Calculating tree-depth
Introduction

Dymitr Lubczyk Wojciech Replin Bartosz Różański

15.04.2020 r.

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 the graph 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 decomposition of G.

Tree-depth is also be defined in recursive manner. It is worth to mention 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 I 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 equals 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 is the same thing.

Antoher thing is that for every graph G a trivialy valid tree-depth depcomposition of G is a path P(where $|P| = |G|$) 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.

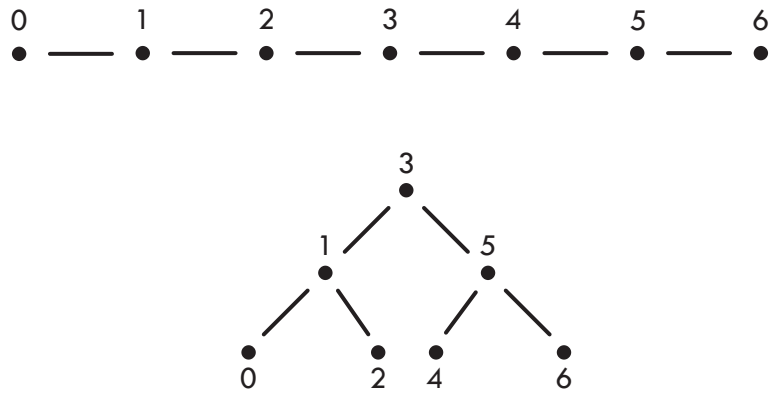

Figure 1: P_7 and its tree-depth decomposition

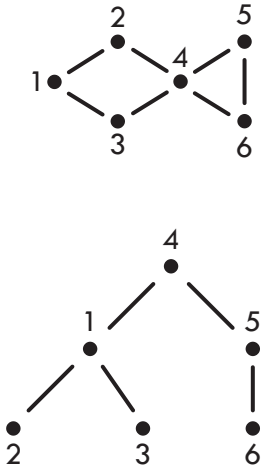

Figure 2: Fish and its tree-depth decomposition

2 Typical approaches

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: $\binom{\{0,\cdots,n-1\}}{k} \to [0; \binom{n}{k})$.
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 has 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 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 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.

3.1.4 The Algorithm

```
void TDDynamicStep(Graph g, size_t step_number, (UnionFind, size_t)[] prev, (UnionFind,
  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 neighbours 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
      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 rep
      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[curr
  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.

3.1.5 Remarks

As we can see, this algorithm provides an obvious improvement in time complexity of $O^*\left(2^{|G|}\right)$ compared to naive 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. So much space is necessary by this algorithm to run, that it can be too much for large graphs. We will try to overcome this difficulty later. 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 try to 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 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.
General rule is: every neighbour of v in G must be either its child or ancestor in a valid treedepth decomposition (as definition of treedepth decomposition states).
During execution of branch and bound algorithm we will keep track of preferences (in subtree of which vertex it 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_ve
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;
  return GenerateFrom(depth + 1, active_vertices, {}, 0);
```

```
}

// 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 active_vertices_new as a child of parent and update both curre
Collision TryVertexAsAChild(int child, int parent) {
 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;
 return null;
}

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

// This function considers vertex from as a child of consecutive vertices from active_ve
Collision GenerateFrom(int depth, Set active_vertices, Set active_vertices_new, int from
  if(history[from] != Taken) {
    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);
      if(collision == null) {
        State current_state = CurrentState.save();
        if(from + 1 < V(g).size()) {
          // Try continuing generation from next vertex
          collision = GenerateFrom(depth, active_vertices, active_vertices_new, from + 1
          // 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(from + 1 < V(g).size())
    return GenerateFrom(depth, active_vertices, active_vertices_new, from + 1);
  return NextLevel(depth, active_vertices_new);
}
```

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 [`?tsp_bnb`] 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.

4 Proposed modifications

5 Expected results

5.1 General thoughts

As we mentioned before we are going to try a few variants with diffrent degree of complexity and in this point we are not sure if we manage to accomplish all of them. The most uncertain variant is branch and bound algorith on GPU as this alogorithm seems to be very hard to be made parallel. As a result it may not be possible to provide GPU implementation for hybrid algorithm, but it is not sure, we will try to figure it out.

5.2 Dynamic

We most likely will manage to do dynamic algorithm both GPU and CPU implementation and 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 be 30 times quicker 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.

5.3 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 polinomial. 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. The need of creating this algorithm came from dynamic algoritm memory limit.

5.4 Hybrid

This algorithm has similar complexity to branch and bound algorith 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 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. Unfortunately, we think that the GPU implementation is very unlikly to be accomplished.

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).