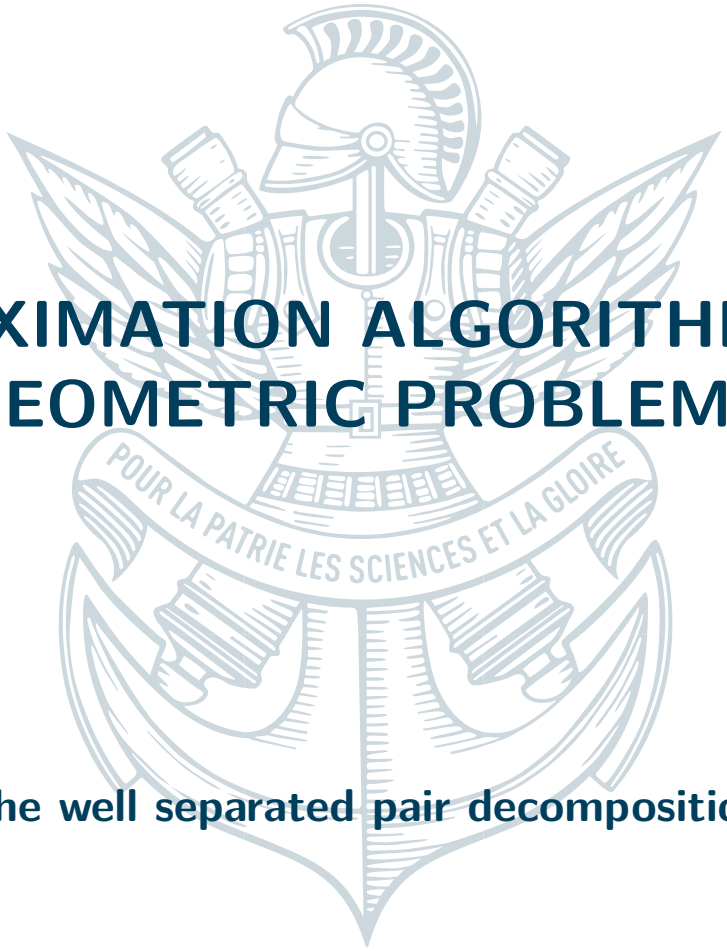

APPROXIMATION ALGORITHMS FOR GEOMETRIC PROBLEMS

The well separated pair decomposition

February 2019

Marc Fersztand, Léo Grinsztajn

INTRODUCTION

In computational geometry, many problems rely on the knowledge of the distances between a large number of points. However, storing all the distances from a set of n points requires a $O(n^2)$ space complexity. Hence there is no hope to devise a subquadratic algorithm for those problems without improving the distances representation. A well-separated pair decomposition(WSPD), is a data structure that enables to store $O(n)$ distances and to infer an approximation of the other distances. Thanks to the WSPD, it is possible to create efficient exact and approximation algorithms for geometric problems in any fixed dimension [2].

In this report we define the notion of octree and the well-separated pair decomposition and we discuss their implementation. Then, we will explain how to apply the WSPD for three famous problems : the closest pair problem, the diameter problem and the computation of a force directed layout. Those problems have naive $O(n^2)$ algorithms, the goal of this project is to solve them in quasilinear time.

1 PRINCIPLE OF THE WSPD

Let P be a set of n points of $\mathbb{R}^3$, a well-separated pair decomposition (WSPD) is intuitively a partition of $P \times P$ such that for (A, B) of the WSPD :

(i) all the distances between a point of A and a point of B are similar

(ii) the distances within A (and within B) are small compared to the distance between A and B.

More formally,

Définition 1.1 (Well-separated sets). Let $(A, B) \in P \times P$ and $s > 0$, we say that (A, B) is s-well separated if it exists $R > 0$ such that

(i) A is included in a ball C_A of radius R

(ii) B is included in a ball C_B of radius R

(iii) $d(C_A, C_B) \leqslant sR$

Définition 1.2 (Well-separated pair decomposition). Let $s > 0$, we say the $\mathcal{W} = \{(A_1, B_1), \ldots, (A_k, B_k)\}$ is a s-well separated pair decomposition(WSPD) if

(i) $\forall i\ A_i, B_i \subset P$

(ii) $\forall i\ A_i \cap B_i = \varnothing$

(iii) $\cup_i (A_i, B_i) = P \times P - \{(p, p),\ p \in P\}$

(iv) for all i, (A_i, B_i) **is s-well separated**

Let's consider two subsets A and B and suppose we know that $A \subset V_A$ and $B \subset V_B$ where V_A and V_B are simple volumes. Then, if we consider the two balls circumscribed to V_A and V_B, we obtain a constant time criteria for not being s-well separated. That's why we want to divide P into subsets of points confined into simple volumes.

The proposed structure in this project is the octree (2^d tree in dimension d). The idea is to divide the (hyper)cube containing all the considered points into eight (2^d) smaller cubes and to repeat it recursively. It creates a tree where each internal node has eight children. A node represents a cube and its children represent the eight smaller cubes it is composed of. We stop dividing the cube when it contains less than two points.

The principle of the algorithm to compute a WSPD of P is :

```
main(P)
construct an octree T of P;
return WSPD (s,T.root,T.root);
WSPD(s,u,v)
if u and v are well separated then
  | add (u,v) to the result;
else
  | Go down the octree and call recursively WSPD;
end
return the list ;
```

Algorithm 1: Algorithm creating a s-WSPD

2 IMPLEMENTATION OF THE WSPD

The subject suggests to implement the octree with a divide and conquer approach. We start from the root and for each internal node we create recursively its children.

Proposition 2.1 The complexity of the octree computation is $O(n \log n)$.

This theorem is quite natural because there are n leaves, and the height cannot be too much because of two phenomena : when the points have a uniform distribution the octree is complete (leaves are all on the same level), there are $O(n)$ nodes ; on the contrary, when the points are concentrated in zones, many branches stop early.

For the WSPD implementation, we follow the algorithm 1. The following points seemed relevant :

— "going down the tree" is done by calling recursively the children of the highest node

— To avoid a costly concatenation (each call of the recursive function should be constant in time), we consider the list as a parameter of the function.

Proposition 2.2 The WSPD function has a complexity of $O(s^3 n)$ and the output has a size $O(s^3 n)$.

Since s is the factor of precision of the representation, when s is large we have a better approximation and we need more space to store it.

Regarding the experimental complexity, in addition to the provided files, we made tests with random uniformly distributed points. We obtain that the designed algorithm is faster than the naive one ($O(n^2)$). However, the theoretical asymptotic complexity is not reached for small n (see figure1 all the data is the average of at least 10 computations of the algorithm). The constant in the WSPD complexity is important. Moreover in this computation, the WSPD function is a lot slower than the octree creation despite having a better asymptotic complexity ! Moreover as predicted the complexity is cubic in s (see figure 2).
We can observe other properties of the algorithm. The size of the output is proportional to the time complexity and is a little bit more than linear for small n. The complexity constant depends on the distribution of the points : for a random set uniformly distributed, the constant is five time higher than in a structured figure like in the examples. Since the WSPD are meant to be used on structured figures, we may obtain better result for real applications than in figure 1.

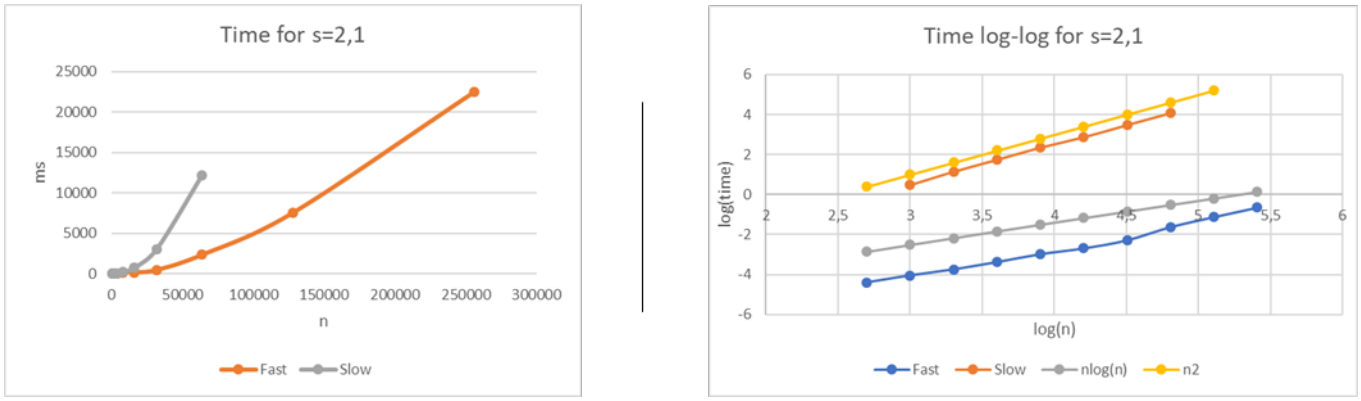

Figure 1 – Naive and WSPD algorithms average times on a random set for the closest pair problem (s=2.1)

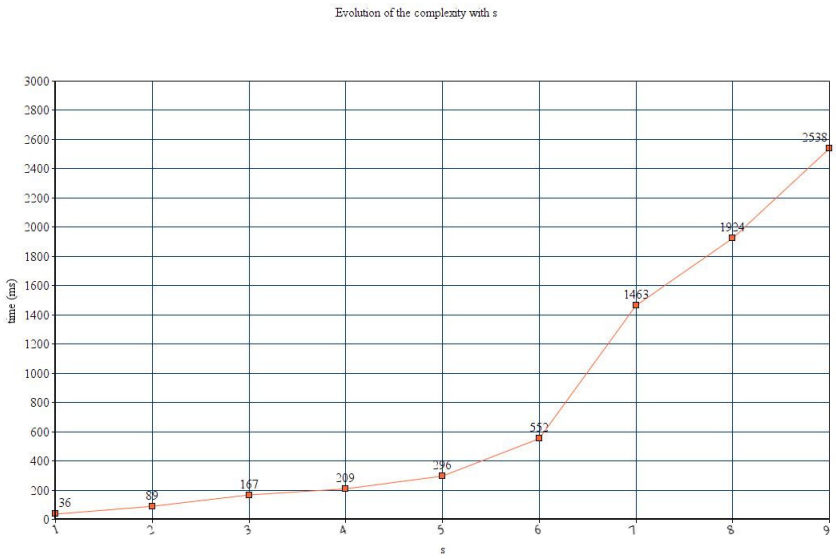

Figure 2 – Average experimental complexity of the WSPD algorithm on $n = 10000$ points and s varying

We devised three optimizations for WSPD algorithm :

— For all the problems we consider in this report, the distance are symmetric. It is then possible to store a partition of 2-subsets of P instead of all the pairs of P. Moreover, we can loosen the definition of WSPD by not imposing the two balls containing A and B to have the same radius and by replacing R by the maximum of the two radii.

— We notice that the Octree construction takes no time compared to the WSPD algorithm. Thus we can add information into the octree nodes. For instance we could compute the smallest centered ball that contains all the points of the node. It does not increase the theoretical complexity since dividing the points into the children is already linear. However, in fact this optimization is effective. Experimental data is available in figure 4. We can go farther by changing the center of the node, we take as center point the center of the bounding box. The principle of this optimization is described in figure 3 (in 2D).

— An other observation is that until the algorithm finds a leaf, it is always the same computation. The idea is to pre-compute a WSPD for an octree whose leaves are all on the level k. And each time WSPD(u,u,s) is called and that all the children of u on k generations are not leaves, instead of going down the tree until the level k, we use the precomputation to go directly to the level k. The precomputation for the level k uses the level $k-1$ and this precomputation depends only on s. For the applications that use a

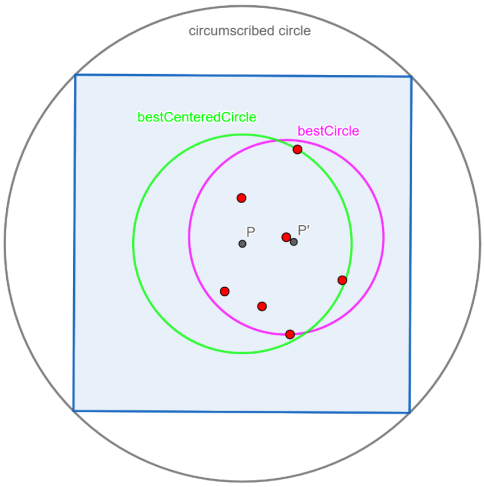

Figure 3 – principle of best radius optimization

single value of s, the pre-computation can be stored in the files of the application. However, in facts this optimization is not very effective for small n.

The two optimizations could be in conflict but the experimental data show that it is not the case. The experimental data for $s = 2.1$ on average on uniformely distributed sets is in figure 4.

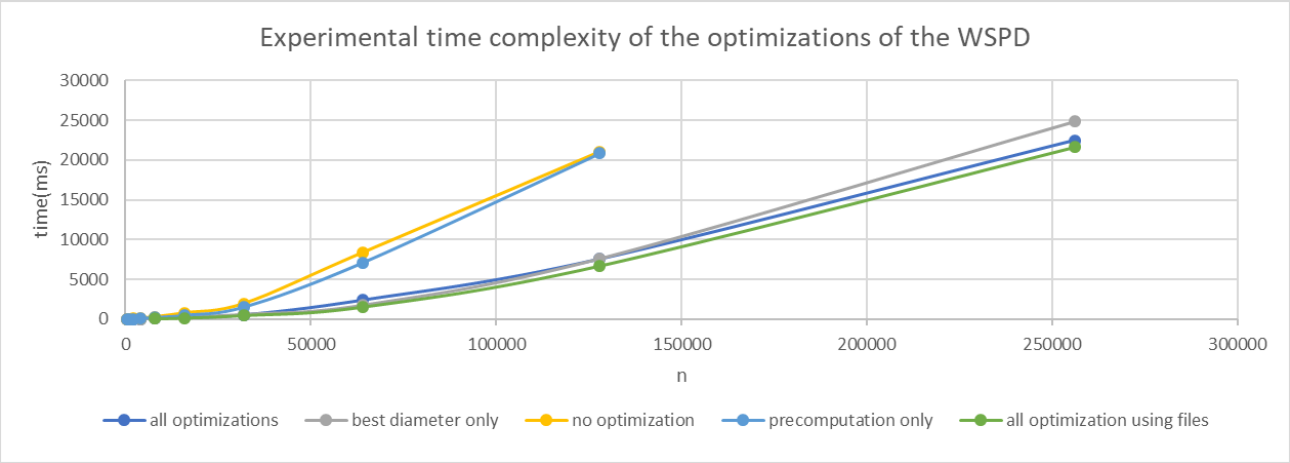

Figure 4 – Experimental complexity for the different optimizations for $s = 2.1$

3 APPLICATIONS

3.1 The closest pair problem

The closest pair problem is a classic problem. It has a naive $O(n^2)$ algorithm and a divide and conquer $O(n \log n)$ algorithm in dimension 2. The WSPD method gives a $O(n \log n + s^d n)$ algorithm in any dimension d.

Proposition 3.1 All s-WSPD with $s > 2$, contains a pair of leaves which realize the smallest pairwise distance.

Démonstration. Let (p, q) realising the smallest distance and let $\mathcal{W}$ a s-WSPD with $s > 2$. $(A, B) \in \mathcal{W}$ of radius R such that $(p, q) \in A \times B$. If $\text{Card}(A) > 1$, there exists $p' \neq p \in A$, then :

$$\mathrm{d}(p, p') \leqslant 2R < sR \leqslant \mathrm{d}(p, q) \leqslant \mathrm{d}(p, p').$$

Hence $\text{Card}(A) = 1$. Similarly $\text{Card}(B) = 1$. □

Since s is relatively small, we obtain here a fast exact algorithm in $O(n \log n)$ (see fig 1 and 2 for an empirical analysis). It will be different in the next problem.

3.2 The diameter problem

The diameter problem is the classic problem of finding the farthest pair in a set of points. With the traditional points implementation, we would have a naïve $O(n^2)$ algorithm. However, if we allow for a small relaxation, we can have a $O(n \log n)$ algorithm thanks to the WSPD paradigm.

Proposition 3.2 Let $\varepsilon \in \mathbb{R}_+^*$ and P a set of point in $\mathbb{R}^d$.
For $s \geq \frac{4}{\varepsilon}$, the pairs of leaves of a s-WSPD contains a pair (q, r) such as $\mathrm{d}(q, r) \geq (1 - \varepsilon)\text{diam}(P)$, with $\text{diam}(P)$ being the length of the farthest pair in P.

Démonstration. Let (p, q) the farthest pair in P, let $\mathcal{W}$ a s-WSPD with $s \geqslant \frac{4}{\varepsilon}$, and let $(u, v) \in W$ the pair containing respectively p and q. Then we have :

$$\mathrm{d}(rep_u, rep_v) \geq \mathrm{d}(u, v) \geq \mathrm{d}(p, q) - \text{diam}(u) - \text{diam}(v).$$

$\mathcal{W}$ being a s-WSPD, we have $s \times \max(\frac{\text{diam}(u)}{2}, \frac{\text{diam}(v)}{2}) \leqslant \mathrm{d}(u, v) \leqslant \mathrm{d}(p, q)$.
Hence,

$$\mathrm{d}(rep_u, rep_v) \geqslant (1 - \frac{4}{s})\mathrm{d}(p, q) \geqslant (1 - \varepsilon)\mathrm{d}(p, q)$$

. □

If we compute a s-WSPD with $s \geqslant \frac{4}{\varepsilon}$, and then check all the WSPD pairs for the farthest, we have a $O(n \log n + n/\varepsilon^d)$ algorithm to find the farthest pair up to a ε error.

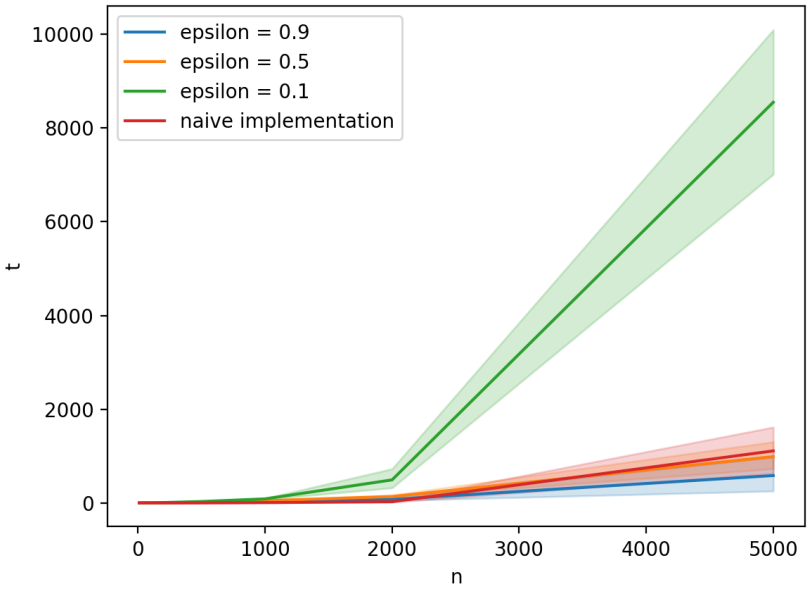

Figure 5 – Time for the farthest pair computation for different precisions ε

3.2.1 • Empirical evaluation

Theoretically, we could expect the octree computation in $O(n \log n)$ to be the longest computation for this algorithm, but as noted in part 1, we find empirically that the WSPD construction from the octree, in $O(s^2 n)$ takes far longer. The computation of the farthest pair from the WSPD, also theoritically in $O(s^2 n)$, is also negligible in practice compared to the WSPD construction, thus the empirical analysis of part 2 in particularly relevant. In practice, we find the constant in the $O(n/\varepsilon^d)$ of the WSPD construction to be quite high, and the algorithm to be quite slow for small ε. In Figure 5, we see that for random point clouds, the fast implementation only outperforms the naïve one for very large epsilon and large n, making it quite useless in practice.

3.3 Fast approximation for repulsive forces

A practical problem often encountered is the one of drawing a graph in 2 or 3D in a way that is both pleasing to the eye and informative to the viewer. One of the most popular paradigm for this purpose is the force-directed graph drawing methods, where we apply physical forces to the vertices of the graph until an equilibrium is reached.
For our subject, we are interested by the most common of these methods, called Fruchterman-Reingold. In this method, there are 2 types of forces :

— Additive forces between vertices linked by an edge in $k \times d$, with d the distance between the 2 vertices. This corresponds to modelling the edges by springs.

— Repulsive forces between each pair of vertices (linked or not) in $\frac{k}{d}$, preventing the graph from collapsing into itself.

Noting n the number of vertices of a graph G and m the number of edges, it is easy to see that the resulting complexity of this computation is naïvely in $O(m + n^2) = O(n^2)$, but thanks to the WSPD concept, we will devise an algorithm in $O(m + n \log n)$.
Indeed, as far as repulsive forces are concerned, we can represent all pairs as a WSPD and only compute the

repulsive forces between the barycenters of each set of point.

From an algorithmic viewpoint, we first compute an octree T from the points corresponding to the graph vertices. Thanks to T, we create a WSPD $\mathcal{W}$, and we compute the repulsive forces between each pair $p \in \mathcal{W}$, and add it to the corresponding octree nodes. Then we go down the octree T, and add the force of a node to its children. Below a more detailed pseudocode of the algorithm.

`FastFruchterman(`G`)`
for *each iteration* **do**
  Compute the attractive forces for each edge;
  Compute an octree T from the points of G, and store for each node the barycenter of its subtree;
  Compute a WSPD $\mathcal{W}$ through T;
  for *each pair* $(u, v) \in \mathcal{W}$*, corresponding to the octree nodes* N_u *et* N_v **do**
    $N_u.force+ = |v| \times \text{RepulsiveForce}(\text{barycenter}(N_u), \text{barycenter}(N_v))$;
    $N_v.force+ = |u| \times \text{RepulsiveForce}(\text{barycenter}(N_v), \text{barycenter}(N_u))$;
  end
  Go down the octree T and add the force of a node to its children;
end

Algorithm 2: Algorithm for "fast" Fruchterman-Reingold

3.3.1 • Empirical evaluation

Empirically, the "fast" implementation of Fruchterman-Reingold through WSPD gives very pleasing results. For instance, we can see Figure 6 that the layout is as pleasing as the "naïve" implementation, even for small s like 1.
Nevertheless, our "fast" implementation quickly outperforms the "naïve" one in term of speed. (Figure 7 for the test on the graphs given for the project).

One optimisation we have played with is to recompute the octree and the wspd not at every iteration, but when $\lfloor \delta_t * \log iteration\_number \rfloor$ increases. The article [1] recommends $\delta_t = 5$, but we found that for our parameters $\delta_t = 20$ achieved better result while being quicker (a better result for a bigger δ_t may seem paradoxical but is confirmed by [1]). See Figure 10 for visual results and Figure 8 and 9 for speed results on the graphs given for the project.

CONCLUSION

The principles of the Well-Separated-Pair-Decomposition, while being a quite simple idea, allows for the improvement of numerous classic algorithm relying on the distances computation of large set of points, and can make intractable $O(n^2)$ algorithm become easily tractable $O(n \log n)$ algorithms (Part 3). Furthermore, the idea of the WSPD can be refined through various optimisations, like the ones we have played with Part 2.

RÉFÉRENCES

[1] Alexander Wolff Fabian Lipp and Johannes Zink. Faster force-directed graph drawing with the well-separatedpair decomposition, 2015.

[2] Sariel Har-Peled. Well separated pairs decomposition, 2008.

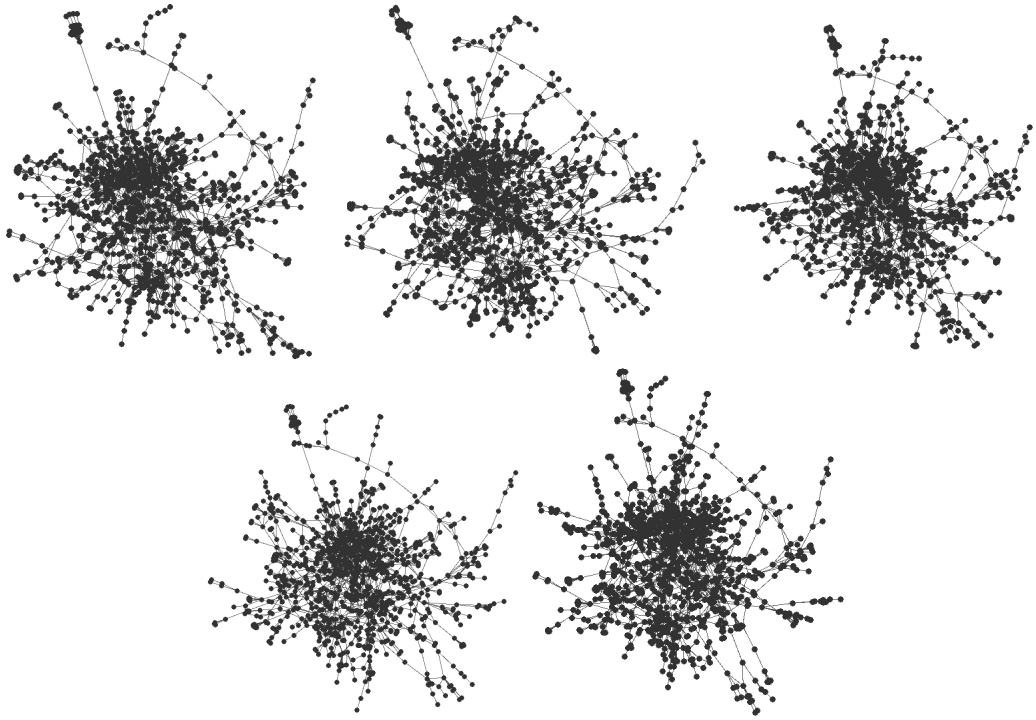

Figure 6 – In left to right order : classic algorithm, and "fast" algorithm for s=10, 1, 0.1, 0.01 for the "Facebook" graph

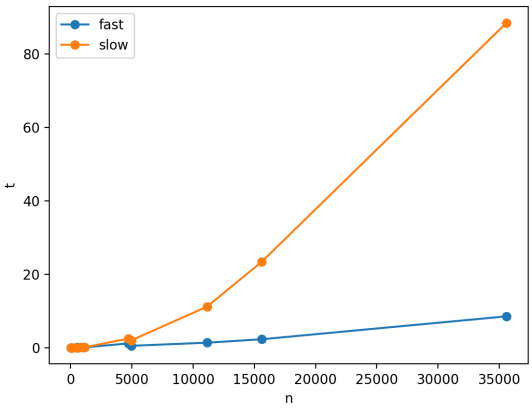

Figure 7 – Time for both implementations for 15 iterations of Fructerman for different number of vertices (s=1 for fast implementation)

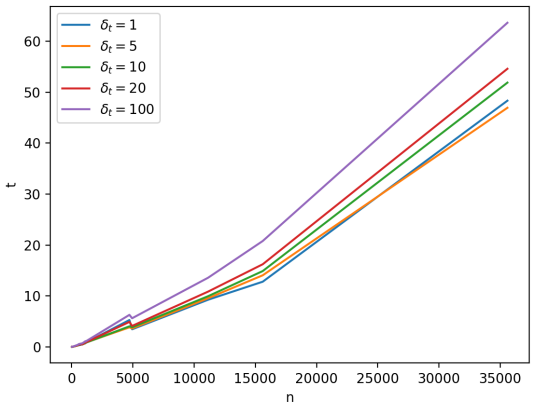

Figure 8 – Time for 150 iterations of Fruchterman with s=1 and variable δ_t for different number of vertices

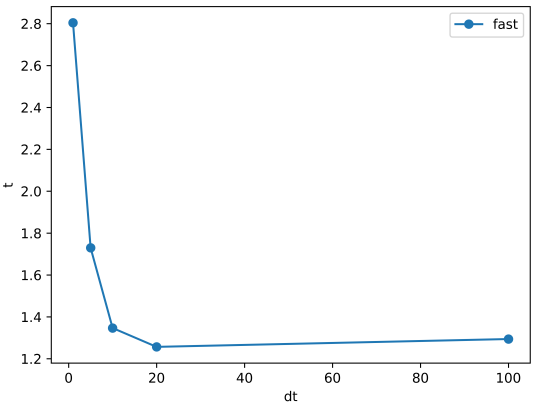

Figure 9 – Time for Fruchterman with s=1 and variable δ_t on the graph "Facebook" (n=1128)

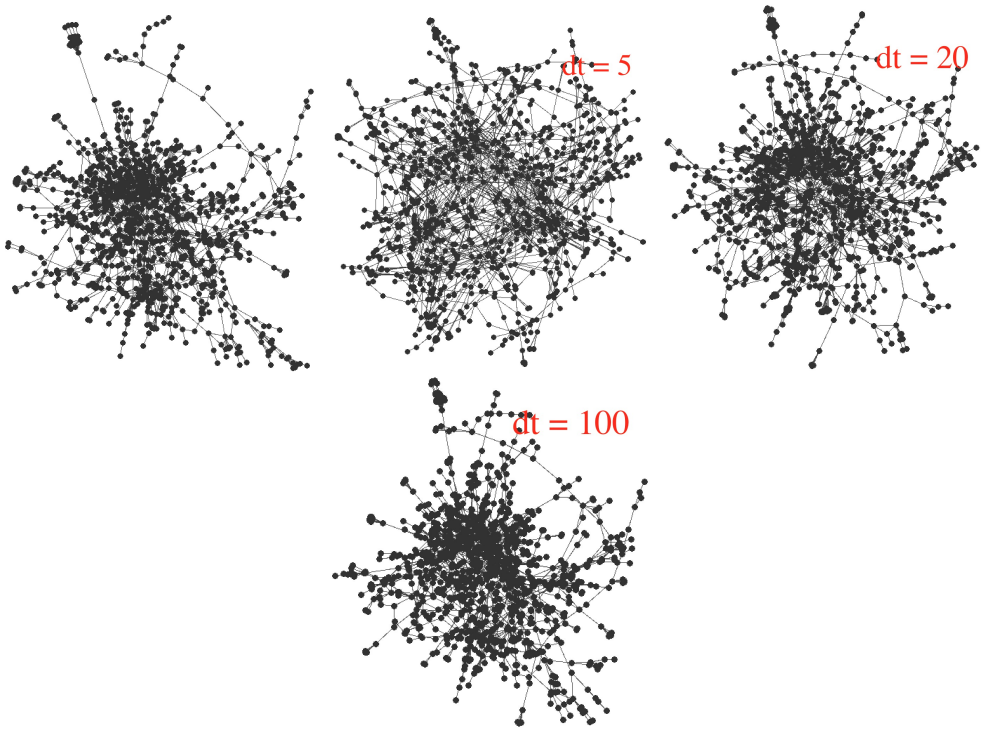

Figure 10 – In left to right order : classic algorithm, and "fast" algorithm for s=1 and δ_t = 5, 20, 100 for the "Facebook" graph