# Calculating tree-depth
# Introduction

Dymitr Lubczyk Wojciech Replin Bartosz Różański

April 26, 2020

**Abstract**

We present several methods for computing treedepth on both CPU and GPU. We also set goals for the future of our project. Presented algorithm is a combination of well known dynamic algorithm and our own innovatory approach using branch and bound technique.

## Contents

# 1 Introduction

## 1.1 Tree-depth definition

For undirected graph $G$, the tree-depth $td(G)$ is a graph invariant. Intuitively, we may think of it as the parameter that says how similar $G$ is to a star. The lower tree-depth is, the more "star-like" $G$ is. Tree-depth value can range from 1 to $|G|$. To explain the most common tree-depth definition, it is necessary to introduce term tree-depth decomposition. Tree-depth decomposition of graph $G$ is a forest $F$ with the following property:

*If there is an edge $uv$ in $G$ then vertices $u$ and $v$ have ancestor-descendant relationship between each other in $F$.*

Using this definition, the tree-depth of $G$ is a depth of the forest with minimal depth among all tree-depth decompositions of $G$.

Tree-depth can also be defined in a recursive manner. It is worth mentioning, because we make strong use of it in our dynamic algorithm, presented later in this documentation. The definition looks as follows:

$$td(G) = \begin{cases} 1 & \text{if |G|=1} \\ 1 + \min_{v \in V} td(G - v) & \text{if G is connected} \\ \max_i td(G_i) & \text{otherwise} \end{cases} \tag{1}$$

## 1.2 Examples

In this paragraph we will provide some obvious facts and examples of tree-depth decompositions in order to give reader a better intuition.

The only connected graph which tree-depth equal to 1 is complete graph $K_1$
Stars are the only connected graphs with tree-depth equal to 2. For these graphs, graph and its best tree-depth decomposition are isomorphic.

Another thing is that for every graph $G$, a trivially valid tree-depth decomposition of $G$ is a path $P_{|G|}$ rooted in its beginning. It is true, because for such a tree every pair of vertices have ancestor-descendant relationship to each other.

The last fact, to point out is that if $G$ is connected then its tree-depth decomposition $F$ is also connected. It is true, because if $F$ had two components, then there would be two groups of vertices in $G$ without edges between each other, which is contradictory to $G$ being connected.

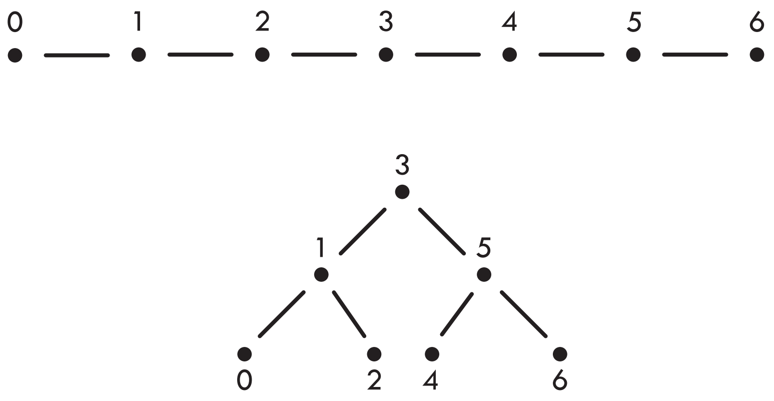

Figure 1: $P_7$ and its tree-depth decomposition

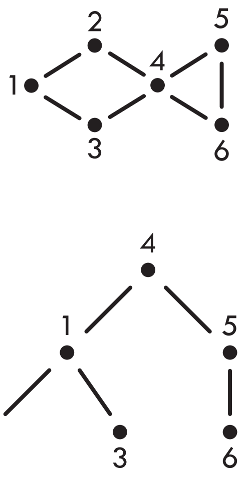

Figure 2: Fish and its tree-depth decomposition

## 2 Typical approaches

### 2.1 Exact

#### 2.1.1 Naïve

Recall the definition of treedepth:

$$td(G) = \begin{cases} 1, & \text{if } |G| = 1; \\ 1 + \min_{v \in V} td(G - v), & \text{if } G \text{ is connected and } |G| > 1; \\ \max_i td(G_i), & \text{otherwise;} \end{cases} \tag{1}$$

Let's say that we have permutation $\sigma$ of vertices of $G$. Let's approximate treedepth using this permutation:

$$td(G, \sigma, i) = \begin{cases} 1, & \text{if } |G| = 1; \\ td(G, \sigma, i + 1), & \text{if } \sigma(i) \notin V(G); \\ 1 + td(G - \sigma(i), \sigma, i + 1), & \text{if } \sigma(i) \in V(G) \text{ and } G \text{ is connected and } |G| > 1; \\ \max_j td(G_j, \sigma, i), & \text{otherwise;} \end{cases} \tag{2}$$

then our approximation would be $td(G, \sigma, 0)$. Let $\sigma_G$ be a permutation that satisfies $td(G) = td(G, \sigma_G, 0)$. Finding $\sigma_G$ takes $O(|G|!)$ time using brute force search. This permutation can be built using a standard definition of treedepth:

- When $|G| = 1$, return permutation of the only vertex of $G$.
- When $G$ is connected and $|G| > 1$, find $v$ such that minimizes $td(G - v)$ and store permutation $\sigma$ returned by $td(G - v)$. Return $v\sigma$.
- Lastly, let $\sigma_i$ be permutation returned by $td(G_i)$. Return $\sigma_0 ... \sigma_k$.

which also requires at least $O(|G|!)$ time.
Having found $\sigma_G$, we can reconstruct treedepth decomposition in $O(|G|)$ time. Method for such reconstruction is very similar to reconstruction described in 'Our approach' in section devoted to dynamic algorithm thus it won't be described here.

#### 2.1.2 Dynamic

To reduce time complexity of an algorithm, one may store intermediate solutions to avoid re-computation. When calculating treedepth, we can trade-off space complexity for quite significant improvement in time complexity (from $O(|G|!)$ to $O^*\left(2^{|G|}\right)$). The most classical approach is described in section 'Our approach' in section 'Dynamic'. In 2013, the team of Fedor V. Fomin, Archontia C. Giannopoulou and Michał Pilipczuk came up with an improvement to classical dynamic algorithm which reduces its time complexity from $O^*\left(2^{|G|}\right)$ to $O^*\left(1.9602^{|G|}\right)$ [2]. As dynamic approach is based on treedepth decompositions of induced subgraphs, they restrict what kind of subgraphs will be considered in subsequent iterations of the algorithm. To ensure pruned subgraphs that would've led to optimal solutions are not lost, it is shown that those graphs share very special properties (minimal trees) and can be recovered easily. For further details about their work please refer to [2].

## 2.2 Approximate

### 2.2.1 Bodlaenders' heuristic [4]

A notion strongly connected to the notion of treedepth is the notion of treewidth $(tw\,(G))$. Treewidth can be described as: *Given a graph G, we can eliminate a vertex v by removing it from the graph, and turning its neighborhood into a clique. The treewidth of a graph is at most k, if there exists an elimination order such that all vertices have degree at most k at the time they are eliminated.*[3]. *Then, the smallest k such that there exists elimination order which satisfies $td(G) \leq k$ is called treewidth of G.* For the definition of treewidth decomposition please refer to [5].
Bodlaenders' approach is based on inequality [2][4]:

$$tw(G) \leq td(G) \leq tw(G) \cdot \log(|G|) \tag{3}$$

First, he finds an approximate treewidth decomposition which is at most $O(\log(|G|))$ times worse than the optimal solution. Then, by finding Georges' nested dissection ordering [6][4] finds treedepth decomposition which is at most $O\left(\log^2(|G|)\right)$ times worse than the optimal one. Note that this algorithm is one of the best approximations of treedepth decompositions and works in polynomial time.

# 3 Our approach

## 3.1 Dynamic Algorithm

### 3.1.1 Binomial Encoding [1]

In this section we shall present a simple method to optimally encode $k$-element subsets of $\{0, ..., n-1\}$ for $k \in [0; n]$ called binomial encoding:

```
size_t Encode(Set s, size_t n, size_t k) {
  size_t ret = 0;
  while (k > 0) {
    --n;
    if (s.contains(n)) {
      ret += NChooseK(n, k);
      --k;
    }
  }
  return ret;
}

Set Decode(size_t code, size_t n, size_t k) {
  Set ret;
  while (k > 0) {
    --n;
    size_t nk = NChooseK(n, k);
    if (code >= nk) {
      ret.insert(n);
      code -= nk;
      --k;
    }
  }
  return ret;
}
```

This method of encoding is optimal because it forms a bijection: $\binom{\{0,...,n-1\}}{k} \to \left[0; \binom{n}{k}\right)$.
It is capable of running in $O(n)$ time, provided that Pascals' Triangle is precomputed.

### 3.1.2 UnionFind data structure

UnionFind data structure is crucial in this approach. It will allow us to compute treedepth efficiently. With each set it associates some value, which will be managed by UnionFind data structure internally. Our implementation shall satisfy following contract:

- It can be constructed from integer $n$. As a result, we get a UnionFind data structure representing $n$ disjoint one-element subsets of $\{0, ..., n-1\}$ and have associated value equal to 1. Each set has an id equal to the element it contains.
- `UnionFind Clone()` clones given object.
- `ValueType GetValue(SetIdType)` returns value associated with given set.
- `ValueType GetMaxValue()` returns the maximum of all values associated with sets represented by the object.
- `SetIdType Find(ElemType)` returns id of a set in which given element is contained.
- `SetIdType Union(SetIdType s1, SetIdType s2)` sums two given sets. As a result of this operation:
  - `s1` will not change its id.
  - Every element of `s2` will be contained in `s1`.
  - Value associated with `s1` is replaced with
    the greatest of values: `GetValue(s1)`, `GetValue(s2) + 1`

The data structure defined this way allows to compute treedepth very efficiently, as values associated with contained sets change in the same way as treedepth if we were to attach one tree to the root of another tree. Complexity of each operation is not much bigger than $O(1)$ (if not exactly $O(1)$) except for construction and copying which have $O(n)$ time complexity.

### 3.1.3 UnionFind implementation

To implement UnionFind data structure we use an array of size $|G|$ (number of disjoint sets at the beginning) and two variables to store currently the highest assigned value to any of sets and the size of the array. In this array we store a forest where each tree is a representation of one of the subsets. In $array[v]$ we store the parent of $v$ node. If $array[v] < 0$ it means that v is the root of a tree and a representative of the set. As not to waste space, we use this negative number to assign a value to this set. $|array[v]|$ is the assigned value.

*Find* is capable of performing *path compression* i.e. each call to this method is changing the structure of forest to flatten the tree which contained sought element as much as possible without causing too much computational overhead.

On the other hand, *Union* method is implemented trivially. We always attach tree representing the set of the second argument to the tree representing the set of the first argument. We do not risk here the uncontrolled growth of the tree, because our algorithm always joins sets where one of them is a single element set.

On GPU our algorithms will work in the same way as on CPU. The only difference will be the layout of used structures in memory. To take into account the architecture of the GPU we will try to use SoA (Structure of Arrays) rather than AoS (Array of Structures) approach to encourage coalescing and minimize memory bank conflicts. This different layout of data should greatly reduce time spent by GPU on memory access.

### 3.1.4 Idea behind the algorithm

Having described all necessary tools, we shall now proceed to describe dynamic approach to computing treedepth value as well as treedepth decomposition. As mentioned earlier, this method trades off space complexity of $O\left(2^{|G|} \cdot |G|\right)$ for time complexity of $O^*\left(2^{|G|}\right)$. The algorithm builds the treedepth decomposition in a bottom-to-top manner. It generates an array of pairs `(UnionFind, size_t)` from another array of the same type. We start with an array of one element: unmodified UnionFind constructed from $|G|$ and index 0. Then, given a UnionFind object, for every inactive vertex $v$ in it, we shall generate a new one by:

- activating $v$ in considered object
- performing `Union(Find(`$v$`), Find(`$w$`))` for every active neighbor $w$ of $v$ in $G$

This very action adds vertex $v$ to some treedepth decomposition of $G$ induced on a set of active vertices. At this point it would be appropriate to explain what an active vertices are and how we will handle them. It is a concept which tells us whether a vertex is in a solution represented by the given UnionFind object. With an index in an array we associate a UnionFind object. This index shall be decoded into a subset of $\{0, ..., n-1\}$ which will tell us what vertices are active in object in question (nevertheless nothing stops us from accessing other elements).
Having generated a new UnionFind we shall store it in output array under the index equal to the encoding of a set of active vertices contained in generated UnionFind object. With it we store an index from which we received the original UnionFind object. It will be useful in treedepth decomposition reconstruction. If there is some UnionFind object in the designated space, we replace the object in it only if our UnionFind has a lower max value.

### 3.1.5 The Algorithm

```
void TDDynamicStep(Graph g, size_t step_number, (UnionFind, size_t)[] prev, (UnionFind,
    size_t)[] next) {
  for(size_t subset_index = 0; subset_index < prev.size(); ++subset_index) {
    // Get set of active vertices
    Set s = Decode(subset_index, V(g).size(), step_number);
    for(ElemType x : V(g)\s) {
      UnionFind uf_x = prev[subset_index].first.Clone();
      Set ids = {};
      // Find trees containing neighbors of v
      for(ElemType v : g.neigh(x))
        if(s.contains(v))
          ids.insert(uf_x.Find(v));
      SetIdType new_set_id = uf_x.Find(x);
      // Ensure that we are still in compliance with treedepth decomposition definition when adding x to active vertices
      for(SetIdType id : ids)
        new_set_id = uf_x.Union(new_set_id, id)
      s.insert(x);
      // Determine designated index of our new object
      size_t dest = Encode(s, V(g).size(), step_number + 1);
      // Check if we have improved current result for combination of active vertices represented by s
      if(next[dest] == null || next[dest].first.GetMaxValue() > uf_x.GetMaxValue())
        next[dest] = (uf_x, subset_index);
    }
  }
}

(UnionFind, size_t)[][] TDDynamicAlgorithm(Graph g) {
  // Starting point of the algorithm: object representing empty set of active vertices
  (UnionFind, size_t)[][] history = new (UnionFind, size_t)[V(g).size()][];
  history[0] = new (UnionFind, size_t)[1];
  history[0][0] = UnionFind(V(g).size());
  for(size_t i = 0; i < V(g).size(); ++i) {
    history[i + 1] = new (UnionFind, size_t)[NChooseK(V(g).size(), i + 1)];
    TDDynamicStep(g, i, history[i], history[i + 1]);
  }
  return history;
}
```

Having completed the dynamic algorithm, the object on the last page of history holds the treedepth of our graph. Based on the history we can reconstruct the treedepth decomposition in $O(|G|)$ time:

```
(Graph, int) TDDecomposition(Graph g, (UnionFind, size_t)[][]history) {
  size_t current_entry = 0, current_page = V(g).size() - 1;
  int depth = history[current_page][current_entry].second;
  // This list will hold permutation of vertices that made up the object in history[current_page][current_entry]
  List indices = {current_entry};
  while(current_page > 0) {
    indices.push_front(history[current_page][current_entry].second);
    current_entry = history[current_page][current_entry].second;
    --current_page;
  }
  Graph tree(V(g).size());
  while(current_page < V(g).size()) {
    size_t current_entry = indices.pop_front();
    Set s = Decode(current_entry, V(g).size(), current_page);
    // Find vertex that had been added in considered transition
    ElemType v = Decode(indices.front(), V(g).size(), current_page)\s;
    UnionFind uf = history[current_page][current_entry].first;
    // Add edges required by the definition of treedepth decomposistion
    for(ElemType w : g.neigh(v))
      if(s.contains(w))
        tree.add_edge(v, uf.Find(w));
    ++current_page;
  }
  return tree, depth;
}
```

This algorithm finds what vertices have been added on consequent steps and joins appropriate trees under common parent vertex.

### 3.1.6 Remarks

As we can see, this algorithm provides an obvious improvement in time complexity of $O^*\left(2^{|G|}\right)$ compared to naïve approach with time complexity of $O\left(|G|!\right)$, but it requires additional $O\left(2^{|G|} \cdot |G|\right)$ space to run. It is particularly important for set encoding to be very efficient as it will be executed many times, therefore, the presented method for such encoding might not make it into final implementation and be replaced with different method or be modified significantly. Space complexity of this algorithm makes it impractical to be run on large graphs. We address this issue in next sections of this document. It also has some useful properties which we will exploit, but this will be also discussed later.

## 3.2 Branch and Bound algorithm

In contrary to dynamic algorithm, this time we will build treedepth decomposition in a top-to-bottom manner. To make this approach efficient, we shall incorporate branch and bound technique. Please note that this approach assumes connected input graph. This assumption is valid as for disconnected graphs we can run this algorithm for every connected component.

### 3.2.1 Treedepth decomposition of incomplete elimination

In this section we shall define notions that will be crucial in further sections of this document.

1. Incomplete elimination $w$ is a word with distinct characters upon alphabet $V(G)$.
2. Treedepth decomposition of incomplete elimination $w$ is a rooted tree $T_w$ such that:
   - Treedepth decomposition of empty elimination (empty word $\varepsilon$) is a single vertex represented by $G$.
   - Treedeph decomposition of incomplete elimination $wv$, is built from $T_w$, in the following way:
     (a) Find leaf $H$ in $T_w$ such that $v \in H$.
     (b) Add every connected component of $H - v$ as a child of $H$.
     (c) Replace node $H$ with $v$.
3. Depth of a node $v$ in $T_w$ is defined as a length of a path between $T_w$ root and $v$ and is denoted as $depth_{T_w}(v)$.

Those notions embrace the history of treedepth decomposition, i.e. how it used to be called elimination tree. The $T_w$ structure really shows how vertex elimination affects the graph while preserving information about the elimination and is capable of illustrating an elimination in progress. There are some observations that can be made to familiarize oneself with incomplete elimination $w$ and $T_w$:

- $w$ tells us which vertices have been removed from $G$ and in which order. $T_w$ pictures state of a graph after elimination of vertices from $w$
- $T_\varepsilon$ is represented by $G$ alone because that is how the graph looks when no vertices were eliminated
- $T_w$ is a tree with leaves being induced subgraphs of $G$. Internal nodes are single vertices from $G$ which are also induced subgraphs of $G$
- $T_w$ root is $G$ when $w = \varepsilon$ otherwise it's the first letter of $w$
- Construction of $T_{wv}$ breaks down exactly one of its leaves into connected components. Those new leaves are one step lower than its ancestor, to show how their treedepth has changed upon elimination of $v$
- Path from root to leaf $H$ shows which vertices had to be removed in order to acquire $H$ as a connected component
- $T_w$ of complete elimination (when $|w| = |G|$) is a treedepth decomposition of $G$ (or maybe rather an elimination tree at this point)

### 3.2.2 The algorithm

As in every branch and bound algorithm, we shall start with some valid solution to our problem. This solution will be improved upon and be used to eliminate some unnecessary attempts. Our initial solution shall be $P_{|G|}$.

Let $L$ be a language of permutations of $V(G)$. We shall traverse prefix tree built upon $L$. Let $w = \varepsilon$. For each $H$ such that $H$ is a leaf of $T_w$, if $lbtd(H) + depth_{T_w}(H)$ is bigger than currently best known upper bound of $td(G)$, then remove last letter from $w$ and continue tree traversal. If $|w| = |G|$, then update current best known treedepth decomposition with treedepth decomposition defined by $w$ if necessary. If $|w| < |G|$, then, for each $v \notin w$, traverse subtree rooted in $wv$.

Note: $lbtd(H)$ denotes lower bound of $td(H)$ and for non-empty graphs is at least 1. When $td(H)$ is known, it can be used instead of $lbtd(H)$.
Note: Upon elimination, one can find leaf $H$ in $T_w$ which maximizes $|H| + depth_{T_w}(H)$. Then $|H| + depth_{T_w}(H)$ is an upper-bound of $td(G)$, possibly improving upon it.

### 3.2.3 Remarks

This algorithm has very high pessimistic time complexity of $O(|G|!)$, but has small (polynomial) space complexity. Knowledge about treedepth value of induced subgraphs of $G$ as well as fast discovery of low-treedepth solutions can greatly benefit the performance of this algorithm.

## 3.3 Modifications

This section is devoted to acknowledging some interesting properties of just described algorithms. Since we aim to incorporate GPUs into our project it revolves around this idea.

### 3.3.1 Dynamic algorithm

The Dynamic algorithm is very easy to parallelize on the level of one step. Each subset can be processed independently of the others. The only synchronization will be needed to update output array but this should impose no problem at all. Another interesting property of this algorithm is the possibility of reconstructing treedepth decompositions in parallel. If we were to terminate this algorithm at, let's say 18-element subsets of a 50-element set, we can, for each of these subsets, reconstruct the treedepth decomposition independently of others, provided that operation `SetId Find(ElemType)` will not change the UnionFind object i.e. will not perform path compression if told not to do so.

### 3.3.2 Branch and Bound algorithm

We will attempt to implement this algorithm on a GPU. It may require some modifications to it, but we will not be deterred. The most important modification will be combining this algorithm and the dynamic algorithm, to create so-called *Hybrid Algorithm.* This modification provides treedepth values of induced subgraphs requested by branch and bound algorithm from the results of incomplete execution of the dynamic algorithm. This modification will reduce branch and algorithms' complexity to $O((|G|-k)!)$, where $k$ is the number of the last iteration performed by the dynamic algorithm.

# 4 Expected results

## 4.1 Dynamic

We most likely will manage to implement dynamic algorithm both for GPU and CPU. The GPU implementation of dynamic algorithm will probably perform very well as long as memory will allow it to do so. The GPU implementation is our first modification to the well-known algorithm and we expect, that it will achieve much better performance results than its CPU version. We predict that the limits on memory will be reached for graphs with around 30 vertices. The exact value depends on union-find structure implementation.

## 4.2 Branch and bound

This algorithm has higher complexity than dynamic algorithm and most likely it will perform worse than dynamic algorithm, however, it does not have limits on memory as its memory complexity is polynomial. We may manage to provide CPU implementation, but because of its recursive nature it may cause a lot of trouble to accomplish GPU one. This algorithm was designed to address memory issues that the dynamic algorithm imposes.

## 4.3 Hybrid

It is expected, that this algorithm will perform much better, than its ancestor, branch and bound algorithm. Its improvement will be strictly determined by the number of iterations the dynamic algorithm was capable of carrying out. This algorithm is planned to be implemented on a GPU thus further reducing its running time.

## 4.4 General thoughts

Having presented our ideas for computing treedepth along with its decomposition, we acknowledge that they highly vary in degree of complexity. Some of them are very easy to be implemented both on GPU and CPU, and some seem to pose a challenge to be implemented on a GPU. Nevertheless we aim to accomplish all of the goals set, including the hardest of all: **hybrid approach implemented on a GPU**.

## References

[1] Yuval Filmus (https://cstheory.stackexchange.com/users/40/yuval-filmus), *Optimal encoding of $k$-subsets of $n$*, Theoretical Computer Science Stack Exchange. URL: https://cstheory.stackexchange.com/q/19330 (version: 2013-10-09).

[2] Archontia C. Giannopoulou Fedor V. Fomin Michał Pilipczuk, *Computing treedepth faster than* $2^n$, Department of Informatics, University of Bergen, Norway. URL: https://www.mimuw.edu.pl/ mp248287/treedepth-talk.pdf.

[3] Tom C. and Bodlaender van der Zanden Hans L., *Computing Treewidth on the GPU*, arXiv e-prints (2017), arXiv:1709.09990, available at `1709.09990`.

[4] Hans L. Bodlaender, *Approximating treewidth, pathwidth, frontsize, and shortest elimination tree.* (1992).

[5] Marek and Fomin Cygan Fedor V. and Kowalik, *Parameterized Algorithms*, Springer Publishing Company, Incorporated, 2015.

[6] Alan George, *Nested Dissection of a Regular Finite Element Mesh*, SIAM Journal on Numerical Analysis **10** (1973), no. 2, 345-363, DOI 10.1137/0710032.