JTDec: A Tool for Tree Decompositions in Soot

Krishnendu Chatterjee[†] Amir Kafshdar Goharshady[†] Andreas Pavlogiannis[†]

[†] IST Austria (Institute of Science and Technology Austria),
Am Campus 1, 3400, Klosterneuburg, Austria
{krishnendu.chatterjee, amir.kafshdar.goharshady, andreas.pavlogiannis}@ist.ac.at

Abstract. The notion of treewidth of graphs has been exploited for faster algorithms for several problems arising in verification and program analysis. Moreover, various notions of balanced tree decompositions have been used for improved algorithms supporting dynamic updates and analysis of concurrent programs. In this work, we present a tool for constructing tree-decompositions of CFGs obtained from Java methods, which is implemented as an extension to the widely used Soot framework. The experimental results show that our implementation on real-world Java benchmarks is very efficient. Our tool also provides the first implementation for balancing tree-decompositions. In summary, we present the first tool support for exploiting treewidth in the static analysis problems on Java programs.

JTDec is available at `http://pub.ist.ac.at/~akafshda/JTDec/` and a conference version of this paper is due to appear in the Fifteenth International Symposium on Automated Technology for Verification and Analysis, ATVA 2017.

1 Introduction

Treewidth of graphs. A very widely studied and well-known concept in graph theory for algorithmic analysis is the notion of *treewidth*, which measures the similarity of a graph to a tree [16, 14]. Along with its mathematical elegance, the treewidth property has great practical relevance, as many NP-complete problems can be solved in polynomial time on graphs of constant treewidth [3, 4].

Constant treewidth in verification and program analysis. The constant treewidth property has not only been studied in the graph algorithmic community, but has been considered in many problems in verification and program analysis.

Verification. The constant-treewidth property has played an important role in logic and verification; for example, MSO (Monadic Second Order logic) queries can be solved in polynomial time [10] (also in log-space [12]) for constant-treewidth graphs; parity games on graphs with constant treewidth can be solved in polynomial time [15]; and there exist faster algorithms for probabilistic models (such as Markov decision processes) [6]. Recently it was shown for problems in quantitative verification the constant treewidth can be exploited to design much faster algorithms [5], as well as improve space usage [7].

Program analysis. A very important class of constant-treewidth graphs is the control flow graphs (CFGs) of goto-free programs of many programming languages [17]. It has also been shown that typically all Java programs have small treewidth [13]. The small treewidth has been used to develop algorithms for (i) register allocation in polynomial time [17, 2], (ii) interprocedural analysis [9], and (iii) intraprocedural analysis of concurrent programs [8].

Relevant algorithmic questions. In the context of program analysis, the relevant algorithmic questions are: (a) given an input CFG of constant treewidth, construct a constant-width

decomposition; and (b) balance a constant-width tree decomposition Balanced tree decompositions are required to support fast dynamic algorithms (i.e., algorithms that support fast updates given small changes in the input graph) [9] as well as for intraprocedural analysis of concurrent programs [8].

Our contributions. Although the treewidth property has been exploited for faster algorithms in many problems in verification and program analysis, there exists no tool support for the algorithmic questions we consider. In this work we present JTDec, a tool for constructing tree decompositions of Java programs. We have implemented existing algorithms for the above algorithmic questions, along with several heuristics that exploit the special structure of programs. Our tool is integrated as a plugin in the widely used Soot framework [18]. Our experimental results show that our implementation on real-world Java benchmarks is very efficient. In summary we present the first tool for tree-decomposition and balanced tree-decompositions of CFGs of programs in Java, which can be used by algorithms that exploit the low-treewidth property of graphs.

2 Definitions

Graphs and Trees. Let $G = (V, E)$ be a finite directed graph (henceforth called simply a graph) where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation. Given a set of nodes $X \subseteq V$, we denote by $G \restriction X = (X, E \cap (X \times X))$ the subgraph of G induced by X. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \ldots, x_k)$ such that $x_1 = u$, $x_k = v$, and for all $1 \leq i < k$ we have $(x_i, x_{i+1}) \in E$. The length of P is $|P| = k - 1$. A set of nodes $X \subseteq V$ is called a *connected component* of G, if for every pair of nodes $u, v \in X$, there is either a path $P_1 : u \rightsquigarrow v$ or a path $P_2 : v \rightsquigarrow u$ in $G \restriction X$. Additionally, X is called *strongly-connected* if both P_1 and P_2 exist. A *tree* $T = (V, E)$ is an undirected graph with a root node u_0, such that between every two nodes there is a unique acyclic path. For a node u, we denote by $\mathsf{Lv}(u)$ the *level* of u which is defined as the length of the acyclic path from u_0 to u. A *child* of a node u is a node v such that $\mathsf{Lv}(v) = \mathsf{Lv}(u) + 1$ and $(u, v) \in E$, and then u is the *parent* of v. A tree T is *k-ary* if every node has at most k-children (e.g., a binary tree has at most two children for every node). Finally, the *depth* of T is the maximum level of its nodes, i.e. $\max_u \mathsf{Lv}(u)$, and T is called *balanced* if its depth is logarithmic on its size, i.e. $\max_u \mathsf{Lv}(u) = O(\log n)$.

Tree-decompositions. A *tree-decomposition* $\mathrm{Tree}(G) = T = (V_T, E_T)$ of a graph G is a tree, where every node B_i in T, which is called a bag, is a subset of nodes of G such that:

- C1 $V_T = \{B_0, \ldots, B_b\}$ with $B_i \subseteq V$, and $\bigcup_{B_i \in V_T} B_i = V$ (every node is covered).
- C2 For all $(u, v) \in E$ there exists $B_i \in V_T$ such that $u, v \in B_i$ (every edge is covered).
- C3 For all i, j, k such that there is a bag B_k that appears in the unique path $B_i \rightsquigarrow B_j$ in T we have $B_i \cap B_j \subseteq B_k$ (every node appears in a contiguous subtree of T).

Conventionally, we call B_0 the root of T, and denote by $\mathsf{Lv}(B_i)$ the level of B_i in T. For a bag B of T, we denote by $T(B)$ the subtree of T rooted at B. A bag B is called the *root bag* of a node u if $u \in B$ and every B' that contains u appears in $T(B)$. We often use B_u to refer to the root bag of u, and define $\mathsf{Lv}(u) = \mathsf{Lv}(B_u)$. A tree decomposition T is called *normal* if for each $B_1, B_2 \in V_T$, such that B_2 is a child of B_1, then $|B_2 \setminus B_1| \leq 1$, i.e. each bag has at most one more node from its parent. The *width* of the tree-decomposition T is the size of the largest bag minus 1. The treewidth t of G is the smallest width among all tree-decompositions of G.

(α, β, γ) **tree-decompositions.** Given a graph G with treewidth t and a fixed $\alpha \in \mathbb{N}$, a tree-decomposition $\mathrm{Tree}(G)$ is called α-*approximate* if it has width at most $\alpha \cdot (t+1) - 1$. Given a real constant $\beta < 1$ and an integer constant $\gamma \geq 1$ we say that $\mathrm{Tree}(G)$ is (β, γ)-balanced, if for every bag B and every descendant B' of B with $\mathsf{Lv}(B') = \mathsf{Lv}(B) + \gamma$, the size of the subtree $T(B')$ is at most β times as large as the size of the subtree $T(B)$. A (β, γ)-balanced tree-decomposition that is α-approximate is called an (α, β, γ) tree-decomposition.

Algorithmic questions. We consider the following algorithmic questions:

Q1: Given a constant-treewidth input graph (being the CFG of a method), construct a tree-decomposition of constant width.

Q2: Convert an input tree decomposition to a balanced one.

Q3: Convert a tree decomposition to an (α, β, γ) tree-decomposition, for a single balancing parameter $\lambda \geq 2$, and $\alpha = 2 \cdot \lambda$, $\beta = 2^{-\lambda+1}$ and $\gamma = \lambda$ (see Remark 1 below).

We note that the solution for Q3 subsumes the solution for Q2 (with any constant λ). In the next section we will present details of the algorithms for Q1 and Q3.

Remark 1 (Significance). (1) The basic tree-decomposition has been used for polynomial-time algorithms for register allocation [17, 2]. (2) The balanced tree-decompositions have been crucial in algorithms for interprocedural analysis [9] and verification of quantitative properties in graphs [5]. (3) The notion of (α, β, γ) tree-decompositions has been used in algorithmic dataflow analysis of concurrent programs [8]. The ideal value for α and γ is 1, and for β is $\frac{1}{2}$. However, this exact combination is not achieved in any of the known algorithms. In Q3 we consider parameters for which efficient algorithms exist and suffice for the problems considered in [8].

3 Algorithms

We present the algorithms for Q1-Q3. First, we focus on Q1 and then consider Q3 (which subsumes Q2). Our tool JTDec implements all the algorithms of this section.

3.1 Tree-decomopsitions of CFGs

There exist several general-purpose tree-decomposition algorithms which operate on arbitrary graphs. Here our focus is on tree-decompositions of CFGs. The main part of this section focuses on outlining a tree-decomposition algorithm that operates on input being the source code, as opposed to arbitrary graphs. In particular, the input to the algorithm is a method in Jimple, which is a standard, 3-address representation of Java methods in the Soot framework. As an example, Fig. 1 depicts a Java method and its Jimple representation, and Fig. 2 shows the CFG of a simplified version of the Jimple representation and a balanced tree-decomposition of the CFG.

A dedicated algorithm for tree-decompositions of CFGs. We consider dedicated algorithms that are specific to CFGs, and operate on the source code rather than on the graph itself [17]. In the following we outline the key steps of one such algorithm. We phrase the algorithm on Jimple source code. We start with the notion of complex listings required for the algorithm.

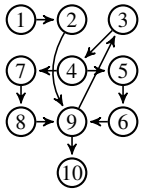
$(\leq k)$-*complex listings.* A $(\leq k)$-complex listing of a graph $G = (V, E)$ is a permutation of the elements of V, such that for each $u \in V$, there exists a set S_u of at most k nodes preceding u in the permutation, and whose deletion from G separates u from all the nodes

```
void threeNPlusOne(int n)
    {
  while(n > 1){
   if(n % 2 == 0){
    n /= 2;
   }
   else{
    n = 3 * n + 1;
   }
  }
 }
```

```
1:    n := parameter0: int
2:    nop
3:    if n > 1 goto nop
4:    goto [?= nop]
5:    nop
6:    temp$0 = n % 2
7:    if temp$0 == 0 goto
      nop
8:    goto [?= nop]
9:    nop
10:    temp$1 = n
11:    temp$2 = temp$1/2
12:    n = temp$2
13:    goto [?= nop]
14:    nop
15:    temp$3 = 3 * n
16:    temp$4 = temp$3
17:    temp$5 = temp$4+1
18:    n = temp$5
19:    nop
20:    goto [?= nop]
21:    nop
22:    return
```

Fig. 1: A Java method (left), and its 3-address code representation in Jimple.

```
1 i0 := parameter0: int
2 goto [?= (branch)]
3 $i1 = i0 % 2
4 if $i1 != 0 goto $i2 = 3 * i0
5 i0 = i0 / 2
6 goto [?= (branch)]
7 $i2 = 3 * i0
8 i0 = $i2 + 1
9 if i0>1 goto $i1 = i0 % 2
10 return
```



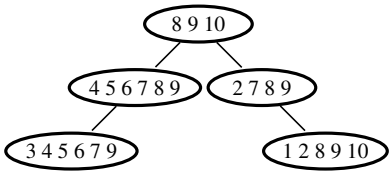


Fig. 2: A simplified Jimple representation of the method in Fig. 1 (left), Its CFG G and an approximate, balanced tree-decomposition $\mathrm{Tree}(G)$ (right).

preceding u in the permutation. In this case S_u is called a *separator* of u. Given a listing and a node u, there is a unique minimal choice for separators of nodes [17]. We always use minimal separators and for brevity drop the word minimal in the sequel. A graph G is called $(\leq k)$-complex if it has a $(\leq k)$-complex listing and is called k-complex if it has a $(\leq k)$-complex listing but no $(\leq k-1)$-complex listings. Given a $(\leq k)$-complex listing L and a pair of distinct nodes u, v, we write $v < u$ to denote that v appears before u in L.

Our main approach for obtaining tree-decompositions from CFGs of Jimple methods is the following theorem and algorithm, which allows us to focus on computing $(\leq k)$-complex listings of the CFG:

Theorem 1. *A graph G has treewidth k if and only if it is k-complex [11]. Given any such listing and separators of all the nodes, a tree-decomposition $\mathrm{Tree}(G)$ of width k can be constructed in linear time [17].*

CFG-specific algorithm. Given a listing L, the above algorithm simply creates one bag B_x per node x, and connects the bag B_u to the bag B_v if $v < u$ and v is the latest element of S_u in the listing or to the root if no such v exists. The bag corresponding to u will contain the set $S_u \cup \{u\}$. Hence, computing a tree-decomposition efficiently reduces to two steps:

1. Finding "good" listings, i.e., listings where each separator has small size.
2. Given a listing L, finding efficiently the separator S_u of every node u.

We provide brief and intuitive description of the two steps (following [17]), and refer to the Appendix for the pseudocode.

Heuristic for good listings. Our algorithm implements a heuristic of [17] for processing programs in the form of 3-address codes, which is guaranteed to create a listing of small separators for CFGs of goto-free programs, and is expected to perform well for structured

programs. Intuitively, given a CFG, a listing with small separators can be obtained using the following two rules for the heuristic:

- The nodes of CFG that correspond to entries and exits of block structures (e.g. if-blocks, while-loops) must appear early in the listing.
- The remaining nodes (e.g. statements within the structures) must appear after the entries and exits of these structures in the listing.

Finding separators in listings. Given a listing L, the separators S_u of nodes of L can be created efficiently in $O(n \lg^* n)$ time. This is achieved by a single traversal of L from right to left, and maintaining a *disjoint-set* data-structure, which keeps track of the strongly-connected components of G formed by the set of nodes that have been examined already.

3.2 Constructing (α, β, γ) tree-decompositions

Given a CFG G and a binary tree-decomposition $\mathrm{Tree}(G)$ of width k, let $\lambda \geq 2$ be the balancing (integer) parameter (c.f. Q3). For $\alpha = 2 \cdot \lambda$, $\beta = 2^{-\lambda+1}$ and $\gamma = \lambda$, the core procedure for constructing an (α, β, γ) tree-decomposition is a recursive one. In each step of the recursion, the algorithm uses one of two rules to split a subtree of $\mathrm{Tree}(G)$ to connected components. Informally, (i) Rule 1 controls the height (i.e., parameters (β, γ)), and (ii) Rule 2 controls the width (i.e., parameter α) of the constructed tree-decomposition. The balancing parameter λ specifies how often each rule is used in the recursion, and thus specifies the trade-off between height and width. The algorithm is a simplified and efficiently implementable version of [8, Section 3] (see Appendix for the pseduocode). Given a tree-decomposition of $O(n)$ bags, the algorithm runs in $O(n \cdot \log n)$ time and $O(n)$ space.

4 Implementation

We build upon the widely used Soot framework and JTDec is an extension to it. Soot is a framework for language manipulation and optimization that provides tools for different problems in static program analysis. Soot is written in Java and has many different intermediate representation schemes for Java programs. We use the Jimple representation which has a typed 3-address format and is the most widely used of the representations. The main class for representing CFGs is `BriefUnitGraph` in which each node of the CFG corresponds to one Jimple statement, and is represented by the `Unit` class.

Our implementation is placed under the package `JTDec`. We provide an implementation of a class `JTDecTree` which is used to store and manipulate tree decompositions and supports basic tree-decomposition operations, e.g., iterating over the bags of the tree, and adding/removing nodes to bags. Based on this, we have implemented the algorithms of Section 3 in another class called `JTDec` in an easy-to-use manner. Specifically, we give the user access to the following functions in `JTDec`:

1. `createTreeDec`: The input is a `SootMethod` method and returns a tree decomposition of the CFG of the method. In the CFG we treat each `Unit` as one node.
2. `normalizeTreeDec`: The input is a tree-decomposition T in form of `JTDecTree` and returns a normalized version of T.
3. `createBalancedTree`: The input is an integer λ and tree-decomposition T in form of `JTDecTree`, and returns a balanced version of T with parameter λ.
4. `process`: The input is a `SootMethod` and (optionally) integer λ, and applies all the above functions in the same order and returns a balanced tree decomposition of the CFG of the method.

Range	#M	Unbalanced			λ = 2			λ = 3			λ = 4			λ = 5		
		T	W	H	T	W	H	T	W	H	T	W	H	T	W	H
[50, 59]	494	0	3.1	29.7	1	8.5	6.9	1	9.9	5.3	1	10.7	5.1	1	11.0	4.6
[60, 69]	343	1	3.2	33.0	2	8.8	7.1	1	10.6	5.7	2	11.2	5.6	1	11.5	5.0
[70, 79]	232	1	3.4	39.3	2	8.6	7.6	2	10.5	6.4	2	11.1	5.8	1	11.4	5.6
[80, 89]	170	1	3.4	42.8	2	9.0	8.0	2	11.0	6.6	2	11.5	5.9	2	11.9	5.8
[90, 99]	128	1	3.6	45.7	3	9.8	8.4	2	12.1	6.8	2	12.5	5.9	2	13.0	5.8
[100, 149]	394	1	3.6	59.5	4	10.0	8.9	3	12.3	7.2	3	13.1	6.4	3	13.4	6.3
[150, 299]	270	2	4.7	90.6	8	11.3	19.4	8	14.1	8.1	7	14.7	7.2	6	15.2	7.1
[300, 999]	81	5	5.4	203.1	22	17.4	11.8	19	26.6	9.4	17	27.1	8.5	16	27.4	8.0
[1000, 2156]	9	46	2.6	1079.7	98	21.3	15.6	82	20.0	12.2	72.0	26.7	10.9	71	26.7	10.6

Table 1: Evaluation results of JTDec. Times are rounded to the nearest millisecond.

Examples of using `JTDec` can be found in the Appendix. The tool and source code are available at `http://pub.ist.ac.at/~akafshda/JTDec/`.

5 Evaluation

Experimental results show that JTDec is very efficient. We used JTDec to obtain tree-decompositions for methods from the DaCapo benchmark suit [1], and to obtain balanced tree-decompositions using different values of the balancing parameter λ. The experiments were run on a laptop with Intel Core i5 5200U Processor (2.7 GHz) and 8 GB of RAM. Table 1 summarizes the results. We divided the benchmark methods based on number of nodes in their CFG to several ranges and for each range we report number of methods in that range (#M), mean execution time of the function `createTreeDec` (T), mean width of the obtained tree-decompositions (W) and their mean height (H). Then for each λ, we report the mean execution time (T) of `createBalancedTree` on the tree-decompositions obtained previously, the mean width of the obtained balanced tree-decompositions (W) and their mean height (H). The table shows the trade-off between the latter two. All times are measured in milliseconds. Soot's analysis typically takes much more time than JTDec.

Acknowledgements. We thank all reviewers for their helpful comments which led to considerable improvements in presentation. The research is partially supported by Vienna Science and Technology Fund (WWTF) ICT15-003, Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE) and ERC Start grant (279307: Graph Games).

References

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, and S. Z. Guyer. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 169–190.

[2] H. Bodlaender, J. Gustedt, and J. A. Telle. "Linear-time register allocation for a fixed number of registers". In: *SODA*. Vol. 98. 1998, pp. 574–583.

[3] H. L. Bodlaender. "A Tourist Guide through Treewidth." In: *Acta cybernetica* (1993).

[4] H. L. Bodlaender. "Discovering Treewidth." In: *SOFSEM*. Vol. 3381. Springer. 2005, pp. 1–16.

[5] K. Chatterjee, R. Ibsen-Jensen, and A. Pavlogiannis. "Faster algorithms for quantitative verification in constant treewidth graphs". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 140–157.

[6] K. Chatterjee and J. Lacki. "Faster algorithms for Markov decision processes with low treewidth". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 543–558.

[7] K. Chatterjee, R. Rasmus Ibsen-Jensen, and A. Pavlogiannis. “Optimal reachability and a space-time tradeoff for distance queries in constant-treewidth graphs”. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 57. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[8] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and A. Pavlogiannis. “Algorithms for algebraic path properties in concurrent systems of constant treewidth components”. In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 733–747.

[9] K. Chatterjee, R. Ibsen-Jensen, A. Pavlogiannis, and P. Goyal. “Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth”. In: *POPL*. ACM, 2015.

[10] B. Courcelle. “The monadic second-order logic of graphs. I. Recognizable sets of finite graphs”. In: *Information and computation* 85.1 (1990), pp. 12–75.

[11] N. D. Dendris, L. M. Kirousis, and D. M. Thilikos. “Fugitive-search games on graphs and related parameters”. In: *Theoretical Computer Science* 172.1-2 (1997), pp. 233–254.

[12] M. Elberfeld, A. Jakoby, and T. Tantau. “Logspace versions of the theorems of Bodlaender and Courcelle”. In: *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*. IEEE. 2010, pp. 143–152.

[13] J. Gustedt, O. Mæhle, and J. Telle. “The treewidth of Java programs”. In: *Algorithm Engineering and Experiments* (2002), pp. 57–59.

[14] R. Halin. “S-functions for graphs”. In: *Journal of geometry* 8.1-2 (1976), pp. 171–186.

[15] J. Obdrzálek. “Fast mu-calculus model checking when tree-width is bounded”. In: *CAV*. Vol. 3. Springer. 2003, pp. 80–92.

[16] N. Robertson and P. D. Seymour. “Graph minors. III. Planar tree-width”. In: *Journal of Combinatorial Theory, Series B* 36.1 (1984), pp. 49–64.

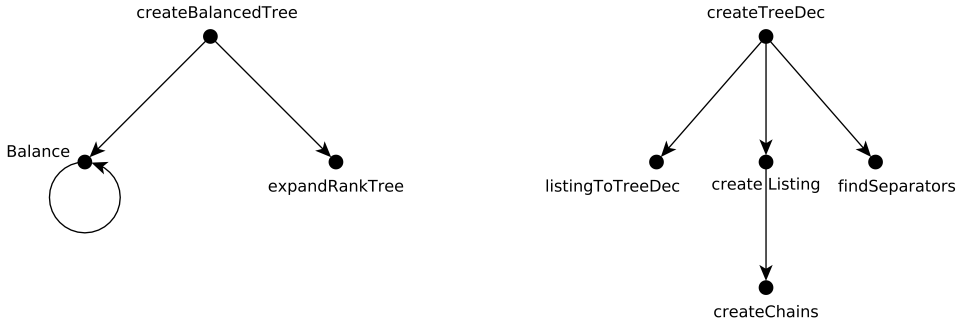
[17] M. Thorup. “All structured programs have small tree width and good register allocation”. In: *Information and Computation* 142.2 (1998), pp. 159–181.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. “Soot-a Java bytecode optimization framework”. In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1999, p. 13.

Appendix

A Pseudocode and Details of Algorithms

In this section we provide psuedocode of all the algorithms implemented in the class `JTDec` along with descriptions of what they do and references. The graph below shows the dependencies between these algorithms. An edge from a procedure A_1 to another procedure A_2 means that A_1 calls A_2 as part of its operation.



A.1 Obtaining a Tree Decomposition from the Control Flow Graph

We use the algorithm introduced in [17] to obtain a listing of constant complexity. We then find the separators of nodes in this listing and convert it to a tree decomposition. The following two algorithms, for creating listings are quoted from [17] and are treated in more details there. The second algorithm finds maximal chains. Given a set $I \subseteq \{1, 2, \ldots, n\}^2$ a chain from i to j in I is a sequence of pairs $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k) \in I$ with $i = i_1$ and $j = j_k$ such that for each $1 \leq l < k$, the pairs (i_l, j_l) and (i_{l+1}, j_{l+1}) are chained together as segments, i.e. $i_l < i_{l+1} < j_l < j_{l+1}$. A chain from i to j in I is called a maximal I-chain if no chain exists from i' to j' such that $i' \leq i$ and $j' \geq j$ and $(i', j') \neq (i, j)$. Maximal chains in the following algorithm correspond to the two intuitive properties of listings as in the heuristic Section 3.1.

In order to find separators of vertices in a given listing, we parse the listing from right to left (end to beginning) and maintain the following parameters as we go left:

- Connected components of the induced subgraph of the CFG on parsed vertices, and
- For each such connected component, a set of its neighbors among non-parsed vertices.

When we reach a vertex u, its separator, S_u is the set of all non-parsed vertices that can be reached from u through a path that contains no other non-parsed vertices. We can find this set by merging u's connected component with components of all parsed neighbors of u and then getting the list of non-parsed neighbors of the resulting component. If we store the connected components in a disjoint set data structure that uses trees and path compression, as is done in the JTDec class `JTDecDSU`, the whole algorithm takes amortized $O(n \lg^* n)$ time, assuming the listing has a constant complexity.

Finally, we use the algorithm from Theorem 1 to convert a listing to a tree decomposition.

Algorithm 1: createListing

Input: A method M containing n nodes $s_1, s_2, \ldots s_n$ and its control flow graph
Output: A listing of the nodes of M

Assign $J \leftarrow$ all pairs (i, j) connected in the control flow graph, i.e. s_i contains a jump to s_j
Assign $S \leftarrow$ symmetric closure of J
createChains(J)
createChains(S)
Assign $i \leftarrow 0$
for j ***from*** n ***downto*** 1 **do**
    if s_j *is not marked* **then**
        Mark s_j with i
        Assign $i \leftarrow i + 1$
    if *there is a maximal* S*-chain from* k *to* j *and* s_k *is not marked* **then**
        Mark s_k with i
        Assign $i \leftarrow i + 1$
    if *there is a maximal* J*-chain from* k *to* j *and* s_k *is not marked* **then**
        Mark s_k with i
        Assign $i \leftarrow i + 1$
end
return nodes in order of their marks

Algorithm 2: createChains

Input: A subset I of $\{1, 2, \ldots, n\}^2$
Output: A set C of pairs (i, j) such that there exists a maximal I-chain from i to j

Assign $C \leftarrow \emptyset$
Assign $s \leftarrow 0$
Assign $i_0 \leftarrow 0$
Assign $j_0 \leftarrow n + 1$
for i ***from*** 1 ***to*** n **do**
    if *there is a* $j > i$*, such that* $(i, j) \in I$ **then**
        Assign $j \leftarrow \max\{j | (i, j) \in I\}$
        while $j_s \leq i$ **do**
            Assign $C \leftarrow C \cup \{(i_s, j_s)\}$
            Assign $s \leftarrow s - 1$
        end
        while $j \geq j_s > i$ **do**
            Assign $i \leftarrow i_s$
            Assign $s \leftarrow s - 1$
        end
        Assign $s \leftarrow s + 1$
        Assign $i_s \leftarrow i$
        Assign $j_s \leftarrow j$
end
return C

Algorithm 3: findSeparators

Input: A listing L of size n
Output: Separators of nodes in L

Initialize a disjoint set data structure for storing connected components
for i ***from*** n ***downto*** 1 **do**
    for $j > i$ *in neighbors of* i **do**
        Merge the connected components of i and j in the disjoint set data structure
    end
    Assign $S_i \leftarrow$ the set of all neighbors k of the connected component of i, such that $k < i$
end
return S_i's

Algorithm 4: listingToTreeDec

Input: A listing $v_1, v_2, \ldots, v_n$ of the nodes of a method M and its nodes' separators, The control flow graph of M
Output: A tree decomposition of the control flow graph of M

Assign $T \leftarrow$ empty tree decomposition
Add $B_1 = \{v_1\}$ as the root of T
for i ***from*** 2 ***to*** n **do**
    Assign $B_i \leftarrow S_{v_i} \cup \{v_i\}$
    Add B_i to T
    Assign $T.parent[B_i] \leftarrow B_1$
    if $S_{v_i} \neq \emptyset$ **then**
        Assign $h \leftarrow max\{j | v_j \in S_{v_i}\}$
        Assign $T.parent[B_i] \leftarrow B_h$
end
return T

Algorithm 5: createTreeDec

Input: A method M and its control flow graph
Output: A tree decomposition of the control flow graph of M

Create a listing of the control flow graph using `createListing`
Find the separators of this listing using `findSeparators`
Convert the listing to a tree decomposition using `listingToTreeDec`
return the obtained tree decomposition

A.2 Obtaining Balanced Tree Decompositions

The algorithm for balanced tree-decompositions is a simplified and efficiently implementable version of [8, Section 3]. The algorithm consists of two conceptual steps.

1. Given a binary tree-decomposition $\mathrm{Tree}(G)$ and a balancing integer parameter $\lambda \geq 2$, a tree of bags T is constructed, which is (β, γ)-balanced (algorithm Balance).
2. T is turned to an α-approximate tree decomposition of G (algorithm `expandRankTree`).

Algorithm Balance is a recursive procedure that operates on inputs $\mathcal{C}$ which are connected components of bags of $\mathrm{Tree}(G)$. Given such a component $\mathcal{C}$, Balance determines a bag $B \in \mathcal{C}$, using one of two rules. Informally, (i) Rule 1 controls the height (i.e., parameters (β, γ)), and (ii) Rule 2 controls the width (i.e., parameter α) of the constructed tree-decomposition. The balancing parameter λ specifies how often each rule is used in the recursion, and thus specifies the trade-off between height and width. In either case of the rules, the removal of B splits $\mathcal{C}$ into components $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$. Then `Balance` is called recursively on those inputs, and the return bags B_1, B_2, and B_3 are made children of B in the constructed tree T. The algorithm `Balance` uses the notion of neighborhood which is defined as follows: The *neighborhood* $\mathsf{Nh}(\mathcal{C})$ of the component $\mathcal{C}$ contains all bag of $\mathrm{Tree}(G)$ that are adjacent to nodes of $\mathcal{C}$. Additionally, given a bag B, we define $\mathsf{Nh}(B) = \mathsf{Nh}(\mathcal{C})$, where $\mathcal{C}$ is the component on which B was chosen by Balance according to Rule 1 or Rule 2. The correctness of the algorithm follows from [8, Theorem 1] (as we consider a simplified version).

Algorithm 6: Balance

Input: A component $\mathcal{C}$ of T, a natural number $\ell \in [\lambda]$
Output: A rank tree $\mathsf{R_G}$

Assign $\mathcal{T} \leftarrow$ an empty tree
if $\mathcal{C} = \{B\}$ **then**
    Make B the root of $\mathcal{T}$
else if $\ell > 0$ **then**
    `// Rule 1`
    Let $B \leftarrow$ a bag of $\mathcal{C}$ whose removal splits $\mathcal{C}$ to $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ with $|\mathcal{C}_i| \leq \frac{|\mathcal{C}|}{2}$
    Assign $T_i \leftarrow \mathsf{Balance}(\mathcal{C}_i, (\ell + 1) \mod \lambda)$
    Make B the root of $\mathcal{T}$ and $\mathcal{T}_i$ the children of $\mathcal{T}$
else
    if $|\mathsf{Nh}(\mathcal{C})| > 1$ **then**
        `// Rule 2`
        Let $B \leftarrow$ a bag of $\mathcal{C}$ whose removal splits $\mathcal{C}$ to $\mathcal{C}_1, \mathcal{C}_2$ with $|\mathsf{Nh}(\mathcal{C}_i) \cap \mathsf{Nh}(\mathcal{C})| \leq \frac{|\mathsf{Nh}(\mathcal{C})|}{2}$
        Assign $\mathcal{T}_i \leftarrow \mathsf{Balance}(\mathcal{C}_i, (\ell + 1) \mod \lambda)$
        Make B the root of $\mathcal{T}$ and $\mathcal{T}_i$ and $\mathcal{T}_2$ the children of $\mathcal{T}$
    else
        Assign $\mathcal{T} \leftarrow \mathsf{Balance}(\mathcal{C}, (\ell - 1) \mod \lambda)$
    end
end
return $\mathcal{T}$

Algorithm 7: expandRankTree

Input: A rank tree $\mathcal{T}$ obtained from `Balance`
Output: A tree decomposition created by expanding $\mathcal{T}$

Assign $\mathcal{T}' \leftarrow \mathcal{T}$
for ***each*** *bag* B'_i ***in*** $\mathcal{T}'$ **do**
    Assign $B'_i \leftarrow \bigcup_{B_j \in \mathsf{Nh}(B_i) \cup \{B_i\}} B_j$
end
return $\mathcal{T}'$

Algorithm 8: createBalancedTree

Input: An unbalanced tree decomposition T and a positive integer $\lambda > 1$
Output: A balanced tree decomposition T'

Assign $T' \leftarrow$ `Balance`(T)
Assign $T' \leftarrow$ `expandRankTree`(T')
return T'

B Example of Use

In this section we use the function `threeNPlusOne` from Section 3.1 as an example to demonstrate the use JTDec. We consider that the input method (function) is obtained as a `SootMethod` and is called `method`. For instructions on how to do this, consult Soot documentation.

All the main methods in JTDec return their outputs as a `JTDecTree`. Each instance of the class `JTDecTree` corresponds to a tree-decomposition and has methods for accessing and manipulating it. Some of these will be addressed in more details in the following example. In each of these tree decompositions, both the bags (vertices of the tree decomposition) and the nodes (of the control flow graph) are numbered and identified by integers. The only requirement is that these integers should be distinct, but our methods create consecutive numbers starting with 1. Also, if a `JTDecTree` is passed as an argument to one of the methods in JTDec, then it must be ensured that the tree is connected, otherwise exceptions and undefined behavior can arise. Note that `JTDecTree` allows users to make local changes to the tree-decomposition (see the comments in the class itself for more information). If such changes are made, then the global variables like width and depth are not updated (due to the high computational cost associated with it). To update these variables, `JTDecTree.traverseTree()` needs to be invoked. In the following example we do not have to care about this since all our `JTDecTree`s are created by JTDec itself.

We begin our example by creating a tree-decomposition of the control flow graph `method`. Recall that `method` is of type `SootMethod`. The following line creates a tree-decomposition named `treeDecomposition`:

```
JTDecTree treeDecomposition = JTDec.createTreeDec(method);
```

There is also an option of passing a `boolean` value to the methods in `JTDec`, to flag that a log should be written in `stderr`. In this case the code to run is:

```
JTDecTree treeDecomposition = JTDec.createTreeDec(method, true);
```

In this case the log in `stderr` is:

```
[JTDec] [createTreeDec] Obtaining CFG of the method
[JTDec] [createTreeDec] CFG obtained
[JTDec] [createTreeDec] The lines were numbered as follows:
[JTDec] [createTreeDec] 1:    n := @parameter0: int
[JTDec] [createTreeDec] 2:    nop
[JTDec] [createTreeDec] 3:    if n > 1 goto nop
[JTDec] [createTreeDec] 4:    goto [?= nop]
[JTDec] [createTreeDec] 5:    nop
[JTDec] [createTreeDec] 6:    temp$0 = n % 2
[JTDec] [createTreeDec] 7:    if temp$0 == 0 goto nop
[JTDec] [createTreeDec] 8:    goto [?= nop]
[JTDec] [createTreeDec] 9:    nop
[JTDec] [createTreeDec] 10:    temp$1 = n
[JTDec] [createTreeDec] 11:    temp$2 = temp$1 / 2
[JTDec] [createTreeDec] 12:    n = temp$2
[JTDec] [createTreeDec] 13:    goto [?= nop]
[JTDec] [createTreeDec] 14:    nop
[JTDec] [createTreeDec] 15:    temp$3 = 3 * n
[JTDec] [createTreeDec] 16:    temp$4 = temp$3
[JTDec] [createTreeDec] 17:    temp$5 = temp$4 + 1
[JTDec] [createTreeDec] 18:    n = temp$5
[JTDec] [createTreeDec] 19:    nop
[JTDec] [createTreeDec] 20:    goto [?= nop]
[JTDec] [createTreeDec] 21:    nop
[JTDec] [createTreeDec] 22:    return
[JTDec] [createTreeDec] Finding successors of each node
[JTDec] [createTreeDec] Creating a list of neighbors of each node
[JTDec] [createTreeDec] successors are: {1=[2], 2=[3], 3=[4, 5], 4=[21],
 5=[6], 6=[7], 7=[8, 9], 8=[14], 9=[10], 10=[11], 11=[12], 12=[13],
  13=[19], 14=[15], 15=[16], 17=[18], 16=[17], 19=[20], 18=[19], 21=[22],
   20=[2], 22=[]}
[JTDec] [createTreeDec] neighbors are:{1=[2], 2=[1, 3, 20], 3=[2, 4, 5],
 4=[3, 21], 5=[3, 6], 6=[5, 7], 7=[6, 8, 9], 8=[7, 14], 9=[7, 10],
 10=[9, 11], 11=[10, 12], 12=[11, 13], 13=[19, 12], 14=[8, 15], 15=[16, 14],
 17=[16, 18], 16=[17, 15], 19=[18, 20, 13], 18=[17, 19], 21=[4, 22],
  20=[2, 19], 22=[21]}
[JTDec] [createTreeDec] J is:[(18, 19), (16, 17), (14, 15), (12, 13),
 (2, 3), (6, 7), (10, 11), (21, 22), (17, 18), (1, 2), (5, 6), (9, 10),
  (3, 5), (8, 14), (19, 20), (11, 12), (3, 4), (7, 9), (7, 8), (4, 21),
   (20, 2), (15, 16), (13, 19)]
[JTDec] [createTreeDec] S is:[(19, 18), (18, 19), (17, 16), (16, 17),
 (15, 14), (14, 15), (13, 12), (12, 13), (3, 2), (2, 3), (6, 7), (7, 6),
  (10, 11), (11, 10), (22, 21), (21, 22), (18, 17), (17, 18), (6, 5),
   (5, 6), (2, 1), (1, 2), (9, 10), (10, 9), (5, 3), (3, 5), (8, 14),
    (14, 8), (20, 19), (19, 20), (12, 11), (11, 12), (4, 3), (3, 4),
     (7, 9), (9, 7), (7, 8), (8, 7), (21, 4), (4, 21), (2, 20), (20, 2),
      (16, 15), (15, 16), (19, 13), (13, 19)]
[JTDec] [createTreeDec] Creating the listing
```

```
[JTDec] [createTreeDec] Created the following listing:
[JTDec] [createTreeDec] [22, 21, 2, 3, 20, 19, 7, 18, 17, 16, 15, 14, 13,
 12, 11, 10, 9, 8, 6, 5, 4, 1]
[JTDec] [createTreeDec] Finding separators
[JTDec] [createTreeDec] Separators are:
[JTDec] [createTreeDec] {1=[2], 2=[21], 3=[2, 21], 4=[3, 21], 5=[3, 6],
 6=[3, 7], 7=[19, 3], 8=[7, 14], 9=[7, 10], 10=[7, 11], 11=[7, 12],
 12=[7, 13], 13=[19, 7], 14=[7, 15], 15=[16, 7], 17=[18, 7], 16=[17, 7],
  19=[3, 20], 18=[19, 7], 21=[22], 20=[2, 3], 22=[]}
[JTDec] [createTreeDec] Nodes of the tree decomposition are: {1=[22],
 2=[21, 22], 3=[2, 21], 4=[2, 3, 21], 5=[2, 3, 20], 6=[19, 3, 20],
  7=[19, 3, 7], 8=[19, 18, 7], 9=[17, 18, 7], 10=[17, 16, 7], 11=[16,
   7, 15], 12=[7, 14, 15], 13=[19, 7, 13], 14=[7, 12, 13],
   15=[7, 11, 12], 17=[7, 9, 10], 16=[7, 10, 11], 19=[3, 6, 7],
    18=[7, 8, 14], 21=[3, 4, 21], 20=[3, 5, 6], 22=[1, 2]}
[JTDec] [createTreeDec] Edges of the tree decomposition are:[(16, 17),
 (14, 15), (2, 3), (4, 5), (6, 7), (8, 9), (10, 11), (13, 14), (1, 2),
  (5, 6), (9, 10), (19, 20), (3, 4), (11, 12), (7, 13), (7, 8), (4, 21),
   (3, 22), (7, 19), (15, 16), (12, 18)]
[JTDec] [createTreeDec] Created this JTDecTree:
------JTDecTree------
width: 2
numberOfVertices: 22
root: 1
height: 12
parents: {1=1, 2=1, 3=2, 4=3, 5=4, 6=5, 7=6, 8=7, 9=8, 10=9, 11=10, 12=11,
 13=7, 14=13, 15=14, 17=16, 16=15, 19=7, 18=12, 21=4, 20=19, 22=3}
children: {1=[1, 2], 2=[3], 3=[4, 22], 4=[21, 5], 5=[6], 6=[7],
7=[19, 8, 13], 8=[9], 9=[10], 10=[11], 11=[12], 12=[18], 13=[14], 14=[15],
 15=[16], 17=[], 16=[17], 19=[20], 18=[], 21=[], 20=[], 22=[]}
bags: {1=[22], 2=[21, 22], 3=[2, 21], 4=[2, 3, 21], 5=[2, 3, 20],
 6=[19, 3, 20], 7=[19, 3, 7], 8=[19, 18, 7], 9=[17, 18, 7], 10=[17, 16, 7],
  11=[16, 7, 15], 12=[7, 14, 15], 13=[19, 7, 13], 14=[7, 12, 13],
   15=[7, 11, 12], 17=[7, 9, 10], 16=[7, 10, 11], 19=[3, 6, 7], 18=[7,
    8, 14], 21=[3, 21, 4], 20=[3, 5, 6], 22=[1, 2]}
rootBags: {1=22, 2=3, 3=4, 4=21, 5=20, 6=19, 7=7, 8=18, 9=17, 10=16,
 11=15, 12=14, 13=13, 14=12, 15=11, 17=9, 16=10, 19=6, 18=8, 21=2, 20=5,
  22=1}
levels: {1=0, 2=1, 3=2, 4=3, 5=4, 6=5, 7=6, 8=7, 9=8, 10=9, 11=10, 12=11,
 13=7, 14=8, 15=9, 17=11, 16=10, 19=7, 18=12, 21=4, 20=8, 22=3}
---End of JTDecTree---
[JTDec] [createTreeDec] Process finished
```

We now go through this log and explain each part. As mentioned earlier, we identify each node of the CFG with an integer. At first the CFG is obtained using Soot and the used node numbering is printed. This is in the same order as obtained by iterating over the control flow graph in its `BriefUnitGraph` format and its default iterator. For readers unfamiliar with `BriefUnitGraph`, the function `JTDec.getUnitsOf` can be used to obtain the same numbering as a `HashMap<Integer, Unit>`. Here is an example code:

```
System.out.println(JTDec.getUnitsOf(method));
```

and its output:

```
{1=n := @parameter0: int, 2=nop, 3=if n > 1 goto nop, 4=goto [?= nop],
 5=nop, 6=temp$0 = n % 2, 7=if temp$0 == 0 goto nop, 8=goto [?= nop], 9=nop,
 10=temp$1 = n, 11=temp$2 = temp$1 / 2, 12=n = temp$2, 13=goto [?= nop],
 14=nop, 15=temp$3 = 3 * n, 17=temp$5 = temp$4 + 1, 16=temp$4 = temp$3,
 19=nop, 18=n = temp$5, 21=nop, 20=goto [?= nop], 22=return}
```

The log continues with S and J as in Algorithm 2 and then the listing obtained from Algorithm 1 and its separators which are found using Algorithm 3. Finally a tree decomposition is created by Algorithm 4 and is printed at the very end of the log. This can be done by simply printing the resulting `JTDecTree`, i.e., the code

```
JTDecTree treeDecomposition = JTDec.createTreeDec(method);
System.out.println(treeDecomposition);
```

outputs the following representation of the tree decomposition:

```
------JTDecTree------
width: 2
numberOfVertices: 22
root: 1
height: 12
parents: {1=1, 2=1, 3=2, 4=3, 5=4, 6=5, 7=6, 8=7, 9=8, 10=9, 11=10, 12=11,
 13=7, 14=13, 15=14, 17=16, 16=15, 19=7, 18=12, 21=4, 20=19, 22=3}

children: {1=[1, 2], 2=[3], 3=[4, 22], 4=[21, 5], 5=[6], 6=[7], 7=[19, 8,
 13], 8=[9], 9=[10], 10=[11], 11=[12], 12=[18], 13=[14], 14=[15], 15=[16],
  17=[], 16=[17], 19=[20], 18=[], 21=[], 20=[], 22=[]}

bags: {1=[22], 2=[21, 22], 3=[2, 21], 4=[2, 3, 21], 5=[2, 3, 20], 6=[19, 3,
 20], 7=[19, 3, 7], 8=[19, 18, 7], 9=[17, 18, 7], 10=[17, 16, 7], 11=[16, 7,
  15], 12=[7, 14, 15], 13=[19, 7, 13], 14=[7, 12, 13], 15=[7, 11, 12],
   17=[7, 9, 10], 16=[7, 10, 11], 19=[3, 6, 7], 18=[7, 8, 14], 21=[3, 21,
    4], 20=[3, 5, 6], 22=[1, 2]}

rootBags: {1=22, 2=3, 3=4, 4=21, 5=20, 6=19, 7=7, 8=18, 9=17, 10=16, 11=15,
 12=14, 13=13, 14=12, 15=11, 17=9, 16=10, 19=6, 18=8, 21=2, 20=5, 22=1}

levels: {1=0, 2=1, 3=2, 4=3, 5=4, 6=5, 7=6, 8=7, 9=8, 10=9, 11=10, 12=11,
 13=7, 14=8, 15=9, 17=11, 16=10, 19=7, 18=12, 21=4, 20=8, 22=3}

---End of JTDecTree---
```

Here the tree is composed of 10 bags, is rooted at bag number 1, has a height of 7 and a width of 3, see Figure 3. After the global properties, parents and children of each bag are printed, e.g. parent of vertex 3 is 2 and children of vertex 2 are 3 and 4. Then the bag data is printed, e.g. bag 4 contains nodes 8 and 9. Then the root bag of each of the nodes is printed and finally the distance between each of the bags and the root of the tree are given. This data can be obtained by calling the respective functions in `JTDecTree` as well.

```
int width = treeDecomposition.getTreeWidth(); //returns 2
int height = treeDecomposition.getHeight(); //returns 12
```

```
int root = treeDecomposition.getRoot(); //returns 1
int x = treeDecomposition.getParent(3); //returns 2
Set<Integer> y;
y = treeDecomposition.getChildren(2); //returns [3]
y = treeDecomposition.getBag(4); //returns [2, 3, 21]
int l = treeDecomposition.getLevel(5); //returns 4
int rb = treeDecomposition.getRootBag(4); //returns 21
int d = treeDecomposition.getDegree(2); //returns 2
```

This tree-decomposition can be normalized by:

```
JTDecTree normalTD = JTDec.normalizeTreeDec(treeDecomposition);
System.out.println(normalTD);
```

In this case the original tree happens to be normal and does not change. Now we can balance this tree-decomposition using `JTDec.createBalancedTree`:

```
JTDecTree balancedTwo = JTDec.createBalancedTree(treeDecomposition,
    2);
JTDecTree balancedThree = JTDec.createBalancedTree(
   treeDecomposition, 3);
System.out.println(balancedTwo);
System.out.println(balancedThree);
```

This balances the tree with $\lambda = 2, 3$ and prints the resulting balanced tree-decompositions (Figures 3, 4 and 5). Here is the output:

```
------JTDecTree------
width: 7
numberOfVertices: 16
root: 1
height: 4
parents: {1=1, 2=1, 3=1, 4=3, 5=4, 6=4, 7=1, 8=7, 9=8, 10=8, 11=1, 12=11,
 13=12, 14=13, 15=12, 16=12}
children: {1=[1, 2, 3, 7, 11], 2=[], 3=[4], 4=[5, 6], 5=[], 6=[], 7=[8],
 8=[9, 10], 9=[], 10=[], 11=[12], 12=[16, 13, 15], 13=[14], 14=[], 15=[],
  16=[]}
bags: {1=[19, 18, 3, 20, 5, 6, 7, 13], 2=[19, 3, 5, 6, 7], 3=[16, 19, 3, 18,
 7, 15], 4=[17, 16, 19, 18, 7, 8, 14, 15], 5=[16, 7, 8, 14, 15], 6=[17, 16,
  18, 7, 15], 7=[19, 3, 7, 10, 11, 13], 8=[19, 7, 9, 10, 11, 12, 13], 9=[7,
   9, 10, 11], 10=[7, 10, 11, 12, 13], 11=[2, 19, 3, 21, 20, 7], 12=[1, 19,
    2, 3, 21, 20, 22], 13=[2, 3, 4, 21, 20], 14=[2, 3, 4, 21, 20], 15=[1, 2,
     21], 16=[2, 21, 22]}
rootBags: {1=12, 2=11, 3=1, 4=13, 5=1, 6=1, 7=1, 8=4, 9=8, 10=7, 11=7, 12=8,
 13=1, 14=4, 15=3, 17=4, 16=3, 19=1, 18=1, 21=11, 20=1, 22=12}
levels: {1=0, 2=1, 3=1, 4=2, 5=3, 6=3, 7=1, 8=2, 9=3, 10=3, 11=1, 12=2,
 13=3, 14=4, 15=3, 16=3}
---End of JTDecTree---
------JTDecTree------
width: 10
numberOfVertices: 15
root: 1
height: 3
```

```
parents: {1=1, 2=1, 3=1, 4=3, 5=3, 6=5, 7=1, 8=7, 9=7, 10=1, 11=10, 12=10,
 13=10, 14=13, 15=13}
children: {1=[1, 2, 3, 7, 10], 2=[], 3=[4, 5], 4=[], 5=[6], 6=[], 7=[8, 9],
 8=[], 9=[], 10=[11, 12, 13], 11=[], 12=[], 13=[14, 15], 14=[], 15=[]}
bags: {1=[16, 2, 19, 3, 21, 5, 6, 7, 11, 12, 15], 2=[19, 3, 5, 6, 7], 3=[17,
 16, 19, 3, 7, 8, 14, 15], 4=[16, 7, 8, 14, 15], 5=[17, 16, 19, 18, 7, 15],
  6=[17, 16, 19, 18, 7], 7=[19, 3, 7, 9, 10, 11, 12, 13], 8=[7, 9, 10, 11,
   12], 9=[19, 7, 11, 12, 13], 10=[19, 2, 3, 4, 21, 20, 7], 11=[2, 3, 4,
    21], 12=[19, 2, 3, 21, 20], 13=[1, 2, 3, 21, 22], 14=[1, 2, 21], 15=[2,
     21, 22]}
rootBags: {1=13, 2=1, 3=1, 4=10, 5=1, 6=1, 7=1, 8=3, 9=7, 10=7, 11=1, 12=1,
 13=7, 14=3, 15=1, 17=3, 16=1, 19=1, 18=5, 21=1, 20=10, 22=13}
levels: {1=0, 2=1, 3=1, 4=2, 5=2, 6=3, 7=1, 8=2, 9=2, 10=1, 11=2, 12=2,
 13=2, 14=3, 15=3}
---End of JTDecTree---
```

Finally, the function `JTDec.process` does all the steps above at once.

```
JTDecTree result = JTDec.process(method, 3);
```

This creates a tree decomposition of the CFG of `method`, normalizes it and then balances it with $\lambda = 3$ and returns the result. The `true` flag can be passed to enable log printing in `stderr`. It should also be noted that one can avoid specifying a λ in which case the default value of 2 will be used.

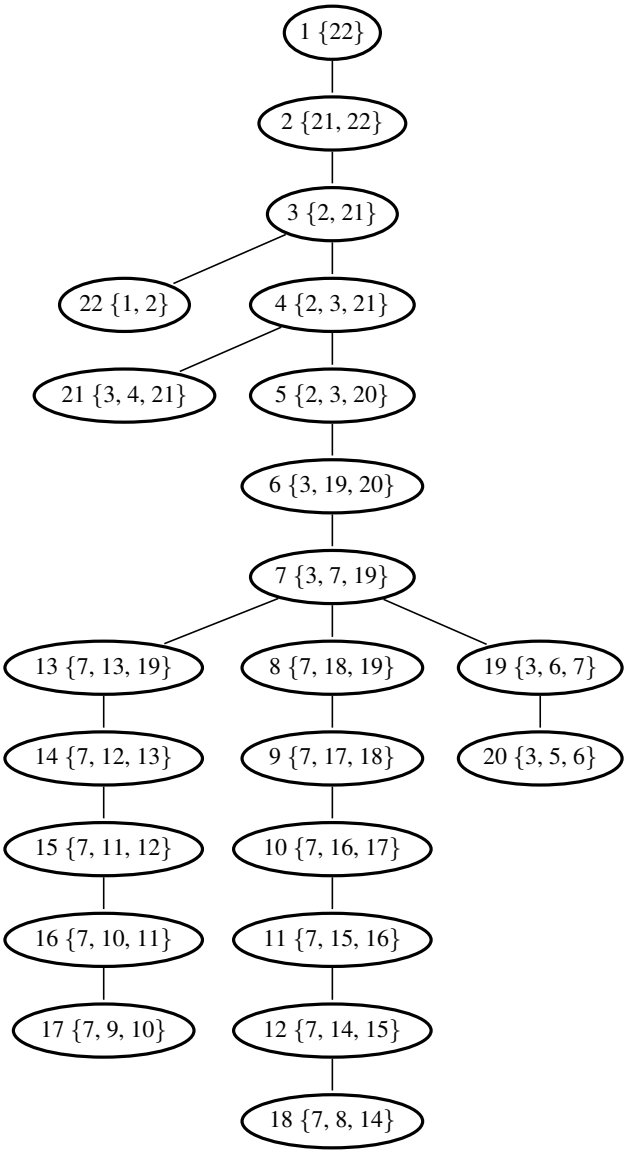


Fig. 3: The initial tree decomposition obtained from `createTreeDec`. Each bag has a number and contains a list of nodes of the CFG.

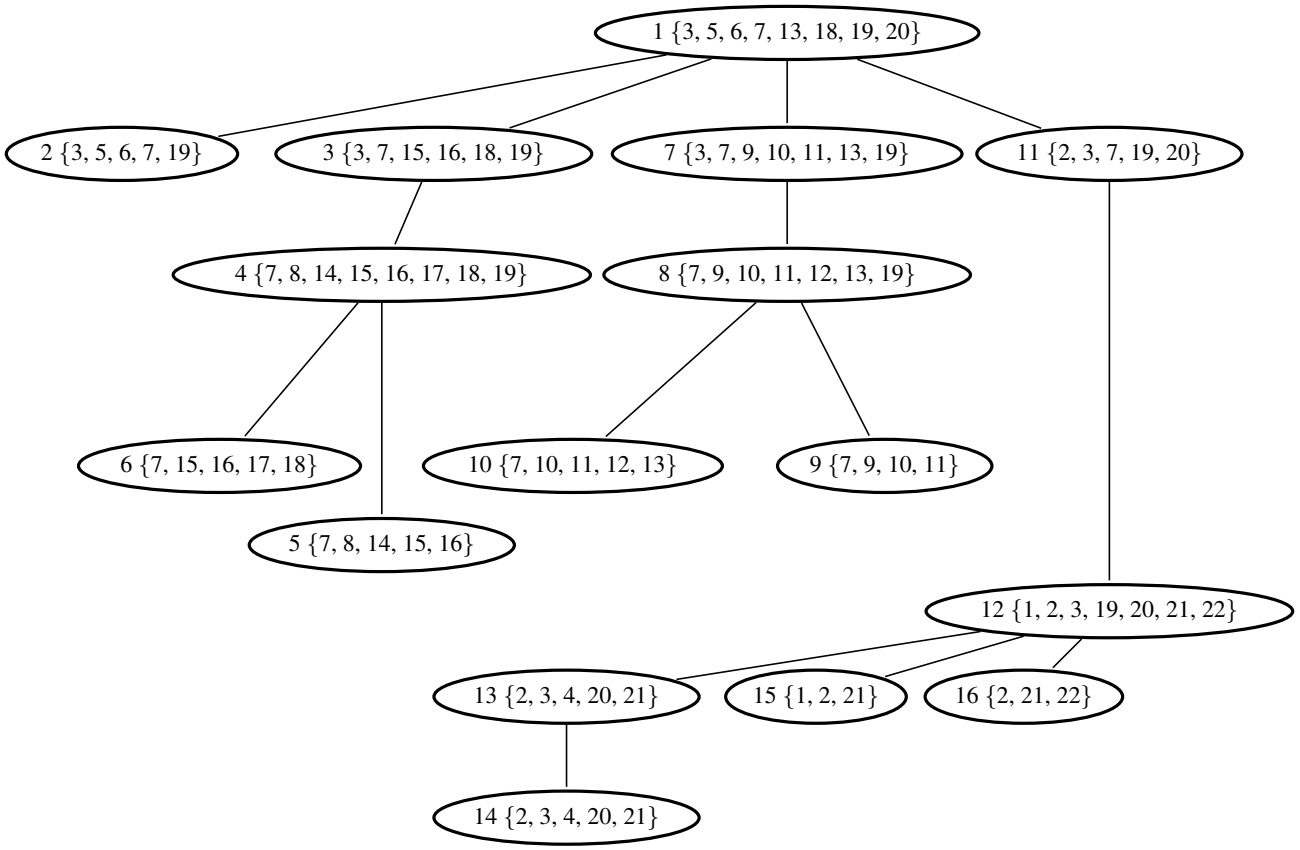


Fig. 4: Result of balancing the tree in Figure 3 with $\lambda = 2$.

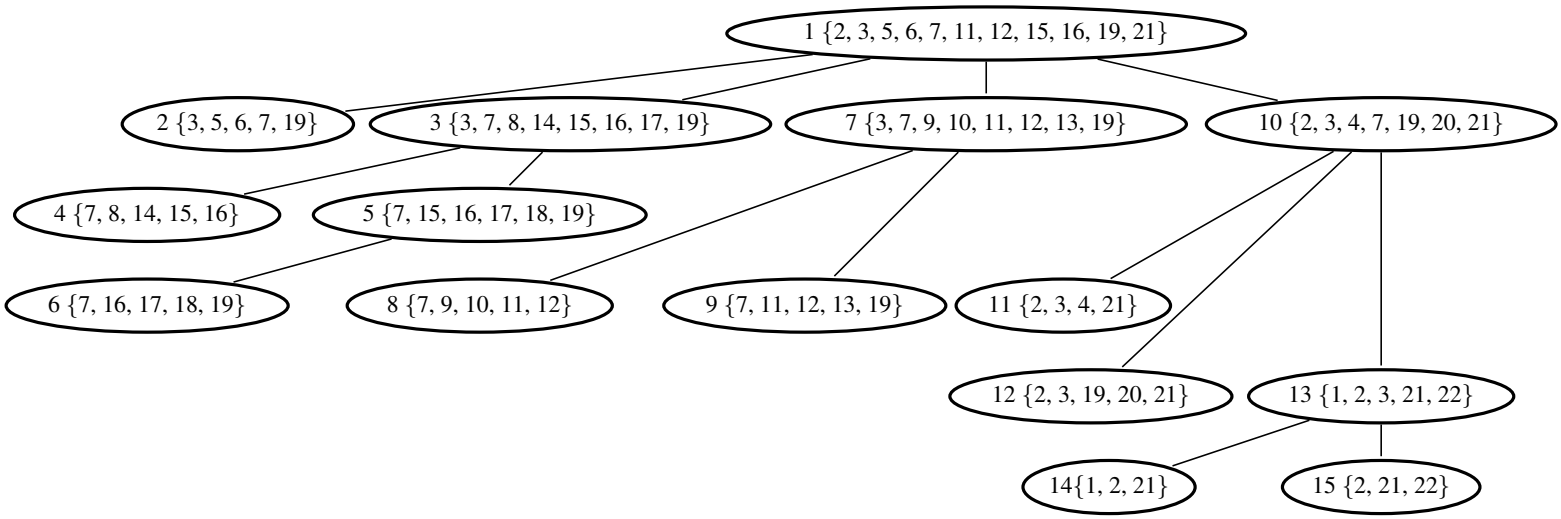


Fig. 5: Result of balancing the tree in Figure 3 with $\lambda = 3$.

C Running JTDec from the Command Line

JTDec packages come with three `jar` files as explained below:

- `JTDec-source.jar` contains the Java source files of JTDec.
- `JTDec-lib.jar` is the library file which can be imported in a java project to use JTDec. This file does not include soot, but depends on it.
- `JTDec.jar` is an executable jar that includes soot and can be used to obtain tree decompositions of Java methods.

In this section we briefly demonstrate how to run `JTDec.jar` from the command line environment.
`JTDec.jar` gets a set of java source or class files[1], runs soot over them and produces tree-decompositions of all the methods in the given classes.
It can be invoked as follows:

```
java -jar JTDec.jar [JTDec Parameters] [Soot Parameters]
```

`JTDec Parameters` consist of the following two:

- The first parameter can be either `true` or `false` and tells JTDec whether it should print a log to `stderr`.
- The second parameter is the number λ as explained in the previous section.

For example, in order to process two files `A.java` and `B.java` (located in the same folder as `JTDec.jar`), where the `main` function is in the former, and create tree decompositions of all the methods with $\lambda = 3$ one can execute:

```
java -jar JTDec.jar false 3 -cp . -pp A B -main-class A
```

[1] Jimple files are also supported.