# A heuristic approach to the treedepth decomposition problem for large graphs

Sylwester Swat$^{[0000-0001-8763-0045]}$ and Marta Kasprzak$^{[0000-0002-9863-5412]}$

Poznan University of Technology, Institute of Computing Science,
Piotrowo 2, 60-965 Poznan, Poland.
`sylwester.swat@put.poznan.pl` `marta.kasprzak@put.poznan.pl`

**Abstract.** In this article, we describe algorithms and techniques used in the method ExTREEm for the treedepth decomposition problem. ExTREEm won the heuristic track of the 5th Parameterized Algorithms and Computational Experiments Challenge (PACE 2020). It searches for a minimum-height treedepth decomposition of a graph via computing graph separators. Among concepts that are incorporated into the approach, we can distinguish a new objective function for evaluating separators, preprocessing based on finding treedepth decompositions in cactus subgraphs and on identification of graphlets, five algorithms for finding separators, a separator minimization method for a refinement of found separators, and a refinement of an obtained treedepth decomposition by merging techniques of tree rotations. This approach enables us to quickly obtain low-depth decompositions of very large graphs.

**Keywords:** Treedepth decomposition · Elimination tree · Separator.

## 1 Introduction

In this paper, we provide a description of algorithms of the method ExTREEm, which is a heuristic approach to the treedepth decomposition problem. The goal of the problem is to find a treedepth decomposition for a given graph with a height as small as possible. A treedepth decomposition of a connected graph $G = (V, E)$ is a rooted tree $T = (V, E_T)$ such that every edge of $G$ connects a pair of nodes that have an ancestor-descendant relationship in $T$. A treedepth of a connected graph is a minimum possible height of its treedepth decomposition. There are many equivalent notions to treedepth. The most commonly used ones include the notion of elimination tree of a graph and corresponding elimination height [14], ordered coloring, vertex ranking [10] and centered coloring [13].

There are many fields where the treedepth decomposition is applicable. One of them is parallel factorization of sparse matrices using the Cholesky factorization method [7]. Elimination trees are also used in routing algorithms such as the Customizable Contraction Hierarchies algorithm [5], where finding good balanced separators and nested dissection orders is of utmost practical importance, especially when operating on graphs with millions of vertices. Such routing algorithms are used in many navigating systems, as well as in the field of computer games, where shortest paths in a given map graph need to be computed many times each second. The treedepth notion is also relevant for theoretical reasons, e.g. in the design of fixed-parameter-tractable algorithms [8].

It was shown that the decision variant of the treedepth problem is NP-complete. There are classes of graphs for which the treedepth problem is solvable in polynomial time; e.g., it is possible for trees [12] and interval graphs [1]. In general, the construction of a minimum height elimination tree for a large graph in reasonable time seems to be very unlikely. Therefore, instead of exact algorithms, heuristics are used to create good decompositions. ExTREEm was designed for such optimization variant of the problem. It enables us to quickly find good treedepth decompositions even for very large graphs.

## 2 Preliminaries

Before we proceed to the description of algorithms, let us fix some natural notations and definitions. For a given connected graph $G = (V, E)$, we denote by $T(G)$ its treedepth decomposition, by $h(T)$ we denote height of tree $T$, and by $root(T)$ we denote the root of $T$. We denote by $G \setminus S$ an induced graph $G[V \setminus S]$, and by $C(G, S)$ the set of connected components of $G \setminus S$. For $a \in V$ we define $N(a) = \{v \in V : \{a, v\} \in E\}$, and for $A \subset V$ we take $N(A) = \bigcup_{v \in A} N(v)$. To indicate that a neighborhood is considered in graph $H$ (and not in $G$), we use notations $N_H(a)$ and $N_H(A)$, respectively. We denote by $size_n(G)$ (or $|V|$) the number of nodes in $G$ and by $size_e(G)$ (or $|E|$) the number of edges in $G$. For $a, b \in V$ we denote by $d(a, b)$ the distance between nodes $a$ and $b$. For sets $A, B \subset V$ we denote by $d(A, B)$ the distance between sets $A$ and $B$, $d(A, B) = \min\{d(a, b) : a \in A, b \in B\}$. A subset $S \subset V$ such that $G \setminus S$ is disconnected is called a *separator*. If additionally, for every $C \in C(G, S)$, the condition $size(C) \leq b \cdot size(G)$ holds, where $size(C)$ is either $size_n(C)$ or $size_e(C)$, then we say that separator $S$ is *b-balanced*. By *balanced* we mean $b$-balanced, where $b$ is a fixed parameter. Given two sets $A, B \subset V$ we denote $G_{A,B} = (A \cup B, \{\{a, b\} \subset E : a \in A, b \in B\})$.

## 3 Algorithms

In ExTREEm we search for a treedepth decomposition using a nested dissection approach[1]. The algorithm works in iterations, each iteration is executed independently of the others and with modified parameters. We refer to those iterations as *main iterations*. In each main iteration we apply some preprocessing to a given graph $G$. Then, we use a set of different heuristics to create a set of separators of $G$. Each of five best (according to a certain criterion) candidates is further refined. After selecting the best one after the refinement, we recursively obtain treedepth decompositions for components in $C(G, S)$. Finally, we merge

[1] ExTREEm is available at `https://doi.org/10.5281/zenodo.3873126`

separator $S$ and the decompositions we found into an elimination tree $T(G)$. At the end we apply some additional improvements to $T(G)$, trying to further minimize height of the treedepth decomposition.

## 3.1 Separator evaluation

There are a few commonly used objective functions used to evaluate separators (see e.g. [9], [3]). We propose a new approach to evaluate separators, based on the estimated height of the whole elimination tree.

To assess the quality of a separator $S$, we need to know values $mn(G,S)$ and $me(G,S)$ denoting, respectively, the maximum number of nodes and the maximum number of edges of a graph from $C(G,S)$. Now, let us define

$$score_n(S) = |S| \cdot \frac{1-\beta^{\lceil -\frac{\log|V|}{\log\beta}\rceil}}{1-\beta}, \quad \text{where } \beta = \frac{mn(G,S)}{|V|}.$$

We analogously define $score_e(S)$ by taking $\beta = \frac{me(G,S)}{|E|}$. Objectives $score_n$ and $score_e$ are used to quickly estimate a total height of the elimination tree, assuming that all subgraphs considered in recursive calls will have roughly the same ratio $\frac{|S|}{size_n(G)}$, respectively $\frac{|S|}{size_e(G)}$. Now, we can define the final objective function used to evaluate the quality of separators:

$$score(S) = \theta \cdot score_n(S) + (1-\theta) \cdot score_e(S),$$

where $\theta \in [0,1]$ is a parameter.

For given two balanced ($b$-balanced) or two unbalanced separators $S_1$ and $S_2$, we say that $S_1$ is better than $S_2$ if $score(S_1) < score(S_2)$. Additionally, a balanced separator is always better than an unbalanced one. The balance parameter $b$ is modified in every main iteration. Greater values are used to enable the objective *score* to find tiny separators that disjoin relatively small subgraphs from the rest of the graph (as it happens, e.g., in road graphs, see [9]), whereas smaller values are used mainly to find separators with better balance at the topmost levels of the decomposition.

More details about objective functions $score_n(S)$ and $score_e(S)$ can be found in Appendix A.

## 3.2 Preprocessing

The preprocessing phase works in two steps. The first step consists in detecting some cactus-subgraphs of a given graph $G$. This is achieved by repetitively performing a vertex contraction operation (see [5]) on a node of degree at most 2, unless its neighbors are already connected by an edge. For each such cactus component, we find a treedepth decomposition using recursively the Articulation Point Separator Creator method (see 3.3). By $G_1$ we denote graph $G$ after the first preprocessing step.

The second step of the preprocessing has two substeps. In the first one we find an independent set $I_3$ of an induced graph $G_1[\{v \in V : deg(v) = 3\}]$ and perform vertex contraction on each $v \in I_3$. In the second substep, we find a maximum independent set $I_4$ of nodes that are "center nodes" of induced subgraphs of $G_1$ isomorphic to some graphlet from the set $\{G_i : i \in \{22, 24, 26, 27, 28, 29\}\}$ (according to the numeration from Fig.2 of [19]). All nodes in the set $I_4$ have degree 4 and the neighborhood of each of those nodes induces a connected subgraph. We proceed with set $I_4$ in the same way as with $I_3$. By $G_2$ we denote graph $G$ after the second preprocessing step. After finding recursively a decomposition of $G_2$, each node $v \in I_4$ is attached to the deepest of its neighbors (in $G_1$), then we analogously proceed with set $I_3$.

Let us now examine how the preprocessing influences the height of a final treedepth decomposition $T(G)$. For each removed cactus subgraph, we attach the root of its corresponding decomposition to the lowest of its neighbors in $G$. This way, for each cactus $C$, the value $h(T(G_1))$ after attaching $T(C)$ increases by exactly $\max\{0, h(T(C)) - h(T(G_1)) + \max\limits_{v \in N(C) \setminus C} d_T(v, root(G_1))\}$. Since the treedepth decomposition of a cactus graph is of size $O(\log |V|)$, for most cactus subgraphs it simply does not cause any increase. Attaching nodes from $I_3$ can increase $h(T(G_2))$ by at most one. The same holds for $I_4$. Hence, we have $h(T(G_1)) \leq h(T(G_2)) + 2$.

Finding decompositions of detected cacti is done in $O(|V| \cdot \log |V|$ time, while the second preprocessing step works in time $O(|E| + |V| \cdot \log |V|)$. The overall complexity of the preprocessing phase is $O(|E| + |V| \cdot \log |V|)$.

## 3.3 Separator creation

After the preprocessing, we generate a set of separator candidates using several heuristics. From that set we select five best ones (with respect to their value of the objective *score*) that are further subjected to a refinement process called *minimization*. As a final separator we take the best one after the minimization.

**Articulation Point Separator Creator** In this method, we find separators that contain only articulation points (cut vertices) of the graph $G$. At the beginning, we find a set $A$ of articulation points of $G$ using Tarjan's algorithm [4] for finding biconnected components. We want to find, for each articulation point $v \in A$, values $mn(G, \{v\})$ and $me(G, \{v\})$. To do this, during the depth-first search we additionally keep track of the number of visited nodes and edges. Whenever we are processing node $v$ and backtracking from node $u$ such that $u$ and $v$ do not belong to the same biconnected component, we are able to count the number of nodes and edges in the component $C \in C(G, \{v\})$ that contains $u$. Hence, for each $a \in A$, we can find values $size_n$ and $size_e$ for every component in $C(G, \{a\})$. It is now easy to obtain values $mn(G, \{v\})$ and $me(G, \{v\})$.

This method of creating separators is used mainly during the preprocessing phase, where its complexity is $O(|V| \cdot \log |V|)$ for creating a treedepth decomposition for each of the processed cactus-subgraphs.

**BFS Separator Creator** Here, we create separators basing on the known fact that set $D_l(B) = \{v \in V : d(v,u) = l, u \in B\}$, for a given set $B \subset V$, is a separator in graph $G$. We propose a modification to this algorithm that makes it useful in practice, especially in the context of the BFS Separator Minimizer (see section 3.4), where the algorithm's running time is crucial.

Given a set $B \subset V$, we run the standard breadth-first search with source-nodes set $B$. Let $L = \max_{v \in V \setminus B} d(B, \{v\})$ and let $G_i = G[V \setminus (\bigcup_{j=0}^{i-1} D_j(B))]$, for $1 \leq i \leq L$. We divide nodes in $D_i(B)$ into blocks, two nodes belong to the same block if they belong to the same connected component of $G_i$. Now, for each such block $X$ we find the minimum vertex cover of a graph $G_{X, D_{i+1} \cap N_{G_i}(X)}$ using Kőnig's theorem and algorithm of Hopcroft and Karp for finding maximum matching in a bipartite graph (see [15]). In order to perform the whole procedure quickly, we process sets $D_i(B)$ in the reverse order (from $L$ to 1), dynamically keeping track of the $size_n$ and $size_e$ values of components in graph $G_i$. We consider all found vertex covers and all blocks as different separator candidates.

We run the described procedure multiple times, for small random subsets $B \subset V$. During each iteration we solve multiple instances of finding a vertex cover of a bipartite graph. By observing that each edge can occur in at most one such bipartite graph, we obtain the bound $O(|E| \cdot |V|^{\frac{1}{2}})$ on the running time.

**Component Expansion Separator Creator** The method described in this subsection is an improvement of another known approach. It consists in fixing some initial set of nodes $B$, then iteratively expanding set $B$ by adding to it a node from $N(B) \setminus B$. We select each time a node with the tightest connection to $B$. In case of a tie, a node with fewer neighbors outside $B$ is preferred. We store a sequence $ord = (v_0, v_1, \dots v_n)$ of nodes added to $B$. We call that sequence an *expansion order*. Creating an expansion order for a given initial set $B$ works in time $O(|E| \cdot \log |V|)$, as the information about node candidates is updated using a binary heap.

Let us denote $P_i = \bigcup_{j=0}^{i} \{v_i\}$. For a given expansion order $(v_0, v_1, \dots v_n)$, we consider separators of the form $S_i = \{v_j \in P_i : |N(v_j) \setminus P_i| > 0\}$. In order to do this quickly, we process nodes from $ord$ in the reverse order and dynamically keep track of all necessary information required to calculate sizes of components in graphs $G \setminus P_i$. By processing the nodes from $ord$ in the original order, we find those information for components in graphs $G[P_i]$. We are therefore able to quickly find values $mn(G, S_i)$ and $me(G, S_i)$ for all $0 < i < n$.

It often happens that found separators are not minimal. To avoid those situations, we want to rearrange nodes in $ord$ in such a way that, when iterating over $i$ from 1 to $n-1$, if $|C(G, P_{i-1})| < |C(G, P_i)|$ then all nodes from smaller components will occur in $ord$ before nodes that are in larger components.

To obtain time complexity better than $O(|E| \cdot |V|)$, we create an auxiliary graph $H$. We initially set $H = (V, \emptyset)$, then we process nodes $v_i$ in the reverse order, dynamically keeping track of the number of nodes and edges in graphs $G \setminus$

$P_i$. For each $i$ we consider the set of representatives of connected components in $G \setminus P_i$ in which $v_i$ has a neighbor, then we sort those representatives by ascending order of the corresponding values $size_e$. Finally, considering representatives $r$ in this order, we add a directed edge $(v_i, v_j)$ to graph $H$, where $j > i$ is the smallest integer such that $v_j$ and $r$ belong to the same connected component in $G \setminus P_i$. After processing all nodes, we run a depth-first search on $H$, starting from $v_0$ and processing neighbors in the order of adding them to $H$. We obtain the rearranged sequence $ord$ by listing nodes in the order in which they were visited during the traversal. We observed that the use of optimized orders for generating separators $S_i$ almost always results in a huge decrease of the value $score(S_i)$.

To create the auxiliary graph $H$, we need to keep track of component sizes and their representatives. We additionally need, for each $0 < i < n$, to sort a set of designated representatives. Fortunately, for given index $i$ only one of found representatives can occur again for another index $j < i$, as the components are merged. Hence, creating a rearranged, optimized order takes time $O(|E| \cdot \log |V|)$. The overall running time thus remains $O(|E| \cdot \log |V|)$.

**FlowCutter Separator Creator** We use our own implementation of a slightly modified version of the FlowCutter algorithm (see [9]). At the beginning, we create a set $L$, by initializing it with a random node and iteratively adding to $L$ a random node $v$ that lies furthest to $L$. We stop adding nodes when $|L| = 50$.

As the initial source node and target node for the FlowCutter iteration we take a random pair of nodes from $L$. Additionally, we enable expansion of the larger of the source-reachable and target-reachable sets only if both grow to size at least $\frac{|V|}{10}$. When the final cut is found, we consider four different expansion orders based on the order of adding graph nodes to sources and targets. For each order we find separators using the Component Expansion Separator Creator. We also consider as a separator candidate a vertex cover of a bipartite graph $G_{X,Y}$, where $X$ and $Y$ denote final sets of sources and targets.

It is necessary to mention here, that we do not use FlowCutter Separator Creator if $score(S)$ is large for the the best separator found by other methods. In those cases graphs seem not to have balanced separators of small size and the algorithm's running time $O(c \cdot |E|)$, where $c$ is the size of the most balanced cut, is too expensive.

**Flow Separator Creator** We consider sets of the form $B = N(N(N(u)))$ and $E = N(N(N(v)))$, where $u, v$ are randomly selected nodes. We find a maximum set of node-disjoint paths that begin in $B$ and end in $E$. This is done by running a unit-flow algorithm with unit capacity constraints imposed on nodes . As a separator we consider the union of all paths and refine that separator using Greedy Minimizer (see section 3.4). It is worth noting that separators created using this method are much worse than those from FlowCutter Separator Creator, but this algorithm works pessimistically in time $O(|E| \cdot \min(|V|^{\frac{2}{3}}, |E|^{\frac{1}{2}}))$, has much smaller constant factor and works well in the context of Flow Minimizer (see 3.4), where sets $B$ and $E$ are not created for random $u$ and $v$.

## 3.4 Separator minimization

After creating separator candidates, we proceed to the refinement step. For each candidate $S$ we try to minimize the value of $score(S)$ using iteratively the following methods, most of which are based on a usage of separator creators with specific initial settings:

1. Vertex Cover Minimizer - in this minimization technique we find a vertex cover of a bipartite graph $G_{S,N(S)\cap C_d}$, where $C_d$ is a subset containing a minimal number of largest components of $C(G,S)$ whose total sum is at least $\frac{|V\setminus S|}{4}$. The algorithm works in time $O(|E|\cdot|V|^{\frac{1}{2}})$.
2. BFS Minimizer and Component Expansion Minimizer - we find separators using BFS Separator Creator and Component Expansion Separator Creator, respectively, with the initial source set containing all nodes from $S$.
3. Greedy Minimizer - we greedily remove nodes from $S$, each time selecting a node $v$ which minimizes $size_e(C)$, where $C$ is a component obtained by merging $v$ and its adjacent components from $C(G,S)$. It is done by operating on an auxiliary weighted bipartite graph with bipartition $(A,B)$, where set $A$ represents nodes in $S$, nodes in set $B$ represent connected components of $C(G,S)$, and weights represent the number of edges between corresponding node and component. By using a binary heap to quickly update size values and removing lazily nodes from the auxiliary graph when a node from $S$ is removed, we achieve running time $O(|E|+|S|^2\cdot\log|V|)$.
4. FlowCutter Minimizer and Flow Minimizer - we find separators using FlowCutter Separator Creator and Flow Separator Creator, respectively. In the first variant, as the initial set of sources we take any subset $B\subset C_d$ with size $|B|=\frac{|C_d|}{2}$ and with the property that there does not exist any $v\notin B$ with $d(\{v\},S)>d(B,S)$ ($C_d$ is taken in the same way as in Vertex Cover Minimizer). We analogously create the initial set of targets, but we take nodes from $V\setminus(S\cup C_d)$. In the second variant, we consider initial sets of the form $\{v\in X : d(\{v\},S)=t\}$, where $X$ is $C_d$ or $V\setminus(S\cup C_d)$, respectively, and $t$ is a small value (usually between 2 and 4). Algorithms work in time $O(|E|+|S|\cdot|V|)$, but with greatly reduced constant factor.

## 3.5 Attaching subtrees

After finding decompositions for all components in $C(G,S)=\{C_1,\ldots,C_k\}$, we need to merge the results to obtain $T(G)$. To do that, we sort all trees $T(C_i)$ by their depths in the nonincreasing order. Then, we create a sequence $S'$ (starting with $S'=\emptyset$) by iteratively adding to $S'$ nodes from $(S\cap N(C_i))\setminus S'$. As the tree $T(G)$ we initially take the tree (path) represented by sequence $S'$, with the root set to its first element. We attach each tree $T(C_i)$ to the last node from sequence $S'$ that occurs in $N(C_i)$. By using counting sort for sorting heights of trees and considering only edges with an end in $S$, the whole procedure works in time $O(|V|+M)$, where $M$ is the number of edges in a graph $G[S\cup N(S)]$.

### 3.6 Tree improvements

In the literature, there were proposed few techniques aiming at the reducing height of the elimination tree and basing on tree rotations. In this section we describe two structural improvements. The first one is a variation of a rotation-based algorithm described in [11] that enables us to implement it very easily. The second algorithm is, to the best of our knowledge, a new approach to the reduction of the height of an elimination tree via tree rotations.

Let us fix some additional notations. In this section, by $T$ we mean tree $T(G)$. For $v \in V$ we denote by $T_v$ the subtree of $T$ with root in $v$. By $depth_T(v)$ we denote the depth of node $v$ in tree $T$. By the block of $v \in V$ in a tree $T(G)$ we mean a maximal path in $T(G)$ which contains $v$, such that each node on that path, apart from the deepest one, has at most one son.

**Block pivots** Let us fix any block $B$ of tree $T(G)$ and let $S$ be a path from $root(T)$ to the topmost node in $B$. We consider $S$ as a separator in graph $G$. Treedepth decompositions $T_i = T(C_i)$ are already constructed for each component $C_i \in C(G, S)$, $1 \leq i \leq k$, they form connected components of $T \setminus S$. We now rearrange nodes in $S$ using algorithm 3.5 for separator $S$ and trees $T_i$. We repeat the algorithm for several blocks $B$ on the longest root-leaf path.

**Hall-set pivots** The algorithm based on block pivots always needs to rearrange order of a given, contiguous sequence of initial nodes on a root-leaf path in tree $T$. In the following method, we propose a rotation-based technique without that constraint, which works exceptionally well for graphs that do not contain balanced separators of small size.

Let us fix any block $B$ in $T(G)$ that lies on a longest root-leaf path $P$ and let $v$ be the topmost node in that block. Let $U(v)$ be a set of nodes on path $P$ from the root to the parent of $v$ and $D(v)$ be a set containing the remaining nodes on path $P$. We now consider a maximum matching $M$ in a bipartite graph $G_{U(v),N(U(v))\cap R}$, where $R = T_v \setminus D(v)$. If matching $M$ does not saturate set $U(v)$, then there exists in graph $G$ a set $H_M \subset U(v)$ with property $|H_M| > |N(H_M) \cap R|$. In our algorithm, from all sets $H_M$ violating Hall's condition for the existence of a matching saturating set $U(v)$, we select the one with maximum size. Set $H_M$ contains all nodes $u \in U(v)$ to which there exists in $G_{U(v),N(U(v))\cap R}$ an $M$-alternating path starting in an unsaturated node from $U(v)$. We now remove from tree $T_v$ all nodes belonging to the set $N(H_M)$. For each node $s$ with $par(s) \in N(H_M) \cap T_v$ we set $par(s)$ to the deepest ancestor of $s$ in $P \setminus (H_M \cup N(H_M))$. Let us now consider set $S = U(v) \cup N(H_M)$ and a set of treedepth decompositions $T(C_i)$ for $C_i \in T \setminus S$. Using algorithm 3.5, we obtain a new treedepth decomposition $T'(G)$.

In the transformation, we removed from path $P$ at least $|H_M|$ nodes and there are at most $|N(H_M) \cap R| < |H_M|$ new nodes in $T'$ that became ancestors of a node from $T_v$. It follows that for each node $w \in T_v$ we have $depth_{T'}(w) < depth_T(w)$. Let us mention here, that it not necessarily means that

$h(T') < h(T)$, as for some other node $l \in T$ we may have $depth_{T'}(l) \geq h(T)$. From our observations, however, the Hall-set pivot technique works very well on graphs for which the ratio $\frac{h(T(G))}{|V|}$ is large.

Running the algorithm for a single block $B$ is dominated by the factor of finding a maximum matching in a graph $G_{U(v),N(U(v))\cap R}$, which takes time $O(|E| \cdot |h(T)|^{\frac{1}{2}})$. The selection of the block $B$ is not unimportant. We therefore consider all blocks $B$ on path $P$ in a leaf-to-root manner. If, at some moment, for the $i$-th considered block $B_i$ we obtain a tree $T'$ with $h(T') < h(T)$, we terminate the algorithm and run it again for $T'$. If, however, valid set $H_M$ does not exist or $h(T') \geq h(T)$, we check the next block $B_{i+1}$ above $B_i$. We can now initialize next matching $M_{i+1}$ with all edges from previous matching $M_i$ that have an endpoint in $H_{M_i} \setminus B_{i+1}$. Such initialization resulted in a considerable performance improvement. It is also worth mentioning that when a node $p \in U$ becomes saturated in a matching $M_i$, then it becomes saturated in all further matchings $M_j$ $(j > i)$, until it leaves $U$. We therefore terminate processing blocks as soon as the whole set $U$ is saturated.

## 4 Results

A thorough comparison of ExTREEm and other methods was made within the heuristic track of the contest PACE 2020: 5th Parameterized Algorithms and Computational Experiments Challenge. There, 55 heuristic methods were submitted and tested on 200 instances differing in size and properties. ExTREEm won this contest. Here, we focus on a subset of these instances, large graphs. More information about the contest and short descriptions of several solvers can be found in [20].

**Other methods** To the comparison we selected four other best heuristics from the contest[2]. They are: FlowCutter [16] (2nd in the contest), Sallow [18] (3rd), Tweed-Plus [17] (4th), and Fluid [2] (5th).

FlowCutter is based on the previously mentioned algorithm of the same name from 2016 for finding separators of a graph [9]. Present version of FlowCutter is additionally supported by two approaches: iterative node contraction and label propagation. Sallow is another method that incorporates the ideas of FlowCutter from 2016, supplemented with greedy heuristics. The order in which vertices are processed in greedy heuristics is based on different criteria and is updated on the fly. Tweed-Plus creates and next improves an elimination tree with two known methods: nested dissection and the minimum-degree ordering algorithm. In each of its phases, if a graph is small enough, the computations are repeated many times with different results due to randomisation. Fluid realizes four separate strategies and selects as a result the best found solution. Two strategies iteratively select a vertex with a best score, two others iteratively search for separators and remove them. The score-based strategies compute an elimination

[2] https://pacechallenge.org/2020/results/#heuristic-track

order, later used in the tree construction, many times with randomness involved. The separator-based strategies use a greedy approach or the asynchronous fluid communities algorithm.

**Data set** From among public instances of PACE 2020 for the heuristic track[3], we chose graphs of more than 1000 vertices. They represent a wide range of graph kinds; the most important groups according to their origin are biological and social networks, road graphs and graphs generated according to different rules. These 66 instances include from 1013 to 1.32 million vertices and have the average vertex degree from 2 to 148, their presumable heights of the minimum treedepth decompositions vary from 3 to several thousands. All the instances are simple connected graphs.

**Comparison** For the purposes of the comparison, the smallest value of the criterion function obtained for a graph in the contest by any of the 55 heuristics is assumed as the optimum value for this graph. We partitioned the data set into groups from the point of view of a few parameters: graph order, optimum tree height, average vertex degree, fraction of vertices with degree at least three. This way it is easy to notice how compared methods deal with smaller groups of similar instances. The groups are presented in Table 1.

**Table 1.** Partition of the set of 66 instances into groups by different parameters. In a row, the cardinality of a subset of instances is followed by the parameter by which the subset has been determined and the range of its values.

| Group of instances | Cardinality | Parameter | Range of values |
|---|---|---|---|
| A | 22 | graph order | $\langle 1000; 7000)$ |
| B | 24 | | $\langle 7000; 50000)$ |
| C | 20 | | $\langle 50000; 1.32 \text{ mln})$ |
| D | 23 | optimum tree height | $\langle 1; 100)$ |
| E | 20 | | $\langle 100; 300)$ |
| F | 23 | | $\langle 300; 78500)$ |
| G | 24 | average vertex degree | $\langle 2; 3)$ |
| H | 22 | | $\langle 3; 8)$ |
| I | 20 | | $\langle 8; 150)$ |
| J | 21 | fraction of $V$ with degree $\geq 3$ | $\langle 13\%; 77\%)$ |
| K | 24 | | $\langle 77\%; 88\%)$ |
| L | 21 | | $\langle 88\%; 100\%\rangle$ |

During the PACE 2020 challenge, every heuristic had the limit of 30 minutes for returning a solution for an instance. Thus, the results are comparable in this

[3] https://pacechallenge.org/files/pace2020-heur-public.tgz

sense. Figure 1 presents the average quality of solutions obtained by the five best methods for the particular groups of instances and for all 66 instances. The quality is calculated as a ratio of the assumed optimum value of the criterion function to the value returned by a given method.

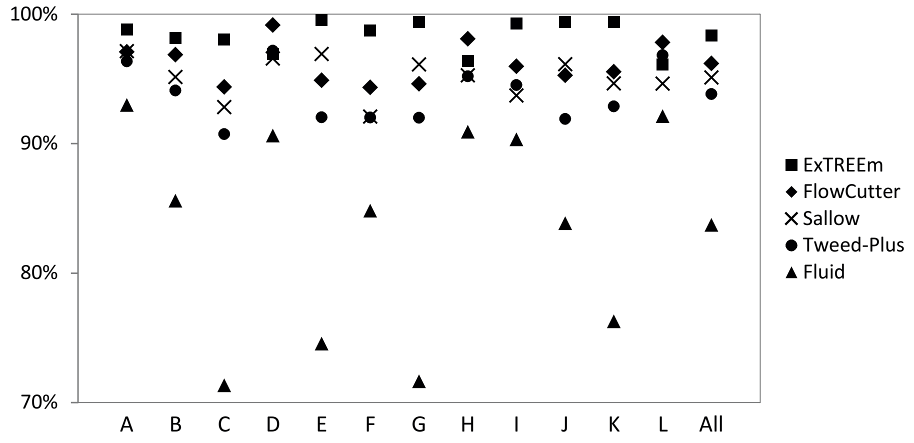

**Fig. 1.** The comparison of the PACE 2020 results for particular groups of large instances. Every marker stands for an average result of a given heuristic for the data set. Y axis shows how close to the optimum the results are.

ExTREEm generated solutions of the best quality (on average) for 9 out of 12 groups of instances A–L and for all large instances as well. For all the instances, ExTREEm achieved the ratio 98.34%, FlowCutter took the second rank with 96.18%, and Sallow was the next with 95.11%.The partitioning of the data set on the basis of graph order led to the same hierarchy. Groups D, H, and L were best solved by FlowCutter, ExTREEm took the second (H) or the third position (D, L). It means that a little harder for our method, in comparison to the others, were instances with a resulting tree of a small height or instances with a small fraction of vertices having only one or two neighbors.

## 5 Conclusions

We proposed the new method for the treedepth decomposition problem, which proved its efficiency in a wide comparison with top and current other algorithms. As ExTREEm very well solves large graphs, it may be useful in practical applications involving wide and complex networks, for example in industry of computer games or navigating systems. On the other hand, the method also deals well with smaller graphs. The public instances of PACE 2020 with less than 1000 vertices were solved by ExTREEm with the quality equal to 99%. Therefore, the applicability of ExTREEm is even wider.

## References

1. B. Aspvall and P. Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT Numerical Mathematics*, 34:484–509, 1994.
2. M. Bannach, S. Berndt, M. Schuster, and M. Wienobst. Solver description of Fluid. In *5th Parameterized Algorithms and Computational Experiments Challenge, PACE 2020*, doi:10.5281/zenodo.3871709, 2020.
3. N. Castillo-Garcia, H. Fraire-Huacuja, J. Flores, R. Rangel, J. Gonzalez Barbosa, J. Carpio Valadez. Comparative study on constructive heuristics for the vertex separation problem. *Studies in Computational Intelligence*, 601:465–474, 2015.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, MIT Press, 2nd ed., 2001.
5. J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. In Proc. of SEA 2014, Lecture Notes in Computer Science, 8504:271-282, 2014.
6. B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7:301–303, 1964. 364099.364331.
7. A. Gupta. Sparse Direct Methods. In *Encyclopedia of Parallel Computing*, D. Padua (eds), Springer, Boston, USA, pp. 1877–1886, 2011.
8. G. Gutin, M. Jones, and M. Wahlstrom. The mixed chinese postman problem parameterized by pathwidth and treedepth. *SIAM Journal on Discrete Mathematics*, 30:2177–2205, 2016.
9. M. Hamann and B. Strasser. Graph bisection with pareto-optimization. In *Proceedings of the Meeting on Algorithm Engineering and Experiments, ALENEX'16, Arlington, USA*, pp. 90–102, doi:10.1137/1.9781611974317.8, 2016.
10. I. Karpas, O. Neiman, and S. Smorodinsky. On vertex rankings of graphs and its relatives. *Discrete Mathematics* 338:1460–1467, 2015.
11. J. W. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.
12. F. Manne. An algorithm for computing an elimination tree of minimum height for a tree. Preprint at ResearchGate, 1998.
13. J. Nesetril and P. Ossona de Mendez. *Sparsity: Graphs, Structures, and Algorithms*, Springer, 2012.
14. J. Pieck. Formele definitie van een e-tree. Technische Hogeschool Eindhoven, memorandum 80-06, 1980.
15. A. Schrijver. A course in combinatorial optimization. Preprint at ResearchGate, 2003.
16. B. Strasser. FlowCutter. In *5th Parameterized Algorithms and Computational Experiments Challenge, PACE 2020*, https://github.com/ben-strasser/flow-cutter-pace20, 2020.
17. J. Trimble. Tweed: a heuristic solver for treedepth. In *5th Parameterized Algorithms and Computational Experiments Challenge, PACE 2020*, doi:10.5281/zenodo.3881441, 2020.
18. M. Wrochna. Sallow - a heuristic algorithm for treedepth decompositions. In *5th Parameterized Algorithms and Computational Experiments Challenge, PACE 2020*, preprint arXiv:2006.07050, doi: 10.5281/zenodo.3870565, 2020.
19. O. Yaveroglu, S. Fitzhugh, M. Kurant, A. Markopoulou, C. Butts, and N. Przulj. Ergm.graphlets: A package for ERG modeling based on graphlet statistics. *Journal of Statistical Software*, 65, 2014.
20. Proceedings of the 15th International Symposium on Parameterized and Exact Computation, IPEC 2020, Leibniz International Proceedings in Informatics 180, 2020.

## Appendix A

Let us define

$$score_n(S) = |S| \cdot \frac{1-\beta^{\lceil -\frac{\log|V|}{\log\beta}\rceil}}{1-\beta}, \quad \text{where } \beta = \frac{mn(G,S)}{|V|},$$
$$score_e(S) = |S| \cdot \frac{1-\gamma^{\lceil -\frac{\log|E|}{\log\gamma}\rceil}}{1-\gamma}, \quad \text{where } \gamma = \frac{me(G,S)}{|E|}.$$

We now show why the proposed objective functions $score_n(S)$ and $score_e(S)$ estimate the height of a treedepth decomposition. We specify only the case of $score_n(S)$, arguments for $score_e(S)$ are analogous.

Let $S$ be a separator of a graph $G = (V, E)$, $\alpha = \frac{|S|}{|V|}$, $\beta = \frac{mn(G,S)}{|V|}$, and let $EH(G, \alpha, \beta)$ be a function that estimates the height of a sought decomposition $T(G)$. It is calculated on the basis of values $\alpha$ and $\beta$, that is on information that we can obtain knowing only graph $G$ and separator $S$. For each $C \in C(G, S)$ the decomposition $T(C)$ will be attached to some node from the set $S \cap N(C)$ (see 3.5), therefore we use the following estimation:

$$EH(G,\alpha,\beta) \le |S| + \max_{C\in C(G,S)} h(T(C)) = \alpha\cdot|V| + \max_{C\in C(G,S)} h(T(C))$$

Assuming that in each recursive call the values of parameters $\alpha$ and $\beta$ do not change, we can replace $h(T(C))$ with $EH(C, \alpha, \beta)$ to obtain the following assessment:

$$\begin{aligned}
EH(G,\alpha,\beta) &\le \alpha\cdot|V| + \max_{C\in C(G,S)} h(T(C))\\
&\approx \alpha\cdot|V| + \max_{C\in C(G,S)} EH(C,\alpha,\beta)\\
&\approx \alpha\cdot|V| + \alpha\cdot\beta\cdot|V| + \max_{C'\in C(C,S')} EH(C',\alpha,\beta)\\
&\approx \alpha\cdot|V| + \alpha\cdot\beta\cdot|V| + \alpha\cdot\beta^2\cdot|V| + \dots\\
&\approx \sum_{i=0}^{\lceil \log_{\beta^{-1}}|V|\rceil} \alpha\cdot|V|\cdot\beta^i = \alpha\cdot|V|\cdot\sum_{i=0}^{\lceil -\frac{\log|V|}{\log\beta}\rceil} \beta^i\\
&\approx |S|\cdot\frac{1-\beta^{\lceil -\frac{\log|V|}{\log\beta}\rceil}}{1-\beta}
\end{aligned}$$

Let us note here that the formulas for the objective functions can be further simplified via the estimation $\beta^{\lceil \log_{\beta^{-1}}|V|\rceil} \approx \beta^{\log_{\beta^{-1}}|V|} = \frac{1}{|V|}$. We found, however, test cases where the replacement made a difference to the evaluation of separators.