3.1 EDGE k-d TREE

Several methods to store spatial information were devised and the major schemes are presented in Chapter 2. The prominent among them are the raster and vector representations, chain codes, pyramids and the numerous variants of the ubiquitous quadtree. In this Chapter, a new data structure is presented, namely the edge k-dimensional binary search tree or edge k-d tree, for short. This is a variant of the conventional k-d tree [Bentley 75] (cf. 2.1.0.2) and the edge quadtree [Ayala et al 85] (cf. 2.1.1.4) but it has the additional features of being balanced, translation and scaling invariant.

When the object is represented in the form of a tree, the nature of the tree is important because all further operations and information retrieval depend on it. In particular, the balanced nature and the height of the tree play a crucial part. The time to retrieve information from the tree or the traversal of the tree directly depend on these. The shorter the height, the less is the time it takes to access any node. Moreover, if the tree is balanced and information is stored systematically, the binary search, with a time complexity of O(log N), can be used for information retrieval. Among the data structures that are presented in Chapter 2, the k-d tree is balanced. But none of the important tree structures is balanced.

*The main results of this chapter were presented in a National Conference. See [Lavakusha et al 86]*
Not many representation schemes are invariant to translation and scaling. A slight change in the position or size of the object, may necessitate drastic reorganization of the representation. In many applications, a translation invariant data structure is necessary. The TID, MAT, polylines and chain-codes are invariant to translation and scaling. In fact, the normalized derivative of the chain-code (Sec. 2.1.1.2) is used as a shape descriptor [Pavlidis 78]. Though the quadtree has many advantages, it is not translation invariant. In Fig. 2.14, how the quadtree structure changes when the object is translated by one unit along the X-axis is shown. Li et al [82] have proposed a normal form of quadtree that is translation invariant. Chien et al [84] presented the normalized quadtree that is invariant to translation, scaling and rotation. But the space and time complexities are of $O(2^{21})$ and $O(1.2^{21})$ respectively for finding the minimal cost quadtree in terms of the number of nodes, by moving an image lying between squares $2^{1-1}$ and $2^1$ in a square of size $2^{1+1}$. Thus though there are several data structures for 2D region representation, no single scheme provides for the two desirable properties such as balanced tree structure with translation invariance. This motivated us to develop an alternative representation, the edge k-d tree, which is balanced and translation invariant.

3.2 CONSTRUCTION OF THE TREE

Now the underlying idea of constructing the edge k-d tree for any simple polygon P of N edges is described. This is a divide-and-conquer algorithm akin to the quicksort algorithm
[Hoare 61]. It is assumed here that the vertices of P are arranged in clockwise order. During the construction, the original list of vertices is recursively divided into smaller lists. We call the list that is to be divided as the object list. Initially the median of the X-coordinate of the vertices in the object list is computed. On this, the object list is partitioned into two sublists. For the first sublist, the X-coordinates of
the vertices is less than the median and for the other sublist, it is greater. Now each of the two sublists will be the object lists and the above process is repeated for each of the object lists, but this time the median is computed for the Y-coordinates. This recursive process of computing the median alternately on X and Y-coordinates and then partitioning the object list is repeated while the number of vertices in any object list is greater than three.

An X-cut is made, when the object list is partitioned into two sublists based on the median of the X-coordinates of the vertices. The vertex at which the cut is made is called object vertex. Similarly for Y-cut. In short, an X-cut is made on the
original object list. On the resulting two object lists, Y-cuts are made and on the resulting four object lists, X-cuts are made and so on while the cardinality of the object list is four or more.

But when the cardinality of the object list is three or less, the vertex that is farthest from the previous cut is chosen for the purpose of partitioning. Because of the cut being made at the farthest point, rather than at the median, the 'balanced' nature of the binary tree is lost. This may result in the tree having a height of \( \log N + 1 \), but still the height of the tree is \( O(\log N) \). The structure of a node in this binary tree is

\[
\begin{array}{cccccc}
\text{coord} & \text{lptr} & \text{rptr} & \text{fptr} & b & c
\end{array}
\]

The 'coord' field represents the X (Y) coordinate where the X-cut (Y-cut) is made. The next three fields are the pointers to the left and right children of the node and the father of the node. For the internal nodes, the fields b and c are blank. The coefficients a, b and c of the line equation \( ax + by + c = 0 \) are stored in the first, the fifth and the sixth fields respectively.

The relation between the list of vertices and the edge k-d tree is described below. The X-coordinate of the vertex where the X-cut is made in the original object list is stored in the 'coord' field of the root of the tree. This results in two object lists. The first sublist for which the X-coordinate is less than the X-coordinate of the object vertex, corresponds to the left branch of the tree and the other sublist corresponds to the right branch. In general, for any internal node, the left (right)
branch corresponds to the sublist created by that X-cut (Y-cut) for which the X (Y) coordinate of the object vertex is less (greater) than the X (Y) coordinate of the object vertex. The edge equations are stored in the leaf nodes, such that for points inside the polygon, the value obtained by substituting the points' coordinates in the edge equation is positive. The polygon is considered to be closed (in the sense of point set topology) i.e., the boundary of the polygon is part of the polygon.

A polygon and its edge k-d tree are shown in Fig. 3.1 and Fig. 3.2 respectively. Here there are nine vertices, given in a clockwise direction. When the array is sorted on X-coordinate, the median vertex is (36,25) and the left and right subarrays are \{ (16,36), (28,44), (28,8), (32,64) \} and \{ (48,4), (55,60), (64,28), (75,40) \} respectively. In the root of the tree, the X-coordinate of the median vertex 36 is stored. Each of the two subarrays is sorted on Y-coordinate. We follow the action on the left subarray. Sorted on Y-coordinate, the left subarray becomes \{ (28,8), (16,36), (28,44), (32,64) \}. The median vertex's Y-coordinate, i.e., 36 is stored in the left son of the root. Now, the left subarray is just one vertex (28,8) and the right subarray is \{ (28,44), (32,64) \}. The X-coordinate 28 is stored in the left son. And for the right branch, there are only two vertices. Here, the vertex farther from the previous cut is taken i.e., the vertex (32,64) is farther of the two vertices from the cut \( Y = 40 \). Hence an X-cut is made at (32,64) and 32 is stored in the right son. The Y-coordinate of the remaining vertex is stored in the left son.

For generating the line equations, the vertex is searched for
in the original array and its neighbors are located and the equation is computed using the formula

\[
\frac{x-x_1}{x_1-x_2} = \frac{y-y_1}{y_1-y_2}
\]

if the given vertices are \((x_1,y_1)\) and \((x_2,y_2)\). After the coefficients of the equation \(a,b,c\) are computed, they are multiplied by \(-1\), if necessary, so that for points inside the polygon \(ax + by + c > 0\).

3.3 The Algorithm

The input to the algorithm is the number of points \(N\), the coordinates of the vertices given in either clockwise or counterclockwise direction and a reference point \(r\). We need the point \(r\) to determine the signs of the coefficients in the edge equations. The algorithm generates the edge k-d tree. We use two arrays of size \(N \times 2\), namely points and dummy, to store the vertices. The sorting of vertices subsequently is done on the array dummy. Each node in the edge k-d tree has fields and a record structure treenode is used.

```c
struct treenode {
    int coord;
    pointer lptr, rptr, fptr;
    int b, c;
}
```

The algorithm in a C-like pseudo code is presented in Alg. 3.1 - 3.3.

```c
main()
begin
    read N and the vertices into the array points;
    copy points into dummy;
```
t = getnode();
dir = 0;
sort the array dummy on X-coordinate and let the median be i;
t.coord = dummy[i, 0];
t1 = getnode();
t2 = getnode();

construct(1, i-1, t1, 1);
construct(i+1, N, t2, 1);

write the tree in preorder in a file;
end

Algorithm 3.1

gtnode creates a record of the type treenode and returns
the address of the new record. The X and Y coordinate of the
vertex j, 1 ≤ j ≤ N, is stored in points[j, 0] and points[j, 1]
respectively. Same is the case with dummy.

construct (f, l, tnode, dir)
/* a recursive function to build the edge k-d tree. f and l are the first and
last indices in the subarray of the array dummy. dir determines whether the
next cut is to be an x-cut or a y-cut */
dif = l-f;
case (dif) of
   begin
      0 : /* there is only one point */
         t1 = getnode();
t2 = getnode();

         for the vertex (dummy[l,0], dummy[l,1]) find the
         previous and the next vertices from points.

         tnode.coord = dummy[l,dir];

         i = search ( dummy[l,0], dummy[l,1] );
         if ( i = N ) then pos = 0;
         else pos = i+1;

         if (points[pos,dir] > points[i,dir] )
            equation (i-1, i, tnode.l.ptr);
            equation (i, i+1, tnode.r.ptr);
         else
            equation (i, i+1, tnode.l.ptr);
            equation (i-1, i, tnode.r.ptr);
   endif;

Algorithm 3.1 (contd.)
/* there are two points. It takes the farthest point from the previous cut and introduces a cut there */

prev_cut = tnode.fptr.coord;
cdir = 1 - dir;

fdif = | prev_cut - dummy[f,dir]|;
ldif = | prev_cut - dummy[l,dir]|;

if ( ldif > fdif ) then pos = l
    else pos = f;

tnode.coord = dummy[pos,dir];
k = search(dummy[pos,0] dummy[pos,1]);

depending on whether the cut is made at f or l, construct is invoked for the left or right son and an equation is created at the other son;

/* there are three vertices. This module introduces a cut at the farthest vertex from the previous cut. construct is called once and the equation once. */

prev_cut = tnode.fptr.coord;

let pos be the vertex ( one of f or l) that is farthest from prev_cut along the direction dir;

k = search (dummy[pos,0] dummy[pos,1] );

depending where the cut is made, i.e., either at f or l, construct is called twice or a construct and an equation are called;

default: /* there are more than three points. This module sorts the vertices along dir, takes the median, make the cut there and invokes construct for the resulting two subarrays. */

sort (f, l ,dir);
dif = (f+l)/2;

pre

tnode.coord = dummy[dif,dir];

construct(f,dif-1,cdir,tnode.lptr);
construct(dif+1,l,cdir,tnode.rptr);

end;

end.

Algorithm 3.2

The function sort(f, l, dir), sorts the vertices in the
array dummy only between the indices f and l, with the primary key on dir. The function search(x,y) searches for the location of the vertex (x,y) in the array points sequentially.

```
equation(i, j, tnode)
begin
  x1 = points[i,0];
  x2 = points[i,1];
  y1 = points[j,0];
  y2 = points[j,1];
  coord = y1 - y2;
  b = x2 - x1;
  c = x1 * (-a) - y1 * b;
  if (r1 * a + r2 * b + c < 0) then sign = -1
  else sign = 1;
  tnode.coord = sign*coord;
  tnode.b = b*sign;
  tnode.c = c*sign;
end
```

Algorithm 3.3

3.4 ANALYSIS

The algorithm makes use of the functions main, construct, sort, equation and search. The time complexity of sort is \(O(n \log n)\) if there are \(n\) records, by Quicksort. The worst case complexity of search is \(O(N)\). The time complexity of equation is \(O(1)\).

The complexity of the function construct(l,f,tnode,dir) depends on the length of subarray of the vertices that it has to organize, i.e., on l-f. Two cases arise here, i.e., when \(l-f \leq 3\) and otherwise.

In the first case, when \(l-f \leq 3\), search is called once and the complexity of the search for a particular vertex in an array of size \(N\) is \(O(N)\).
In the other case, when \(1 - f > 3\), the complexity depends on the sort and the complexity of sort is of \(O(N \log N)\), when there are \(N\) vertices.

Hence, the complexity of construct is \(\max(O(N), O(N \log N))\), i.e., \(O(N \log N)\).

The initial array of size \(N\) is sorted and construct is invoked once for each of the two subarrays of size \(N/2\) and four times for subarrays of size \(N/4\) and so on. i.e., construct is invoked \(2^i\) times for subarrays of size \(N/2^i\). Hence the complexity of the function main and hence of the algorithm is

\[
(N \log N) + 2\left(\frac{N}{2} \log \frac{N}{2}\right) + 4\left(\frac{N}{4} \log \frac{N}{4}\right) + \ldots + 2 \log N
\]

\[
= \sum_{i=0}^{\log N} 2^i \left(\frac{N}{2^i} \log \frac{N}{2^i}\right)
\]

\[
= \sum_{i=0}^{\log N} N \left(\log N - \log 2^i\right)
\]

\[
= N \log^2 N - \sum_{i=0}^{\log N} N \cdot i
\]

\[
\approx N \log^2 N - \frac{(N \log^2 N)}{2}
\]

Hence the time complexity of constructing the edge k-d tree is \(O(N \log^2 N)\).

3.5 OPERATIONS ON EDGE K-D TREES

In this section, the operations of answering the point membership problem, the translation of the polygon by a vector \((t, s)\) and the scaling of the polygon by a constant \(s\) are
addressed using the edge k-d tree.

3.5.1 POINT MEMBERSHIP

The 'point-in-polygon' problem i.e., given a polygon P with N sides, and a point p, to determine whether p is inside P, can be answered in $O(\log N)$ time using the edge k-d tree. $O(\log N)$ is the best known bound for any simple polygon [Preparata 88]. The point-in-polygon problem is also referred to as the point membership problem. Given a point, $p = (p_1, p_2)$ and the edge k-d tree of the polygon P, if the point p is inside P is checked. The algorithm for answering this problem is a simple tree traversal method. We compare $p_1$, the X-coordinate of p with the 'coord' of the root and take the left branch if $p_1$ is less than the coord of the root. Else, the right branch. Next, we compare $p_2$, the Y-coordinate of p, with the coord of the node and depending on

---

**Fig. 3.3**: To determine the membership of the point $p=(24, 40)$, the nodes visited are shown. The edge equation is $8X - 12Y + 304 = 0$. 
whether $p_2$ is less or greater than the coord field, we take the left or the right branch. Thus, alternating between $p_1$ and $p_2$, we keep comparing with the coord field and take the appropriate branch for the next comparison. On coming to the leaf, we have an edge equation. By substituting $(p_1, p_2)$ in the edge equation, if the result is greater than 0, then $p$ is inside $P$, else outside. The algorithm is given in Alg. 3.4.

```c
function member(ptr, p) /* ptr is pointer to the root of the edge k-d tree and p=(p_1,p_2) */
begin
  i = 1;
  while (ptr is not leaf) do
    begin
      if $p_i < ptr$.coord
      then ptr = ptr.lptr
      else ptr = ptr.rptr;
      i = 3 - i;
    end;
    /* this while loop determines the block containing the point p. From this leaf we can access the corresponding (a,b,c) of the edge, this leaf represents */
    k = a * p_1 + b * p_2 + c;
    if $k > 0$ then print "Inside"
    else print "Outside";
  end.
```

Algorithm 3.4

From the algorithm, it is easily seen that, by traversing the binary tree of height $O(\log N)$, the point membership problem is answered. Hence, the complexity (best, worst and average) is of $O(\log N)$ and this is the best known bound for any general simple polygon [Preparata 88].
3.5.2 SCALING

If the polygon $P$ is scaled by a factor $s$, i.e., a vertex $v = (v_1, v_2)$ becomes $(sv_1, sv_2)$, then the edge $k$-d tree for the scaled polygon can be directly obtained from $P$. For this, in the internal nodes of the edge $k$-d tree, the coord field is multiplied by $s$ and in the leaf nodes, the field ‘$c$’ in the edge equation $aX + bY + c = 0$ is multiplied by $s$ i.e., $c$ is replaced by $sc$. The formal algorithm is presented in Alg. 3.5.
function scale (ptr,s) 
/ * ptr is the pointer to the root of the edge k-d tree and s is the scaling factor */
begin
  for (each internal node) do
    coord = coord * s
  for (each leaf node) do
    c = c * s;
end

Algorithm 3.5
The algorithm is a binary tree traversal and hence can be performed in O(m) time, where m is number of the nodes.

3.5.2 TRANSLATION
The algorithm for the translation of the polygon P represented by its edge k-d tree is also direct. When it is required to translate the polygon by a vector (t,s), we traverse the binary tree and make the necessary changes. For the internal nodes, the coord field is updated by adding t or s depending on whether it represents an X-cut or a Y-cut. For the leaf nodes, the field c is replaced by ‘-a * s -b * t -c’. We formally present the algorithm in Alg. 3.6.

function translate (ptr,t,s) begin
  for (each internal node) do
    if (level of node is even)
      then coord = coord + t
     else coord = coord + s;
  for (each leaf node) do
    c = m -a * s -b * t -c;
end.

Algorithm 3.6
The level of the root is 0 and the level of a node in the tree is 1 more than the level of its father. Since this algorithm is a tree traversal, the time complexity is of $O(m)$, where $m$ is the number of nodes.

3.6 CONCLUSIONS

It is observed in the previous sections that the edge k-d tree representation facilitates efficient method for solving the point membership problem, translation and scaling operations. In the conventional vector representation, the operations of translation and scaling become trivial, but the point membership problem can not be answered easily. Similarly, though the construction of a region quadtree is relatively easy, this structure is sensitive to scaling and translation. Moreover, the region quadtree is not balanced and hence its size is not a function of the number of vertices of the polygon.

Thus the edge k-d tree can be the most suitable data structure for region representation in situations like automated cartography, GIS and robotics where the point membership, scaling and translation are required frequently. In the polygon partitioned by the X- and Y- cuts in Fig. 3.5, the X-cut at the vertex is singular. Should the edge k-d tree be constructed for the polygon, how should the edges $e_1$ and $e_2$ be stored? In an ideal situation, the edges $e_1$ and $e_2$, were to lie on either side of the cut for answering point membership query. But here, since $e_1$ and $e_2$ lie on the same side, the present algorithm does not work. Thus, the proposed method of construction of the edge k-d tree works successfully for class of polygons which includes the
**Fig. 3.5:** For this polygon, point membership cannot be answered through an edge k-d tree.

convex class and the monotone class.

The representation schemes that are used to store 2D object information, in some cases, can be extended to store 3D object information. So it is natural to ask whether the edge k-d tree concept can be extended to represent 3D polygons. The answer is in the negative. This is because, in the edge k-d trees, cuts parallel to one of the axes is made at vertices and the polygon is broken up into smaller and smaller polylines. But in the case of a 3D polygon, where should a cut be made? And, further, should the cut be made by a line or plane?

If we make the cut at a vertex, it may partition the faces in the middle and if the cut is made at an edge, the plane may not be parallel to any of the coordinate planes.