Clustering in 2D - Insert algorithm

Jan Lukeš
xlukesj@fel.cvut.cz

Department of Computer Science and Engineering, Faculty of Electrical Engineering
Czech Technical University
Prague / Czech Republic

Abstract

The goal of this work is to improve speed of searching for intersection of a point, line segment, rectangle, etc. with a group of bounding-boxes in 2D. In case of successive passing of bounding-boxes one by one, the complexity is O(n). Our work present a way with complexity O(log(n)). One of many algorithms that we call "Insert algorithm" is described in details, the others are mentioned only marginally.

KEYWORDS: clustering, cluster analysis, computational geometry.

1 Introduction

For the given task, the following data are specified: a list of bounding-boxes of 2D geometrical objects and a tested object (point, line segment, rectangle, etc.) as an input. The output is a list of bounding boxes which have an intersection with the tested object. Searching within bounding-boxes takes O(n) complexity. Instead of it we will build a kind of hierarchy (tree). Then we will traverse this hierarchy. The basic idea is to take two bounding-boxes and group them together, it means to create one bigger bounding-box around them. This can be represented as one node of a tree. The whole thing is about what kind of a tree is the best and how to create it simply.

The paper is organized as follows: Chapter 2 contains presumptions, Chapter 3 describes our solution, Chapter 4 discuss complexity, Chapter 5 presents results of measurements and the last Chapter contains summary.

2 Presumptions

We presuppose two-dimensional Euclid's space and Cartesian rectangular system of co-ordinates. Than we have chosen the following constraints:

- bounding boxes are rectangles parallel to co-ordinate axes

- a hierarchy is represented by a general tree (not restricted to binary)
- bounding-boxes can overlap themselves
- a number of tested objects is much bigger than a number of bounding boxes
- the distribution of probability for location of tested object is constant in the whole space, it means that it has no relation to the distribution of bounding boxes

Time complexity should be better than O(n), it means better than processing bounding-boxes one by one.

There are several methods how can be this problem solved, we have chosen only one for this text, which is based on inserting bounding-boxes into the tree one by one.

It is necessary to point out, that building a tree takes a lots of time. For overall efficiency is advantageous to traverse the tree many times.

3 Solution

3.1 Minimization of number of comparisons

We want to build a hierarchy, which will be used for fast searching. The structure we have chosen is a tree (not limited to binary). The leaves represent the initial bounding-boxes, internal nodes represent bounding-boxes that include their successors.

A distribution of probability for location of tested object is the same in the whole plane. Figure 1 shows a typical case.

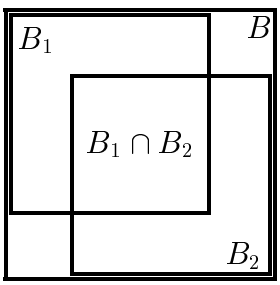

Figure 1: Clustering two overlapping bounding-boxes into a bigger one

Presume the situation in the picture and that we should make a test in the extent of bounding-box B. At the beginning let's take only a binary tree. Then we have:

$P_1 = \frac{S(B_1 - B_2)}{S(B)}$	the probability, that we have to test the interior of bounding-box B_1 which is outside B_2.
$P_2 = \frac{S(B_2 - B_1)}{S(B)}$	the analogical case for the bounding-box B_2
$P_3 = \frac{S(B_2 \cap B_1)}{S(B)}$	the probability, that we have to test interior of both bounding-boxes B_1 and B_2
$P_4 = \frac{S(B - B_1 - B_2)}{S(B)}$	the probability, that we needn't test anything else

where $S(B)$ is the area of the bounding-box B and total sum $P_1 + P_2 + P_3 + P_4 = 1$.

The number of comparisons performed in the scope of bounding-box B is

$$C(B) = 2 + P_1 \cdot C(B_1) + P_2 \cdot C(B_2) + P_3 \cdot (C(B_1) + C(B_2))$$

Number two in this equation means that we have to test both bounding-boxes B_1 and B_2 for intersection with the tested object. Then we may have to test the interior of one or both of them. For leaves of a tree (initial bounding-boxes) $C(B) = 0$.

When many overlaps between bounding-boxes occur, the component $P_3 \cdot (C(B_1) + C(B_2))$ has a high value and therefore we often have to pass both successors. This is worse than consecutive processing of bounding-boxes. We can solve this using a general tree, not only binary. Then we get a general expression:

$$C(B) = n + \sum_{i=1}^{n} P_i \cdot C(B_i), \quad P_i = \frac{S(B_i)}{S(B)}$$

where n is number of successors for a given node B. In case of repeated overlapping we use sequential passing instead of inefficient tree (more comparisons in the scope of one node).

3.2 Building a hierarchy

The question is how to find the structure of a tree in order that the number of comparisons will be minimal (i.e. how to minimize the function $C(B)$). There are two basic approaches:

1. *bottom-up method*, in each step we find two "closest" bounding-boxes, group them together and continue with new one instead of them. There are many variants of what we can choose as relation "closest" e.g.:
   - the shortest distance between given bounding-boxes
   - the smallest area of resultant bounding-box
   - the biggest ratio of the area of resultant bounding-box and the area of the union of given bounding-boxes
   - combination of previous
2. *insert algorithm* - consequent adding bounding-boxes one by one into the tree

Generally there is a disadvantage of both approaches: if there are many overlaps among initial bounding-boxes the tree structure will be not efficient.

We have chosen the second approach - the insert algorithm, because it solves at least partially the problem of many overlapping bounding-boxes. It allows to create trees with more than two successors in one node. This can't be done easily by the first approach, because the number of combination is too large.

The basic idea of the insert-algorithm is to place a given bounding-box locally into the tree but not to modify the tree structure in a global way. The complex modifications would be necessary when building an optimal tree structure. But for our data is suboptimum solution good enough. See section 5 for details.

3.3 Inserting

We take one of initial bounding-boxes as a base of the future tree, doesn't matter which one. Then the other bounding-boxes are put into the tree one by one (in no particular order).

(Note: In fact the difference between the tree that we will get and the ideal tree depends just on the order and the selection of the first one, but we didn't solve this problem.)

Let's have a tree of bounding boxes and one bounding-box (B) that we want to insert into the tree. (Starting with the root A). See the Fig. 2.

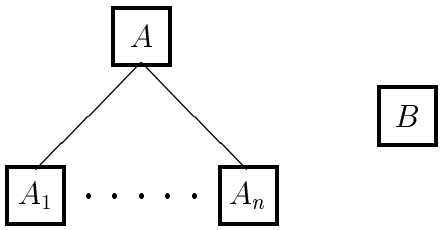

Figure 2: Inserting bounding-box in the tree

Than we have three possibilities how to do it:

1. Create new bounding box over A and B. See Fig. 3(a).
2. Put B as one of the successors of A. See Fig. 3(b).
3. Insert bounding-box B somewhere inside one of the successors A_i.

Now we will express the function C (which means number of comparisons) for each of the situations above.

1. Inserted bounding-box at the same level as original root. $C(B) = 0$ because B is a leaf.

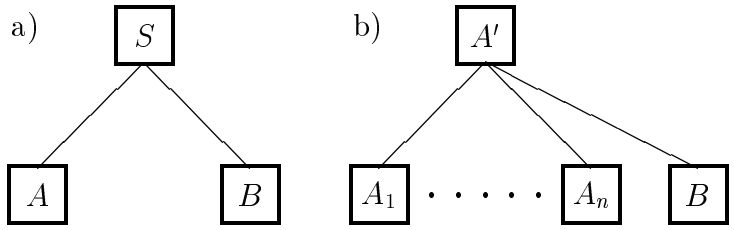

Figure 3: Inserting bounding-box at the same level as original root (a) and as one of successors (b)

$$
\begin{aligned}
C_1(S) &= 2 + \frac{S(A-B)}{S(S)} \cdot C(A) + \frac{S(B-A)}{S(S)} \cdot C(B) + \frac{S(A \cap B)}{S(S)} \cdot (C(A) + C(B)) \\
C_1(S) &= 2 + \frac{S(A-B)}{S(S)} \cdot C(A) + \frac{S(B-A)}{S(S)} \cdot 0 + \frac{S(A \cap B)}{S(S)} \cdot (C(A) + 0) \\
C_1(S) &= \underline{2 + \frac{S(A)}{S(S)} \cdot C(A)}
\end{aligned}
$$

2. Inserted bounding-box as one of successors.

$$
\begin{aligned}
C_2(A') &= (n+1) + \sum_{i=1}^{n} \frac{S(A_i)}{S(A')} \cdot C(A_i) + \frac{S(B)}{S(A')} \cdot C(B) \\
C_2(A') &= (n+1) + \sum_{i=1}^{n} \frac{S(A_i)}{S(A')} \cdot C(A_i)
\end{aligned}
$$

now substitute from relation for the original root:

$$
\begin{aligned}
C(A) &= n + \sum_{i=1}^{n} \frac{S(A_i)}{S(A)} \cdot C(A_i) \\
\sum_{i=1}^{n} \frac{S(A_i)}{S(A)} \cdot C(A_i) &= C(A) - n
\end{aligned}
$$

then we get:

$$
\begin{aligned}
C_2(A') &= (n+1) + \frac{1}{S(A')} \cdot (C(A) - n) \cdot S(A) \\
C_2(A') &= \underline{(n+1) + \frac{S(A)}{S(A')} \cdot (C(A) - n)}
\end{aligned}
$$

3. Insert bounding-box B somewhere inside one of the successors A_i.

   This means complete recursion. It seems like a big disadvantage that we have to parse the whole tree every time, but as the tree has complexity only $O(n)$, it is not so significant.

   The result from this situation is a value $C_3(A')$, which is the lowest from all possible values for placing B inside A_1 or A_2,...,A_n.

In the end we'll choose that variant, which has the lowest value of C, it means the minimum number of comparisons.

3.4 Implementation of insert algorithm

Representation: in each node we have the relevant bounding-box and the value of function C. We also remember the path to the place where the inserted bounding-box will be added.

Algorithm for building a tree

1. tree = one of the bounding-boxes (arbitrary)
2. successively for all other bounding-boxes
   (a) find a place in the tree where the bounding-box should be added
   (b) add the bounding-box there

ad 2.(a) - finding a place

1. Evaluate function C_1.
2. If the tree is only initial bounding-box (has no successors), then there are no other possibilities. The result is C_1
3. Evaluate function C_2.
4. Evaluate function C_3. Try to insert into each successor and then select the variant with the lowest C (and remember the path).
5. choose the best one from C_1, C_2, C_3 (the lowest value $\sim$ minimum number of comparisons)

ad 2.(b) - adding

According to selected variant and remembered path incorporate bounding-box into the tree. Then it is necessary to figure out new values of C on the path.

Algorithm for traversing a tree

1. At the beginning we have to test the bounding-box in the root of the tree. If this test fails, the tested object is outside all bounding-boxes.
2. The next steps depend on type of node:
   - for internal nodes
     Test the related bounding-box in the node. If it fails then stop – there is no intersection with input bounding-boxes. Otherwise call testing recursively for each of successors.
   - for leaves
     Test the bounding-box. If there is no intersection then stop. Otherwise add this bounding-box in the resultant list.

As a result we have a list of bounding-boxes intersecting or including the tested object.

4 Complexity

Let's expect specification:

- n ... number of initial bounding-boxes
- q ... number of tested objects ($q >> n$)
- complexity of test of intersection between tested object and bounding-box $= O(1)$

Now we can look at particular methods for solution:

successively

- time complexity: $O(q \cdot n \cdot 1) = \underline{O(q \cdot n)}$
- memory complexity: $\underline{O(1)}$

binary tree

- building a tree
  1. figure out values for all pairs of bounding-boxes $\Rightarrow O(n^2)$
  2. sort these values $\Rightarrow O(n^2 \cdot \log n)$
  3. in each step group two bounding-boxes together: $O(n)\times$
     (a) select maximum value - two selected bounding-boxes $\Rightarrow O(1)$
     (b) remove all combinations with two selected bounding-boxes $\Rightarrow O(n)$
     (c) inserting combinations with new bounding-box $\Rightarrow O(n \cdot \log n)$
  4. all together: $O(n^2 \cdot \log n)$
- time complexity: the best $\underline{O(q \cdot \log n)}$, the worse $\underline{O(q \cdot n)}$
  1. building a tree: $O(n^2 \cdot \log n)$
  2. searching: the best $O(q \cdot \log n)$, the worse $O(q \cdot n)$ (distribution of tree, overlapping bounding-boxes)
  3. be aware of condition: $O(q \cdot \log n) >> O(n^2 \cdot \log n)$, otherwise it is not efficient (better than successively)
- memory complexity: $\underline{O(n)}$ – tree
- disadvantage:
  1. If there are many overlapping bounding-boxes, we have to traverse larger parts of the tree.

insert algorithm

- building a tree: We incorporate n bounding-boxes one by one into the tree. The tree has $O(n)$ nodes, which we have to pass every time. Modification of the tree has complexity $O(\log n)$. All together it is $O(n \cdot (n + \log n)) = \underline{O(n^2)}$

- time complexity: the best $\underline{O(q \cdot \log n)}$, the worse $\underline{O(q \cdot n)}$
  1. building a tree: $O(n^2)$
  2. searching: the best $O(q \cdot \log n)$, the worse $O(q \cdot n)$
  3. the important condition: $O(q \cdot \log n) >> O(n^2)$
- memory complexity $\underline{O(n)}$ – tree

example of one another approach - only for comparison

Top-down method, dividing space by projection of bounding-boxes in axes.

- time complexity: $\underline{O(q \cdot \log n)}$ always
- memory complexity: $\underline{O(n^2)}$
- advantages: unambiguous in searching, faster, better for very overlapping bounding-boxes
- disadvantage: bigger consumption of memory

5 Testing and measurements

5.1 Implementation

Computer: IBM/Cyrix P166+, 32MB RAM

Operating system: Microsoft DOS

Programming language: C++

Compilers: Borland C++, DJGPP

5.2 Example

Conditions:

- random generated bounding boxes
- processing all pixels in raster as tested objects (640×480)

1. Difference between values of function C and successive passing

   See Table 1. and Figure 4.

   n ... number of initial bounding-boxes

   C_{seq} ... number of comparisons for successive passing ($= n$)

   C_{insert} ... value of function C (number of comparisons) for the tree which was created with insert algorithm

   $\frac{C_{insert}}{C_{seq}}$ ... characteristic ratio in %

n	C_{seq}	C_{insert}	$\frac{C_{insert}}{C_{seq}}$
2	2	1.05	52.5 %
5	5	1.71	34.2 %
10	10	3.96	39.6 %
20	20	6.71	33.5 %
50	50	9.55	19.1 %
100	100	15,22	15.2 %
200	200	20.43	10.2 %
500	500	33.43	6.9 %
1000	1000	51.15	5.1 %
2000	2000	78.10	3.9 %
5000	5000	149.02	3.0 %
10000	10000	254.64	2.5 %

Table 1: Difference between values of function C

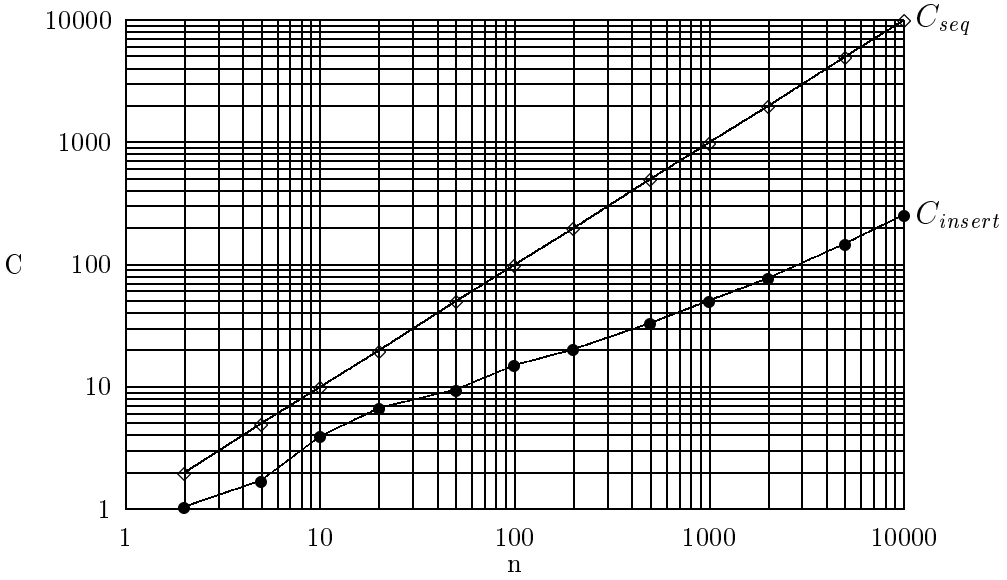

Figure 4: Difference between values of function C

2. Time complexity See Table 2. and Figure 5.

   n ... number of initial bounding-boxes

   t_{seq} ... time spended with successive passing

   $t_{building}$ ... time needed for building a tree with insert algorithm

   $t_{searching}$ ... time needed for searching within the created tree

   $t_{insert} = t_{building} + t_{searching}$ ... total sum for insert algorithm

   $\frac{t_{insert}}{t_{seq}}$ ... characteristic ratio in %

3. Memory complexity - exactly linear, it means $O(n)$

n	$t_{seq}[s]$	$t_{building}[s]$	$t_{searching}[s]$	$t_{insert}[s]$	$\frac{t_{insert}}{t_{seq}}$
2	16	0	0	0	—
5	17	0	1	1	5.8%
10	18	0	2	2	11.1%
20	22	0	2	2	9.1%
50	31	0	3	3	9.8%
100	47	0	4	4	8.5%
200	80	1	5	6	7.5%
500	183	2	9	11	6.0%
1000	342	7	15	22	6.4%
2000	1242	27	22	49	3.9%
5000	—	160	45	205	—
10000	—	636	466	1102	—

Table 2: Difference between time complexities

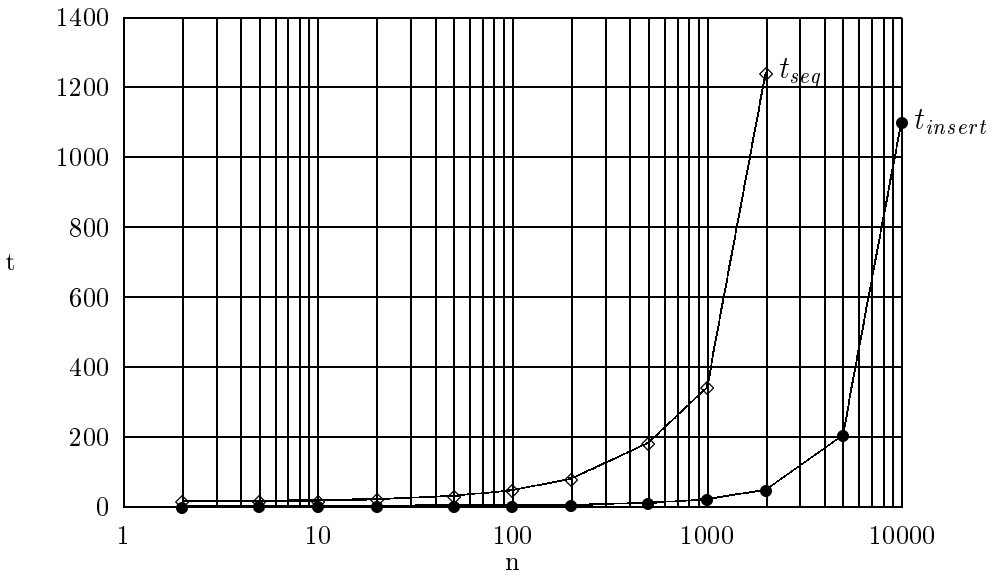

Figure 5: Difference between time complexities

6 Summary

The insert algorithm as described above is a suboptimum solution for the problem of searching for intersecting bounding-boxes within 2D space. Mostly it has much better time complexity than successive passing of bounding-boxes ($O(q \cdot \log n)$ vs. $O(q \cdot n)$). And it has only $O(n)$ memory consumption. It can be easily extended for 3D space if we take ratios of volumes instead of areas.

References

[1] J. Šarmanová A. Lukasová. *Metody shlukové analýzy.* SNTL, 1985.

[2] Multimedia Theory and Application, Bilateral student workshop CTU Prague-HTW Dresden, http://cs.felk.cvut.cz/~xobitko/d/xlukesj/. *Clustering in 2D - division approach*, December 1998.