# 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 "starlike" $G$ 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.

Antoher thing is that for every graph $G$, a trivialy valid tree-depth depcomposition 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 contradictiory to $G$ being connected.

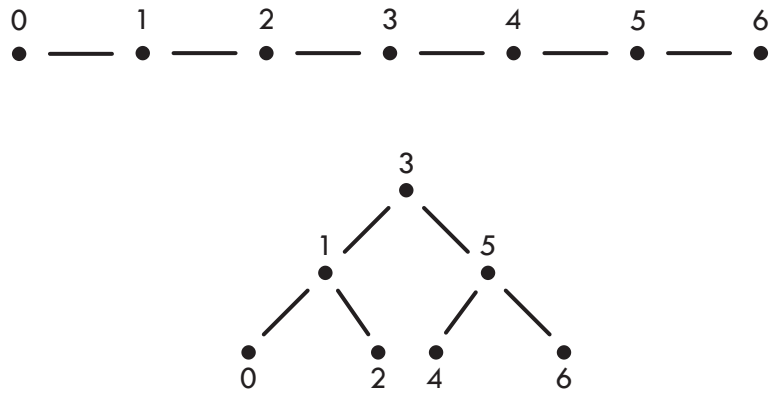


Figure 1: $P_7$ and its tree-depth decomposition

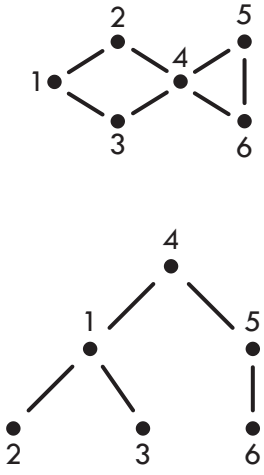


Figure 2: Fish and its tree-depth decomposition

# 2 Typical approaches

## 2.1 Exact

### 2.1.1 Naïve

Recall the defintion 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 build using 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 simillar to reconstruction described in 'Our approach' in section devoted to dynamic algorithm thus it won't be described here.

### 2.1.2 Dynamic

In order to reduce time complexity of an algorithm, one may store intermediate solutions to avoid recomputation. 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, 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. In order to ensure pruned subgraphs which 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]:

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

First, he finds an approximate treewidth decomposition which is at most $O\left(\log\left(|G|\right)\right)$ times worse than the optimal solution. Then, by finding Georges' nested dissection ordering [6][4] finds treedepth decomposition which is at most $O\left(\log^2\left(|G|\right)\right)$ times worse than the optimal one. Note that this algorithm is one of the best approximations of treedepth decmpostions 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 beacuse it forms a bijection: $\binom{\{0,\dots,n-1\}}{k} \to [0; \binom{n}{k})$.
It is capable of running in $O(n)$ time, provided that Pascals' Traingle 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 $n$ disjont 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 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 $n$ (number of disjoint sets at the beggining) 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 subsets.

*Find* is capable of performing *path − compression* i.e. each call to this method is changing the structure of forest to maximaly flatten the tree which contained sought element.

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

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 minimaze memory bank conflicts. This different layout of data in memory should greatly reduce time spent by GPU on memory access activities.

To slightly reduce memory usage of this structure instead of storing the value assigned to each tree next to them and using another integer, we store this value in a root as a negative value which tells us that this is a root and its absolute value is assigned value.

### 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 bottom-to-top manner. It generates an array of pairs `(UnionFind, size_t)` from another array of the same type. We start off 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 neighbour $w$ of $v$ in $G$

This very action adds vertex $v$ to some treedepth decomposition of $G$ induced on 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 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 encoding of a set of active vertices contained in generated UnionFind object. With it we store an index from which we recieved 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 lower max value.

### 3.1.5 The Algorithm

```
1  void TDDynamicStep(Graph g, size_t step_number, (UnionFind, size_t)[] prev, (UnionFind,
       size_t)[] next) {
2    for(size_t subset_index = 0; subset_index < prev.size(); ++subset_index) {
3      // Get set of active vertices
4      Set s = Decode(subset_index, V(g).size(), step_number);
5      for(ElemType x : V(g)\s) {
6        UnionFind uf_x = prev[subset_index].first.Clone();
7        Set ids = {};
8        // Find trees containing neighbours of v
9        for(ElemType v : g.neigh(x))
10         if(s.contains(v))
11           ids.insert(uf_x.Find(v));
12       SetIdType new_set_id = uf_x.Find(x);
13       // Ensure that we are still in compliance with treedepth decomposition definition
       when adding x to active vertices
14       for(SetIdType id : ids)
15         new_set_id = uf_x.Union(new_set_id, id)
16       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 beed 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, 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 algorithmm 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 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 Idea behind the algorithm

As in every branch and bound algorithm, we shall start with some valid solution to our problem. We can try to find some good heuristic solution, but let's choose $P_{|G|}$ for now. Along this solution we must store its treedepth (equal to $|G|$) as we will improve this number and eliminate some unnecessary attempts. We can now proceed to generating our solutions. First, we will choose a root of our tree. Then, we will choose its children. Then we will choose their children and so on. We will be building our solution generation after generation (like generations of a family). When chosen generation is too deep in our tree, we terminate current attempt and try different configuration.

### 3.2.2 Determining which vertices are valid as children

Not every vertex will be suitable as a child of another vertex in treedepth decomposition. Take as an example $P_5$ with vertices $\{0, 1, 2, 3, 4\}$ and its decomposition generation. Let's say that we chose as a root of our tree vertex 2. Then we know, for sure, that vertices 1 and 3, as neighbours of 2, must be in a subtree of 2. Let's say that we decided, that 1 and 3 will be the only children of 2. Now we know for sure, that vertex 0 must be in a subtree of 1 and vertex 4 must be in a subtree of 3. A tree when vertex 0 is a child of vertex 3, will not be valid treedepth decomposition as it condradicts treedepth decomposition definition. Our algorithm is equipped with a collsion detection mechanism, which can react to such contradictory actions. Upon detection, many trees can be discarded from further consideration, thus decreasing runtime.

During execution of branch and bound algorithm we will keep track of preferences (in subtree of which vertex must be placed) of each vertex along with information about what vertices are already in the tree.

### 3.2.3 The Algorithm

```
// Global variables used by the algorithm (current state)
const int Taken = -1;
const Graph g = GetConnectedGraph();
int[] preference = new int[V(g).size()];
Graph best_tree = Path(V(g));
int best_td = V(g).size();
Graph current_tree = Graph();

// Finds treedepth decomposition with its treedepth using branch and bound technique
(Graph, int) TDBnB() {
  for(int root = 0; root < V(g).size(); ++root) {
    // Vertices which will have its children generated next. Right now it's just root.
    Set active_vertices = {root}
    for(int y = 0; y < V(g).size(); ++y)
      // Every vertex will end up in a subtree of trees' root.
      preference[y] = root;
    current_tree = Graph();
    current_tree.add_vertex(root);
    preference[root] = Taken;
    NextLevel(1, active_vertices);
  }
  return best_tree, best_td;
}

// This function is called whenever a level in current_tree has been finished.
    active_vertices is a set of vertices which will have its children chosen next. Depth
    tells us how deep finished level is. It checks whether we have a chance improving
    previous results and starts generation of next level.
Collision NextLevel(int depth, Set active_vertices) {
  if(V(current_tree).size() == V(g).size() && depth < best_td) {
    (best_tree, best_td) = (current_tree, depth)
    return null
  }
  if(active_vertices.empty() || depth + 1 >= best_td)
    return null;
  int from = 0;
  while(preference[from] == Taken)
    ++from;
  return GenerateFrom(depth + 1, active_vertices, {}, from);
}

// Checks if v2 is a child of v1 in tree
```

```
bool InSubtreeOf(Graph tree, int v1, int v2) {
  while(v2 != tree.root()) {
    if(v1 == v2)
      return true;
    v2 = tree.parent(v2);
  }
  return v1 == v2;
}

// Tries to add child to av as a child of parent and update both current_tree and preferences. When it contradics with treedepth decomposition definition, it returns an error and makes no changes to current state.
Collision TryVertexAsAChild(int child, int parent, Set av) {
 if(!InSubtreeOf(current_tree, preference[child], parent))
   return {};
 for(int v : g.neigh(child))
   if(preference[v] != Taken && !InSubtreeOf(current_tree, preference[v], parent))
     return Collision{v1: preference[v], v2: child, w: v};
 current_tree.add_vertex(child);
 current_tree.add_edge(child, parent);
 for(int v : g.neigh(child))
   if(preference[v] != Taken)
     preference[v] = child;
 preference[child] = Taken;
 av.insert(child);
 return null;
}

// Determines whether given collision makes current state contradictory with treedepth decomposition definition
bool Critical(Collision collision) {
  ...
  return false;
}

// This function considers vertex from as a child of consecutive vertices from active_vertices. At the time of calling this function, every vertex v < from has been considered on this level to some extent. Note that calling this function does not change current state i.e. after this function returns state is how it was before the call. It is required that preference[from] != Taken.
Collision GenerateFrom(int depth, Set active_vertices, Set active_vertices_new, int from) {
  int next_available = from + 1;
  while(next_available < V(g).size() && preference[next_available] == Taken)
    ++next_available;
  for(int x : active_vertices) {
    // Dump all information that might change as a result of TryVertexAsAChild
    State previous_state = CurrentState.save();
    Collision collision = TryVertexAsAChild(from , x, active_vertices_new);
    if(collision == null) {
      State current_state = CurrentState.save();
      if(next_available < V(g).size()) {
        // Try continuing generation from next available vertex
        collision = GenerateFrom(depth, active_vertices, active_vertices_new, next_available);
        // Rollback changes made
        previous_state.apply();
        if(Critical(collision))
          return collision;
        current_state.apply();
      }
      // Try finishing generating this level
      collision = NextLevel(depth, active_vertices_new);
      // Rollback changes made
      previous_state.apply();
      if(Critical(collision))
        return collision;
    }
    else if(Critical(collision))
      return collision;
  }
  // Try not adding from as a child of any vertex on this level if it makes sense
  if(next_available < V(g).size() && depth + 1 < best_td)
```

```
104     return GenerateFrom(depth, active_vertices, active_vertices_new, next_available);
105   return NextLevel(depth, active_vertices_new);
106 }
```

#### 3.2.4 Remarks

Despite very high pesymistic time complexity being at most $O^*\left(|G|^{|G|-2}\right)$ it is expected for this algorithm to perform reasonably well, just as TSP algorithm using branch and bound technique does. With every improvement of initial heuristic this algorithm will significantly reduce its running time. It has small space complexity of $O\left(|G|^2\right)$ which is much better than dynamic algorithms' space complexity of $O\left(2^{|G|} \cdot |G|\right)$. The depth of recursion of this algorithm is at most $2 \cdot |G|$ which is small enough to not worry about stack overflow when implementing it on a standard machine.

### 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 level of one step. Each subset can be processed independently of the others. The only synchorization will be needed to update output array but this should impose no problem at all. Another interesting property of this algorithm is possibility of reconstructing treedepth decompositions in parallel. If we were to terminate this algorithm at, let's say 18-element subsets of 50-element set, we can, for each of these subsets, reconstruct the treedepth decomposition independently of anothers, 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

Obiously we will make an 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 together, to create so called *Hybrid algorithm*. It goes as follows:

- Run the dynamic algorithm until we have insufficient amount of memory for the next iteration
- Reconstruct intermediate solutions (so-called *endings*) provided by the history of execution of the dynamic algorithm
- Launch Branch and Bound algorithm taking *endings* into account

The last step is yet unclear, but poses final modification to Branch and Bound algorithm. It shows how those to approaches can be combined together, in hope of increasing efficiency to a maximum. Of course it will be implemented on a GPU to further decrease its running time. The method of combining is in development as of writing this document.

# 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 well-known algorithm and we expect, that it will achive 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. It is very possible that we will 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

This algorithm has similar complexity to branch and bound algorithm as branch and bound is main part of it, but we assume that it will perform way better than its basic version because of *endings* precalculated by dynamic algorithm and at the same time it will not have the limit on memory, so this is our idea to handle the dynamic algorithm memory issue. Most likely we will provide CPU implementation.We suspect, that GPU implementation will be very hard to accomplish. Nevertheless we are highly motivated to come up with GPU implementation of this algorithm even if it requires launching seperate project.

## 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.