Well-Separated Pair Decomposition

Thorbjørn Søgaard Christensen
Department of Computer Science, University of Copenhagen

November 2, 2015

1 Introduction

A Well-separated pair decomposition is a geometric structure with multiple applications. The first part of this report explains the structure, how to compute it, and gives an application of it. The algorithm presented is based on algorithms from the book *Geometric Spaner Networks* (Narasimhan and Smid 2007) and the paper *The Well-Separated Pair Decomposition and Its Applications* (Smid 2007). The second part of this report describes the implementation of the algorithms in Java.

2 Well-Separated Pair Decomposition

To define the structure Well-separated pair decomposition (WSPD), we introduce the concept of well-separated set of points.

Well-separated Whether two sets are well-separated depends on a separation factor $s > 0$. We say that two sets of points A and B are well-separated, with respect to a separation factor s iff there exist two balls C_A and C_B, of radius ρ such that the smallest axis-parallel bounding box of A is contained in C_A and the smallest axis-parallel bounding box of B is contained in C_B, and the smallest distance between C_A and C_B is greater or equal to $\rho \cdot s$.
This definition is illustrated in Figure 1, where the points are in the plane, but the definition holds for points in arbitrary dimensions.
We note that two point sets A and B, both of size 1, always will be well-separated, since the radius of the balls C_A and C_B is 0. Thus, two sets of point A containing only point a, and B containing only point b, where $dist(a, b) = 0$, is well-separated for every choice of s, since the radius of C_A and C_B are 0, and our well-separated condition $dist(C_A, C_B) \geq \rho \cdot s$ is always satisfied.

Having defined well-separated pairs, we are now ready to define a Well-separated pair decomposition.

Well-separated pair decomposition For a given set of points $S \in \mathbb{R}^d$, where $|S| > 1$, a Well-separated pair decomposition is a collection of pairs P_i, each consisting of two **well-separated** sets of points $A_i \subset S$ and $B_i \subset S$. For a well-separated pair decomposition the following condition must hold:

- For each two points $q \in S$ and $p \in S$, where $q \neq p$, there exists **exactly** one i, such that $p \in A_i$ and $q \in B_i$ or $q \in A_i$ and $p \in B_i$.

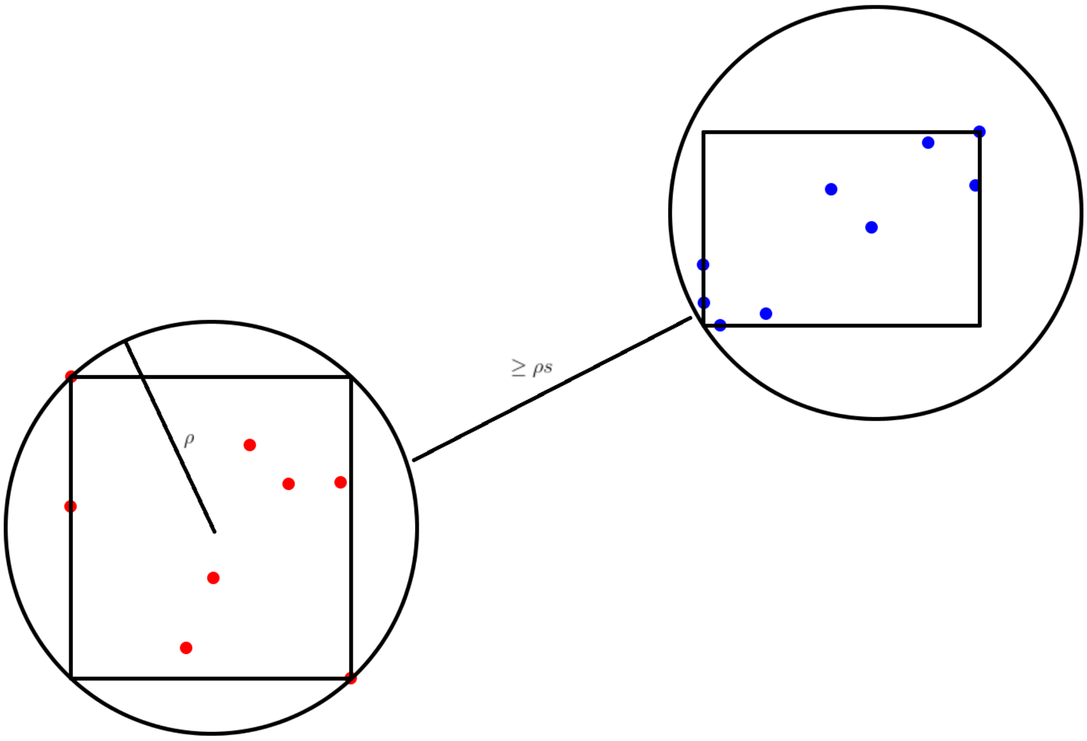


Figure 1: The red and the blue points are well-separated with respect to s iff the smallest distance between the circles are greater than or equal to $s \cdot \rho$, where ρ is the radius of the two circles. Notice that both circles have the same radius.

We define the **size** of our decomposition as the number of pairs, and does not allow any pairs containing an empty set.

A WSPD exists for any point set, which we can show by the following construction: For every two points $p, q \in S$ where $p \neq q$, let the WSPD contain the pair $(\{p\}, \{q\})$. This construction is a valid WSPD, since every two sets of size 1 is well-separated, as discussed in the previous section. The size of this construction will be $O(n^2)$, since we take all possible pairs of points. In this report we show how to construct a WSPD of size $O(n)$, when fixing separation factor s and number of dimensions d as constants.

To get a better understanding of WSPD, before discussing how to compute it and which applications it have, we look at some examples. We say that the pair P_i **separates** points a and b, if $p \in A_i$ and $q \in B_i$ or $q \in A_i$ and $p \in B_i$. The most basic example is a WSPD for a set of size 2. If $S = \{a, b\}$ then there exists a WSPD of size 1, where $P_0 = (\{a\}, \{b\})$. So why does the size have to be exactly 1? If the size was 0, no P_i could separate a and b, and the first condition for WSPD would be violated. Similar if the size was greater than 1, then there must be more than one i where P_i separates a and b, and the second condition would be violated. From our discussion of well-separated sets of size 1, we know that the two sets $\{a\}$ and $\{b\}$ are well-separated for any separation factor s. Thus a WSPD of size 1 exists for any S, where $|S| = 2$. However the case where $|S| = 2$ is not very interesting for two reasons: 1) The separation factor is not used, and 2) The locations of the points are not used.
To obtain a better understanding of the definition, let us look at a WSPD for a set S, where $|S| > 2$, and for the illustration purpose let the points be in the plane.

Figure 2 shows a WSPD for a set of 6 points for four different values of s. Each line segment L_i illustrates a pair P_i in the decomposition, and the sets A_i and B_i are all the points inside or on the edge, of the two circles connected by L_i. In the case where either A_i or B_i has size 1, the circle is just illustrated as the point itself.
On the figure we can validate that for any two points p and q there exist exactly one line segment L_i, connecting C_{A_i} and C_{B_i} where $p \in A_i$ and $q \in B_i$ or $q \in A_i$ and $p \in B_i$.

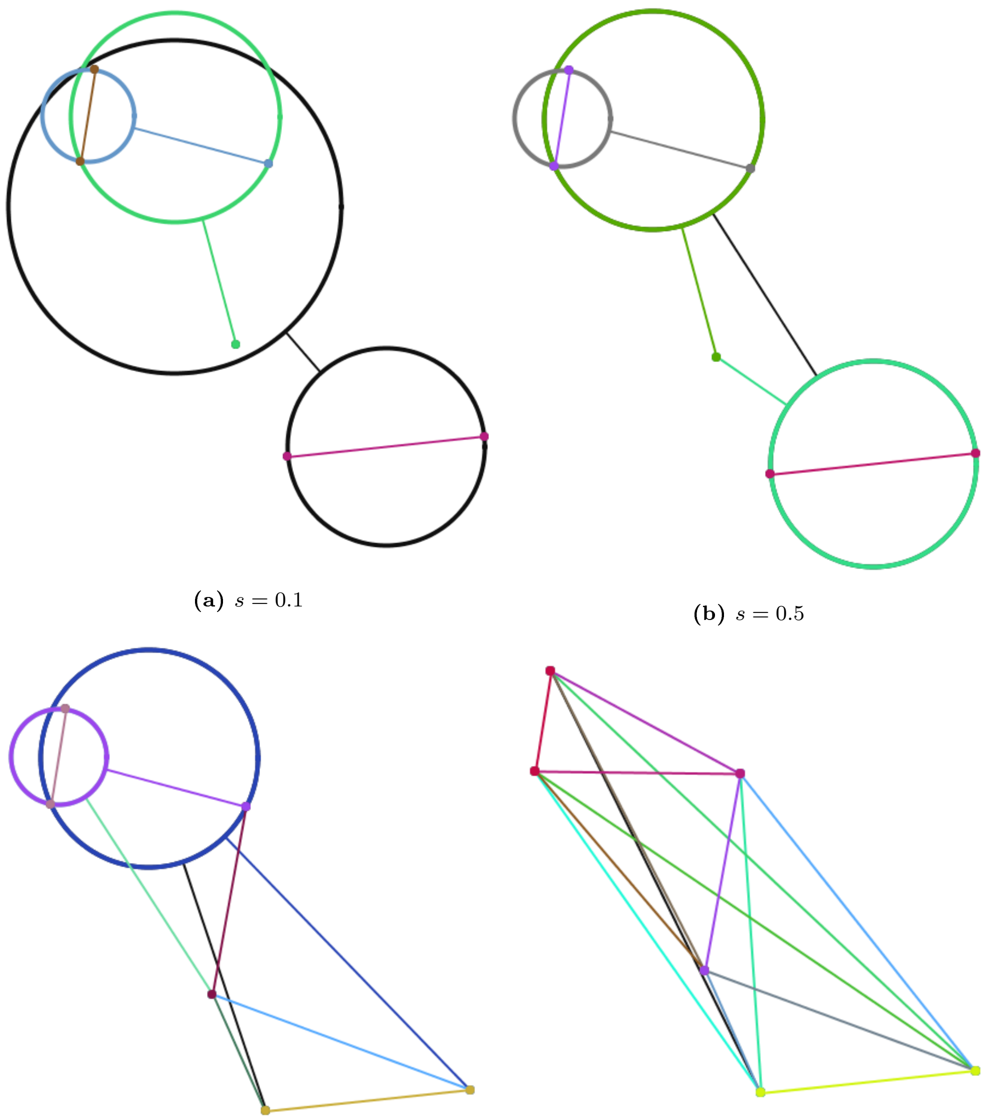
2.1 Computing WSPD

To compute a Well-separated pair decomposition, we need to compute another structure called a **Split tree** for the point set. This structure does not depend on the separation factor s. From the split tree and the separation factor s, we can then compute the WSPD. We start by defining the split tree, and how we can compute it.

Split tree Let $R(S)$ define the smallest axis-parallel bounding box of a point set S. A split tree T for a point set S, is a binary tree defined recursively by the following conditions:

- If $|S| = 1$ then T consists only of a single node storing the single point in S.
- If $|S| > 1$ we split $R(S)$ into two boxes R_1 and R_2, by splitting $R(S)$ in the middle in the dimension where $R(S)$ has the maximum length. Let S_1 be the set of points inside or on the edge of R_1, and let $S_2 = S \backslash S_1$. The split tree for S is then a root, which children are split trees defined recursively for S_1 and S_2.

To use a split tree to compute the Well-separated pair decomposition, we store the following information inside each node v in our tree:


(a) $s = 0.1$

(b) $s = 0.5$

(c) $s = 2$

(d) $s = 10$

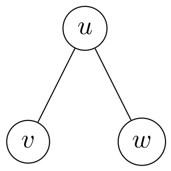
Figure 2: WSPD of a point set of size 6 for different values of s.

1. A point set S, which consists of all points contained by some leaf, in the subtree of v.
2. The smallest axis-parallel bounding box for S, denoted $R(S)$.

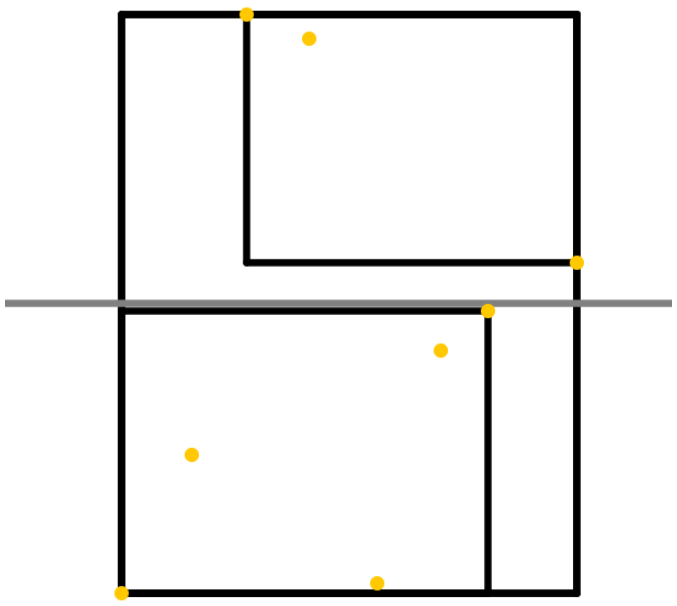
Computing the split tree From the definition of the split tree we get an idea of how to compute a split tree for a point set: A recursive algorithm, where the base case is $|S| = 1$. This algorithm is presented in **Algorithm 1**.
Figure 3 shows how the algorithm works in one step, where it splits the big rectangle (by the grey line) in the middle in the dimension, where the rectangle is longest. After this splitting the

Algorithm 1: A recursive algorithm for computing the Split tree, with the additional information used to compute the Well-separated pair decomposition.

```
1  SplitTree ($S$):
     Input  : A point set $S$ of size $n$ in $\mathbb{R}^d$.
     Output: The *root* of the split tree for $S$, where each node $v$ contains all the additional
             information listed above.
2  if $n = 1$ then
3      Create new node $u$
4      Compute bounding box $R(u)$ of the set $S$
5      Store $R(u)$ with the node $u$
6      return $u$
7  else
8      Compute minimum axis parallel bounding box $R(S)$, by running through each point of
       $S$, and determining the upper and lower bounds in each dimension
9      Let $i$ be the dimension where $R(S)$ has its longest side
10     Let $l_i$ be the minimum of $R(S)$ in the $i$'th dimension, and let $r_i$ be the maximum
11     Let $H$ be the hyperplane with equation: $x_i = (l_i + r_i)/2$
12     Use $H$ to split $R(S)$ into $R_1$ and $R_2$
13     Let $S_1$ be the subset of $S$, where the points are contained in $R_1$
14     Let $S_2 = S \backslash S_1$
15     $v = SplitTree(S1)$
16     $w = SplitTree(S2)$
17     Create new node $u$
18     Store $R(S)$ with the node $u$
19     $u.leftChild = v$
20     $u.rightChild = w$
21     return $u$
22 end
```



(a) The split tree after the first iteration.



(b) The splitting used in the first iteration.

Figure 3: Showing the first step of the SplitTree algorithm for a point set of size 8 in the plane.

recursive calls compute the two bounding boxes for the subset. The next step of the algorithm is to split the two smaller rectangles, but for illustration purposed only the first step of the recursion is shown here. The big rectangle in the figure corresponds to the root u and the two smaller rectangles correspond to the two children of u, v and w.

Running time of the SplitTree algorithm Lines 8-21 in the algorithm take $O(|S|)$, since we run through every point to compute the bounding box and to split S into S_1 and S_2. If we look at one layer in the recursion, using the sets $\{S_1, \dots S_m\}$, we know that

$$|S_1| + \cdots + |S_m| \leq |S| = n$$

since each recursion splits $|S|$ into distinct subsets. Thus, the running time of one layer in the recursion is $O(n)$. The total running time must then be $O(n \cdot h)$, where h is the height of the computed split tree. Since the splitting only guarantee that each of the subset has size at least 1, the splitting could result in sets of size 1 and $n - 1$ each time, giving a worst-case height of $O(n)$. Thus, the worst case running time for this algorithm is $O(n^2)$.
It is possible to compute the same split tree in worst case $O(n \log n)$-time, by doing the splitting in a specific order. To obtain this order we introduce a partial split tree.

Partial split tree A partial split tree is define in the same way as a split tree, with the exception that the leaves can contain up to $|S|/2$ points, instead of only a single point. The algorithm to compute a partial split tree efficiently needs the following input:

1. The point set S of size n, where $S \subseteq \mathbb{R}^d$ and $n \geq 2$.

2. A collection of doubly-linked lists LS_i for $0 \leq i \leq d$, where LS_i contains the n points sorted in non-decreasing order of their i'th coordinate. In addition, any point p in one of the LS_i lists has $d-1$ cross-pointers to the same point in all lists LS_j where $j \neq i$.

Before diving into how the algorithm proceeds, we will see how the given collections of doubly-linked lists look for a given set. For a point set $A \subset \mathbb{R}^4$, containing the following 5 points: $p_1 = (1,0,4,6)$, $p_2 = (2,4,9,1)$, $p_3 = (5,6,2,4)$, $p_4 = (7,9,7,7)$ and $p_5 = (9,2,1,5)$, Figure 4 shows the given collection of doubly-linked lists LS_i for $1 \leq i \leq 4$, where LS_i contains the points in non-decreasing order of the i'th coordinate.
Figure 5 shows the cross-pointers, for LS_1, where every point p in the list has a cross-pointer to the same point p in every other lists. Notice that every LS_i for $1 \leq i \leq 4$, has these cross-pointers, but only cross-pointers from LS_1 are shown in this figure. When explaining the algorithm, we will continue to use this set A to illustrate each step of the algorithm.

LS_1 :	(1,0,4,6)	(2,4,9,1)	(5,6,2,4)	(7,9,7,7)	(9,2,1,5)
LS_2 :	(1,0,4,6)	(9,2,1,5)	(2,4,9,1)	(5,6,2,4)	(7,9,7,7)
LS_3 :	(9,2,1,5)	(5,6,2,4)	(1,0,4,6)	(7,9,7,7)	(2,4,9,1)
LS_4 :	(2,4,9,1)	(5,6,2,4)	(9,2,1,5)	(1,0,4,6)	(7,9,7,7)

Figure 4: The lists LS_i, $0 \leq i \leq 4$, for the point set A.

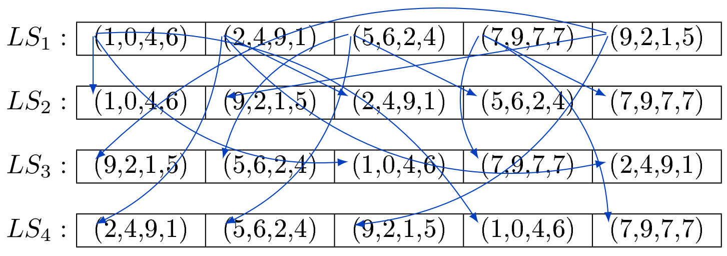
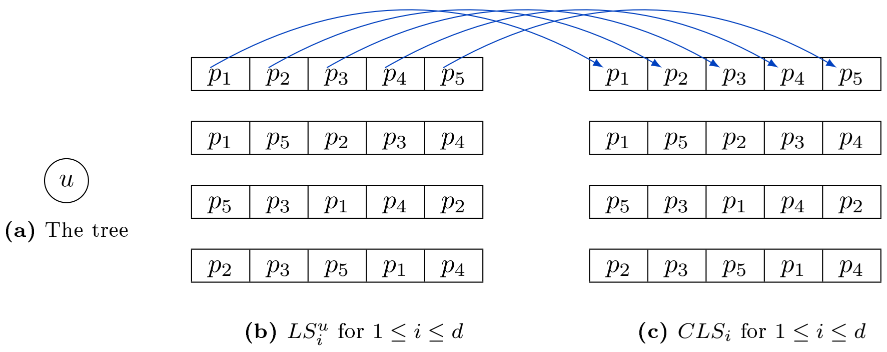


Figure 5: The cross pointers from the list LS_1.

We divide the algorithm for computing a partial split tree into 3 parts, **(1)** a preprocessing step, only running a single time for each call to the algorithm, **(2)** the main part of the algorithm, which can run multiple times and splits the point set until every leaf contains at most $n/2$ points. The final part of the algorithm is **(3)** a post-processing step, which only runs a single time. We denote the algorithm **PartialSplitTree**$(\boldsymbol{S},\ (\boldsymbol{LS_i})_{\boldsymbol{1 \leq i \leq d}})$[1] and each part of the algorithm is explained below:

[1]While all other algorithms is written in pseudo-code, this algorithm is written in text, since (1) the complexity of the pseudo-code would be large, and (2) the algorithm is easier to understand, when each step is explained along with an example.



u
(a) The tree
p_1 p_2 p_3 p_4 p_5
p_1 p_5 p_2 p_3 p_4
p_5 p_3 p_1 p_4 p_2
p_2 p_3 p_5 p_1 p_4
(b) LS_i^u for $1 \leq i \leq d$
(c) CLS_i for $1 \leq i \leq d$

Figure 6: The structures after the preprocessing step for the set A. Only cross-pointers from LS_1^u to CLS_1 is showed, the rest is omitted.

(1) Preprocessing: Given the lists LS_i, $1 \leq i \leq d$ we copy those and their cross-pointers into CLS_i. For each point in any of the LS_i lists, we add a cross-pointer to the corresponding point in CLS_i. We create a node u, which is going to be the root of the partial split tree, and rename S as S_u and LS_i, $i \leq i \leq d$ as LS_i^u. We define $size = n$. Figure 6 shows cross-pointers between LS_1^u and CLS_1 after the preprocessing step. Internal cross-pointers in LS_i^u and CLS_i for $1 \leq i \leq d$, and cross-pointers between LS_i and CLS_i, $2 \leq i \leq d$ is omitted in Figure 6.
By copying the lists, we can ensure that we run through at most $O(n)$ points in LS_i, $1 \leq i \leq d$, by deleting points we no longer need, while CLS_i, $1 \leq i \leq d$ keeps the original structure of the points, which we will use to track which leaf a particular point is added to.

(2) The main part: The main part starts by computing the minimal axis-parallel bounding box $R(S_u)$ for S_u, by using the lists LS_i^u for $1 \leq i \leq d$. We then compute the index j, such that $R(S_u)$ has the maximal length in the j'th dimension. We define $x_j = (l_j - r_j)/2$, where l_j is the lower bound of $R(S_u)$ in the j'th dimension, and r_j is the upper bound in the j'th dimension.
We define p_f as the first point of LS_j^u, and p_l as the last, and use the notation that $p(i)$ denotes the i'th coordinate of the point p. We notice that $p_f(j) \leq x_j$ and $p_l(j) \geq x_j$ since LS_j^u is sorted in non-decreasing order by the j'th coordinate. Now we use the fact that LS_j^u is a doubly-linked list, to run in parallel through the list from both ends, until we either find a point from the beginning of the list p_b where $p_b(j) \leq x_j$, or we find a point from the end of the list p_e where $p_e(j) \geq x_j$.
We now look at an example of this step using the set A previously introduced. We start by computing the bounding box of A, which we can do by running through LS_i^u for $1 \leq i \leq d$, where the lower and upper bound in dimension i, is respectively the first and last point of LS_i^u. If we express the bounding box as the lower and upper bound in each dimension from 1 to d, we get that $R(A) = [1, 9], [0, 9], [1, 9], [1, 7]$. Since $R(A)$ has maximal length in the second dimension, we define $x_2 = (l_2 - r_2)/2 = (9 - 0)/2 = 4.5$. Figure 7 shows how we run through the list LS_2 from both ends, until we find a point from the beginning where the second dimension is greater than or equal to 4.5, or we find a point from the end where the second dimension is smaller than or equal to 4.5. In this particular case our search stops when encountering a point in the middle of the list, but in general this search could stop anywhere between the 2nd point from the beginning of the list and the 2nd point from the end of the list.

Later we will delete some points S_{del} from LS_i, $1 \leq i \leq d$, and by encountering the list in parallel from the start and the end, we only encounter $O(|S_{del}|)$ points, which is crucial for the efficiency of this algorithm.

(1,0,4,6)	(9,2,1,5)	(2,4,9,1)	(5,6,2,4)	(7,9,7,7)

(a) We start in both ends of the list, p_f is the first point, p_l is the last. We know that $p_f(2) \leq 4.5$ and $p_l(2) \geq 4.5$.

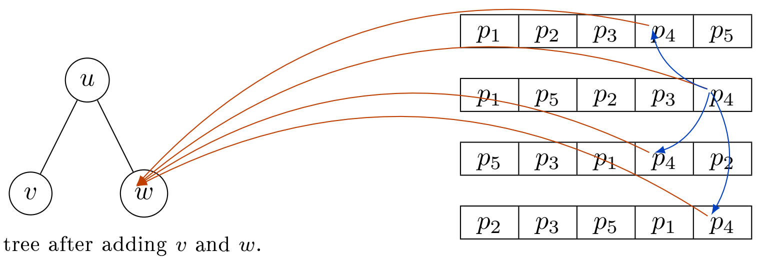
(1,0,4,6)	(9,2,1,5)	(2,4,9,1)	(5,6,2,4)	(7,9,7,7)

(b) We continue by looking at the successor p'_f of p_f and the predecessor p'_l of p_l. Since $p'_f(2) \leq 4.5$ and $p'_l(2) \geq 4.5$ we continue.

(1,0,4,6)	(9,2,1,5)	(2,4,9,1)	(5,6,2,4)	(7,9,7,7)

(c) In this step we see that the predecessor p''_l of p'_l does not no longer satisfy that $p''_l(2) \geq 4.5$, so we stop.

Figure 7: The parallel run through of LS_2, in the example with the set A.



u
v
w
p_1 p_2 p_3 p_4 p_5
p_1 p_5 p_2 p_3 p_4
p_5 p_3 p_1 p_4 p_2
p_2 p_3 p_5 p_1 p_4

(a) The tree after adding v and w.

(b) CLS_i for $1 \leq i \leq d$.

Figure 8: We run trough LS_2 backwards, and for each point we follow the cross-pointers to the copy z_c in CLS_2. For each z_c we follow all its cross-pointers to all other instances of z_c in any list CLS_i, from where we add a pointer to the node w. Here we show the used cross-pointers (blue lines) and the added pointers to u (red lines) for the first point encountered (p_4 in CLS_2).

The next step depends on whether we found a point p_f where $p_f(j) \geq x_j$ from the beginning of the list, or we found a point p_l where $p_l(j) \leq x_j$ from the end of the list. For now, assume the latter is the case, as in our example, and let us see how we handle the other case later.
Let p_l be the point that we found from the end of the list, and let p'_l be the successor of p_l. Notice that p'_l is now the leftmost point in LS^u_j whose j'th coordinate is greater than x_j.
Now we create the nodes v and w and set them to respectively the left and right child of u. We run through the list LS^u_j from right to left until we reach p'_l, and for each point z we encounter (including p'_l) we do the following:

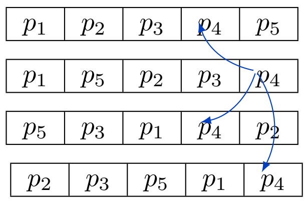
S.1 From z follow the cross-pointer to occurrence of z in each list CLS_i, $1 \leq i \leq d$, and store a pointer to node w in each of them.

S.2 Use the cross-pointers from z to all other occurrence of z_i in each list LS^u_i where $i \neq j$, and delete z_i from LS_i.

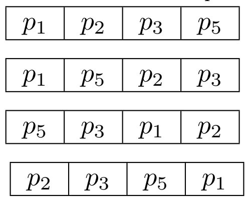
S.3 Delete z from LS^u_j.

Figure 8 shows how step **S.1** works in our example with set A, and Figure 9 shows how **S.2** and **S.3** works. Notice that the figures only show the step for p_4, and that the step should be repeated for p_3 as well.

Let $size'$ be the number of points z we just encountered. We know that $size' < n/2$ since $size'$ is the size of the smaller set of the partitioning of S_u. Because of this, there are less than $n/2$ points in CLS_i for any $1 \leq i \leq d$, which have a pointer to node w. Since the number of points pointing


p_1 p_2 p_3 p_4 p_5
p_1 p_5 p_2 p_3 p_4
p_5 p_3 p_1 p_4 p_2
p_2 p_3 p_5 p_1 p_4

(a) Use the cross-pointers from LS_2 to find the points to delete in the other lists



(b) LS_i for $1 \leq i \leq d$ after deleting z in all the lists.

Figure 9: Using the cross-pointers in LS_i for $1 \leq i \leq d$ to find z in every list, and delete it.

to w is less than $n/2$, we do not need to split w any more. Now we update $size = size - size'$, such that $size$ is the number of points remaining in LS_i^u for any $1 \leq i \leq d$. If $size \leq n/2$ we let the remaining points have a pointer to v, and we continue with the post-processing step. If this is not the case, we set $u = v$ and recursively run the main part of the algorithm.

Before explaining the post-processing step, we clarify how the last step changes, if we encounter a point p_f where $p_f(j) \geq x_j$ from the beginning of the list, instead of a point p_l where $p_l(j) \leq x_j$ from the end of the list. Instead of running the list backwards until p_l was encountered, we run the list forward until p_f is encountered. Instead of taking the successor of p_l, we take the predecessor of p_f. Instead of adding pointers to w, we add pointers to v, and instead of setting $u = v$ before calling recursively, we set $u = w$.
In our example with point set A, we have to call recursively one time, since we have 3 points remaining and $3 \geq |A|/2$. The final partial split tree for A is showed in Figure 10.

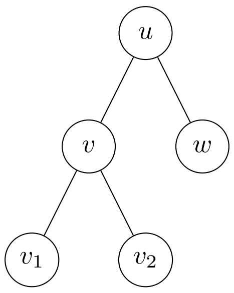


Figure 10: The partial split tree for A, where each leaf contains at most $n/2$ points.

Post-processing In the following we denote a_{cp} as the point in CLS_i, where the point a in LS_i^u has a cross-pointer to.
LS_i^u for $1 \leq i \leq d$ only contains points z, where z_{cp} does not point to any node, since the rest was deleted. We start by going through each of the lists LS_i^u for $1 \leq i \leq d$, and for each point z we set z_{cp} to point at node u. Let T denote the tree constructed so far. We have now established

the following: For each leaf l of T, there is at most $n/2$ points in any list CLS_i for $1 \leq i \leq d$, which has a pointer to l, and every point has a pointer to some leaf.
For each leaf l we create empty lists LS_i^u for $1 \leq i \leq d$.
Then we walk forward along CLS_i for $1 \leq i \leq d$ and for each point z_i we encounter, we add z_i to the end of LS_i^l, where l is the leaf that z_i points to.
The only thing left to do, is to compute the minimum axis-parallel bounding box $R(l)$ for each leaf l of T. This is easily done, since LS_i^l is sorted for each dimension, meaning that the bounds of $R(l)$ in dimension i is at the first and last element of LS_i^l. We store $R(l)$ and LS_i^l for $1 \leq i \leq d$ with the leaf l.

Figure 11 shows how LS_i^w for $1 \leq i \leq d$ looks in our example with point set A. Since we did not change any ordering in the list, LS_i^w is sorted in the i'th dimension. After the post-processing step we have established similar lists for the leafs v_1 and v_2. We will later see, how we use these lists along with the fact that they are sorted, to compute a split tree from a partial split tree efficiently.

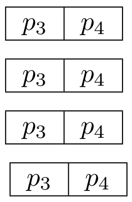


Figure 11: LS_i^w for $1 \leq i \leq d$, where cross-pointers is omitted.

Let S_u denote the set of points in LS_i^u for any i, $1 \leq i \leq d$. Now we have an algorithm, which computes a partial split tree T for a set S, where each leaf u has $|S_u| \leq |S|/2$. Each node v of T has the minimum axis-parallel bounding box $R(v)$ stored, and leaves u of T has in addition LS_i^u for $1 \leq i \leq d$ stored, where points associated with u is sorted for each of the d dimensions.

Running time of PartialSplitTree(S, $(LS_i)_{1 \leq i \leq d}$): The preprocessing step takes $O(n)$ time. The main step takes $O(\sum_u |S_u|)$ time, where u is a leaf in the computed tree. Since $S_{u_1} \cap S_{u_2} = \emptyset$ for any two distinct leaves u_1 and u_2, we get that $O(\sum_u |S_u|) = O(n)$. The post-processing step can be performed in $O(n)$. So the total time to compute a partial split tree is $O(n)$.

Computing a split tree faster Since we are able to compute a partial split tree in $O(n)$ time, we can compute a split tree in $O(n \log n)$ time by the algorithm showed in **Algorithm 2**. This algorithm starts by sorting all points d-times and add cross-pointers. After this we can use our PartialSplitTree algorithm. The reason that we divide part of the algorithm into a sub-routine, is due to the following crucial observation: We only need to sort the points and add cross-pointers before the first call to PartialSplitTree. For each leaf u in the computed partial split tree, the points are already sorted in the lists LS_i^u, $1 \leq i \leq d$ and cross-pointers are already established. The recursion will stop when $n = 1$, meaning that each leaf in the computed tree will contain only a single point and thus, the returned tree is a valid split tree.

Now that we know how to compute a split tree efficiently, we can focus on how to compute a Well-separated pair decomposition from a split tree.

Algorithm 2: An algorithm for computing the split tree by using partial split trees.

```
1  FastSplitTree ($S$):
   Input  : A point set $S \subseteq \mathbb{R}^d$.
   Output: The root of the split tree for $S$.
2  for $i$ from 1 to $d$ do
3      Create doubly linked list $LS_i$
4      Add all points from $S$ to $LS_i$
5  end
6  Add cross-pointers for each point in $(LS_i)_{1 \leq i \leq d}$ to the same point in all lists $(LS_j)_{j \neq i}$
7  for $i$ from 1 to $d$ do
8      Sort $LS_i$ in by the $d$'th coordinate in non-decreasing order
9  end
10 return SubRoutine($S, (LS_i)_{1 \leq i \leq d}$)
11 SubRoutine ($S, (LS_i)_{1 \leq i \leq d}$):
12 if $|S| = 1$ then
13     return SplitTree($S$)
14 else
15     $r$ = PartialSplitTree($S, (LS_i)_{1 \leq i \leq d}$)
16     Let $T$ be the tree, where $r$ is root
17     for each leaf $u$ in $T$ do
18         $u$ = SubRoutine($S_u, (LS_i^u)_{1 \leq i \leq d}$)
19     end
20     return $r$
21 end
```

Getting a WSPD from a split tree From a split tree T and a separation factor s, we can compute a Well-separated pair decomposition, by **Algorithm 3**, where $L_{max}R(S_v)$ is denoting the maximum length of any sides of the hyperrectangle $R(S_v)$.
Since each leaf of T contains only a single point, and two point sets of size 1 always are well-separated we are guaranteed that the algorithm will terminate. We recall the 2 conditions for WSPD, (1) every pair consists of well-separated sets and (2) for every two distinct points there exist exactly one pair, which separate them. **Algorithm 3** computing the WSPD W from a split tree T satisfy these 2 conditions since: (1) we only add well-separated pairs to W, and (2) for each two points p and q, where the leaf l_p containing p comes before the leaf l_q containing q in a pre-order traversal of T, we have added some pair (A_i, B_i) to W, where A_i corresponds to either the leaf l_p or some ancestor of l_p, and B_i corresponds to either the leaf l_q or some ancestor of l_q.

Running time of the WSPD algorithm (Narasimhan and Smid 2007) uses a packing argument to show that the total number of insertions into the result set is $O(s^d n)$, and by using the additional bounding box information of each split tree node, we are able to test if two sets (corresponding to two nodes of T) is well-separated in $O(1)$-time, since we have the axis-parallel minimum bounding box of both sets. This gives us a total running time of $O(n)$ for computing a WSPD from a split tree, when d and s are fixed constants. Since it is possible to compute the split tree in $O(n \log n)$-time, we will be able to compute a WSPD for any given set in $\mathbb{R}^d$ in $O(n \log n)$ time.

Algorithm 3: An algorithm for computing the WSPD from a split tree.

```
1  ComputeWSPD $(T, s)$:
   Input  : A split tree $T$ for point set $S$, and a separation factor $s$.
   Output: A WSPD for the point set $S$.
2  Initialize empty $result$ set
3  Initialize empty queue $Q$
4  for each internal node $u$ in $T$ do
5      $v = u.leftChild$
6      $w = u.rightChild$
7      insert$(v, w)$ into $Q$
8  end
9  while $Q$ is not empty do
10     $(v, w) = Q.next$
11     if $v$ and $w$ is well-separated with respect to $s$ then
12         Add $(v, w)$ to the $result$ set
13     else
14         if $L_{max}(R(S_v)) > L_{max}(R(S_w))$ then
15             $v_1 = v.leftChild$
16             $v_2 = v.rightChild$
17             Add $(w, v_1)$ and $(w, v_2)$ to $Q$
18         else
19             $w_1 = w.leftChild$
20             $w_2 = w.rightChild$
21             Add $(w_1, v)$ and $(w_2, v)$ to $Q$
22         end
23     end
24 end
25 return the $result$ set
```

3 Applications

Having defined the Well-separated pair decomposition, and showed how to compute it efficiently, let us give an example of an application for WSPD.

3.1 t-spanners

A t-spanner is a graph $G = (S, E)$ for a point set S in $\mathbb{R}^d$ with the following property: For any two points $p, q \in S$, we have that $dist(pq) \leq t \cdot dist_G(pq)$, where $dist(pq)$ denotes the distance between p and q, and $dist_G(pq)$ denotes the length of the shortest path in G, between p and q. We call t the stretch factor of the spanner. Before we determine how to obtain a t-spanner from a WSPD, we need the following corollary, which basically states that for any point sets A and B, which are well-separated with respect to some large s, (1) the distance between two points in A is much smaller than the distance between an arbitrary point in A and an arbitrary point in B, and (2) distances between every pair of points (a, b), where $a \in A$ and $b \in B$ are approximately equal.

Corollary 3.0.1 (Narasimhan and Smid 2007) *Let two point sets A and B be well-separated with respect to some $s > 0$. Let $p, p' \in A$ and $q, q' \in B$ then*

1. $|pp'| \leq (2/s)|pq|$
2. $|p'q'| \leq (1 + 4/s)|pq|$

Proof: Since A and B are well-separated, there exist balls C_A containing A and C_B containing B, with the same radius ρ, so the smallest distance between C_A and C_B is greater than or equal to ρs. Since C_A contains p and C_B contains q we get that $|pq| \geq \rho s$, and since C_A contains both p and p' we have that $|pp'| \leq 2\rho$. Combining these gives us

$$|pp'| \leq 2\rho \leq 2 \cdot (|pq|/s) = (2/s)|pq|$$

proving **(1)**. A symmetric argument gives that $|qq'| \leq (2/s)|pq|$. By using the triangle inequality and **(1)** we get

$$\begin{aligned} |p'q'| &\leq |p'p| + |pq| + |qq'| \\ &\leq 2 \cdot (2/s)|pq| + |pq| \\ &= (4/s)|pq| + |pq| \\ &= (1 + 4/s)|pq| \end{aligned}$$

proving **(2)**.

Now, we can show the following lemma which gives us a way to easily compute a t-spanner from a Well-separated pair decomposition:

Lemma 3.1 (Smid 2007) *Given a WSPD $W = \{A_0, B_0\}, \ldots, \{A_m, B_m\}$ for S with a separation factor $s > 4$, define $G = (S, \{(a_i, b_i) : 1 \leq i \leq m\})$, where a_i and b_i are respectively arbitrary points from A_i and B_i. Then G is a $(s + 4)/(s - 4)$-spanner for S.*

Proof: Let $t = (s+4)/(s-4)$ for some $s > 4$. The proof is by induction on increasing distances between points. We consider two points p and q, where $p = q$. Then $dist_G(pq) = dist(pq)$, which satisfy $dist_G(pq) \leq t \cdot dist_G(pq)$. Now we consider two points p and q where $p \neq q$, and for any two points p' and q' where $dist(p', q') < dist(p, q)$, we assume that $dist(p', q') \leq t \cdot dist_G(p', q')$. Let i be the index such that $p \in A_i$ and $q \in B_i$. By using the triangle inequality and Corollary 3.0.1 we get that

$$\begin{aligned}
dist_G(pq) &\leq dist_G(pa_i) + dist_G(a_ib_i) + dist_G(b_iq) \\
&= dist_G(pa_i) + dist(a_ib_i) + dist_G(b_iq) \\
&\leq t \cdot dist(pa_i) + dist(a_ib_i) + t \cdot dist(b_iq) \\
&\leq t \cdot (2/s)dist(pq) + (1 + 4/s)dist(pq) + t \cdot (2/s)dist(pq) \\
&= (2t/s + (1 + 4/s) + 2t/s)dist(pq) \\
&= \left(\frac{4(t+1)}{s} + 1\right) dist(pq)
\end{aligned}$$

By rewriting s to $s = 4(t+1)/(t-1)$ we get

$$\begin{aligned}
\left(\frac{4(t+1)}{s} + 1\right) dist(pq) &= \left(\frac{4(t+1)}{4(t+1)/(t-1)} + 1\right) dist(pq) \\
&= \left(4(t+1)\frac{(t-1)}{4(t+1)} + 1\right) dist(pq) \\
&= (t - 1 + 1)dist(pq) \\
&= t \cdot dist(pq)
\end{aligned}$$

We have showed that $dist(pq) \leq t \cdot dist_G(pq)$ if $s > 4$, for any choice of $p, q \in S$, proving that G is a t-spanner for S.

To obtain a t-spanner for a set S, we start by computing a WSPD $W = \{A_0, B_0\}, \ldots, \{A_m, B_m\}$ for S, where the separation factor $s = 4(t+1)/(t-1)$. Let E be an empty edge set, and for every i, $0 \leq i \leq m$, add edge (a_i, b_i) to E, where a_i is any points in A_i and b_i is any point in B_i. $G = (S, E)$ is then a t-spanner for S, computed in $O(n \log n)$-time.
Figure 12 shows an example of obtaining a t-spanner from a WSPD. In this example $t = 5$, and thus, we have to compute the WSPD with a separation factor of 6.

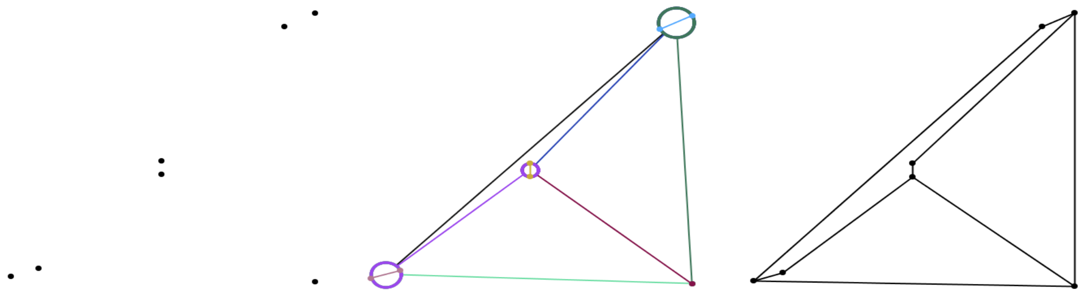

(a) Point set S. **(b)** WSPD for S, where $s = 6$. **(c)** A 5-spanner for S.

Figure 12: Using a WSPD of S, to get a t-spanner for S

4 Implementing WSPD in Java

The algorithms presented in last section is implemented in Java. The code for this implementation is found here:
`https://github.com/thorebear/Well-Separated-Pair-Decomposition`
The implementation has 3 different ways of computing a Well-separated pair decomposition:

- **2d-Slow:** Computing WSPD in 2 dimensions, using the worst-case $O(n^2)$-algorithm (Algorithm 1) for computing the split tree. This 2-dimension implementation has two advantages (1) The code is easier to understand in opposition to the general n-dimension implementation, which makes it a good entry point to understand the code base, and (2) the WSPD can be illustrated by circles and lines (as in Figure 2).
- **nd-Slow:** Computing WSPD in n dimensions, using the worst-case $O(n^2)$-algorithm (Algorithm 1) for computing the split tree.
- **nd-Fast:** Computing WSPD in n dimensions, using the worst-case $O(n \log n)$-algorithm (Algorithm 2) for computing the split tree.

The code includes 3 samples, which show how to use each of these different ways of computing a Well-separated pair decomposition. The sample showing how to use the algorithm for point sets in the plane will also show a drawing of the WSPD. How to run these samples are described on the GitHub repository.

4.1 Usage of ProGAL

The code uses the Java-library ProGAL (`www.diku.dk/~rfonseca/ProGAL/`) for doing some geometric computations and to illustrate the WSPD in the plane.

4.2 Code structure

The code is structured similar to the ProGAL code and contains the following Java-packages:

- `geom2d`: Contains general code for geometric computations in 2 dimensions.
- `geom2d.wspd`: Contains WSPD specific code for computations in 2 dimensions.
- `geomNd`: Contains general code for geometric computations in n dimensions.
- `geomNd.wspd`: Contains WSPD specific code for computations in n dimensions.
- `dataStructures`: Contains structures that can be used by both the 2d and nd implementation.

4.3 Evaluation

Table 1 shows the approximately number of points for which, we can compute a WSPD in 10 seconds, with **nd-fast** for dimensions 2 to 8 and assuming that the separation factor is low enough, such that the time for computing the WSPD from the split tree is dominated by the time for computing the split tree. Obviously if s is very large, we cannot compute the WSPD in 10 seconds for $n = 100000$ points since the size of the WSPD will be $\binom{n}{2}$. Of cause these numbers are hardware depended, but they show how much harder the problem is to solve, when the number dimensions increases.

Table 1: Showing approximately how many points we can compute a WSPD for using nd-fast (in 10 seconds).

Dim	Points
2	100000
3	70000
4	45000
5	28000
6	19000
7	10000
8	5000

References

Narasimhan, Giri and Michiel Smid (2007). “The Well-Separated Pair Decomposition.” In: *Geometric Spanner Networks, Cambridge Press, 151-177.*

Smid, Michiel (2007). “The Well-Separated Pair Decomposition and Its Applications”. In: *Handbook of Approximation Algorithms and Metaheuristics (Teofilo Gonzalez, editor).*