Dendrogram-based Algorithm for Dominated Graph Flooding

Claude Tadonki

Joint work with

Fernand Meyer and François Irigoin

Centre de **R**echerche en **I**nformatique - **C**entre de **M**orphologie **M**athématique

Mines ParisTech - FRANCE

International Conference on Computational Science - ICCS 2014

10-12 June, 2014

CAIRNS, AUSTRALIA

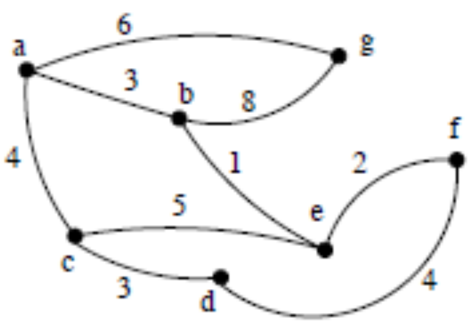
The problem

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* <u>*maximal*</u> *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 $\boxed{1}$ illustrates the flooding of a weighted graph with 7 vertices and 9 edges.*



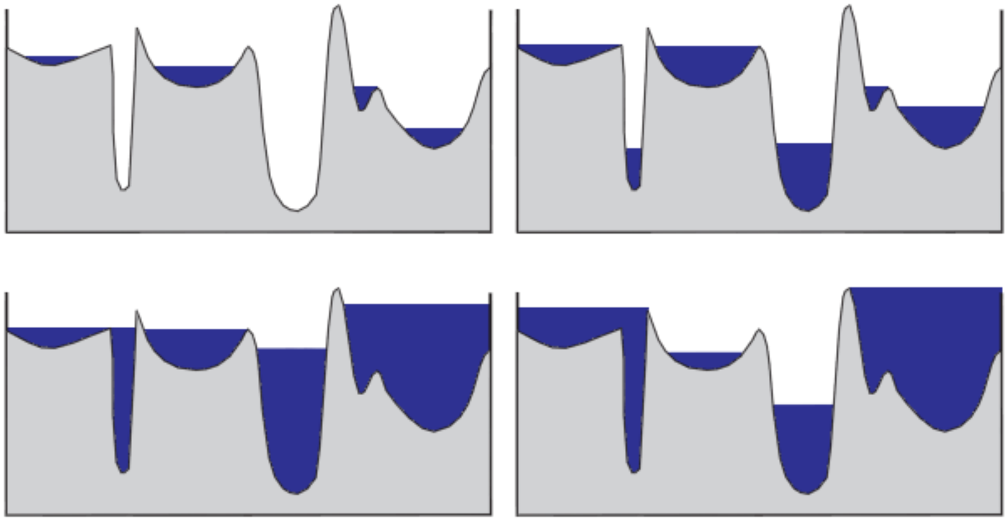
With the ceiling $\quad \omega(\{a; b; c; d; e; f; g\}) = \{9; 3; 6; 7; 9; 4; 5\}$
we have the flooding $\quad \tau(\{a; b; c; d; e; f; g\}) = \{3; 3; 4; 4; 3; 3; 5\}$

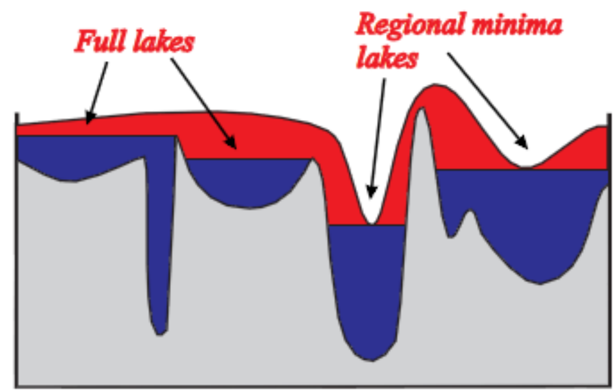
Figure 1: Sample flooding

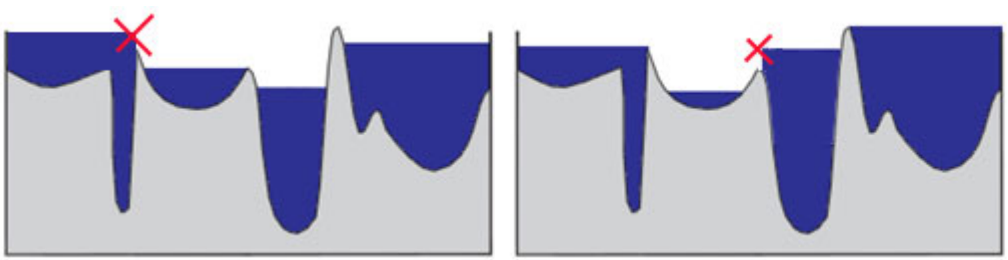
Two classical algorithms exist in the litterature

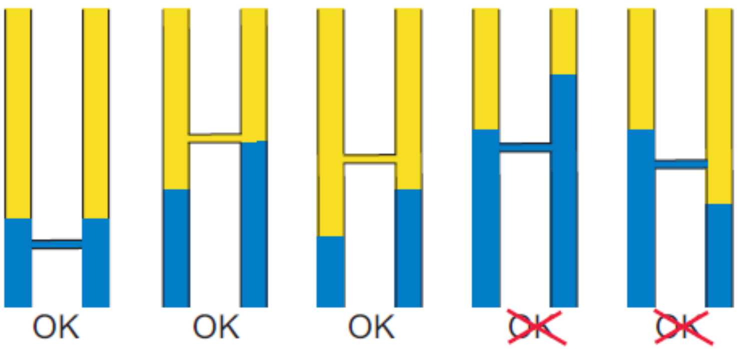
- **Dijkstra** algorithm is <u>greedy</u>
- **Berge** algorithm is <u>dynamical programming</u>

Illustrations









- We study a <u>decomposition</u> algorithm based on the structure of <u>dendrogram</u>

Definition. *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) $\forall U \in E, (\underbrace{\{V \in E | V \supseteq U\}}_{\text{elements of the dendrogram}}, \subseteq)$ *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

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

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

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

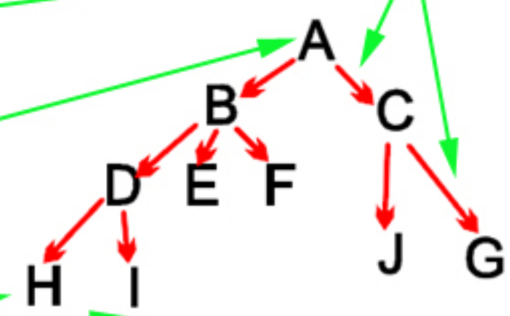
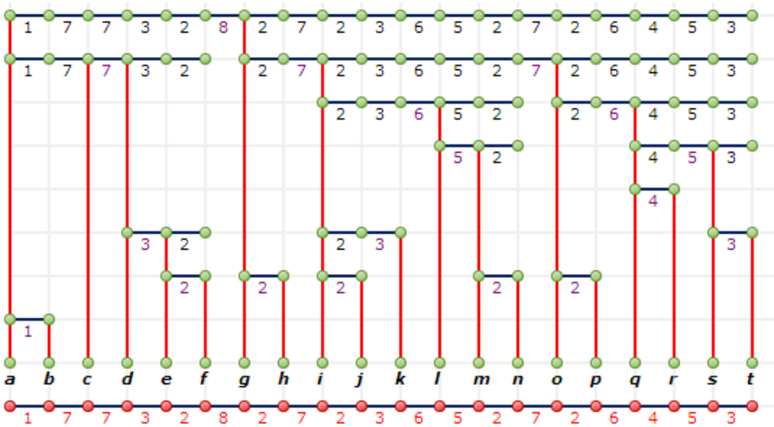


Figure: A dendrogram



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**.

Theses 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).

Live demo !!!

Other algorithms are global, so will always process with and for the whole graph,

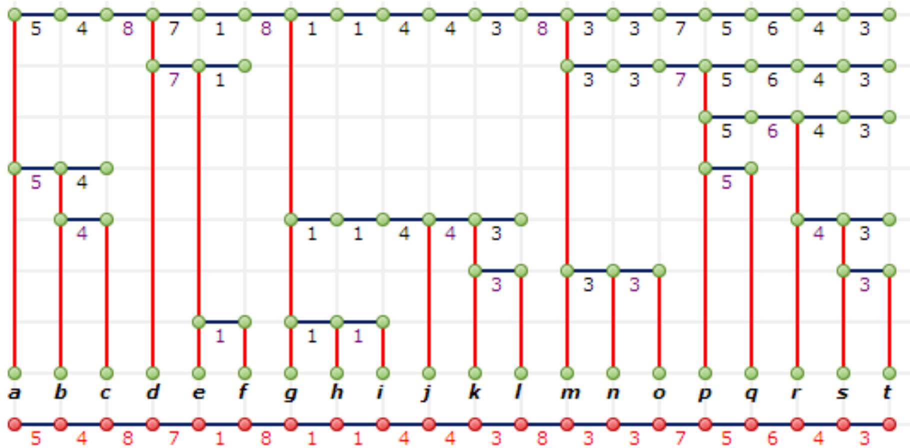
```
D ← ∅
S ← {all edges (u, v, w) of the graph}
while(S ≠ ∅ ){
   //we select the edge with minimum cost
   (u, v) ← min_w(S); //we can sort the list of edges and select on top
   //we remove that edge from S
   S ← S - {(u, v)};
   //we get the id the of the root subdendrogram containing u
   d1 ← 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 subdendrogam d1 and d2 to form a new one (parent)
   if(d1 ≠ d2)
     D ← D ∪ d;
     d ← dendrogram_merge(d1, d2);
   endif
}
```

Technical remarks

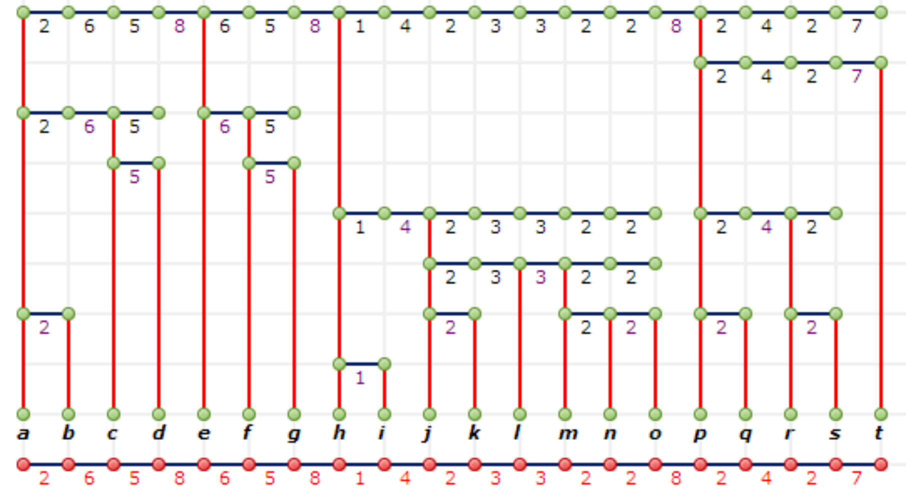
- *dendrogram_singleton*({u}) creates a subdendrogram with singleton {u}
- If *u* and *v* belong to an existing subdendrogram, then we **avoid recreating** it
- *id_root_subdendrogram*(u) is obtained by climbing from *dendrogram_singleton*({u}) to the maximal subdendrogram following the parent (successor) relation.

This function is the most time consuming of the construction. Its global impact depends on the depth or height of the dendrogram tree.

Properties of the dendrogram that impact on performances



(a) Dendrogram constructed from a linear graph
Depth = 7



(b) Dendrogram constructed from a linear graph
Depth = 8

Technical remarks

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

- 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 containg 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 twe start the flooding process
d ← leaf_subdendrogram(x)
//we go up while the ceiling is still greather that the diameter
while((!is_root(d))&&(ceil(d) > diam(d))) d ← pred(d);
//we have reached a root and still get a ceiling greather that the diameter
//we set the definitive flooding values of this subdendrogram to ceil(d)
if(ceil(d) > diam(d)) set_flooding_level(d, ceil(d));
else dismantle_ancestors(d, ceil(d));
```

flood_from_vertex (x)

Technical remarks

- The *dismantling* process breaks the (sub)dendrogram into independent root subdendrograms.
- Newly created root subdendrograms during the *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
FIFO_root_to_explore[0] ← lastly_created_subdendrogram
nb_root_to_explore ← 1
for(i = 0; i < nb_root_to_explore; i++)
  flood_from_vertex(get_vertex_with_min_id(FIFO_root_to_explore[i]);
```

- The FIFO will be populated during the dismantling processes and `nb_root_to_explore` will be incremented accordingly.

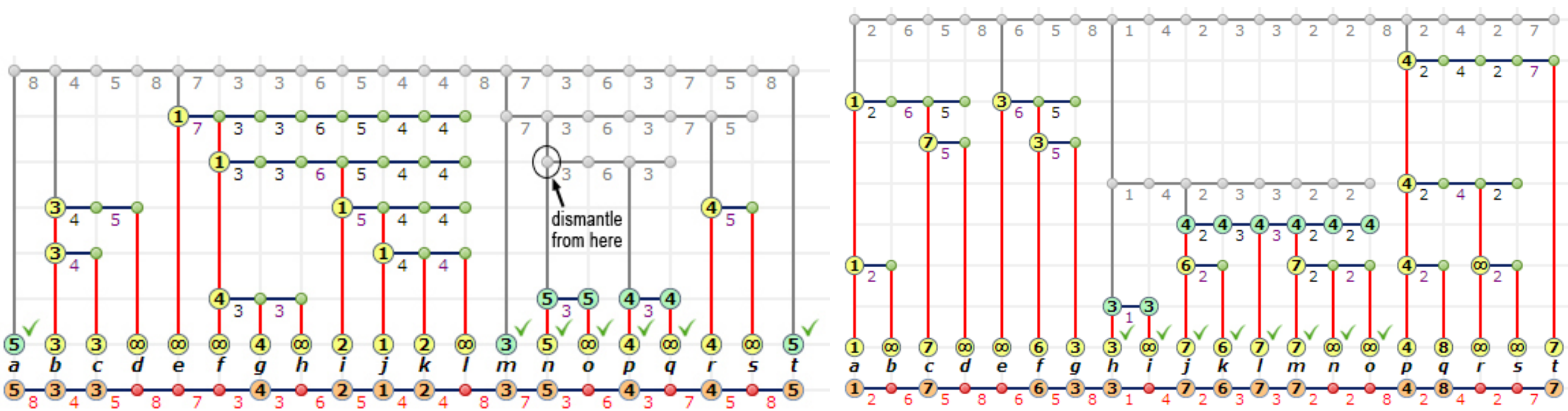
In which order should we explore the sub-dendrograms ? Does this impact on the decomposition ? Perf ?

```
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.



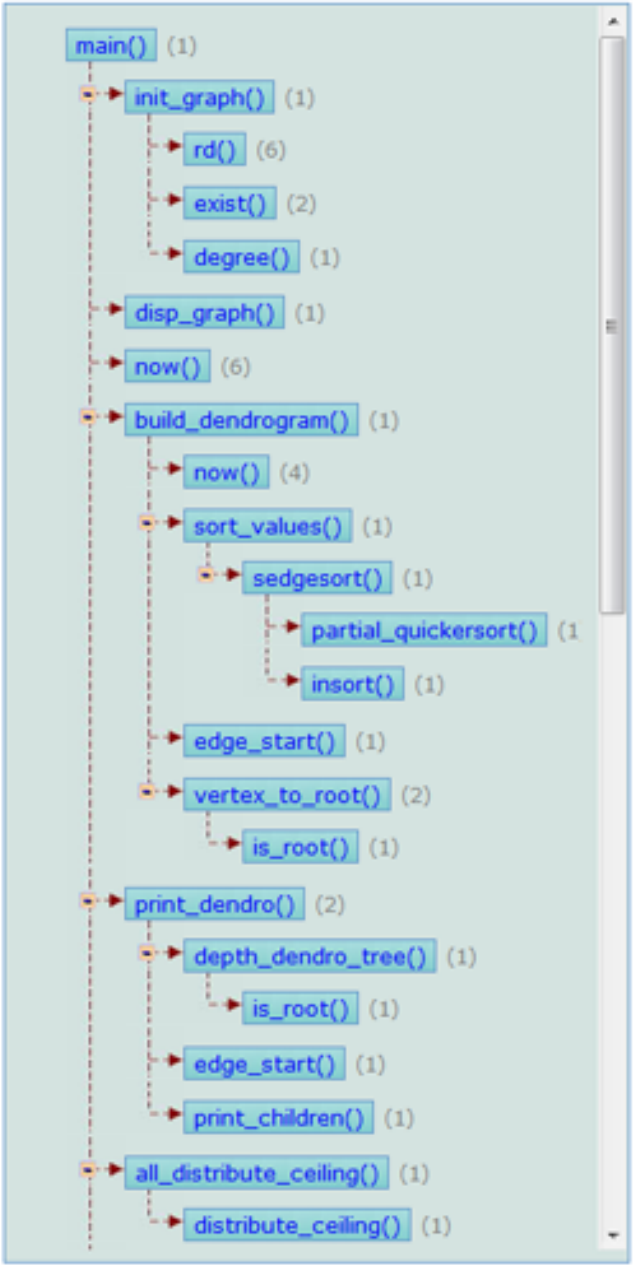
Main data structures

```c
typedef struct
{
   int edge_id;          // the id of the edge used to cerate 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;
```

```c
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;
```

General statistics		Number of lines	
Subfolders	1	Total number of code lines	684
Total number of files	1 file(s)	Total number of comment lines	128
Total number of lines	916 line(s)	Total number of mixed lines	60
Total size	34 Kbyte(s)	Total number of blank lines	104
Number of routines		**Extremal routines sizes**	
Total number of routines	36 routine(s)	Biggest routine size (lines)	96
Total number of routine calls	65	Smallest routine size (lines)	4
		Average routine size (lines)	23
Extremal number of variables in routines		**Number of variables**	
Maximum number of local variables	29	Total number of local variables (all occurrences)	367
Minimum number of local variables	2	Total number of local variables (unique occurrences)	103
Average number of local variables	10		

Dependences graph and sample code (*building the dendrogram*)



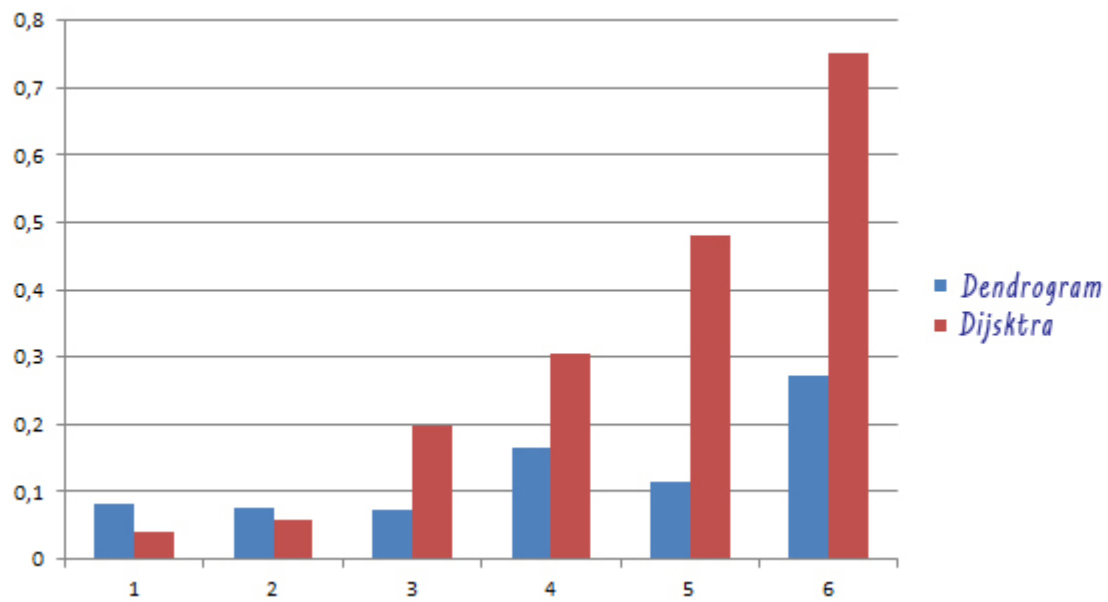

```c
void build_dendrogram(graph g){
   int *rank ;
   int i , j , a , b ;
   int p , q ;
   dendro d ;
   double s , t ; s = 0 ;
   int *roots = malloc(g.nb_nodes*sizeof(int)) ;
   rank = malloc(g.nb_edges*sizeof(int)) ;
   all_dendro = malloc(g.nb_edges*sizeof(dendro)) ;
   starting_dendro = malloc(g.nb_nodes*sizeof(dendro)) ;
   for(i = 0 ; i < g.nb_nodes ; i ++ ) {starting_dendro[i] = - 1 ; roots[i] =- 1 ; }
   t = now() ;
   sort_values(rank , g) ; nb_dendro = 0 ;
   printf("Time for sorting edges = %f\n" ,(now() - t)) ;
   for(i = 0 ; i < g.nb_edges ; i ++ ){
      j = rank[i] ; a = edge_start(j , g) ; b = g.neighbors[j] ;   // current edge is(a , b)
      t = now() ;
      p = vertex_to_root(a , roots[a] , g) ; roots[a] = p ;
      q = vertex_to_root(b , roots[b] , g) ; roots[b] = q ;
      s +=(now() - t) ;
      if((p ==- 1)||(q ==- 1)||(p != q)){
         // We now create the dendrogram which merges the above ones
         d.id = j ;   // the rank of the edge used to create this dendrogram
         d.rootlev = g.values[rank[g.nb_edges - 1]] + 1 ;   // max of the edge values + 1 ==> INFINITY
         d.diam = g.values[j] ;   // the curent edge is necessary the diameter of this dendrogram
         d.ceil = 0 ;   // not really defined
         d.flood = 0 ;   // not really defined
         d.pred = - 1 ;   // For the moment
         d.size = 0 ;   // For the moment
         d.min_outedge = d.rootlev ;   // will be update next time we visit an outgoing edge
         d.smallest_vertex = g.nb_nodes ;   // INFINITY for the moment
         if(p ==- 1) {
            d.child_left = a ; d.is_leaf_left = '1' ; starting_dendro[a] = nb_dendro ; d.size += 1 ;
            if((vertices_min_outedge[a] ==- 1)||(g.values[j] < vertices_min_outedge[a])) vertices_min_outedge[a] = g.values[j] ;
         }
         else{
            d.child_left = p ; d.is_leaf_left = '0' ; all_dendro[p].pred = nb_dendro ; d.size += all_dendro[p].size ;
            if(g.values[j] < all_dendro[p].min_outedge) all_dendro[p].min_outedge = g.values[j] ;
         }
         if(q ==- 1) {
            d.child_right = b ; d.is_leaf_right = '1' ; starting_dendro[b] = nb_dendro ; d.size += 1 ;
            if((vertices_min_outedge[b] ==- 1)||(g.values[j] < vertices_min_outedge[b])) vertices_min_outedge[b] = g.values[j] ;
         }
         else{
            d.child_right = q ; d.is_leaf_right = '0' ; all_dendro[q].pred = nb_dendro ; d.size += all_dendro[q].size ;
            if(g.values[j] < all_dendro[q].min_outedge) all_dendro[q].min_outedge = g.values[j] ;
         }
         all_dendro[nb_dendro] = d ; nb_dendro ++;
      }
      else{
         all_dendro[p].diam = g.values[j] ;
         /*all_dendro[p].id = j ;(not need to change this)*/
      }
      /* The sub - dendrogram existed , but it's diameter should be updated using the current edge , necessary bigger*/

   }
   free(roots) ;
   printf("S = %f\n" , s) ;
}
```

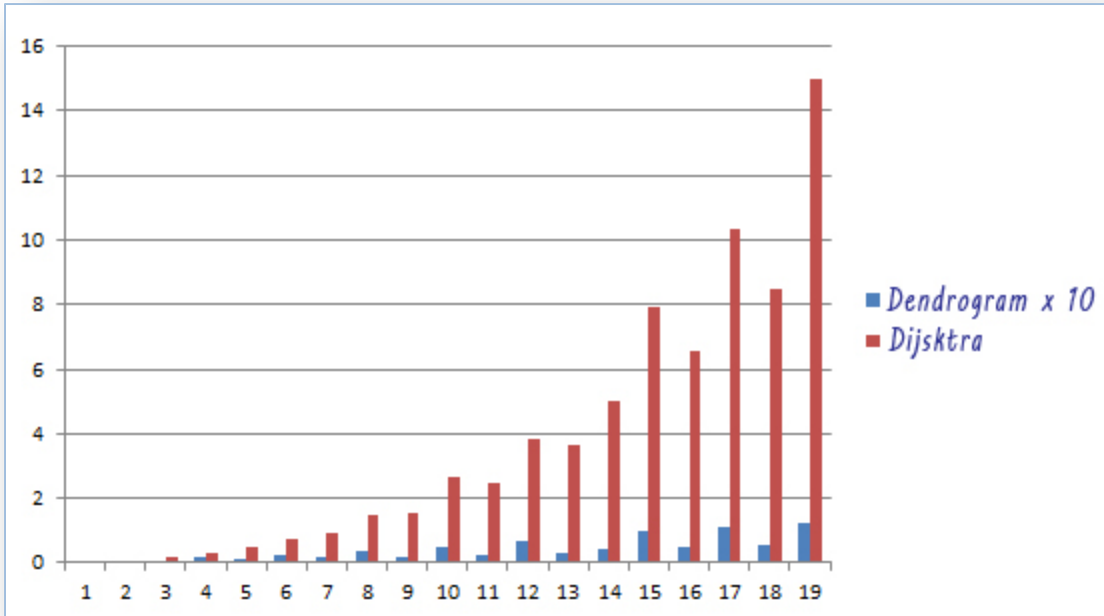
	$\lvert X\rvert$	$\lvert V\rvert$	c	h_tree	Execution time(s) Dendrogram	Flood	Total	Dijktra	↑
1	10000	15024	5	9	0.0073	0.001050	0.0083	0.0406	4.9
2	10000	27667	10	33	0.0072	0.000427	0.0076	0.0578	7.6
3	10000	40192	15	168	0.0166	0.000411	0.0170	0.0742	4.4
4	10000	52672	20	316	0.0355	0.000419	0.0359	0.0863	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

- The height of the dendrogram is moderate
- Building the dendrogram predominates
- We outperform Dijkstra by factor > 2



Performance on some random graphs (timings are in seconds)



					Execution time(s)				
					Dendrogram	Flood	Total	Dijktra	↑
					0.0073	0.001040	0.0083	0.0404	4.9
					0.0072	0.000425	0.0076	0.0577	7.6
					0.0064	0.000908	0.0073	0.1969	26.8
					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

- 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

Performance on an image neighborhoods graph (Fontainebleau Palace)

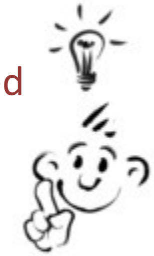
Neighborhood graph generated by **morph-m**

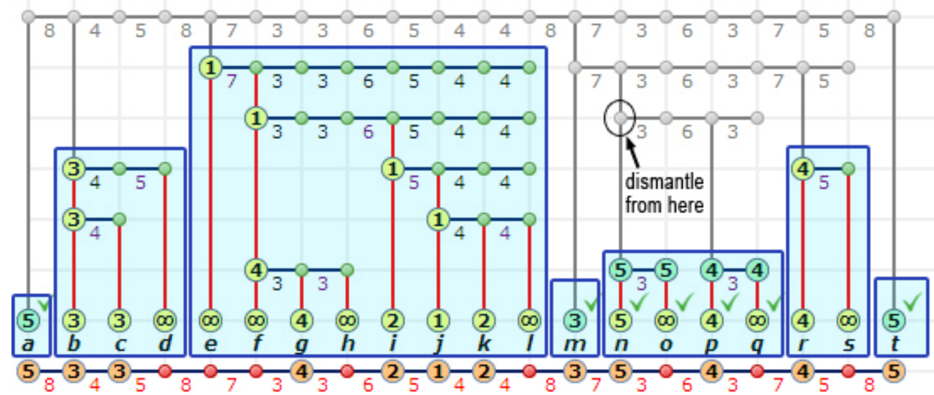
Number of nodes: 24 532
Number of edges: 96 138

Flooding values computed by the dendrogram-based algorithm

Constructing the dendrogram: 1.737 s
Flooding process: 0.002 s
Whole algorithm: 1.739 s
Basic Dijsktra algorithm: 59.042 s

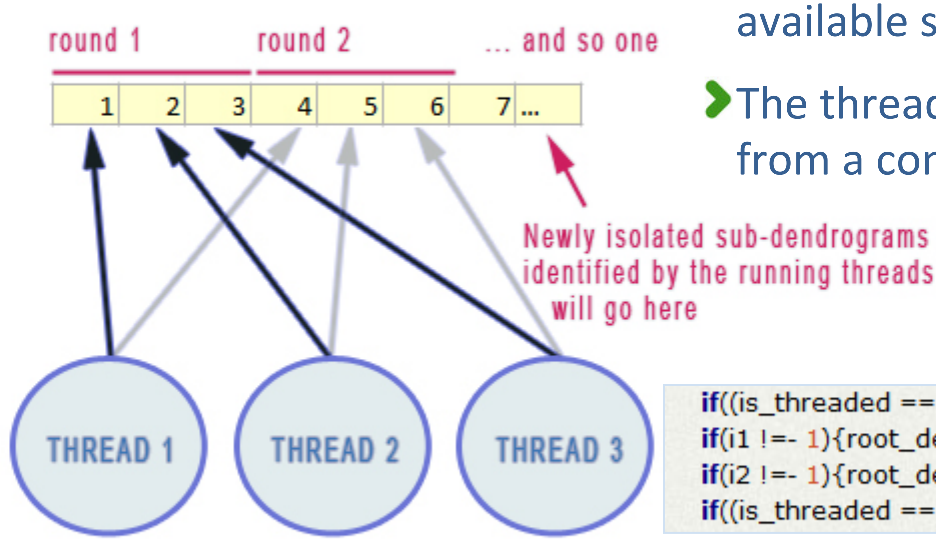
- Dismantling isolates independent subdendrograms which can be explored in parallel
- The flooding step can thus be parallelized
- Care about threads creation overhead
- Contend the effect of unbalanced load



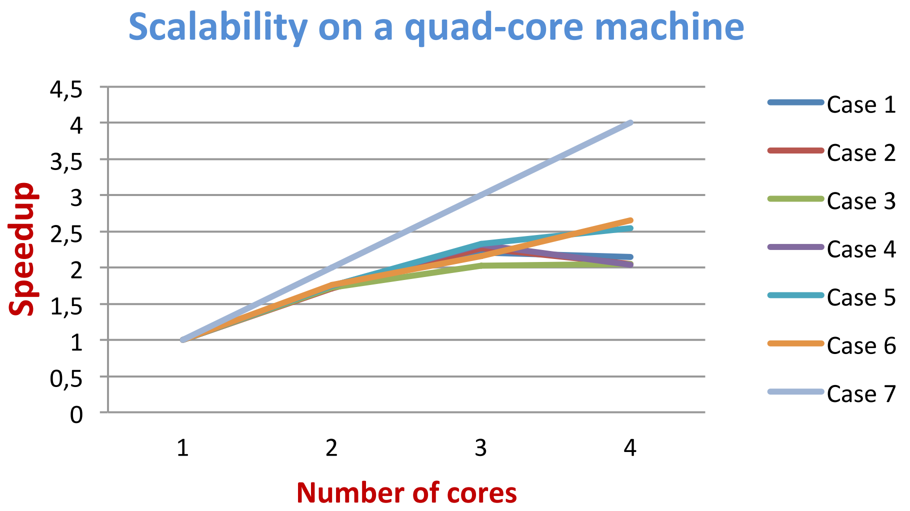


OUR SOLUTION

- 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.




```
if((is_threaded == 1)&&((i1 !=- 1)||(i2 !=- 1))) pthread_mutex_lock(&mutex_dendro_id2) ;
if(i1 !=- 1){root_dendro[nb_root_dendro] = i1 ; nb_root_dendro ++; }
if(i2 !=- 1){root_dendro[nb_root_dendro] = i2 ; nb_root_dendro ++; }
if((is_threaded == 1)&&((i1 !=- 1)||(i2 !=- 1))) pthread_mutex_unlock(&mutex_dendro_id2) ;
```



	$\|X\|$	$\|V\|$	P	T	σ
1	3M	3 749 136	1	0.369	1.00
2	3M	3 749 136	2	0.214	1.73
3	3M	3 749 136	3	0.167	2.21
4	3M	3 749 136	4	0.172	2.15
5	5M	6 250 057	1	0.639	1.00
6	5M	6 250 057	2	0.374	1.71
7	5M	6 250 057	3	0.281	2.27
8	5M	6 250 057	4	0.312	2.05
9	7M	8 747 179	1	0.920	1.00
10	7M	8 747 179	2	0.536	1.72
11	7M	8 747 179	3	0.453	2.03
12	7M	8 747 179	4	0.450	2.05

	$\|X\|$	$\|V\|$	P	T	σ
13	9M	11 249 740	1	1.241	1.00
14	9M	11 249 740	2	0.709	1.75
15	9M	11 249 740	3	0.535	2.32
16	9M	11 249 740	4	0.609	2.04
17	10M	12 500 809	1	1.404	1.00
18	10M	12 500 809	2	0.802	1.75
19	10M	12 500 809	3	0.603	2.33
20	10M	12 500 809	4	0.550	2.55
21	20M	24 996 258	1	3.105	1.00
22	20M	24 996 258	2	1.762	1.76
23	20M	24 996 258	3	1.438	2.16
24	20M	24 996 258	4	1.171	2.65

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

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 decaration
  *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
  graph_flood_dendro(N, L, X, Y, W, C, F,1);
  everywhere as needed in your program
- The typical compilation line is
  gcc -o your_executable graph_flood_dendro.o your_code.c -lpthread

The object file mainly contains the function descibed below

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

Simulator http://www.cri.ensmp.fr/TIMC/dendrogram/index.htm

Results & code http://www.cri.ensmp.fr/TIMC/dendrogram/flooding.htm

Report http://www.cri.ensmp.fr/classement/doc/E-329.pdf

- 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