursion tree *bottom-wideness*. However, this observation alone is not enough to provide good amortized bounds. For instance, Fig. 2.6a, b have bottom level nodes (leaves) taking $O(1)$, but the amortized complexity is still $O(n)$, the maximum cost among the nodes. In both cases, there were a sudden decrease in the computation times.

In the tree of Fig. 2.6c each internal node has two children, all the nodes in the same level have the same cost, and the cost of each node decreases by a constant at each level. It is not hard to show that the average complexity per node in this case is $O(1)$. That is, there was a reduction from $O(n)$ to $O(1)$ when considering the amortized complexity. Lemma 2.2 presents a generalization of this example, every node has two children and the costs are proportional to the height of the node. Technical Lemma 2.1 is used in the proof of the Lemma 2.2.

Lemma 2.1 *For any polynomial $p(x) = \sum_{k=0}^{m} a_k x^k$, there exists δ and $\alpha < 1$, such that $\frac{p(x+1)}{2p(x)} < \alpha$, for all $x > \delta$.*

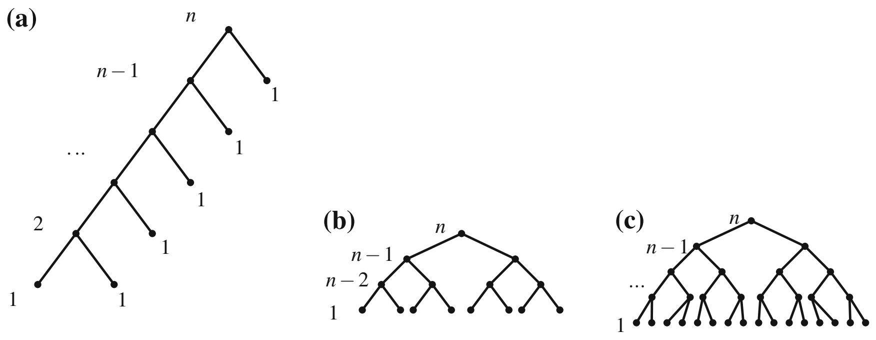

Fig. 2.6 Recursion trees with the cost of each node. **a** The cost of the internal nodes decrease by 1 and the all leaves have cost 1. **b** The nodes in the same level have the same cost, decreasing by 1 from n until $n - 2$. The leaves have cost 1. **c** The nodes in the same level have the same cost, decreasing by 1 from n until 1

Proof It is easy to prove that $\lim_{x\to\infty} \frac{p(x+1)}{2p(x)} = 1/2$. Thus, from the definition of limit, there exist ϵ and δ (depending only on ϵ), such $\frac{p(x+1)}{2p(x)} - 1/2 < \epsilon$ for all $x > \delta$. Choosing $\epsilon < 1/2$, we have that $\frac{p(x+1)}{2p(x)} < \epsilon + 1/2 = \alpha < 1$ for all $x > \delta$.

Lemma 2.2 *Let T be a recursion tree with height n, such that every internal node has degree 2; and the cost for each node is $O(p(i))$, where $p(i)$ is a polynomial and i is the height of the node. Then, the amortized cost for each node is $O(1)$.*

Proof The number of nodes with height i is 2^{n-i}, since the internal nodes have degree 2. The cost of each level i of the tree is bounded by $2^{n-i}p(i)$ and the total cost of the tree is $\sum_{i=1}^{n} 2^{n-i}p(i)$. Let us consider the ratio of the cost of two adjacent levels in the tree,

$$r(i) = \frac{2^{n-(i+1)}p(i+1)}{2^{n-i}p(i)} = \frac{p(i+1)}{2p(i)}.$$

By Lemma 2.1, $r(i) < \alpha < 1$, for all $i > \delta$. Implying that the cost of each level $i \geq \delta$ decrease by α and the sum of the costs of all levels $i \geq \delta$ is bounded by the sum of the geometric series, i.e.

$$\sum_{\delta\leq i\leq n} 2^{n-i}p(i) < 2^{n-\delta}p(\delta) \sum_{\delta\leq i\leq n} \alpha^{i-\delta} < 2^{n-\delta}p(\delta)\sum_{i=0}^{\infty}\alpha^i = \frac{2^{n-\delta}p(\delta)}{1-\alpha}.$$

Therefore, amortized cost of each node in the levels above δ is

$$\frac{\sum_{\delta\leq i\leq n} 2^{n-i}p(i)}{\sum_{\delta\leq i\leq n} 2^{n-i}} < \frac{2^{n-\delta}p(\delta)}{1-\alpha}\frac{1}{2^{n-\delta}} = \frac{p(\delta)}{1-\alpha} = O(1).$$

The last equality follows from the fact that α and $delta$ are constants. Moreover, the cost of the nodes with $i \leq \delta$ is also $O(1)$.

Theorem 2.1 is a straightforward generalization of Lemma 2.2.

Theorem 2.1 *Let T be a recursion tree with height n, such that every internal node has degree* at least *2; and the cost for each node is $O(p(i))$, where $p(i)$ is a polynomial and i is the height of the node. Then, the amortized cost for each node is $O(1)$.*

2.4.1.1 Enumerating Connectivity Elimination Orderings of a Connected Graph *G*

Given a connected graph $G = (V, E)$, a *connectivity elimination ordering* is an ordering of the vertices such that the removal of each vertex keeps the remaining graph connected. Algorithm 11 enumerates all connectivity elimination orderings.

Each call of the algorithm takes $O(|V|^3)$ time, since for each $v \in V$ it checks if $G - v$ is connected. Moreover, for any connected graph there are at least two vertices such that their removal maintain the graph connected. Therefore, the hypothesis of Theorem 2.1 are satisfied, and the amortized complexity per node is $O(1)$. Since, in this case, the number of nodes is at most 2 times the number of leaves, the amortized complexity per solution is also $O(1)$.

Algorithm 11: ENUMORDERINGS($G = (V, E), X$)

```
   Input: A graph $G$ and sequence $X$ that is a prefix of a connectivity elimination order.
   Output: The set of all connectivity elimination orders of $G$.
1  if $V = \emptyset$ then
2      output $X$
3  end
4  foreach $v \in V$ do
5      if $G - v$ is connected then
6          ENUMORDERINGS($G - v, \langle X, v \rangle$)
7      end
8  end
```

2.4.2 Amortization by Children

The basic amortization strategy presented in the previous section, in the form of Theorem 2.1, requires that every leaf has the same depth, and the cost of each node depends uniformly on the height. Though there are applications for Theorem 2.1, these requirements are global, they depend on the tree as whole, imposing a very strict structure in the recursion tree. In this section, we start developing amortization techniques with weaker hypothesis, so that they can be applied also in the case of more biased trees. For this, we focus on local tree structure. Theorem 2.2 presents a simple amortization scheme using only the parent-children structure.

Theorem 2.2 *Let T be a recursion tree and $T(x)$ the cost of node $x \in T$. The amortized cost for each node is $O(\max_{z \in T} \frac{T(z)}{|N^+(z)|+1})$.*

Proof We divide the cost $T(x)$ between the node x and its children $N^+(x)$. In this way the new cost of x is $\frac{T(x)}{|N^+(x)|+1}$ plus the cost received from its parent y. Thus,

$$\begin{aligned} T'(x) &= O\left(\frac{T(x)}{|N^+(x)|+1} + \frac{T(y)}{|N^+(y)|+1}\right) \\ &= O\left(\max_{z \in \{x,y\}} \frac{T(z)}{|N^+(z)|+1}\right) = O\left(\max_{z \in T} \frac{T(z)}{|N^+(z)|+1}\right). \end{aligned}$$

2.4.2.1 Enumerating All Simple Paths of G Starting from s

Given a graph $G = (V, E)$ and a vertex $s \in V$, we consider the problem of enumerating all simple paths of G that start on s. Algorithm 12 solves this problem. Each call outputs a solution and takes $O(|N(s)|)$ time, since we have to explore all edges from s. Moreover, each edge from s generates a recursive call. Therefore, applying Theorem 2.2 we have that the amortized cost per node is $O(\max_{z \in T} \frac{|N(z)|}{|N(z)|+1}) = O(1)$.

Algorithm 12: ENUMPATHS($G = (V, E), s, \pi$)

```
Input: A graph $G$, a vertex $s$ and a path $\pi$ from $s$.
Output: The set of all paths from $s$ in $G$ with prefix $\pi$.
1 output $\pi$
2 foreach $v \in N(s)$ do
3     ENUMPATHS($G - s, v, \langle \pi, s \rangle$)
4 end
```

2.4.3 *Push Out Amortization*

In Lemma 2.2 the key property was that the total cost on each level increases with a constant factor, by going to the next deeper level. Intuitively, the increase of computation time is good because it forbids a sudden decrease, as the one of the trees in Fig. 2.6a, b. In this section, we apply the same idea locally. Instead of comparing the total cost of two adjacent levels we compare the cost of a node with the total cost of its children. Lemma 2.3 gives a precise statement for this local increase property. Afterwards, Theorem 2.3 generalizes Lemma 2.3, by combining it with the amortization by children of Theorem 2.2.

Lemma 2.3 *Let T be a recursion tree, such that the cost of each leaf is $O(T^*)$; and there exist $\alpha > 1$ such that every internal node $x \in T$ satisfy $\sum_{y \in N^+(x)} T(y) \geq \alpha T(x)$, where $T(x)$ is the cost of x. Then, the amortized cost for each node is $O(T^*)$.*

Proof Consider a node $x \in T$ and define $C(x) = \sum_{y \in N^+(x)} T(y)$. We divide the cost $T(x)$ proportionally among the children, so that each $y \in N^+(x)$ receives $T(x)\frac{T(y)}{C(x)}$. Observe that $\sum_{y \in N^+(x)} T(x)\frac{T(y)}{C(x)} = T(x)$, i.e. all the cost $T(x)$ is divided among the children. By doing this division recursively, starting from the root, we have that a node $z \in T$ receives from its parent at most $\frac{T(z)}{\alpha-1}$.

Let us prove the last claim by induction on the depth of the node. Assume that all nodes w with depth $d-1$ receive at most $\frac{T(w)}{\alpha-1}$ from its parent. The base case is trivial, because the root receives no cost. Let z be a node with depth d, its parent w has depth $d-1$. By the induction hypothesis, the cost received by w from its parent is $\frac{T(w)}{\alpha-1}$ and the total cost of w is $T(w) + \frac{T(w)}{\alpha-1}$. Thus, the cost received by z is:

$$\frac{T(z)}{C(w)}\left(T(w)+\frac{T(w)}{\alpha-1}\right)=T(z)\frac{T(w)}{C(w)}\frac{\alpha}{\alpha-1}\leq T(z)\frac{1}{\alpha}\frac{\alpha}{\alpha-1}=\frac{T(z)}{\alpha-1}.$$

The inequality follow from $\frac{T(w)}{C(w)}=\frac{T(w)}{\sum_{y\in N^+(w)}T(y)}\leq\frac{1}{\alpha}$. Completing the proof of the claim.

In the end of the cost division process the only nodes that have non-zero cost are the leaves. For any leaf the cost received from its parent is at most $\frac{T^*}{\alpha-1}$. Therefore, the total cost is $O(T^*+\frac{T^*}{\alpha-1})=O(T^*)$, since α is a constant.

Theorem 2.3 is a generalization of Lemma 2.3. In the theorem, for every node x satisfying item 2 we can use the same amortization strategy of the lemma, i.e. proportionally divide all the cost $T(x)$ among the children. However, instead of stopping this process only on the leaves, we stop on the first node satisfying item 1 or 3. If the node x satisfies item 1 and its ancestrals satisfy item 2, we know that the cost pushed to x is $O(\frac{T^*}{\alpha-1})$ and the total cost of x is $O(T^*+\frac{T^*}{\alpha-1})=O(T^*)$. On the other hand, if a node x satisfies item 3 we can amortize $T(x)$ among the $\Omega(\frac{T(x)}{T^*})$ children or solutions. In this way, each child receives $O(T^*)$ (that is not passed to its grandchildren), so that the cost of x is $O(T^*)$ and we have the same case of item 1.

Theorem 2.3 *Let T be a recursion tree, such that each node $x\in T$ satisfy one of the following properties:*

1. $T(x)=O(T^*)$;
2. $\sum_{y\in N^+(x)}T(y)\geq\alpha T(x)$, *where* $\alpha>1$ *is a constant;*
3. x *has* $\Omega(\frac{T(x)}{T^*})$ *children, or outputs* $\Omega(\frac{T(x)}{T^*})$ *solutions.*

Then, the amortized cost for each node is $O(T^*)$.

2.4.3.1 Matching Enumeration

Given an undirected graph $G=(V,E)$, a *matching* M in G is a set of pairwise non-adjacent edges, i.e. for any two edges $(u,v)\neq(x,y)\in M$, we have that $\{u,v\}\cap\{x,y\}=\emptyset$. The set of all matchings of G is denoted by $\mathscr{M}(G)$. Several variants of matching enumeration have been studied: perfect (every vertex has an incident edge in M) matching enumeration in bipartite graphs [26, 43–45], perfect matching in general graphs [46], maximal matchings in bipartite graphs [45] and maximal matchings in general graphs [47]. In this section, we consider the problem of enumerating all matchings of a graph. First, we present a simple algorithm (Algorithm 13) that correctly enumerates all matchings. Then, we modify it (Algorithm 14) to satisfy the hypothesis of Theorem 2.3, so we can use it to improve the algorithm complexity.

Algorithm 13 uses the binary partition method, each recursive call partitions $\mathscr{M}(G)$ in two sets: matchings not including the edge (u, v) and the ones including it. In the first case (line 6), we remove (u, v) from G and leave M unchanged. In the second case (line 7), we add (u, v) to the matching M and remove from G all edges adjacent to (u, v). The cost of a node is bounded by, the number of edges removed, $O(|V|)$. Now let us analyse the structure of the recursion tree. The conditional of line 1 ensures that only leaves output solutions. Moreover, there is always a matching including (u, v) and one not including it, so every node leads to a solution. Thus, in the recursion tree all internal nodes have exactly two children and every leaf output a solution. Therefore, the number of nodes is bounded by $2|\mathscr{M}(G)|$, and Algorithm 13 takes $O(|V|)$ time for each matching.

Algorithm 13: EnumMatching($G = (V, E), M$)

```
Input: A graph $G$ and a matching $M$ (eventually empty)
Output: $\mathscr{M}(G)$, the set of matchings of $G$
1 if $E = \emptyset$ then
2     output $M$
3     return
4 end
5 choose an edge $(u, v) \in E$
6 EnumMatching($G - (u, v), M$)
7 EnumMatching($G - \{(x, y) \in E | x \in \{u, v\}\}, M \cup \{(u, v)\}$)
```

Actually, each node in the recursion tree of Algorithm 13 takes $O(|N(u)|+|N(v)|)$ time. Consider a node x with the input graph $G = (V, E)$, the input graph of its children contain $|E| - 1$ and $|E| - |N(u)| - |N(v)|$ edges. Hereafter, for the sake of clear analysis, we bound the computation time of each node by $c|E|$. In this way, the cost for x is $T(x) = c|E|$ and, $T(y_1) = c(|E| - 1)$ and $T(y_2) = c(|E| - |N(u)| - |N(v)|)$ for each child. Based on this costs we cannot apply Theorem 2.3. The leaves take $O(1)$ time, satisfying item 1. However, the internal nodes do not satisfy item 2 or 3. Each internal node has exactly two children and do not output solutions, so that item 3 is not satisfied. On the other hand, the total computation time of the children is not increasing by constant factor over the parent, i.e. there is no constant $\alpha > 1$ such that $T(y_1) + T(y_2) \geq \alpha T(x)$.

In order to satisfy item 3 of Theorem 2.3 we need $|N(u)| + |N(v)|$ to be bounded, so that $T(y_2)$ is not too small. The key property is that for any graph either there is an edge (u, v) such that $|N(u)| + |N(v)| < |E|/2$ or there is a vertex u with $|N(u)| \geq |E|/4$. Algorithm 14 is a modified version of Algorithm 13 that uses this property. If there exists an edge (u, v) such that $|N(u)| + |N(v)| < |E|/2$ (line 5), we have that

$$T(y_1) + T(y_2) \geq c(|E| - 1) + c\frac{|E|}{2} \geq \frac{3}{2}T(x),$$

satisfying item 2. Alternatively, if there exists u such that $|N(u)| \geq |E|/4$ (line 8), we create at least $|E|/4$ children, satisfying item 3. Therefore, applying Theorem 2.3 and using that the number of internal nodes is bounded by $2|\mathscr{M}(G)|$, we have that Algorithm 14 takes $O(1)$ time amortized for each matching enumerated.

Algorithm 14: EnumMatching($G = (V, E), M$)

```
Input: A graph $G$ and a matching $M$ (eventually empty)
Output: $\mathscr{M}(G)$, the set of matchings of $G$
1  if $E = \emptyset$ then
2      output $M$
3      return
4  end
5  if $\exists (u, v) \in E$ s.t. $|N(u)| + |N(v)| < |E|/2$ then
6      EnumMatching($G - (u, v), M$)
7      EnumMatching($G - \{(x, y) \in E | x \in \{u, v\}\}, M \cup \{(u, v)\}$)
8  else
9      choose $u$ s.t. $|N(u)| \geq |E|/4$
10     EnumMatching($G - u, M$)
11     foreach $v \in N(u)$ do
12         EnumMatching($G - \{(x, y) \in E | x \in \{u, v\}\}, M \cup \{(u, v)\}$)
13     end
14 end
```

2.5 Data-Driven Speed up

Polynomial delay algorithms, especially linear delay, can be considered as being very close to optimal algorithms in practice. Indeed since the delay is defined as a function of the input size, it can be often considered very small or negligible with respect to the usual exponential number of outputted solutions. However, there are certain applications, particularly in data mining, in which the input data are very large (frequent pattern mining, candidate enumeration, community mining, feasible solution enumeration). In such a context, when the number of outputted solutions is expected to be small (polynomial in the input size), the problem is said to be *large-scale tractable*. This is the case of enumerating peripheral or central vertices in large graphs, as shown in Chap. 7.

On the other hand, when the number of solutions increases exponentially with a linear increase in the instance size, we usually have that many solutions are similar and therefore redundant. In these cases, a post processing step is required to suppress the redundant solutions. Since the number of solutions is huge, the post processing is very time consuming or even infeasible, so these problems are considered intractable. This could be potentially the case of the frequent itemset problem, that is the problem of enumerating all the patterns appearing frequently in a large database, where a pattern can be a sequence of items, a short string, or a subgraph (any subset of a frequent itemset is also frequent). However this intractability is very often linked to the application: for example, even if in theory the number of maximal cliques can be exponential in the size of the graph, in practice for large sparse graphs this number is usually polynomial in the size of the graph. On the other hand, however, the number of independent sets in a graph is huge both in theory and in practice.