Dendrogram-based Algorithm for Dominated Graph Flooding

Claude Tadonki
Joint work with
Fernand Meyer and François Irigoin
Centre de Recherche en Informatique - Centre de Morphologie Mathématique
Mines ParisTech - FRANCE

International Conference on Computational Science - ICCS 2014
10-12 June, 2014
CAIRNS, AUSTRALIA
2.1 Graph flooding

Definition 1. Given a weighted undirected graph $G = (X, E, v)$ and a ceiling function $\omega : A \to \mathbb{R}$. A valid flooding function of $G$ under the ceiling $\omega$ is the maximal function $\tau : A \to \mathbb{R}$ satisfying

$$\forall x, y \in A : \quad \tau(x) \leq \min(\max(v(x, y), \tau(y)), \omega(x)).$$

(1)

Example 1. Figure 1 illustrates the flooding of a weighted graph with 7 vertices and 9 edges.

Two classical algorithms exist in the literature:
- **Dijkstra** algorithm is greedy
- **Berge** algorithm is dynamical programming
We study a decomposition algorithm based on the structure of dendrogram.
**Definition.** Let $A$ be a given set and $E$ a subset of $\mathcal{P}(A)$. $E$ defines a dendrogram if

(i) $\forall U, V \in E, \exists W \in E$ s.t. $(U \subseteq W) \land (V \subseteq W)$,

(ii) $\forall U \in E, \{V \in E | V \supseteq U\}, \subseteq$ is totally ordered.

**Definition.** Considering $U, V, W$ three elements of a dendrogram $E$,

(i) $U$ is a successor of $V$ (resp. $V$ is a predecessor of $U$) if

$$(U \not\subseteq V) \land (\exists W \in E$ s.t. $U \not\subseteq W \subseteq V).$$

(ii) $V$ is maximal if $\nexists W \in E$ s.t. $(W \supseteq V)$.

(iii) $V$ is minimal if $\nexists W \in E$ s.t. $(W \subseteq V)$.

**Figure:** A dendrogram

**Dendrogram of a graph**
(1) Build the dendrogram (this is a n-ary tree, we have considered a binary correspondance)
(2) Distribute the ceiling values of the vertices among the subdendrograms (this is a mintree)
(3) Flood the dendrogram from its leaves until we get the flooding levels of all vertices.

These are the main steps of the dendrogram-based algorithm.

Dendrogram based algorithm suits because
- can be used to generate information from a local input (flooding from a single vertex)
- exposes parallelism (when dismantling subdendrograms)
- several floodings of the same graph can be performed using its dendrogram structure. This aspect is particularly interesting because flooding from the dendrogram is very fast compared to the cost of constructing the dendrogram structure itself.
- is potentially efficient because key information are handled at the level of the sets (instead of individual vertices).

Other algorithms are global, so will always process with and for the whole graph, Live demo !!!
Constructing the dendrogram

D ← Ø
S ← {all edges (u, v, w) of the graph}
while(S ≠ Ø ){
  // we select the edge with minimum cost
  (u, v) ← min_w(S); // we can sort the list of edges and select on top
  // we remove that edge from S
  S ← S – {(u, v)};
  // we get the id the of the root subdendrogram containing u
  d1 ← id_root_subdendrogram(u);
  // we get the id the of the root subdendrogram containing v
  d2 ← id_root_subdendrogram(v);
  // we create a singleton subdendrogram if no one was so far created
  if(d1 == NULL) d1 ← dendrogram_singleton({u});
  if(d2 == NULL) d2 ← dendrogram_singleton({v});
  // we merge the two subdendrogram d1 and d2 to form a new one (parent)
  if(d1 ≠ d2)
    D ← D ∪ d;
    d ← dendrogram_merge(d1, d2);
  endif
}

Technical remarks

- \textit{dendrogram\_singleton}({\{u\}}) creates a subdendrogram with singleton \{u\}
- If \textit{u} and \textit{v} belong to an existing subdendrogram, then we avoid recreating it
- \textit{id\_root\_subdendrogram}({u}) is obtained by climbing from \textit{dendrogram\_singleton}({\{u\}}) to the maximal subdendrogram following the parent (successor) relation.

\textit{This function is the most time consuming of the construction. Its global impact depends on the depth or height of the dendrogram tree.}
In (a), getting the root from node s will cost 1, 2, 3, 4, and 5 steps respectively.

In (b), getting the root from node s will cost 1, 2, 3, and 4 steps respectively.

Going from a given leave to the root of its containing sub-dendrogram is so repeated that it costs. We should move from the previous root (so, store the roots!).

(a) and (b) are linear graphs, so each edge leads to a subdendrogram. This is not the case with any graph, like those containing cycles.

For each subdendrogram, we keep the outgoing edge with minimum cost. Having the list of edges sorted makes this easy, since the minimum outgoing edge is exactly the one connecting the subdendrogram to its parent.
// we get the leaf subdendrogram from which we start the flooding process
\( d \leftarrow \text{leaf_subdendrogram}(x) \)

// we go up while the ceiling is still greater than the diameter
while(\(!\text{is_root}(d)\) \&\& \(\text{ceil}(d) > \text{diam}(d)\))
\( d \leftarrow \text{pred}(d) \);

// we have reached a root and still get a ceiling greater than the diameter
// we set the definitive flooding values of this subdendrogram to \( \text{ceil}(d) \)
if(\(\text{ceil}(d) > \text{diam}(d)\))
\( \text{set_flooding_level}(d, \text{ceil}(d)) \);
else
\( \text{dismantle_ancestors}(d, \text{ceil}(d)) \);

\text{flood_from_vertex}(x)

**Technical remarks**

- The \textit{dismantling} process breaks the (sub)dendrogram into independent root subdendrograms.
- Newly created root subdendrograms during the \textit{dismantling} process are put into a FIFO queue.
- Each root subdendrogram is flooded through its vertex with the minimum id (value into the FIFO).
- The complete flooding process is achieved using the following loop

// the last subdendrogram we have created is maximal, thus a root
\( \text{FIFO\_root\_to\_explore}[0] \leftarrow \text{lastly\_created\_subdendrogram} \)
\( \text{nb\_root\_to\_explore} \leftarrow 1 \)
for(\( i = 0; i < \text{nb\_root\_to\_explore}; i++ \))
\( \text{flood\_\_observe}(\text{get\_vertex\_with\_min\_id}(\text{FIFO\_root\_to\_explore}[i])); \)

- The FIFO will be populated during the dismantling processes and \( \text{nb\_root\_to\_explore} \) will be incremented accordingly.

In which order should we explore the sub-dendrograms? Does this impact on the decomposition? Perf?
while(!is_root(d)){
  for(i = 0; i < nb_children(d); i++){
    e ← get_child_subdendrogram(d, i);
    cut_relationship(e, d); // e is no longer a child of d (dismantling)
    // the min_out_edge is set to max(min_out_edge, ceil(d)) VERY IMPORTANT!!!
    if(min_out_edge(d) < ceil(d)) set_min_out_edge(e, ceil(d));
    if(ceil(e) > min_out_edge(e)) set ceil(e, min_out_edge(e)); // update of ceil(e)
    if(ceil(e) > diam(e)) set_flooding_level(e, ceil(e));
    else{FIFO_root_to_explore[nb_root_to_explore] = e; nb_root_to_explore++;}
  }
  d ← pred(d);
}

dismantling_ancestors(d)

Technical remarks

- The minimum outgoing edge is compared to the ceiling of the parent, and we take the maximum.
- The dismantling process can either terminate the flooding of a subdendrogram or make it independent.
typedef struct {
    int edge_id;  // the id of the edge used to create this dendrogram (by merging)
    char is_leaf_left;  // tells if the left child is a (sub)dendrogram or vertex
    char is_leaf_right;  // tells if the right child is a (sub)dendrogram or vertex
    double diam;  // diameter of the (sub)dendrogram
    double min_outedge;  // the outgoing edge with the minimum cost
    int size;  // number of vertices of the support of this (sub)dendrogram
    double ceil;  // global ceiling of the dendrogram (obtained when propagating the input ceiling values)
    double flood;  // flooding value of the dendrogram (TO BE COMPUTED)
    int smallest_vertex;  // we keep the id of the vertex with the smallest ceiling
    int pred;  // the predecessor of this (sub)dendrogram (its parent in the hierarchical structure)
    int child_left;  // a dendrogram is obtained by fusing two subdendrograms (left, right)
    int child_right;  // right child
} dendro;

typedef struct {
    int nb_nodes;
    int nb_edges;
    int max_degree;
    double *weight;  // weight of the vertices (if any)
    int *neighbors;  // neighborhood of the nodes (array of size nb_nodes*max_degree)
    double *values;  // values in the edges (array of size nb_nodes*max_degree)
} graph;

Claude TADONKI
F. MEYER & F. IRIGOIN
Dendrogram-based algorithm for Dominated Graph Flooding
ICCS 2014 CAIRNS - AUSTRALIA
Dendrogram-based algorithm for Dominated Graph Flooding

ICCS 2014 CAIRNS - AUSTRALIA
The height of the dendrogram is moderate
Building the dendrogram predominates
We outperform Dijkstra by factor > 2
The height of the dendrogram is more related to the density.

Could we eliminate inoffensive edges.

The flooding step is noticeably fast (10% of the overall time).

We significantly outperform Dijkstra.
Neighborhood graph generated by **morph-m**

Flooding values computed by the **dendrogram-based algorithm**

*Number of nodes: 24,532*

*Number of edges: 96,138*

*Constructing the dendrogram: 1.737 s*

*Flooding process: 0.002 s*

*Whole algorithm: 1.739 s*

*Basic Dijkstra algorithm: 59.042 s*
Dismantling isolates independent subdendrograms which can be explored in parallel.
The flooding step can thus be parallelized.
Care about threads creation overhead.
Contend the effect of unbalanced load.

We consider a multithread implementation using pthread.
We create our threads once and each iterates on available subdendrograms isolated during dismantling.
The threads get their exploration tasks (subdendrograms) from a common pool in a round robbing way.
Efficiency of the parallelization on a multicore machine

Scalability on a quad-core machine

![Scalability graph]

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
<th>Case 5</th>
<th>Case 6</th>
<th>Case 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.241</td>
<td>0.709</td>
<td>0.535</td>
<td>0.609</td>
<td>1.404</td>
<td>0.802</td>
<td>0.603</td>
</tr>
<tr>
<td>1.00</td>
<td>1.75</td>
<td>2.32</td>
<td>2.04</td>
<td>1.00</td>
<td>1.75</td>
<td>2.33</td>
</tr>
<tr>
<td>1.00</td>
<td>1.72</td>
<td>2.65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Scalability of our algorithm on a quad-core machine
Materials

GET THE BINARY CODE

Download (linux object file)

In order to use this routine, you should follow the steps below (download an example)

- add the declaration
  ```c
  void graph_flood_dendro(int N, int L, int *X, int *Y, float *W, float *C, float *F, int th);
  ```
  on top of your C file
- then call
  ```c
  graph_flood_dendro(N, L, X, Y, W, C, F, 1);
  ```
  everywhere as needed in your program
- The typical compilation line is:
  ```c
  gcc -o your_executable graph_flood_dendro.o your_code.c -lpthread
  ```

The object file mainly contains the function described below

```c
graph_flood_dendro(N, L, X, Y, W, C, F, 1);
```

Computes the flooding levels of the weighted graph of size $N$, whose edges are $(X(i), Y(i), W(i))$, $i=0,...,L-1$

**INPUT**

- $N$: number of vertices of the graph
- $L$: number of edges provided
- $X, Y, W$: arrays of lengths $L$ defining the edges $(X(i), Y(i), W(i))$
- $C$: the array of the ceiling values (array of $N$ float)
- $th$: Number of threads (a value $< 2$ means sequential version, a negative value tells the code to use the available number of cores)

**OUTPUT**

- $F$: The array of flooding levels (array of $N$ float), this will be populated by the routine

The graph is assumed to be symmetric, however we systematically check and fix it if needed
The vertices of the graph should be labeled with integer number starting from 0

TIMC webpage [http://www.cri.ensmp.fr/projet_timc.html](http://www.cri.ensmp.fr/projet_timc.html)


Results & code [http://www.cri.ensmp.fr/TIMC/dendrogram/flooding.htm](http://www.cri.ensmp.fr/TIMC/dendrogram/flooding.htm)

Consider load balanced from the size of the subdendrograms (instead of their number)

- Parallelize the construction of the dendrogram

How to get the dendrogram of a modified graph from that of the original?

Thank you so much for your attention