Homework 1 – Bisector trees

Jan Kohlhase

jan.kohlhase@est.fib.upc.edu

April 2, 2020

1 Introduction

The *bisector tree* proposed in [1] is a data structure towards the *closest point problem*. The *closest point problem* calls for organizing a set S of points $\{x_1, x_2, ..., x_n\}$ in $\mathbb{R}^k$ and producing an algorithm for finding the nearest point $q \in S$ to a new point z, the *query point*. The figure 1 shows an example set in $\mathbb{R}^2$. The efficiency of a solution to this

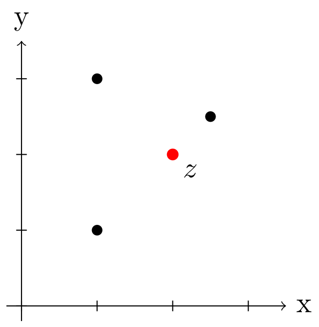


Figure 1: The *closest point problem*. Finding the nearest point to a query point z out of a set of points all in the same plane.

problem can be measured regarding the *preprocessing time* $P(n)$ (number of operations required to construct the data structure), the amount of *storage* $S(n)$ required by the data structure and the *search time* $Q(n)$ (the number of operations required to find the closest point). The most naive solution would be running through all points in S, calculating their distances to z and picking the nearest. $P(n)$, $S(n)$ and $Q(n)$ would in this case $\mathcal{O}(n)$.

The solution presented in [1] uses a tree as data structure and makes use of the triangle inequality for finding the nearest point. The triangle inequality states that for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side. If a, b and c are the length of the sides of a triangle, with no side being greater than c, then the triangle equality states that $c \leq a + b$.

2 Definitions

A *Bisector Tree* is defined as a binary search tree T in which each node N consits of either one point p_L or two points p_L, p_R in the plane. The left and right subtree is denoted as N_L and N_R respectively. The tree is constructed in a way such that if it is consisting of only one node N_L and N_R are empty. For a query point z let $\texttt{DL} = |z - p_L|$ and $\texttt{DR} = |z - p_R|$ denote the distances between z and the points stored in the node. Let $\texttt{LRADIUS} = max\{|w - p_L|, \forall w \in N_L\}$ and $\texttt{RRADIUS} = max\{|w - p_R|, \forall w \in N_R\}$ denote the radius of each subtree.

Let q' denote the current nearest point to z such that $\texttt{NDIST} = |z - q'|$ is defined as the current nearest distance. As long as $\texttt{DL (DR)} - \texttt{LRADIUS (RRADIUS)} < \texttt{NDIST}$ the corresponding subtree(s) need to be queried. Or in terms of the triangle equality: if $\texttt{DL (DR)} - \texttt{LRADIUS (RRADIUS)} \geq \texttt{NDIST}$), then no point in the corresponding subtree is closer to z than q'.

3 Algorithms

In this section the algorithms used to build and query the search tree will be described formally. The description is based on the original work but is not containing any pseudo-code. For a more detailled view on the algorithms and their pseudo-code the reader is referred to [1]. The source code is available for authenticated users at the FIBs internal GitLab[1].

3.1 Tree structure

The set of points S is inserted into a binary tree T. When the tree is full, each node consists of two points $p_L, p_R \in S$. Each node contains two pointers `LCHILD` and `RCHILD` pointing to the children and a pointer `ANCESTOR` pointing to the parent of the node. The nodes structure is visualized in figure 2. Furthermore a node also stores their

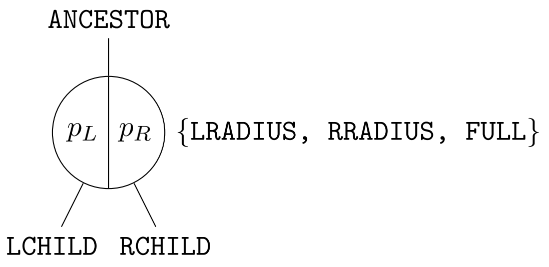


Figure 2: Node structure in a bisector tree. Each node stores up to two points, the range of its subtrees and contains pointers to its ancestor and descendants.

subtrees range `LRADIUS` and `RRADIUS` (as defined in section 2) and a boolean variable `FULL` indicating whether or not the node contains two points.

[1]`https://gitlab.fib.upc.edu/jan.kohlhase/data-structures`

3.2 Insertion of new points

Positioning a new point p into the tree works as follows: starting at the root node the algorithm first calculates the distance from p to p_L and p_R. Whether closer to the p_L or p_R the algorithm continues in direction of `LCHILD` or `RCHILD`. The algorithm stops when there is no node in the direction it is about to go (`LCHILD` or `RCHILD` is `NIL`) or it reached a node which is not containing two points so far (`FULL` is `false`). In the first case the algorithm creates a new node as the corresponding child and places p in p_L. In the second case p will be placed in p_R and `FULL` will be set to `true`. Both cases are shown in figure 3. While traversing down the tree, `LRADIUS` and `RRADIUS` in each node are permanently updated if necessary by terms of their definition (see section 2).

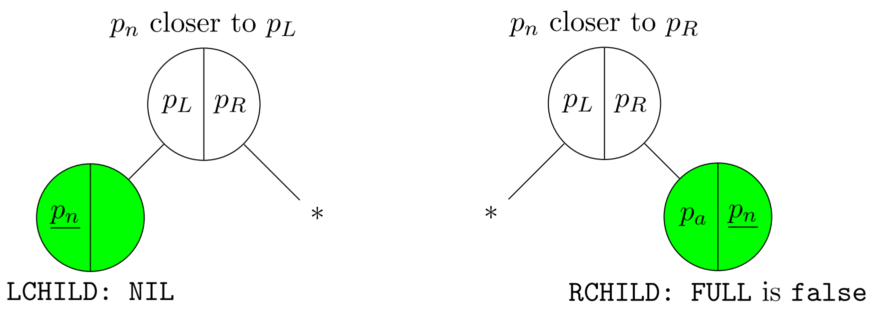


Figure 3: Insertion procedure of a new point p_n into a bisector tree. Traversion through the tree by terms of a distance function to find an "empty spot".

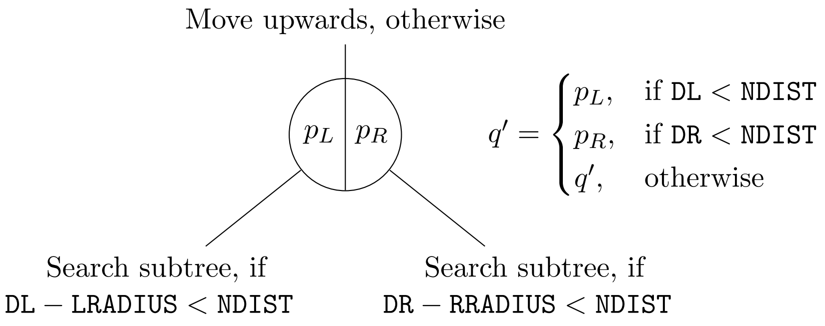
3.3 Querying the closest point

Given the query point z, the search algorithm finds the point q in the tree closest to z. While proceeding through the tree, at each node the algorithm needs to make a decision in which direction to move next: up (U), left (L) or right (R). This decision is based on the following conditions.

$\mathtt{DL} - \mathtt{LRADIUS} < \mathtt{NDIST}$	Yes	Yes	No	No
$\mathtt{DR} - \mathtt{RRADIUS} < \mathtt{NDIST}$	No	Yes	Yes	No
$\mathtt{DR} \leq \mathtt{DL}$	L	R*	R	U
$\mathtt{DR} > \mathtt{DL}$	L	L*	R	U

An asteriks indicates that both conditions hold. In this case the algorithm creates a flag in the node represented by a pointer `TEMPSTORAGE`. It points to a record containing the distances `DL`, `DR` and a boolean variable `WENTRIGHT` indicating the direction the algorithm took.

As while inserting a new point starting at the root node the search algorithm first calculates the distances `DL` and `DR` from z to p_L and p_R. The global variables representing the current closest point q' and its distance to z as `NDIST` are initialized regarding the closest point to z stored in the root node. For each node the algorithm follows the decision procedure described above. The algorithm is proceeding in the direction the decision is pointing to while updating `NDIST` and q' if necessary. The whole process is summarized in figure 4. While moving upwards, the algorithm checks whether the node


Move upwards, otherwise
p_L
p_R
Search subtree, if $\texttt{DL} - \texttt{LRADIUS} < \texttt{NDIST}$
Search subtree, if $\texttt{DR} - \texttt{RRADIUS} < \texttt{NDIST}$

$$q' = \begin{cases} p_L, & \text{if } \texttt{DL} < \texttt{NDIST} \\ p_R, & \text{if } \texttt{DR} < \texttt{NDIST} \\ q', & \text{otherwise} \end{cases}$$

Figure 4: Procedure for finding the closest point. Movement decision and update of the current closest point q'. The update of `NDIST` is not included in the figure but works equivalent to q'.

is flagged to be searched in both subtrees. If the node is flagged and the condition (L) or (R) for the direction not visited still holds, the algorithms moves downwards otherwise it moves upwards. In both cases the flag is disposed. The search algorithm stops when it reaches the root node and returns the closest point $q = q'$.

4 Experiments

In this work some experiments regarding the *preprocessing time* $P(n)$ and the *search time* $Q(n)$ were conducted. The experimental setup was the following: a bisector tree is constructed by inserting a set S of randomly drawn points from the interval $[0, 100] \times [0, 100]$. The construction were repeated for 10 different sets S and for different numbers of points $n \in \{256, 512, 1024, 2048, 4096, 8192, 16384\}$.

The insertion of points follows a binary decision process and therefore $P(n) = \mathcal{O}(nlogn)$. The *preprocessing time* P was computed as the number of nodes visited while constructing the tree. The following table shows the average, best and worst P for different values of n.

n	$nlogn$	Avg. P	Best P	Worst P
256	2048	1731	1665	1877
512	4608	4033	3893	4266
1024	10240	9239	9055	9662
2048	22528	20723	20333	21435
4096	49152	46127	44993	47469
8192	106496	102049	100448	104560
16384	229376	226778	221185	243277

The results support this statement as, except for the worst case for the value of $n = 16384$, P in all cases is limited by $nlogn$.

For each tree constructed from the different sets of points, several search queries were performed. Therefore a set of search points were created consisting of each value z in the interval $[-100, 200] \times [-100, 200]$. The *search time* Q was computed as the number of nodes visited while traversing the tree for the closest point for each z. The following table summarizes the results showing the average, best and worst case search time.

n	$logn$	Avg. Q	Best Q	Worst Q
256	8	31	10	80
512	9	38	12	90
1024	10	47	14	106
2048	11	54	12	140
4096	12	69	18	164
8192	13	86	18	222
16384	14	100	16	268

The results shows a, relative to the size, small number of nodes the search algorithm needs to visit. For example, when there are 4096 points in the tree, there will be at least 2048 nodes, and on the average the algortihms needs to visit less than two percent of them. Here it is to be said that the results obtained in the original work [1] were notably smaller, two times in the average case.

As in the given example, a bisector tree constructed from n points will at least consist of $\frac{n}{2}$ nodes. But when constructing it from randomly drawn points the number of nodes is with a certain probability higher. From the trees created in the above mentioned experiments, the average number of nodes, the average number of leaf nodes and the average tree height were extracted. The following table is summarizing this values.

n	$logn$	# Nodes	# Leafs	Height
256	8	145	35	10
512	9	293	75	12
1024	10	584	144	13
2048	11	1168	289	15
4096	12	2337	579	17
8192	13	4650	1108	19
16384	14	9234	2084	22

A balanced tree is defined as having a maximum height $h = logn$. Therefore the table shows that the trees constructed in the experiments are (on average) not balanced. Nevertheless the average height of trees is growing similar to $logn$.

5 Conclusion

As mentioned in the introduction the efficiency of a solution to the closest point problem can be measured regarding the *preprocessing time* $P(n)$, the amount of *storage* $S(n)$ and the *search time* $Q(n)$. The original work showed that $P(n) = \mathcal{O}(nlogn)$ and $S(n) = \mathcal{O}(n)$. The storage growing depending on the number of points is easy to see. The experimental results also support the logarithmic growth of $P(n)$ expected by a binary decision process.

In [1] the authors concluded from their experimental results that the search time $Q(n)$ behaves like $logn$. For smaller sets $|S| \leq 4096$ the results of the experiments conducted in this work show a similar behavior. The figure 5 shows the average search time (real computation time in (ms)) of the naive approach against the bisector tree for different values of n. As n is growing logarithmically the axes are logarithmic scaled. The naive

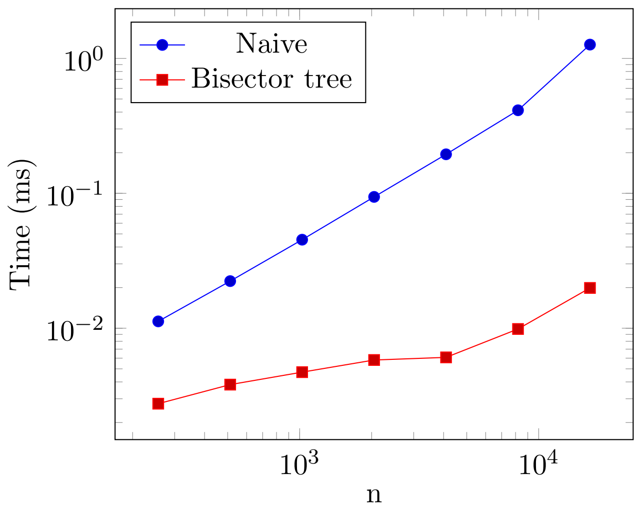


Figure 5: Query time plot. Naive search time versus bisector tree search time. This plot shows average time values in ms. The values are plotted in logarithmic scale.

approach, just running through all points in S finding the minimum distance, shows a linear growth. For sets with $|S| \leq 4096$ the bisector tree curve shows a logarithmic growth, but is notably increasing afterwards. This leads to the question if the authors assumptions regarding the search time growth hold.

As the experimental results for the search time are in general notably higher (see section 4) compared to the results reported in [1], some side effects originated by the implementation cannot be ruled out. Nevertheless the results encourage further work towards the analysis of the search time.

References

[1] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, (5):631–634, 1983.