Dendrogram Based Algorithm for Dominated Graph Flooding*

Fernand Meyer[2], Claude Tadonki[1] and François Irigoin[1]

[1] Mines ParisTech - Centre de Recherche en Informatique (CRI)
35, rue Saint-Honoré, 77305, Fontainebleau Cedex
claude.tadonki@mines-paristech.fr
[2] Mines ParisTech - Centre de Morphologie Mathématique (CMM)
35, rue Saint-Honoré, 77305, Fontainebleau Cedex
fernand.meyer@mines-paristech.fr

Abstract

In this paper, we are concerned with the problem of flooding undirected weighted graphs under ceiling constraints. We provide a new algorithm based on a hierarchical structure called *dendrogram*, which offers the significant advantage that it can be used for multiple flooding with various scenarios of the ceiling values. In addition, when exploring the graph through its dendrogram structure in order to calculate the flooding levels, independent sub-dendrograms are generated, thus offering a natural way for parallel processing. We provide an efficient implementation of our algorithm through suitable data structures and optimal organisation of the computations. Experimental results show that our algorithm outperforms well established classical algorithms, and reveal that the cost of building the dendrogram highly predominates over the total running time, thus validating both the efficiency and the hallmark of our method. Moreover, we exploit the potential parallelism exposed by the flooding procedure to design a multi-thread implementation. As the underlying parallelism is created on the fly, we use a queue to store the list of the sub-dendrograms to be explored, and then use a dynamic round-robin scheduling to assign them to the participating threads. This yields a load balanced and scalable process as shown by additional benchmark results. Our program runs in few seconds on an ordinary computer to flood graphs with more that 20 millions of nodes.

1 Introduction

A flooding algorithm is typically an algorithm for distributing material to every node of a connected network [1]. Graph flooding is thus a classical combinatorial problem, which has many applications in number of fields including *mathematical morphology* [6], *computer network* [3], *mathematical modeling (maze problem* [2]), and *hydrodynamic simulation*. The complexity of the problem mainly depends on the density of the input graph. Depending on the algorithm, the ceiling levels may also influence the effective running time. Anyway, real-time processing expectation for this problem, due to its applications and the large-scale instances under consideration, requires faster algorithms and associated implementations. In some cases, we need to compute the flooding levels for the same graph using different scenarios of the ceiling values. An algorithm for which we do not have to restart the whole flooding process from scratch with each scenario is clearly of a great value. This is the contribution of the current paper.

We provide a new algorithm for dominated graph flooding, based on a structuration of the graph in the form of a dendrogram. Then, given a set of ceiling values, we compute the flooding levels by performing a bottom-up exploration of the dendrogram organized as a binary tree. Our method is a kind of *divide and conquer* algorithm. Note that, Dijkstra procedure is a *greedy algorithm* [5], while Berge solution follows a *dynamical programming* approach [4]. Since our algorithm creates independent sub-structures on the fly, a potential for parallelism comes up. Efficient sequential and parallel implementations are really expected.

*TIMC

The rest of the paper is organized as follows. Section 2 provides the necessary background on graph flooding and the dendrogram structure. Next, our method is fully described in section 3 . Sequential and parallel implementations are detailed in section 4. Section 5 provides some benchmark results and section 6 concludes the paper.

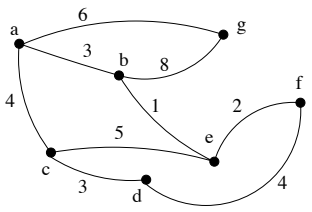
2 Background and preliminaries

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 ω is the maximal function $\tau : A \to \mathbb{R}$ satisfying*

$$\forall x, y \in A : \quad \tau(x) \leq \min(\max(v(x, y), \tau(y)),\ \omega(x)). \tag{1}$$

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



With the ceiling $\omega(\{a; b; c; d; e; f; g\}) = \{9; 3; 6; 7; 9; 4; 5\}$
we have the flooding $\tau(\{a; b; c; d; e; f; g\}) = \{3; 3; 4; 4; 3; 3; 5\}$

Figure 1: Sample flooding

Classical algorithms for graph flooding include Djikstra and Berge procedures. We restate the Djisktra algorithm for self-containedness, as we use it for validation and comparison purposes.

Algorithm 1 Dijkstra greedy algorithm for flooding a graph $G = (X, V, v)$

```
S ← X
while (S ≠ ∅) do
  x ← arg min_{x∈S} τ(x)
  S ← S − {x}
  for all y ∈ S such that (x, y) ∈ V do
    if (τ(y) < max(v(x, y), τ(x))) then
      τ(y) ← max(v(x, y), τ(x))
    end if
  end for
end while
```

Since each edge is visited once, i.e. the time one of its vertices is selected, the algorithm has a complexity of $O(|V| + |X|^2)$, where the term $|X|^2$ is for the selections of the *min* from X to $\emptyset$. The selections of of the minimum values can be made faster by implementing a so-called *mintree*. In that case, each selection of the minimum will cost $O(1)$, and the update of each of the corresponding neighbors will cost $O(\log(|S|)$ (insertion into a *mintree* containing $|S|$ nodes). If the degree of each node is atmost K, then the updates of the *mintree* will cost

$$K \sum_{p=1}^{|X|} \log(p) = K \log(|X|!) \leq K|X| \log(|X|). \tag{2}$$

Thus, for a graph with n nodes and m edges, Dijsktra algorithm has an asymptotic complexity of $m\log(n)$.

Another classical algorithm is due to Berge. The algorithm is a dynamical programming procedure, where the flooding values are iteratively updated from ω to the final results using the correction (3)

$$\tau_x \leftarrow \min(\tau_x, \max(v(x,y), \tau_y), \text{ for all } y \text{ such that } (x,y) \in V. \tag{3}$$

2.2 Dendrogram

Definition 2. *Let A be a given set and E a subset of $\mathscr{P}(A)$. E defines a dendrogram if*

(i) $\forall U, V \in E, \exists W \in E$ *s.t.* $(U \subset W) \wedge (V \subset W)$,

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

Definition 3. *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

$$(U \subsetneq V) \wedge (\nexists W \in E \text{ s.t. } U \subsetneq W \subsetneq V). \tag{4}$$

(ii) V is maximal if

$$\nexists W \in E \text{ s.t. } (W \supsetneq V). \tag{5}$$

(iii) V is minimal if

$$\nexists W \in E \text{ s.t. } (W \subsetneq V). \tag{6}$$

Definition 4. *Given a dendrogram E, we define $supp(E)$, the support of E by*

$$supp(E) = \bigcup_{X \in E} X. \tag{7}$$

Note that the maximal element of a dendrogram is not necessarily unique. This is the case when the graph is not strongly connected. However, without loss of generality, we will assume the uniqueness of the maximal element for the rest of the paper.

Definition 5. *Given a dendrogram E, any subset F of E that also defines a dendrogram is called a subdendrogram of E.*

A dendrogram can be viewed as a tree of its sub-dendrograms. In such a tree, the children of a node are all its successors. Thus, the *leaves* correspond to the *minimal* elements of the dendrogram, while the *root* corresponds to its maximal element. In addition, each sub-dendrogram is represented by the corresponding subtree, whose the root can thus be used as an identifier of the subdendrogram. Figure 2 displays an example of dendrogram and its hierarchical representation.

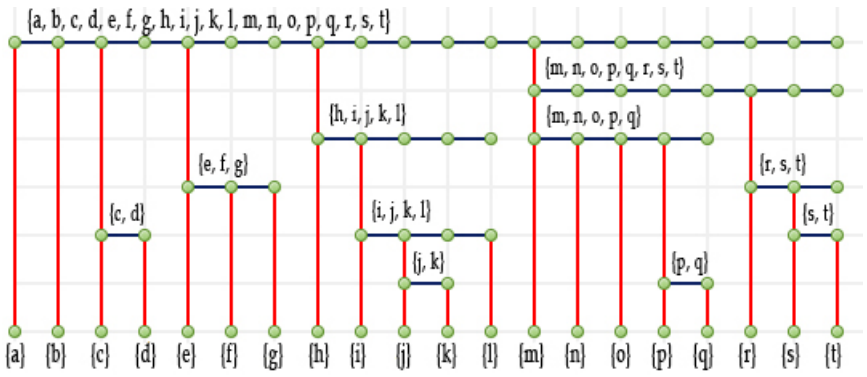


Figure 2: Sample dendrogram and its tree representation

The representation of a dendrogram as a tree can be used for its exploration following the rules a given algorithm. Typically, the dendrogram is constructed through its tree structure. Proposition 1 provides the key for a bottom-up agglomerative way to build the hierarchical structure of the dendrogram.

Proposition 1. *Given n distinct dendrograms E_i, $i = 1, 2, \cdots, n$, built over the same set A, a new dendrogram can be constructed as follows*

$$E = \bigcup_{i=1}^{n} E_i \cup \{\bigcup_{i=1}^{n} root(E_i)\}. \tag{8}$$

Proposition 1 can be used with $n = 2$ for a binary tree, which is more simpler to handle, although higher. Moreover, the way the pairwise selections for merging are performed (clustering) is algorithm dependent and impacts on the shape of the dendrogram representation.

We now describe our dendrogram-based graph flooding algorithm.

3 Description of our algorithm and its ingredients

3.1 Construction of the dendrogram

Our flooding algorithm operates on a dendrogram constructed from the input graph using the set of vertices as the support and the weights of the edges to select the sub-dendrograms to be merged. Algorithm 2 describes how we construct the dendrogram from a given weighted graph.

Algorithm 2 Construction of the dendrogram associated to a weighted a graph $G = (X, V, v)$

```
k = 1
for all x ∈ X do
  D_k ← {{x}} {Singleton dendrograms}
  k ← k + 1
end for
S ← V
while (S ≠ ∅) do
  μ = min(V(S))
  for all (x, y) ∈ S such that v(x, y) = μ do
    S ← S − {(x, y)}
    a ← get_id_root_dendrogram(x) {id of the root dendrogram containing {x}}
    b ← get_id_root_dendrogram(y) {id of the root dendrogram containing {y}}
    if (a ≠ b) then
      D_k ← merge(D_a, D_b, v(x, y))
      k ← k + 1
    else
      {We just need to update the properties of D_a using the weight v(x, y)}
      update_dendrogram_properties(D_a, v(x, y))
    end if
  end for
end while
```

Figure 3 illustrates our procedure on a linear graph whose the set of nodes is $\{a, b, c, \cdots, t\}$ and the weights are the values displayed in red at the bottom. Blue (horizontal) segments represent the nodes of the dendrogram and the associated subgraphs, while red (vertical) lines depict the hierarchical relationship within the dendrogram. Each node of the dendrogram is created by merging the roots of the sub-dendrograms that are linked by edges having the current minimum weight, the corresponding values are in purple color.

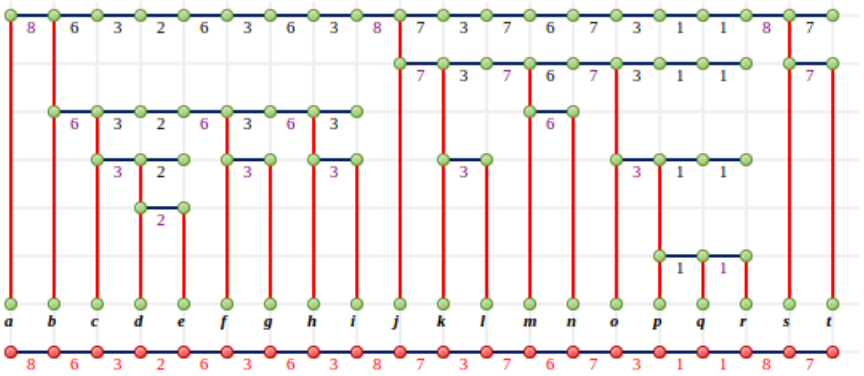


Figure 3: Sample dendrogram built using Algorithm 2

Since the nodes of a dendrogram are subsets of its support, we need to handle some basic information about the corresponding subgraphs. We now define two of them that will be very important for our algorithm.

Definition 6. *Given a weighted graph $G = (X, V, v)$ and a subset Y of X, we define*

$$\varphi(Y) = \min\{v(a,b) : (a,b) \in V, a \in Y, b \in X - Y\} \tag{9}$$

$$diam(Y) = \max\{v(a,b) : (a,b) \in V, a \in Y, b \in Y\} \tag{10}$$

Remark 1. *The main idea when using a dendrogram to flood a given graph is that, flooding levels will be assigned to subsets of vertices rather than to individual vertices. When a flooding value is assigned to a sub-dendrogram, it means that all the vertices of its support receive the same final flooding level equals to that value. In such case, the bigger is the sub-dendrogram the faster is the flooding process.*

Note that the dendrogram, including all its properties, only depends on the structure of the graph and the weights. The ceilings will come across to drive the flooding process as we will shortly describe.

3.2 Considering the ceiling levels

In a dominated flooding, flooding levels are constrained by upper bounds called *ceiling levels*. On a given vertex, the final flooding level should not exceed that of its ceiling level. The set of the ceiling levels is the main input for a flooding algorithm in addition to the weighted graph. An unconstrained vertex is said to have an infinite ceiling, hence the notation ∞ used in our figures. In practice, we can consider the maximum value between the highest ceiling level and the biggest weight of the edges.

Assume the dendrogram is constructed and the values of the ceiling levels are provided. Our algorithm starts by performing a bottom-up propagation of the ceiling values over the hierarchical structure of the entire dendrogram, following the principle of a *mintree*. The ceiling level of a node is the minimum between the ceiling levels of its children. Figure 4 illustrates the procedure on a sample linear graph. Propagated (resp. initial) ceilings are inside yellow (resp. orange) disks. A ceiling level not provided means that the corresponding vertex is unconstrained.

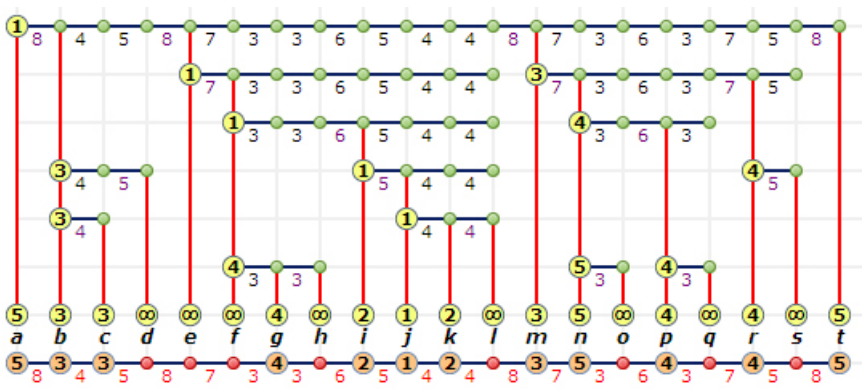


Figure 4: Distribution of the ceiling values among nodes of the dendrogram

3.3 The main algorithm

Once the nodes of the dendrogram are tagged with their initial ceiling levels, our algorithm performs a bottom-up exploration of its hierarchical structure and computes the desired flooding levels. This is done by refining the ceiling levels of the visited nodes of the dendrogram as long as the convergence criteria is not met. Algorithm 3 describes the process from a leaf node.

Algorithm 3 Exploration of the dendrogram from a given leaf node $\{x\}$

$d \leftarrow subdendrogram(\{x\})$ {this is the sub-dendrogram reduced to the singleton$\{x\}$}
while $((!is\_root(d)) \wedge (\omega(d) > diam(d)))$ **do**
  $d \leftarrow pred(d)$ {pred(d) is the parent of d in the tree}
end while
{Either d is a root sub-dendrogram or $(\omega(d) \leq diam(d))$}
if $(\omega(d) > diam(d))$ **then**
  {The vertices of the support of d receive the final flooding level $\omega(d)$, we're done with this!}
  $flood\_subdendro(d, \omega(d))$
else
  {We cut the branch between d and its children, we repeat the same with all its ancestors}
  $dismantle(d)$
end if

Let us outline the validation arguments of the algorithm:

- If $\omega(d) \leq diam(d)$, then $\omega(pred(d)) \leq diam(pre(d))$, since ω (resp. $diam$) is a decreasing (resp. increasing) sequence along the precedence path. Thus, once we have $\omega(d) \leq diam(d)$ in a bottom-up exploration, this will ever remains true. Since having the ceiling level lower than the diameter of the subgraph does not provide any useful information for the flooding, we stop our exploration here and disable the corresponding links in the structure.

- if $\omega(d) > diam(d)$ then all the nodes in the support of d should have the same flooding level (this is typically a lake) dominated by $\omega(d)$. Thus, if d is a root within the current dendrogram hierarchy (probably dismantled one or several times), then we can consider $\omega(d)$ as the flooding level of all associated nodes. Indeed, if d is root, then the corresponding sub-dendrogram is isolated, thus its flooding level can be computed without taking care of any potential influence on the complementary part of the entire dendrogram. However, for sake of consistency, we need to update ω accordingly during the dismantling process as we are going to explain.

3.4 The dismantling procedure

Considering the fact that our dendrogram is constructed by selecting the edges of the graph in an ascending order, we obtain that the diameter of any sub-dendrogram is lower than the weight of any outgoing or outside edge. Thus, the minimum weight of all outgoing edges (i.e. φ) of the subgraph restricted to the support of a sub-dendrogram is greater that its diameter. Consequently, for a given sub-dendrogram d, we have

$$(\omega(d) > diam(d)) \implies \min(\omega(d), \varphi(d)) > diam(d). \tag{11}$$

Following (11) and the fact that we consider a sub-dendrogram d as a lake in case $\omega(d) > diam(d)$, in order to be able to consider the current ceiling level a valid flooding level, we need to make sure that it does not impact any outside vertex. This is made by applying the following correction

$$\omega_d \leftarrow min(\omega_d, \varphi(d)). \tag{12}$$

Our dismantling procedure can thus be expressed as follows by Algorithm 4.

Algorithm 4 Dismantling procedure from a node d of the dendrogram

```
repeat
  {Here we dismantle the links to all children of d}
  for all (p successor of d) do
    remove_link(p, d) {p is now a root independent sub-dendrogram}
    φ_p ← min(φ_p, ω(d)) {Since p is made independent of d, we should not exceed its ceiling}
    ω_p ← min(ω_p, φ_p) {We correct the ceiling of p following equation (12)}
    if (ω_p > diam(p)) then
      flood_lake(p, ω_p) {p is a lake with the final flooding level ω_p}
    else
      put_queue_to_be_explore(p) {p is an independent sub-dendrogram to be explored later}
    end if
  end for
until (is_root(d))
```

The following update

$$\varphi_p \leftarrow min(\varphi_p, \omega(d)) \tag{13}$$

corresponds to the canonical requirement of a valid flooding as expressed by equation (1). Its aim is to put the ceiling of a sub-dendrogram at the level of his parent from which it has been made independent. Once we create an independent sub-dendrogram, we immediately check if its ceiling is greater than its diameter, in that case we have a lake and the flooding level is the ceiling. Otherwise, we put that sub-dendrogram into the queue of the (independent) sub-dendrograms to be explored. The exploration process is then repeated with each sub-dendrogram taken from the queue. Figure 5 illustrates our dismantling procedure on a sample linear graph. We have dismantled from the indicated node, i.e. itself and its ancestors. Gray lines represent the disable links and nodes. Green disks correspond to final flooding levels obtained during the dismantling.

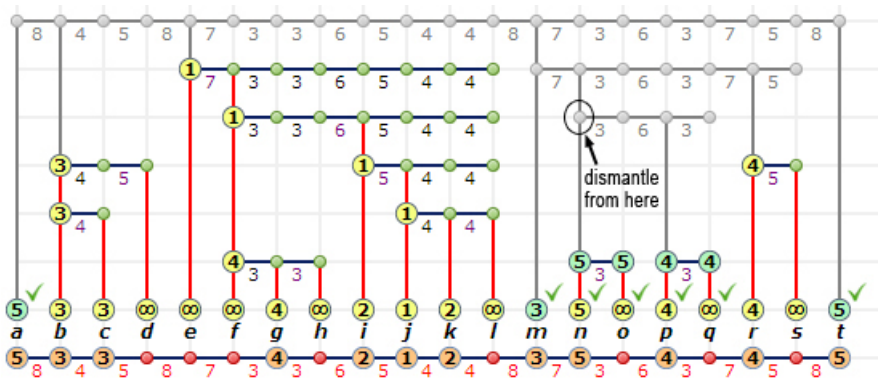


Figure 5: Illustration of the dismantling procedure

3.5 The flooding process and the whole algorithm

The pool of the independent sub-dendrograms to be treated is dynamically supplied during each individual exploration. Thus, we just need to iterate on that pool until it becomes empty. Algorithm 5 describes the process.

Algorithm 5 Flooding through the dendrogram

$Q \leftarrow \{id\_main\_dendrogram\}$ {We start with the main dendrogram}
while $(Q \neq \emptyset)$ **do**
  $d \leftarrow select\_one\_subdendrogram(Q)$ {We pick up one sub-dendrogram to be expored}
  $Q \leftarrow Q - \{d\}$
  $d \leftarrow get\_leaf\_subdendrogram(d)$ {We always explore from a leaf}
  $flood\_from(d)$ {Apply the bottom-up flooding exploration following Algorithm 3}
  {The queue Q has potentially been supplied with created orphan sub-dendrograms}
end while

Our complete algorithm can be summarized as follows:

- create the dendrogram
- propagate the ceiling values among the nodes of the dendrogram organized as a tree
- perform the flooding process by running Algorithm 5

If we need to proceed with another scenario of ceiling levels on the same graph, then we just need to perform the last two steps on its dendrogram. Moreover, since we create independent sub-dendrograms to be explored, we get a natural way for parallelism. However, we need to take care of load balancing and the overhead of inadequate thread creations. We will explain our approach to efficiently deal with this parallelism in the next section, which starts with general implementation details and programming tricks.

4 Implementation

4.1 Data structures and programming methodology

4.1.1 Data structure

First, the leaves of our dendrogram are airs of (connected) vertices, instead of singletons as it should be. Thereby, since we also consider a pairwise merging in our bottom-up agglomerative algorithm to build the dendrogram, our structure is a binary tree, where each of the two children

of a given node is either a node of the dendrogram or simply a vertex. The choice of a binary tree simplifies both the information storage and the exploration process. Moreover, we chose to have the support of each dendrogram represented by its vertex with the smallest id. The flooding of a sub-dendrogram is performed through this representative vertex of its support (any other choice would have been valid too). Thus, a node of our dendrogram is represented by the following data structure:

```
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 dendrogram or a vertex
   char is_leaf_right;  // tells if the right child is a dendrogram or a vertex
   float diam;          // diameter of the dendrogram
   float min_outedge;   // the outgoing edge with the minimum cost
   int size;            // number of vertices in the support of this dendrogram
   float ceil;          // the ceiling level (provided as input)
   float flood;         // the flooding level (to be computed)
   int smallest_vertex; // we keep the id of the vertex with the smallest ceiling
   int pred;            // the predecessor of this dendrogram (its parent)
   int child_left;      // we create dendrogram by fusing two subdendrograms (left, right)
   int child_right;     // right child
} dendro;
```

4.2 Foundations of our implementation

Our implementation relies on the following basis

(1) The number of nodes in our dendrogram is bounded by the number of edges of the graph. Thus, we can use a fixed size array of `dendro` (bounded by $|V|$) to store all nodes of the dendrogram.

(2) Each sub-dendrogram is created from a single edge (a, b) of the graph by merging the roots of the two sub-dendrograms containing $\{a\}$ and $\{b\}$. Thus, each node dendrogram has two children referenced by (`child_left` and `_right`. So, our structure is a binary tree.

(3) We technically consider the *leaves* of our dendrogram as the nodes containing two (neighbor) vertices only. In that case, `child_left` and `child_right` are the indexes of these two vertices (within the global array of the vertices), instead of the indexes to its child sub-dendrograms.

(4) Since we visit the edges of the graph in an ascending order of their weights, the diameter of a newly created sub-dendrogram is the weight of the edge used for the corresponding merging. Later on, each time we select an edge whose the vertices belong to the same dendrogram (*inside* edge), we just set the diameter of that sub-dendrogram to the weight of the selected edge (since it is necessarily higher). Other basic information are updated in the same occasion.

(5) We manage the connectivity of the graph as follows. We assume that the number of neighbors for each vertex is bounded by a given constant (this is a fixed value for regular stencil applications). This constant is either provided as a parameter, or is calculated when reading the input graph. With N vertices and an indegree bound K, we use an array of length NK to store the neighborhood information, each vertex being associated to the corresponding chunk of length K. Thereby, we get a mechanism to seek basic information like the associated vertices of a given edge or the neighborhood of a given vertex.

(6) Since each sub-dendrogram is explored in a bottom-up way from any vertex of its support following the `pred` field, we choose to proceed with the vertex with the smallest *id* stored in the `smallest_vertex` field.

(7) The smallest outgoing edge of a sub-dendrogram is the one used when creating its parent dendrogram, because edges are selected in an ascending order of their weights.

(8) Dendrogram based graph flooding algorithm has several advantages. Among them, we point that

- it can be used to generate information from a local input (e.g. flooding from a vertex)
- it naturally exhibits parallelism (independent sub-dendrograms from the dismantling)
- other algorithms are global, thus will always process with and for the whole graph

4.2.1 Programming aspects

In addition to the bais of our implementation as previously reported, we now describe some important programming strategies.

For a given vertex v of the graph, we need to be able to get the corresponding root sub-dendrogram (the one for which v belongs to the support). This can be done by starting from the leaf sub-dendrogram containing v, and then move from parent to parent (using the `pred` field) until we reach a root sub-dendrogram. The complexity of this procedure is proportional to the increasing height of the whole dendrogram so far constructed. However, since this is repeated each time we have to merge two sub-dendrograms from a selected edge, it is thus vital to make it as efficient as possible. For this purpose, we can observe that, for a vertex that we are visiting for the second time, we can start our way to the root from the previously identified root instead of restarting from a leaf. Such an incremental processing significantly improves the time of the construction of the dendrogram, which is a good point as this is the most heavy part of our algorithm, especially with dense graphs.

4.3 Parallelisation of our method

Regarding the parallelisation of our algorithm, we only consider shared memory multiprocessing. Technically, we focus on a multi-threaded implementation for multicore machines. We have so far claim that we can naturally consider parallelism when flooding a graph through its dendrogram, because we create independent sub-dendrograms that can thus be explored in parallel. This is true indeed, but we need to take care about the potential load unbalance of the scheduling, which is dynamic in this case with an unpredictable flow of (independent) tasks. Our solution is to consider a queue, where we store the references to the (independent) sub-dendrograms isolated, which are then explored in a round robbing distribution by the threads. So, each threads has to continuously pick up a sub-dendrogram from the pool and explore it, putting the corresponding isolated sub-dendrograms into the common pool. Figure 6 illustrates our scheduling mechanism, which is implemented using the *mutual exclusion* feature for the section where the pool is provisioned.

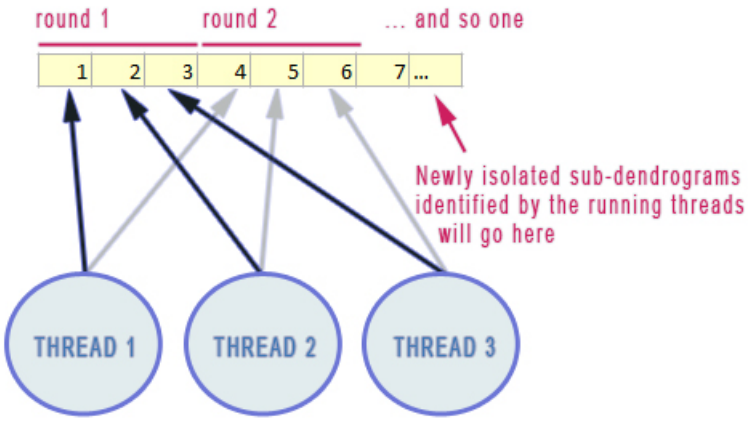


Figure 6: Parallel scheduling

5 Validation and performances

We present the results of our benchmark, performed on an Intel Core i7-2600 processor with 4 cores and up to 8 threads (HyperTreading). Our graphes are randomly generated from two parameters: the number of vertices ($|X|$) and the maximum indegree per vertex (c). $|V|$ is the number of edges ($|V| \leq c|X|$) and h_tree is the height of the constructed dendrogram (the longest path from a leaf to the root). For the timings, the total running time of our algorithm is decomposed into two parts: the time to build the dendrogram and the time to calculate the flooding levels from the dendrogram. We compare our solution with a reasonably optimised implementation of the Dijkstra algorithm by ourself.

5.1 Sequential version

We discuss both the characteristics of the dendrogram and the performances. Table 1 (resp. 2) provides our benchmark results on graphs with a fixed (resp. increasing) number of vertices and an increasing number of edges.

					Execution time(s)				
	$\|X\|$	$\|V\|$	c	h_tree	Dendrogram	Flood	Total	Dijktra	↑
1	10000	15024	5	9	0.0073	0.001050	0.0083	0.0406	4.9
2	10000	27667	10	33	0.0072	0.000427	0.0076	0.0578	7.6
3	10000	40192	15	168	0.0166	0.000411	0.0170	0.0742	4.4
4	10000	52672	20	316	0.0355	0.000419	0.0359	0.0863	2.4
5	10000	65138	25	861	0.0480	0.000387	0.0484	0.0946	2.0
6	10000	76676	30	1205	0.0641	0.000381	0.0645	0.1027	1.6

Table 1: Performances of our algorithm on a graph with different densities

For the dendrogram, we see that its height is always moderate even if the structure is a binary tree. In any case, we outperform Dijktra by factor 2 or more. In addition, the time for building the dendrogram is clearly predominant, and the flooding step is noticeably fast. This last point is a key aspect for our contribution. Indeed, if we have to flood the same graph using several sets of ceiling values, our algorithm is indubitably the best choice. We now see in Table 2 what happens with bigger graphs, both in size and density.

					Execution time(s)				
	$\|X\|$	$\|V\|$	c	h_tree	Dendrogram	Flood	Total	Dijktra	↑
1	10000	15024	5	9	0.0073	0.001040	0.0083	0.0404	4.9
2	10000	27667	10	33	0.0072	0.000425	0.0076	0.0577	7.6
3	20000	29887	5	16	0.0064	0.000908	0.0073	0.1969	26.8
4	20000	54510	10	50	0.0156	0.000885	0.0165	0.3039	18.4
5	30000	44881	5	17	0.0099	0.001405	0.0113	0.4802	42.4
6	30000	82161	10	63	0.0256	0.001513	0.0271	0.7515	27.7
7	40000	60167	5	13	0.0137	0.002007	0.0157	0.9264	59.2
8	40000	110092	10	38	0.0368	0.002204	0.0390	1.4696	37.7
9	50000	74884	5	10	0.0176	0.002787	0.0203	1.5183	74.6
10	50000	137184	10	46	0.0477	0.003016	0.0507	2.6930	53.1
11	60000	89942	5	11	0.0220	0.003674	0.0257	2.4953	97.2
12	60000	165030	10	85	0.0638	0.004175	0.0680	3.8444	56.6
13	70000	104838	5	9	0.0273	0.004662	0.0319	3.6343	113.8
14	80000	119923	5	17	0.0342	0.005967	0.0402	4.9978	124.3
15	80000	219919	10	44	0.0914	0.006746	0.0981	7.9271	80.8
16	90000	134790	5	15	0.0394	0.007540	0.0470	6.5709	139.9
17	90000	247540	10	40	0.1027	0.008173	0.1109	10.3673	93.5
18	100000	150168	5	8	0.0458	0.009133	0.0549	8.4721	154.3
19	100000	275448	10	33	0.1138	0.009807	0.1236	15.0032	121.4

Table 2: Performances of our algorithm on a graph with various sizes and densities

We observe that the height of the dendrogram is not really affected by the number of vertices, but slightly by the density. One could imagine a preprocessing which eliminates the edges whose weight does not influence the flooding levels. Once again, we see that the overall computation is dominated by the construction of the dendrogram and we significantly outperform Dijkstra algorithm.

Now, let us examine the results of our parallel benchmark.

5.2 Parallel version

Here, we mainly focus on the scalability. We emphasize on the fact the only part that is parallelized is the flooding step, because of its natural potential of parallelism. Table 3 displays our timings and speedups with various graphs and 1 to 4 threads.

	$\|X\|$	$\|V\|$	nb_threads	time(s)	speedup
1	3 000 000	3 749 136	1	0.369	1.00
2	3 000 000	3 749 136	2	0.214	1.73
3	3 000 000	3 749 136	3	0.167	2.21
4	3 000 000	3 749 136	4	0.172	2.15
5	5 000 000	6 250 057	1	0.639	1.00
6	5 000 000	6 250 057	2	0.374	1.71
7	5 000 000	6 250 057	3	0.281	2.27
8	5 000 000	6 250 057	4	0.312	2.05
9	7 000 000	8 747 179	1	0.920	1.00
10	7 000 000	8 747 179	2	0.536	1.72
11	7 000 000	8 747 179	3	0.453	2.03
12	7 000 000	8 747 179	4	0.450	2.05
13	9 000 000	11 249 740	1	1.241	1.00
14	9 000 000	11 249 740	2	0.709	1.75
15	9 000 000	11 249 740	3	0.535	2.32
16	9 000 000	11 249 740	4	0.609	2.04
17	10 000 000	12 500 809	1	1.404	1.00
18	10 000 000	12 500 809	2	0.802	1.75
19	10 000 000	12 500 809	3	0.603	2.33
20	10 000 000	12 500 809	4	0.550	2.55
21	20 000 000	24 996 258	1	3.105	1.00
22	20 000 000	24 996 258	2	1.762	1.76
23	20 000 000	24 996 258	3	1.438	2.16
24	20 000 000	24 996 258	4	1.171	2.65

Table 3: Scalability of our algorithm on a quad-core machine

Our scheduling strategy really improves the load balance from the point of view of the set of independent sub-dendrograms, not necessarily from the number of vertices to be examined. This second aspects is difficult to predict, as it really depends on both the structure of graph and the distribution of the ceiling levels. Nevertheless, our speedups look good and promising.

6 Conclusion

Classical algorithms for dominated graph flooding are no longer sufficient for real-time processing when it comes to large and dense graphs. Moreover, when we have to flood the same graph with different set of ceiling values, it is common to restart from scratch. In this paper, we provide an algorithm, which is competitive and well suited for multiple flooding. At this point of our work, we see two perspectives. The first one is about how to get the dendrogram of a modified version of a given graph by just adapting its original dendrogram. The second one is related to the parallelization. We admit that, from a global point of view, parallelizing the construction of the dendrogram would be more rewarding than parallelizing the flooding step. Thus, investigating on a parallel version of our algorithm for constructing the dendrogram is to be done.

References

[1] Fernand Meyer, *Flooding edge or node weighted graphs*, eprint arXiv:1305.5756, 05/2013.

[2] D. C. Dracopoulos, *Neural Robot Path Planning: The Maze Problem*, Neural Computing & Applications, (7)115-120, 1998.

[3] K. Erciyes, *Distributed Graph Algorithms for Computer Networks*, ISBN: 978-1-4471-5172-2, Springer, 2013.

[4] C. Berge, *Theorie des graphes et ses applications*, Dunod, Paris, 1958.

[5] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik 1, 269-271, 1959.

[6] B.T.M. Roerdink and A. Meijster, *The Watershed Transform: Definitions, Algorithms and Parallelization Strategies*, Fundamenta Informaticae (41)187-228, 2001. ...