Programming project: Well Separated Pair Decompositions

Timothée Chauvin & Jean-Stanislas Denain

February 2019

Contents

Introduction **1**

1 Building a Well Separated Pair Decomposition **1**

1.1 Building an Octree from the point set P 1

1.2 Obtaining a WSPD from an Octree 2

2 Applications **3**

2.1 Closest pairs 3

2.2 Diameter 4

2.3 Network visualization 6

Introduction

1 Building a Well Separated Pair Decomposition

1.1 Building an Octree from the point set P

Attributes We added attributes to the `OctreeNode` class. The `quadrant` attribute gives the position of the subcube corresponding to the `OctreeNode` within the larger cube (stored in the `father` field). The `center` and `L` attributes stand for the center and side length of this subcube. The attribute `p` is an arbitrary point in P belonging to the subcube. We use it as a point representative of the `OctreeNode`.

Procedure Since `Octree` is but a wrapper around `OctreeNode`, it suffices to compute the `OctreeNode` associated with P. We start by finding the side length of the smallest cube containing P. Next, we build the octree by repeatedly adding all the points in the input array. There are 5 situations we can find ourselves in when adding a point (`point`) to an `OctreeNode` (`this`):

- The `OctreeNode` is empty: it has neither children nor representative. In this case the resulting `OctreeNode` only contains this point as representative and has no children.
- The `OctreeNode` is a lonely point: it has a representative `this.p` but no children.
  - If `point` and `this.p` belong to different quadrants within `this`, then we add 2 children to `this`, each containing either `point` or `this.p` as representative.
  - If `point` and `this.p` belong to the same quadrant within `this`, then we add a child to `this` in this quadrant with representative `this.p`, and recursively add `point` to this child.
- The `OctreeNode` is neither empty nor a lonely point: it has a representative and at least one child.
  - If there is a child whose quadrant within `this` is the same as that of `point`, then we recursively add `point` to this child.
  - If `this` has no children whose quadrant is the same as that of `point`, then we build such a child and make `point` its representative.

Children as Arrays vs Children as LinkedLists Initially, we had implemented the children of a given `OctreeNode` as an array of eight `OctreeNodes`, initialized as empty. Though such a choice seemed convenient at first, we found that storing the children in a linked list of `OctreeNodes` took about as long and used much less space (*cf* `results/test.csv`).

1.2 Obtaining a WSPD from an Octree

Sufficient condition for well separatedness We define a function `areWellSeparated` determining whether the subsets of P corresponding to two `OctreeNodes` `n1` and `n2` are well separated with respect to some parameter `s`. The function proceeds as follows [1] :

[1] Under the definition in [2] the diameter of a cube of side length L is $\sqrt{3}L$. The diameter is L if defined as twice the radius of the ball in $(\mathbb{R}^3, || - ||_\infty)$. Only the former definition guarantees the well separatedness of the pairs accepted by `areWellSeparated`. We therefore only use a sufficient condition for well separatedness.

- If neither `n1` nor `n2` has any children, then the function assumes that the corresponding representatives are distinct (the case where they are not is handled separately). In this case `n1` and `n2` are well separated.
- If `n1` has no children then the distance `d` between its lone point and the subcube corresponding to `n2` is computed. `n1` and `n2` are well separated iff `d` ≥ `sqrt(3)*s*n2.L`.
- Symmetrically, if `n2` has no children then the distance `d` between its lone point and the subcube corresponding to `n1` is computed. `n1` and `n2` are well separated iff `d` ≥ `sqrt(3)*s*n1.L`.
- Finally, if both `n1` and `n2` have children, then the distance `d` between the corresponding subcubes is computed. `n1` and `n2` are well separated iff `d` ≥ `sqrt(3)*s*max(n1.L,n2.L)`.

Procedure The test of well separatedness described above is equivalent to that described in [2]. We therefore follow the algorithm given there to build the decomposition. We first define a recursive function `WSPDrec` with two `OctreeNodes` as arguments, and apply it to the root of the full `Octree`.

WSPD as a LinkedList vs WSPD as a HashSet We had to choose a data structure to store the pair decomposition. At first, we decided to use instances of `HashSet<OctreeNode>` in order to automatically handle redundancies. We found that using `LinkedLists` of `OctreeNodes` instead and ignoring redundancies yielded better results. Redundancies mean that it is hard to estimate the size of the WSPD, which could be as small as half the length of the returned linked list.

2 Applications

2.1 Closest pairs

Why should $s = 2$? It seems to us that the following generalization of section 3.2.4 in [2] is also true.

Theorem 2.1. *Let P be a set of points in $\mathbb{R}^3$, $s > 1$ and $\{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ be a well separated pair decomposition of P with parameter $s > 1$. Then the two closest points p and q in P are such that $\{p\} = A_i$ and $\{q\} = B_i$ for some $1 \leq i \leq m$.*

Finding a necessary and sufficient condition would involve solving the Minimum Enclosing Ball problem (which can be solved at best in linear time, and would take us a trifle too far).

Proof. Let $1 \leq i \leq m$ be such that $p \in A_i$ and $q \in B_i$. Suppose that A_i or B_i is not a singleton: without loss of generality we assume that there exists a point $r \in A_i \backslash \{p\}$. Since A_i and B_i are well separated sets, there exist two balls $\mathscr{B}_A$ and $\mathscr{B}_B$ of respective diameter d_A and d_B, such that $A_i \subset \mathscr{B}_A$, $B_i \subset \mathscr{B}_B$, and $d(\mathscr{B}_A, \mathscr{B}_B) \geq s \cdot \max(d_A, d_B)$.

Both $p \in \mathscr{B}_A$ and $r \in \mathscr{B}_A$: therefore $||p - r|| \leq d_A \leq \max(d_A, d_B) \leq \frac{1}{s}(\mathscr{B}_A, \mathscr{B}_B) \leq \frac{1}{s}||p - q||$. Since $s > 1$, $||p - r|| < ||p - q||$, which contradicts our initial assumption that $\{p, q\}$ is the closest pair.

□

Therefore in order to find the closest pair, we need only determine an s-WSPD with $s > 1$: values of s smaller than 2 are therefore allowed, and significantly speed up the computation of the WSPD.

Closest pair results The figure below summarizes our results for the task. All our results are available in `results/test.csv`. Though more costly for small point clouds, using WSPDs proved useful for the larger `horse.off` point cloud.

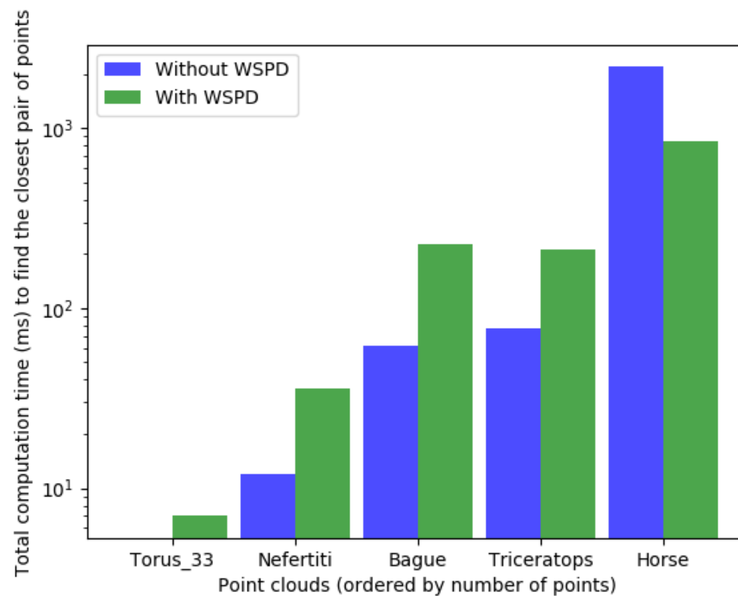

Figure 1: Closest pair computation time, with and without WSPD (logarithmic scale), $s = 1.001$

2.2 Diameter

Why should $s = \frac{4}{\epsilon}$? Here is another generalization of a result in [2]:

Theorem 2.2. *Let $P \subset \mathbb{R}^3$ be a set of points, $\epsilon > 0$ and $\{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ be a well separated pair decomposition of P with parameter $s \geq \frac{2}{\epsilon}$. define for every $1 \leq i \leq m$ a representative α_i (respectively β_i) of A_i (respectively B_i). Call d_P the diameter of P, the greatest distances between two points in P. Then $\max_{1\leq i\leq m} ||\alpha_i - \beta_i|| \geq (1-\epsilon)\cdot d_P$.*

Proof. Let p and q be the farthest points in P, and $1 \leq i_0 \leq m$ be such that $p \in A_{i_0}$ and $q \in B_{i_0}$. Then $||p-q|| \leq ||\alpha_{i_0} - \beta_{i_0}|| + ||p - \alpha_{i_0}|| + ||q - \beta_{i_0}||$.

Since A_{i_0} and B_{i_0} are well separated sets, there exist two balls $\mathscr{B}_A$ and $\mathscr{B}_B$ of respective diameter d_A and d_B, such that $A_{i_0} \subset \mathscr{B}_A$, $B_{i_0} \subset \mathscr{B}_B$, and $d(\mathscr{B}_A, \mathscr{B}_B) \geq s \cdot \max(d_A, d_B)$.

Then $||p - \alpha_{i_0}|| \leq d_A \leq \frac{1}{s} d(\mathscr{B}_A, \mathscr{B}_B) \leq \frac{1}{s}||p-q||$. Similarly $||q - \beta_{i_0}|| \leq \frac{1}{s}||p-q||$. Therefore $\max_{1\leq i\leq m} ||\alpha_i - \beta_i|| \geq ||\alpha_{i_0} - \beta_{i_0}|| \geq ||p-q|| - ||p-\alpha_{i_0}|| - ||q - \beta_{i_0}|| \geq (1-\frac{2}{s})\cdot||p-q|| \geq (1-\epsilon)\cdot d_P$. □

Therefore asking that $s = \frac{4}{\epsilon}$ when computing the diameter seems unnecessarily stringent: it suffices to take $s \geq \frac{2}{\epsilon}$. Since the parameter of the WSPD has a sizeable influence on the speed of its computation, relaxing this constraint turned out useful.

Diameter results The figure below summarizes our results for the task. These results are somewhat disheartening, as using WSPDs does not allow us to reap the expected benefits. We observed that computing the WSPD was the main bottleneck: the octree was generated rapidly, and once the WSPD had been determined, our program could swiftly complete the task.

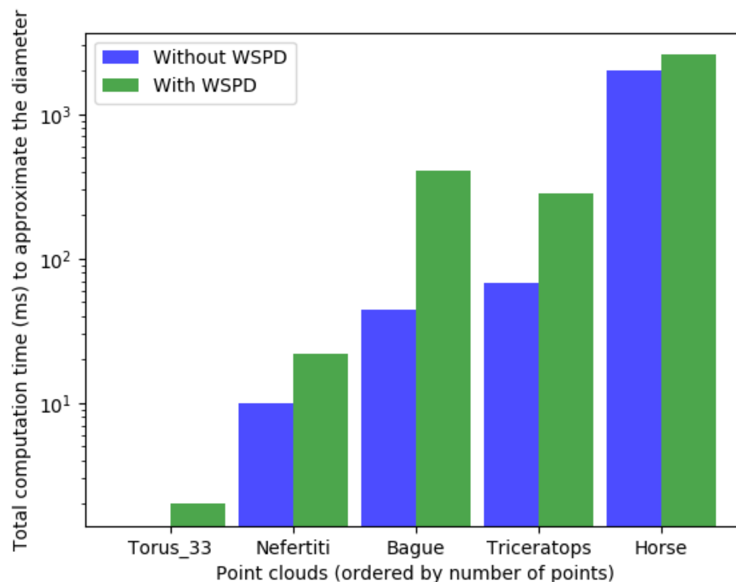

Figure 2: Diameter approximation time with and without a WSPD (logarithmic scale), for $s = 0.7$

2.3 Network visualization

Procedure The only difference between the approach we used and the original method of Fruchterman and Reingold [1] is the computation of the repulsive forces between nodes. We followed the pseudocode in [3]. Here we give a few additional details on our implementation.

We added a few attributes to the `OctreeNode` class. The `numberPoints` field gives the number of points in the subcube, or equivalently the number of leaves in the tree. The `barycenter` attribute stores the barycenter of all the leaves. Finally, given a WSPD, the `repForce` field contains an approximation of the force exerted on the barycenter of the leaves of the `OctreeNode`.

The function `computeAllRepulsiveForces` returns an array of 3D vectors corresponding to the repulsive forces exerted on every point in the network. It works as follows:

- Create a HashMap `hm` where the keys are points (positions of the network's vertices) and the values are vectors (repulsive forces).
- Compute an s-WSPD of the set of vertices in the network. As in [3] we tried $s = 1$, $s = 0.1$, and $s = 0.01$.
- For every pair of `OctreeNodes` in the WSPD, update the `repForce` field of each `OctreeNode`.
- Call the `recTraversal` method that recursively runs through the tree (performing a DFS), and:
  - for every internal node, adds the node's `repForce` to that of its child
  - for every leaf `n`, adds it to the HashMap `hm`, with key `n.p` and value `n.repForce`.
- Initialize an array `repForces` of vectors that will eventually be the result and fill it up vertex by vertex, using the HashMap to get for each vertex the force corresponding to the position of the vertex.

Displaying results Though it may be our ultimate goal, displaying large networks is quite time consuming. Consequently, in order to test the speed of our algorithms, we built another class `GraphDrawingResults` displaying only the time taken by the different parts of our code at each step of the force directed drawing.

Network visualization results The figures below compare the time necessary to perform 15 iterations of the Fruchterman and Reingold algorithm, with and without using an s-WSPD, with values of s in $\{1, 0.1, 0.01\}$, for networks of different size. As in [3] we noticed that $s = 0.1$ or $s = 0.01$ provides better results than $s = 1$.

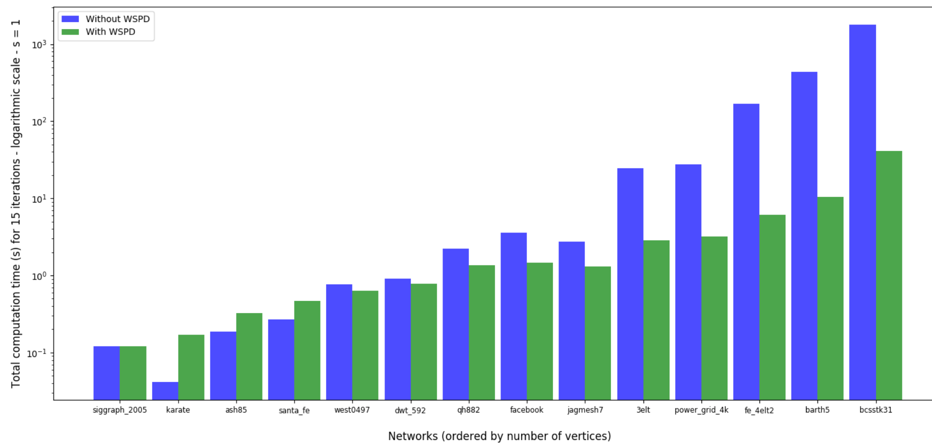

Figure 3: Graph drawing computation time (logarithmic scale) for $s = 1$

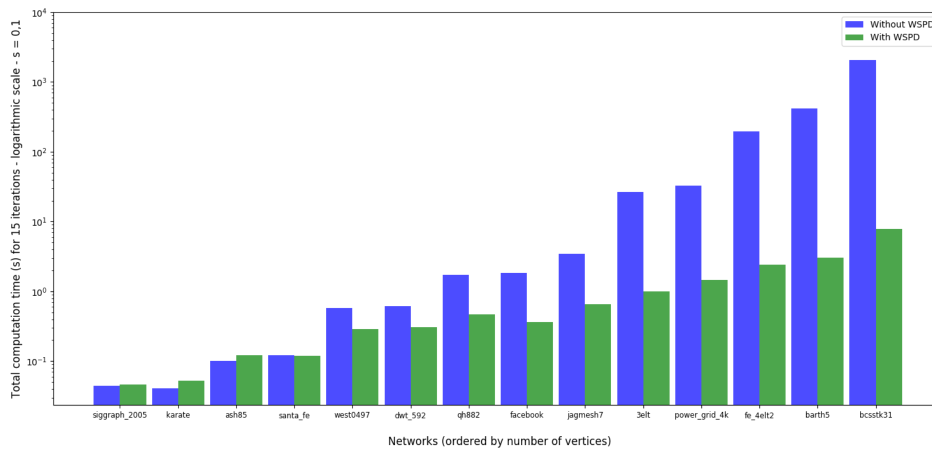

Figure 4: Graph drawing computation time (logarithmic scale) for $s = 0.1$

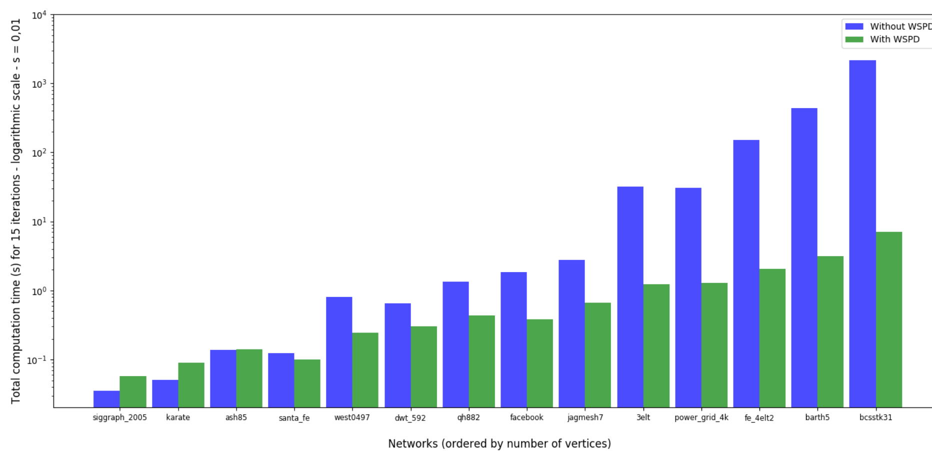

Figure 5: Graph drawing computation time (logarithmic scale) for $s = 0.01$

References

[1] Thomas MJ Fruchterman and Edward M Reingold. “Graph drawing by force-directed placement”. In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164.

[2] Sariel Har-Peled. “Geometric Approximation Algorithms”. In: 2008. Chap. 3. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.9927&rep=rep1&type=pdf.

[3] Fabian Lipp, Alexander Wolff, and Johannes Zink. “Faster Force-Directed Graph Drawing with the Well-Separated Pair Decomposition”. In: *Algorithms* 9.3 (2016), p. 53.