Fast computation of distances in a tree

Marc Pierrot Deseilligny

Fast computation of distances in a tree

Marc Pierrot Deseilligny[1]

[1]Laboratoire LaSTIG, Université Gustave Eiffel

May 5, 2020

Abstract

Computation of distances between two submits of a tree is an operation that occurs in some pattern recognition problem. When this operation has to be done thousands of times on millions of trees, the linear standard algorithms in $\mathcal{O}(N)$ for each pair may be a bottleneck to the global computation.

This note present recursive spliting method with a complexity of $\mathcal{O}(log(N))$ on each pair in worst case, and $\mathcal{O}(1)$ in average on all pair, once a pre-computation $\mathcal{O}(Nlog(N))$ has been done on the whole tree. A commented `C++` implementation is published as a companion to this note.

1 Motivation

Computation of distance between two submits of a graph is a basic well known problem of graph theory for which there exist efficient methods. This is even easier in the case where the graph is known to be tree or a forest (i.e. no cycle), in this case the time of basic algorithm are proportionnal to N where N is the number of submits. However there are cases where this operation has to be done a huge number of time and for which these linear performance may be limitative.

This problem can occur for example in photogrammetry in the computation of global orientation from a set of relative orientations (see for example [Govindu, Venu 2006]) . We give a short description, illustrated by figure 1 :

— we want to orientate a set of N images ;

— for certain pairs of images I, J we have computed the relative orientation $r_{i,j}$ of I relatively to J using some photogtrammetric method after computing tie-points;

— now we want to compute a global solution correspond to equation 1.

$$R_i = r_{i,j} R_J \tag{1}$$

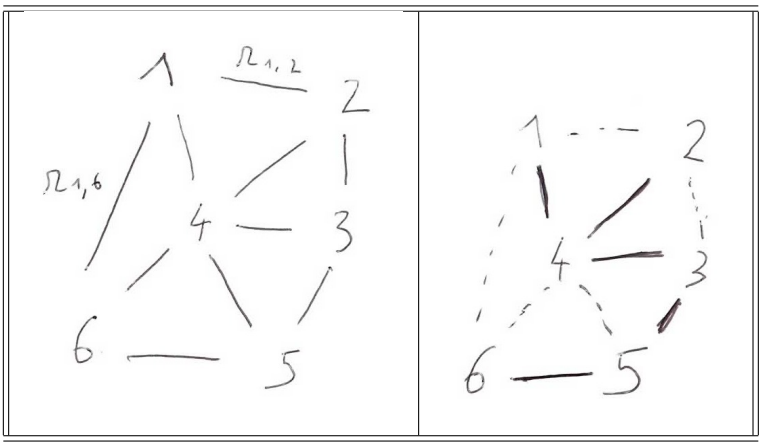

Figure 1 – A graph with relative rotations; a random tree extracted from previous graph

$$E = \sum (R_i - r_{i,j} R_J)^2 \tag{2}$$

As there is more constraints than unkown, the standard method is to minimize some criterion E like equation 2. Before doing that, we need to get an initial solution and remove the outlier. A possible way to do it this one :

— generate a "huge" quantity of random tree;
— for each tree , set arbitrarily $R_1 = Id$ and generate by propagation the unique solution satisfying 1;
— use equation 1 to compute by accumulation the reliable $r_{i,j}$.

Suppose we generate M tree, and note R_i^k solution tree k, with $k \in [1, M]$, a basic way to compute the reliability/quality $Q_{i,j}$ of $r_{i,j}$ is :

$$Q_{i,j} = \frac{\sum_{k=1}^{M} |R_i^k - r_{i,j} R_j^k|}{M} \tag{3}$$

However we see that formula 3 is not completely satisfying :

— when $r_{i,j}$ belongs to the tree , obviously residual of equation 1 is zero, but this has no signification;
— more generally, the highest is the distance between i and j in the tree, the longest is the chain of propagation and the highest is naturally expected to be the residual of equation 1;
— for example in figure 1, the distance between R_1 and R_6 is 4, and this reflect that the chain that goes from one to the other is $r_{1,4} \rightarrow r_{4,3} \rightarrow r_{3,5} \rightarrow r_{5,6}$;

So the formula of equation should be replaced by another one taking into account the distance in the tree; for example, something like equation 4 :

$$Q_{i,j} = \sum \frac{|R_i^k - r_{i,j}^k R_J^k|}{\sqrt{D_{i,j}^k - 1}} \tag{4}$$

There is much more to say, and formula 4 should probably need more attention. However the fact is that to test formula like 4 we need a way to compute for a given tree the distance $D_{i,j}^k$ of many pair of submits inside this tree.

In this paper we present an efficient method with the following performance :

— pre-computation in $\mathcal{O}(Nlog(N))$;
— for each pair S_1, S_2 , the cost of computation is $\mathcal{O}(log(N))$ in the worst case ;
— the cost of computation on all pair is $\mathcal{O}(N^2)$, so in average the cost for one pair is in $\mathcal{O}(1)$.

We give also an implementation in `C++` of the algorithm in a single file that aims to be easily integrated in user code.

2 Recursive split

Figure 2 illustrate the notion used for the computation :

— let A be a any submit (we will see in next section how chosing it);
— starting from A we can easily compute for each submit the distance to A and also propagate a labeling of the connected components we would obtain if we would supress all edges connected to A from the tree;
— left image of figure 2 present the tree and the name of submit we will use;
— middle image present the distance to submit A;
— right image present the labeling L_A of connected components of the graph we would obtain if were supressing all edges starting from A.

The presented method is based on the elementary remark that if two submit B and G do not belong to the same component, then the shortest past from B to G cross A, and $D(B, G) = D(B, A) + D(A, G)$. This is formalized by equation 5 :

$$L_A(B) \neq L_A(G) \Rightarrow D(B, G) = D(B, A) + D(A, G) \tag{5}$$

The following, recursive, split algorithm can then be formulated : we first make a recursrive pre-computation for the whole graphe (using algorithm 1), we then use this precomputation to compute distance between pairs using the algorithm 4 :

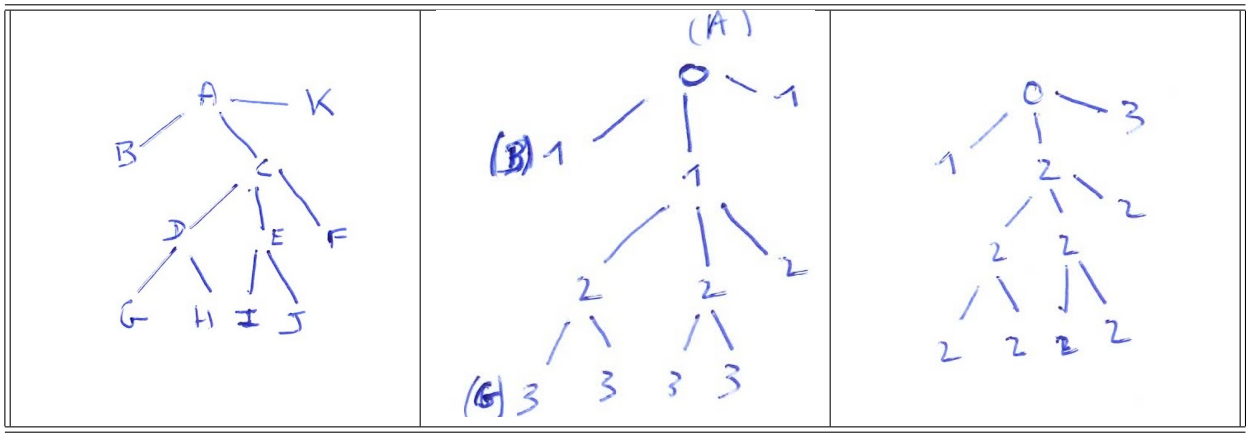

Figure 2 – Notation for a Tree; Distance to submit A; labeling of component after suppressing A

Algorithm 1 *PreComputeDist* (Graph $\mathcal{G}$,Level L)

if $Card(\mathcal{G}) = 1$ **then**
  we are done
else
  select a *pivot* submit A
  $\forall s \in \mathcal{G}$ compute $Dist(A, s)$ and $Lab_A(s)$ and save it at level L
  for all $\mathcal{CC}$ connected component of $\mathcal{G} - \{A\}$ **do**
    PreComputeDist ($\mathcal{CC}$,$L + 1$)
  end for
end if

Algorithm 2 *ComputeDist* (I,J)

if I=J **then**

  return 0
else
  for all L **do**
    if $Lab_L(I) \neq Lab_L(J)$ **then**

      return $Dist_L(I) + Dist_L(J)$
    end if
  end for
end if

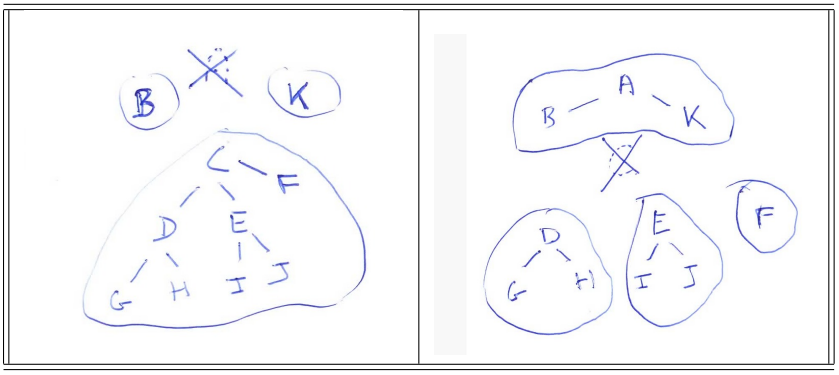

Figure 3 – Two possible split of the same tree : arround A and arround C

3 Selecting the best split

3.1 Exposure of the problem

Untill now we have not discussed how we select the *pivot* submit. Obviously this may have a main influence on the efficiency of algorithm 1. This is illustrated on figure 3. On this figure we see that the choice of A as pivot does not decrease so much the size of the biggest remaining graph, while the choice of C seems much more pertinent leading to sub-graph which maximal size is 3. In the extreme case where the tree is a chain, we see that :

— if we make systematically the "bad" choice and select an extremity to split the graphe, we will have to compute N level;

— on the other hand, if we make systematically the "good" choice and select the middle of the chain, we will only have to compute $log(N)$ level.

3.2 Criteria for spliting

To answer to the question of the *pivot* selection, we must must solve two issues :

— define a criteria on each submit that caracterize how good it would be to choose it as *pivot*;

— ensure that this criteria can be computed fastly enough .

For the first issue we want, after split, to avoid having too big connected component. So we simply take as criteria to select A as pivot, the maximum size of all connected components after supression of A. Once computed on all $\mathcal{G}$, we will select the submit minimizing this criteria.

Algorithm 3 *NaivePivotQual* (S)

```
Qual=0
for all CC connected component of G − {S} do
  Qual = Max(Qual,CardCC)
end for

return Qual
```

If "naive" definition of algorithm 4 is satifying, we cannot simply compute it for each submit, as the cost would be N for each submit, which would result in N^2 for all the graph. This is why in next section, we describe a recursive modification approach leading to a global cost of N for the computation of *pivot quality* on the whole graph.

3.3 Fast computation

Figure 4 illustrate the fast computation. In a first step, we select an arbitray submit, A here, and consider the oriented tree with A as head, an compute the function $H_A(S)$ which, for each submit, contain the size of the subtree under S. This computation can easily be done in linear time.

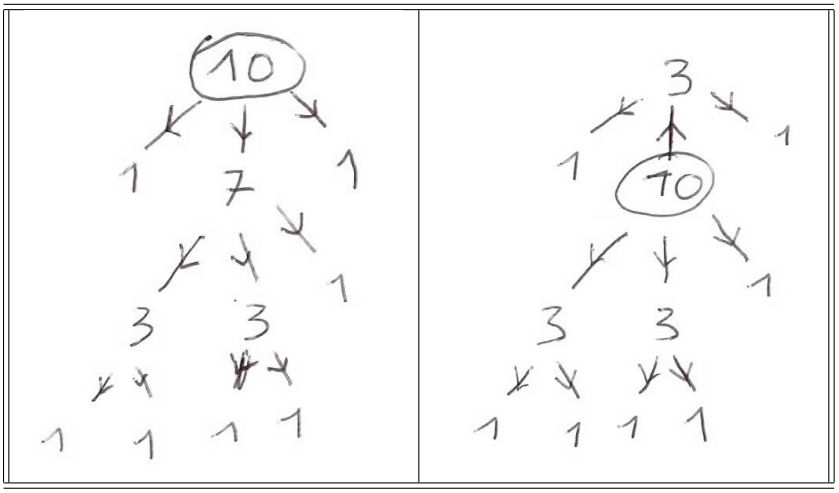

Figure 4 – Using tree of figure 2, function H_A (left) and H_C (right) representing sizes of subtree fonction with A and C as head

Now we have the two interesting property :

1. *PivotQuality*(A) can be easily computed from H_A, as it simply the max of $H_A(N)$ for all neighoor of A;
2. for all neighboor N of A , H_N can easily be computed from H_A;

For second property, we see for example, that if we want to compute H_C knowing H_A, we have only two operation to do :

1. $H_C(C) = H_A(A) = N$, in fact this is always the same value (the number of submit of the graph);
2. $H_C(A) = N - H_A(C)$ because the subtree correponding to $H_C(A)$ and $H_A(C)$ are the two complementary subtree we obtain by supressing edge (A, C).

Algorithm 4 *RecursPivotQual* (S,F)

```
{Before, compute the quality whith H_A of first submit}
QMax=0
for all n neighboor of S do
  QMax = Max(QMax,H(n))
end for
Qual[S] = QMax
{Now recursive exlporation}
for all n neighboor of S, n ≠ F do
  {hs,hn save values to restore them at end of recursive call}
  hs = H[S]
  hn = H[n]
  H[S] = Nb-H(n)
  H[n] = Nb
  RecursPivotQual(n,S)
  {restore values}
  H[S] = hs
  H[n] = hn
end for
```

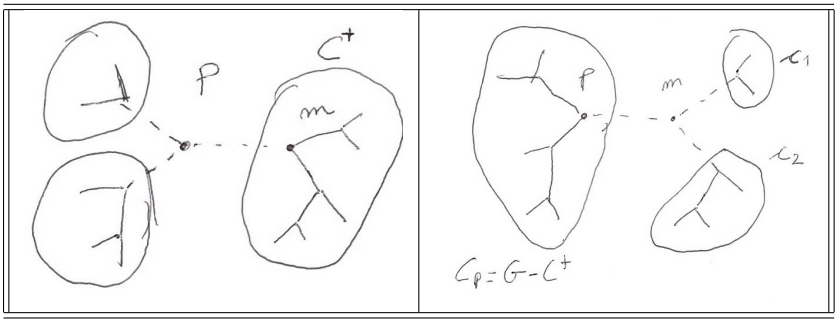

Figure 5 – Notation for analyzing the cardinal when pivot change from p to m

4 The implementation

Attached to this note is an implementation in `C++` of this algorithm. The "librairy" is contained in a single file `TreeDist.h` that can be include. The file `TreeDist.cpp` can generate an executable program that run the test, using the following command (on Gnu/Linux) :

```
g++ TreeDist.cpp -std=c++11
```

The main interface for user is the class `cFastTreeDist`. The usage of the class consist of essentially of 3 methods :

- `cFastTreeDist(int NbSom)` : the constructor, allocate the memory;
- `void MakeDist(const std::vector<int> & aVS1,const std::vector<int> & aVS2);` : make all the pre-computation for a given graph ,see comment in the code for graph representation; 1
- `int Dist(int aI1,int aI2) const;` compute the distance bewteen two submit once the pre-computation has been done (correspond to algorithm 4).

5 Theoreticall and empiricall analyse of complexity

To analyse the complexity, we must first compute the maximal level of recursion that can be reached.

By the definition we use of *pivot* quality, we are ensured that at each decomposition the connected component have a size inferior to half of the graph. Let prove it, illustrated by figure 5 :

- let N be the number of submit of $\mathcal{G}$, $N = \overline{\mathcal{G}}$;
- let p be the pivot;
- let $\mathcal{C}^+$ be the connected componet of $\mathcal{G} - \{p\}$ with maximal number of submit and m the submit connecting $\mathcal{C}^+$ to p;
- if we had $= \overline{\mathcal{C}^+} > \frac{N}{2}$, then if we were taking m as pivot , connected of $\mathcal{G} - \{m\}$ would be :
  - one component $\mathcal{C}_p = \mathcal{G} - \mathcal{C}^+$ and $\overline{\mathcal{C}_p} \leqslant \frac{N}{2}$;
  - other components c_i each being included in $\mathcal{C}^+ - \{m\}$ so $\overline{c_i} < \overline{\mathcal{C}^+}$
- so m would be a better pivot than p which is a contradiction.

As each component, in the recursive split, has a size inferior to the initial size, the maximal number of level is $log_2(N)$. At each level the computation is linear. So :

- the pre-computaion has a complexity of $Nlog(N)$;
- the computation of each distance, in algorithm 4 is in worst case equal to the maximum level, so $log(N)$ is the cost for computation of distance in worst case;

The computation of distance in average case is more complex to analyse. We can make the following, not 100% formal, reasoning in algoritm 4:

- the probabilty that $Lab_1(I) = Lab_1(J)$ is inferior to $\frac{1}{2}$ because size of maximal component is inferior to $\frac{1}{2}$;
- the probabilty that $Lab_2(I) = Lab_2(J)$ is inferior to $\frac{1}{4}$ because size of maximal component is inferior to $\frac{1}{4}$;
- ...
- so the average cost is bounded by $\sum_{k=1}^{\infty} k\frac{1}{2}^k = 2$

NbSom	16	64	144	256	400	576	784
AvgT	1.34	1.46	1.49	1.53	1.50	1.50	1.53
AvgLowT	1.63	2.42	3.03	3.53	3.79	4.06	4.36
Low/Log	0.59	0.58	0.60	0.63	0.63	0.63	0.65

Figure 6 – Computation time of distance between pair as function of the number of submits : (1) **AvgT**= average (2) **AvgLowT**=average for "low" distance ($\leqslant 3$), (3) **Low/Log** = average of "low" /Log(NbSom)

Figure 6 present the experimental computation time we obtain using the command `StatTimeBenchFastTreeDist`, that generate random trees and evaluate the average level reached in algorithm 4 , this level beign proportionnal to the time. The line `AvgT` correspond to the average, on all pair of the graph, we see that this time if almost constant .

By the way, in some situaion the average on all pair may be unrealistic and too optimistic. In all case, the pair corresponding to low distances are generally splited at higher level and correspond to a longer time of computation. For example in the case of photogrammetry exposed in [Govindu, Venu 2006], it will be current that, due to spatial correlation, the majority of edges will correspond to low distances and that the average time is more important that predicted when taking all the pair. This is the reason of the two last lines :

— **AvgLowT** this line present an average on the pair corresponding to "low" distances, here the threshold is arbitrarily 3, we clearly see the time is growing with the number of submit;

— **Low/Log** is the previous line divided by $log(NbSom)$, we see that it is almost constant and validate that the time for low distance a modelisation in $log(N)$, like worst case, is coherent.

References

[Govindu, Venu 2006] Govindu, Venu. "Robustness in Motion Averaging." Computer VisionACCV 2006 (2006): 457-466.