Fast Algorithms and Efficient Statistics: N–point Correlation Functions

A. J. Connolly[1]**,**
Department of Physics & Astronomy, University of Pittsburgh
ajc@phyast.pitt.edu
C. Genovese,
Department of Statistics, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
genovese@stat.cmu.edu
A. Gray,
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
agray@cs.cmu.edu
A. W. Moore,
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
awm@cs.cmu.edu
R. C. Nichol,
Department of Physics, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
nichol@cmu.edu
J. Schneider,
Robotics Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
schneide@cs.cmu.edu
A. Szalay,
Department of Physics & Astronomy, Johns Hopkins University
szalay@pha.jhu.edu
I. Szapudi,
CITA, Canada
szapudi@cita.utoronto.edu
L. Wasserman,
Department of Statistics, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA-15213
larry@stat.cmu.edu

Abstract

This paper is about new algorithms based on *k*d-trees decorated with cached sufficient statistics that permit relatively fast computation of 2-point, 3-point and 4-point statistics. These algorithms allow exact

[1]Authors' names are given in alphabetical order.

computations of the statistics at far greater speed than a naive non-tree-based implementation. We also describe how to achieve greater acceleration by permitting some approximations in the computation. These approximations have two distinguishing features in comparison with earlier approximate methods based on binning the data in grids. First, the approximation is *aware:* the algorithm can prove what its worst-case error is. Second, the method works at all scales: if the separations under consideration are very small, a grid-based method may need to search billions of bins (and trillions in more than two dimensions). These new algorithms give significant speedups at all magnitudes of separations, and in more than two dimensions, with no special case code required.

1 Introduction

The n-point function is widely used to characterize mass distribution in sky surveys.

The simplest case is the two point function. Consider the dataset in Figure 1. Imagine that each dot corresponds to the (ra,dec) coordinates of a galaxy. We may wish to ask the question: "How many pairs of galaxies lie within a separation of 1 parsec of each other?". The answer is depicted in Figure 2: XX. In general, we will usually be interested in many separations, and will ask to construct a histogram of the form in Figure 3, where the kth bar in the histogram denotes the number of pairs of galaxies that lie within separation lo_k and hi_k where lo_k and hi_k are the distances covered by the k'th bar.

Note that each pair of galaxies accounts of one count in one of the bars, and so the total sum of the heights of the bars is equal to

$$\binom{R}{2} \tag{1}$$

where we have R galaxies.

Why are these two point functions useful? The answer is that they provide useful summaries of mass distribution in a dataset, or the tendency of the points in a dataset to clump. Consider the two large datasets of Figures 4 and 5. They are visually very different distributions. Their two-point

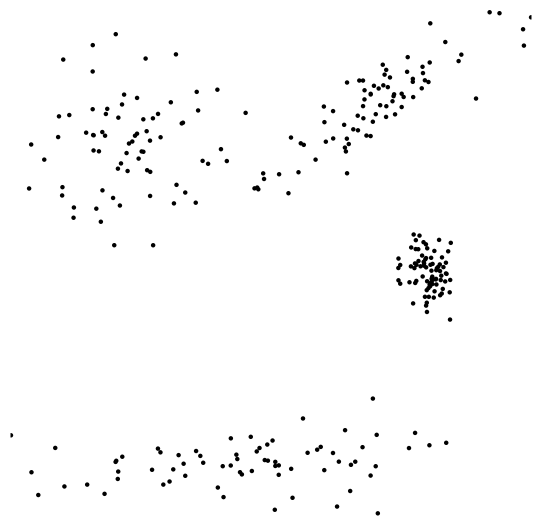

Figure 1: A set of 300 datapoints in 2-d space

functions—shown in Figures 6 and 7 are also very different. Suppose that the uniform data were from a simulation of the universe and the dots were from real data. A comparison of the two-point functions forms an objective way to dismiss the hypothesis that they come from the same underlying generative process.

Having explained the two-point function, we must now discuss some criticisms of it that will lead to a need for more sensitive characterizations in the form of 3-point and higher functions. There are two problems with two-point. First, it is possible to invent mass-distributions that visually have extremely different appearences but which have identical two-point functions. Figures **??** and **??** show two such distributions, and Figures **??** and **??** shows their two point functions. Second, we usually need some analysis of how significantly two 2-pt functions differ. To do that we need a 4-point function.

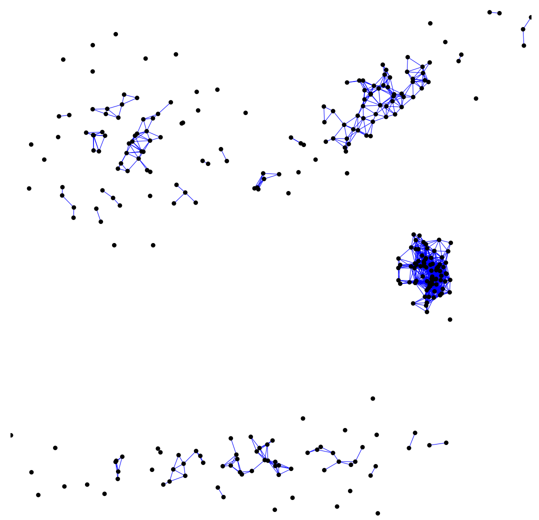

Figure 2: All pairs of points connected by a distance of less than one twentieth of the width of the figure are denoted by line segments.

So what is an n-point function? One "bar" in an n-point histogram concerns the question: "How many n-tuples of datapoints $(\mathbf{x}1, \mathbf{x}2, \ldots \mathbf{x}n)$ obey the following matching predicate?"

$$\begin{array}{rcl}
lo_{12} \leq & |\mathbf{x}_1 - \mathbf{x}_2| & < hi_{12} \\
lo_{13} \leq & |\mathbf{x}_1 - \mathbf{x}_3| & < hi_{13} \\
lo_{14} \leq & |\mathbf{x}_1 - \mathbf{x}_4| & < hi_{14} \\
 & \vdots & \\
lo_{1n} \leq & |\mathbf{x}_1 - \mathbf{x}_n| & < hi_{1n} \\
lo_{23} \leq & |\mathbf{x}_2 - \mathbf{x}_3| & < hi_{23}
\end{array}$$

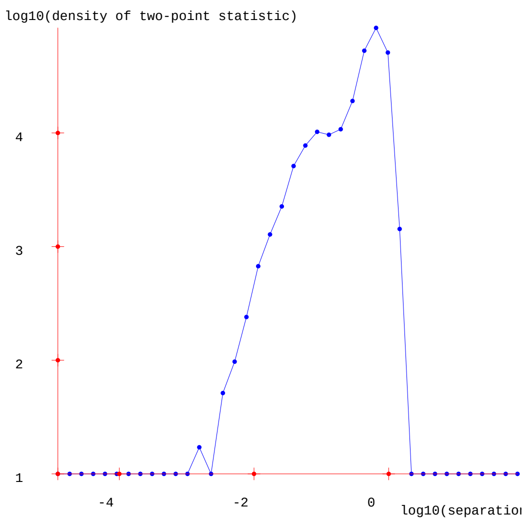

Figure 3: The 2-point function for the data in Figure 1. Notice that both axes use a logarithmic scale.

$$\begin{aligned} lo_{24} \leq \quad & |\mathbf{x}_2 - \mathbf{x}_4| & < hi_{24} \\ & \vdots & \\ lo_{2n} \leq \quad & |\mathbf{x}_2 - \mathbf{x}_n| & < hi_{2n} \\ lo_{n-2n-1} \leq \quad & |\mathbf{x}_{n-2} - \mathbf{x}_{n-1}| & < hi_{n-2n-1} \\ lo_{n-2n} \leq \quad & |\mathbf{x}_{n-2} - \mathbf{x}_n| & < hi_{n-2n} \\ lo_{n-1n} \leq \quad & |\mathbf{x}_{n-1} - \mathbf{x}_n| & < hi_{n-1n} \end{aligned}$$

For example, to find out how many nearly-equilateral triangles with side length of close to 2 units there are, we might run a three point query with the following predicate: How many triplets of points $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ have

$$1.9 \leq \quad |\mathbf{x} - \mathbf{y}| \quad < 2.1 \text{ and}$$

$$
\begin{aligned}
1.9 \leq \quad & |\mathbf{x} - \mathbf{z}| & < 2.1 \text{ and} \\
1.9 \leq \quad & |\mathbf{y} - \mathbf{z}| & < 2.1 \text{ ?}
\end{aligned}
$$

.

2 Computation of an n-point statistic

Suppose we wished to compute the number of pairs of records in a dataset of size R which have a separation between s_{lo} and s_{hi}. Here is the naive algorithm:

- C := 0
- for i = 1 to R do
  - for j = 1 to i do
    - if $s_{lo} \leq |\mathbf{x}_i - \mathbf{x}_j| < s_{hi}$ then
      - C := C + 1

This is an $O(R^2)$ algorithm, requiring $\binom{R}{2}$ of the "if ... then" tests.

Table 1 shows how the cost of this computation varies as a function of R, the number of records, and s_{hi} for the data of Figure 5.

The naive algorithm is hopelessly expensive for large R. There is one piece of good news. If we wish to build a histogram of 30 buckets, such as those of Figures 6 and 7, the cost need not be 30 times the cost of computing the count for one bucket. This is because the doubly nested for loop can simply decide, for each pair of points, which bucket should be incremented by that pair.

The bad news is that this acceleration is nevertheless pitiful in the face of quadratic cost growth. Much worse still, the cost for computing the n-point correlation naively for $n > 2$ is $O(R^n)$, so even for the mere 3-point function, unthinkable for only 100,000 datapoints.

3 Previous approaches to accelerating n-point computations

Which will be written by Istvan.

points	separation	secs
5000	0.01	3
10000	0.01	15
15000	0.01	41
20000	0.01	78
5000	0.1	4
10000	0.1	15
15000	0.1	40
20000	0.1	76
5000	1	3
10000	1	17
15000	1	42
20000	1	78

Table 1: The empirical cost of running the naive implementation of 2-pt for various separations and dataset sizes. Unsurprisingly, the separation has no effect and the number of points has a quadratic effect.

4 Accelerating n-point computations with kd-trees

4.1 Review of kd-trees

The kd-tree data structure was introduced by (Friedman, Bentley, & Finkel, 1977; Bentley, 1980). It is a way of organizing a set of datapoints in k-dimensional space in such a way that once built, whenever a query arrives requesting a list all points in a neighborhood, the query can be answered quickly without needing to scan every single point.

The root node of the kd-tree owns all the data points. Each non-leaf-node has two children, defined by a splitting dimension $n.\textsc{SplitDim}$ and a splitting value $n.\textsc{SplitValue}$. The two children divide their parent's data points between them, with the left child owning those data points that are strictly less than the splitting value in the splitting dimension, and the right child owning the remainder of the parent's data points:

$$\mathbf{x}_i \in n.\textsc{Left} \quad \Leftrightarrow \quad \mathbf{x}_i[n.\textsc{SplitDim}] < n.\textsc{SplitValue} \text{ and } \mathbf{x}_i \in n \tag{2}$$

$$\mathbf{x}_i \in n.\textsc{Right} \quad \Leftrightarrow \quad \mathbf{x}_i[n.\textsc{SplitDim}] \geq n.\textsc{SplitValue} \text{ and } \mathbf{x}_i \in n \tag{3}$$

As an example, some of the nodes of a *k*d-tree are depicted in Figures 8–11.

*k*d-trees are usually constructed top-down, beginning with the full set of points and then splitting in the center of the widest dimension. This produces two child nodes, each with a distinct set of points. This procedure is then repeated recursively on each of the two child nodes.

A node is declared to be a leaf, and is left unsplit, if the widest dimension of its bounding box is ≤ some threshold, MinBoxWidth. A node is also left unsplit if it denotes fewer than some threshold number of points, $r_{\min}$. A leaf node has no children, but instead contains a list of k-dimensional vectors: the actual datapoints contained in that leaf. The values MinBoxWidth $= 0$ and $r_{\min} = 1$ would cause the largest *k*d-tree structure because all leaf nodes would denote singleton or coincident points. In that case the tree has $O(R)$ nodes and so requires $O(kR)$ memory, and (with some care) the construction cost is $O(kR \log R)$. In practice, we set MinBoxWidth to 1% of the range of the data point components and $r_{\min}$ to around 10. The tree size and construction thus cost considerably less than these bounds because in dense regions, tiny leaf nodes are able to summarize dozens of data points. Note too that the cost of tree-building is amortized—the tree must be built once, yet can be re-used for many different analysis operations.

Since the introduction of *k*d-trees, many variations of them have been proposed and used with great success in areas such as databases, computational geometry (Preparata & Shamos, 1985), graphics and robotics (Latombe, 1991). R-trees (Guttman, 1984) are designed for disk resident datasets and efficient incremental addition of data. Metric trees (Uhlmann, 1991; Omohundro, 1991) place hyperspheres around tree nodes, instead of axis-aligned splitting planes. In all cases, the algorithms we discuss in this paper could be applied equally effectively with these other structures. For example, (Moore, 2000) shows the use of metric trees for accelerating several clustering and pairwise comparision algorithms.

4.2 Range Searching

Before proceeding to fast n-point calculations, we will begin with a very standard *k*d-tree search algorithm that could be used as a building block for fast 2-point computations.

For simplicity of exposition we will assume the every node of the kd-tree contains one extra piece of information: the bounding box of all the points it contains. Call this box $n.\textsc{BoundBox}$. The implication of this is that every node must contain two new k dimensional vectors to represent the lower and upper limits of each dimension of the bounding box. The range search operation takes two inputs. The first is a k-dimensional vector $\mathbf{q}$ called the *query point*. The second is a separation distance s_{hi}. The operation returns the complete set of points in the kd-tree that lie within distance s_{hi} of $\mathbf{q}$.

- **RangeSearch**$(n, \mathbf{q}, s_{hi})$
  Returns a set of points S such that

$$\mathbf{x} \in S \Leftrightarrow \mathbf{x} \in n \text{ and } |\mathbf{x} - \mathbf{q}| \leq s_{hi} \tag{4}$$

- Let $\textsc{MinDist}$:= the closest distance from $\mathbf{q}$ to $n.\textsc{BoundBox}$.

- If $\textsc{MinDist} \geq s_{hi}$ then it is impossible that any point in n can be within range of the query. So simply return the empty set of points without doing any further work.

- Else, if n is a leaf node, we must iterate through all the datapoints in its leaf list. For each point, find if it is within distance s_{hi} of $\mathbf{q}$. If so, add it to the set of results.

- Else, n is not a leaf node. Then:

  - Let $S_{\text{left}} :=$ **RangeSearch**$(n.\textsc{Left}, \mathbf{q}, s_{hi})$
  - Let $S_{\text{right}} :=$ **RangeSearch**$(n.\textsc{Right}, \mathbf{q}, s_{hi})$
  - Return $S_{\text{left}} \cup S_{\text{right}}$.

Figure 12 shows the result of running this algorithm in two dimensions. Many large nodes are pruned from the search. 117 distance calculations were needed for performing this range search, compared with 499 that would have been needed by a naive method.

Note that it is not essential that kd-tree nodes have bounding boxes explicitly stored. Instead a hyper-rectangle can be passed to each recursive call of the above function and dynamically updated as the tree is searched.

Range searching with a kd-tree can be much faster than without if the range is small, containing only a small fraction of the total number of datapoints. But what if the range is large? Figure 13 shows an example in which kd-trees provide little computational saving because almost all the points match the query and thus need to be visited. In general this problem is unavoidable. But in one special case it *can* be avoided—if we merely want to count the number of datapoints in a range instead of explicitly find them all.

4.3 Range Counting and Cached Sufficient Statistics

We will add the following field to a kd-treenode. Let n.NumPoints be the number of points contained in node n. This is the first and simplest of a set of kd-tree decorations we refer to as *Cached Sufficient statistics*. In general, we frequently stored the centroid of all points in a node and their covariance matrix.

Once we have n.NumPoints it is trivial to write an operation that counts the number of datapoints within some range without explicitly visiting them.

- **RangeCount**$(n, \mathbf{q}, s_{hi})$
  Returns an integer: the number of points that are both inside the n and also within distance s_{hi} of $\mathbf{q}$.
- Let MinDist := the closest distance from $\mathbf{q}$ to n.BoundBox.
- If MinDist $\geq s_{hi}$ then it is impossible that any point in n can be within range of the query. So simply return 0.
- Let MaxDist := the furthest distance from $\mathbf{q}$ to n.BoundBox.
- If MaxDist $\leq s_{hi}$ then every point in n must be within range of the query. So simply return n.NumPoints.
- Else, if n is a leaf node, we must iterate through all the datapoints in its leaf list. Start a counter at zero. For each point, find if it is within distance s_{hi} of $\mathbf{q}$. If so, increment the counter by one. Return the count once the full list has been scanned.
- Else, n is not a leaf node. Then:

– Let $C_{\text{left}} := \textbf{RangeCount}(n.\textsc{Left}, query, s_{hi})$

– Let $C_{\text{right}} := \textbf{RangeCount}(n.\textsc{Right}, query, s_{hi})$

– Return $C_{\text{left}} + C_{\text{right}}$.

The same query that gave the poor range search performance in Figure 13 gives good performance in Figure 14. The difference is that a second type of pruning of the search is possible: if the hyperrectangle surrounding the n is either entirely outside *or inside* the range then we prune.

4.4 The Single Tree approach to 2-point computation

We can use repeated range counting to quickly answer a two point query. Given a minimum and maximum separation s_{lo} and s_{hi} we run the following algorithm:

- $\textbf{SingleTree2Point}(\mathbf{X}, n, s_{lo}, s_{hi})$
  Input $\mathbf{X}$ is a dataset, represented as a matrix in which the kth row corresponds to the kth datapoint. $\mathbf{X}$ has R rows and k columns. Input n is the root of a kdtree built from the data in $\mathbf{X}$. Output integer: the number of pairs of points $(\mathbf{x}_i, \mathbf{x}_j)$ such that $s_{lo} \leq |\mathbf{x}_i - \mathbf{x}_j| < s_{hi}$.

- C := 0

- For i between 1 and R do:

  – $C := C + \textbf{RangeCount}(n, \mathbf{x}_i, s_{hi}) - \textbf{RangeCount}(n, \mathbf{x}_i, s_{lo})$

Note that in practice we do not use two range counts at each iteration, but one slightly more complex rangecount operation

$$\textbf{RangeCountBetweenSeparations}(n, \mathbf{q}, s_{lo}, s_{hi})$$

that directly counts the number of points whose distance from $\mathbf{q}$ is between s_{lo} and s_{hi}.

Table 2 shows how the cost of this computation varies as a function of R and s_{hi}.

points	separation	secs
5000	0.01	0
10000	0.01	0
15000	0.01	0
20000	0.01	0
5000	0.1	0
10000	0.1	0
15000	0.1	1
20000	0.1	1
5000	1	0
10000	1	0
15000	1	0
20000	1	1

Table 2: The empirical cost of running the naive implementation of 2-pt for various separations and dataset sizes.

References

Bentley, J. L. (1980). Multidimensional Divide and Conquer. *Communications of the ACM*, *23*(4), 214—229.

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, *3*(3), 209–226.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. Assn for Computing Machinery.

Latombe, J. (1991). *Robot Motion Planning*. Kluwer.

Moore, A. W. (2000). The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. In *Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press.

Omohundro, S. M. (1991). Bumptrees for Efficient Function, Constraint, and Classification Learning. In Lippmann, R. P., Moody, J. E., &

Touretzky, D. S. (Eds.), *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann.

Preparata, F. P., & Shamos, M. (1985). *Computational Geometry*. Springer-Verlag.

Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, *40*, 175–179.

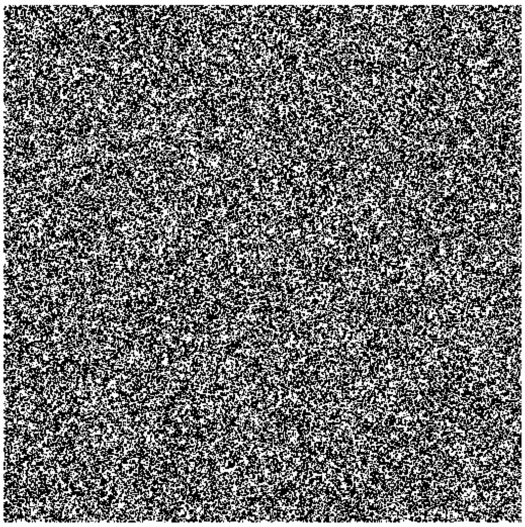

Figure 4: 20,000 uniformly distributed datapoints.

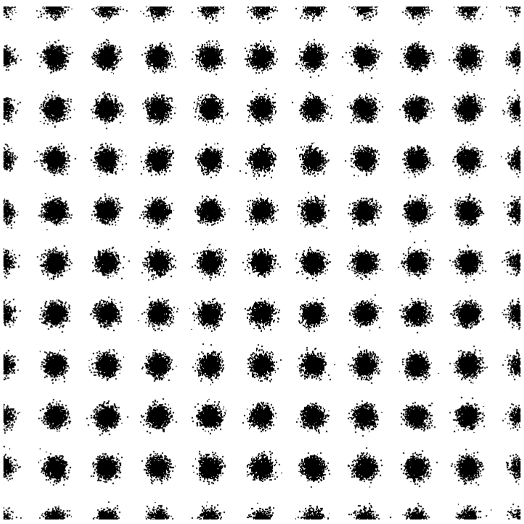

Figure 5: 20,000 datapoints in a non-uniform distribution.

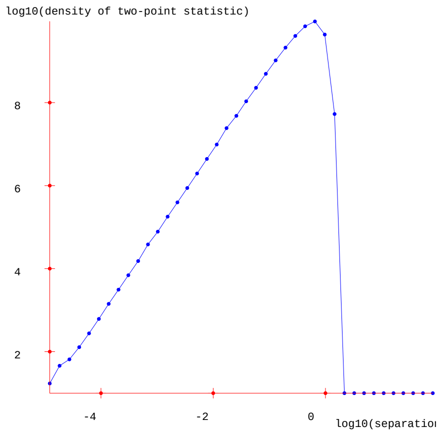

Figure 6: The 2-point function of the uniform data in Figure 4.

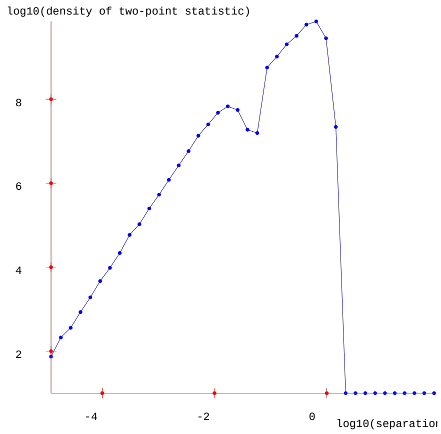

Figure 7: The 2-point function of the non-uniform data in Figure 5.

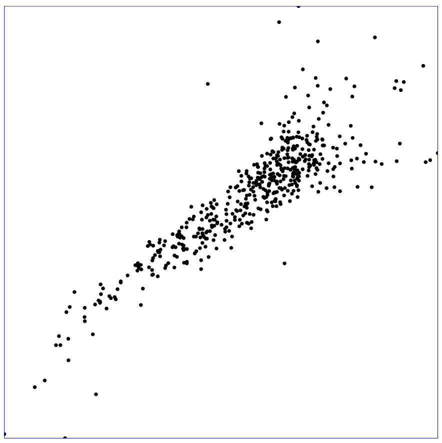

Figure 8: The top node of a kdtree is simply a hyper-rectangle surrounding the datapoints.

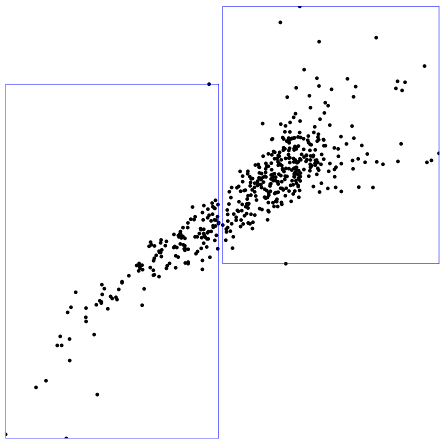

Figure 9: The second level contains two nodes.

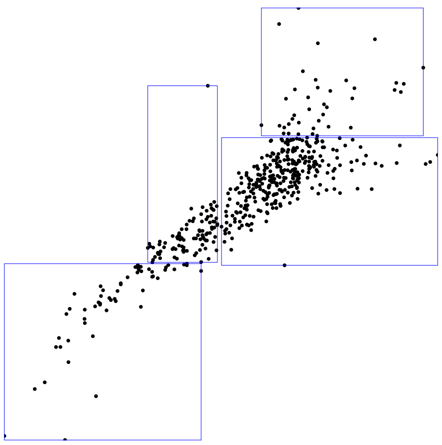

Figure 10: The third level contains four nodes. Note how a parent node creates its two children by splitting in the centers of its widest dimension

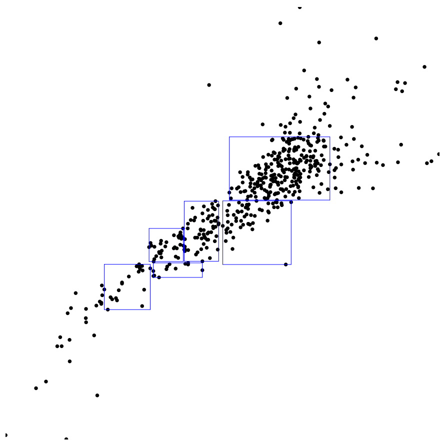

Figure 11: The set of nodes in the sixth level of the tree.

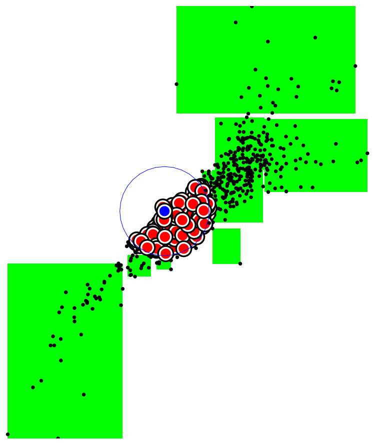

Figure 12: The shaded rectangles denote nodes that were pruned during a search for the set of points that lie inside the circle.

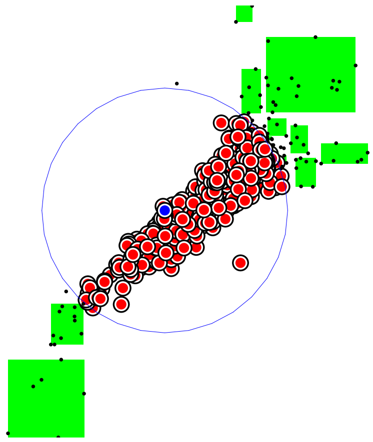

Figure 13: When the range is larger there are fewer opportunities for pruning.

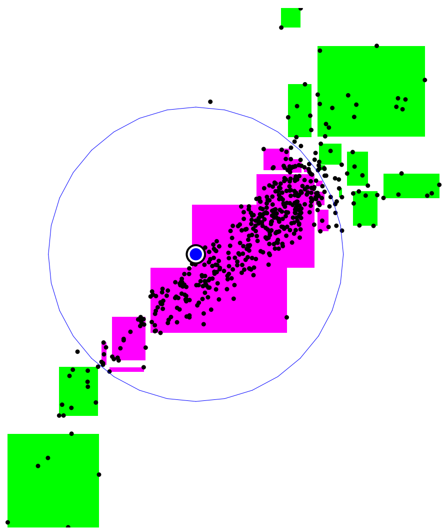

Figure 14: When doing a range count, nodes entirely within range can also be pruned.