Five Balltree Construction Algorithms

STEPHEN M. OMOHUNDRO

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704
Phone: 415-643-9153
Internet: om@icsi.berkeley.edu

November 20, 1989

Abstract. Balltrees are simple geometric data structures with a wide range of practical applications to geometric learning tasks. In this report we compare 5 different algorithms for constructing balltrees from data. We study the trade-off between construction time and the quality of the constructed tree. Two of the algorithms are on-line, two construct the structures from the data set in a top down fashion, and one uses a bottom up approach. We empirically study the algorithms on random data drawn from eight different probability distributions representing smooth, clustered, and curve distributed data in different ambient space dimensions. We find that the bottom up approach usually produces the best trees but has the longest construction time. The other approaches have uses in specific circumstances.

Introduction

Many tasks in robotics, vision, speech, and graphics require the construction and manipulation of geometric representations. Systems which build these representations by learning from examples can be both flexible and robust. We have developed a data structure which we call the *balltree* which is well suited to geometric learning tasks. Balltrees tune themselves to the structure of the represented data, support dynamic insertions and deletions, have good average-case efficiency, deal well with high-dimensional entities, and are easy to implement. In this report we compare five different algorithms for building these structures from data. We discuss the trade-off between the efficiency of the construction algorithm and the efficiency of the resulting structure. Two of the algorithms are on-line, two analyze the data in a top-down fashion, and one analyzes it in a bottom up manner.

The balltree structure is related to other hierarchical representations such as *k-d* trees [Friedman, *et. al.*, 1977] and oct-trees [Samet, 1984], but has specific advantages in the domains of interest. We are applying these structures to representing, learning, and manipulating point sets, smooth submanifolds, nonlinear mappings, and probability distributions. Some of these applications are described in [Omohundro, 1987, 1988, 1989] in the context of *k-d* trees. The operations that are efficiently supported include nearest neighbor retrieval, intersection and constraint queries, and probability maximization. The basic construction techniques described here should be applicable to a wide variety of other hierarchical geometric data structures in which balls are replaced by boxes, cubes, ellipsoids, or simplices.

Balltrees

We refer to the region bounded by a hyper-sphere in the n-dimensional Euclidean space $\mathfrak{R}^n$ as a *ball*. We represent balls by the $n+1$ floating point values which specify the coordinates of its center and the length of its radius. A *balltree* is a complete binary tree in which a ball is associated with each node in such a way that an interior node's ball is the smallest which contains the balls of its children. The leaves of the tree hold the information relevant to the application; the interior nodes are used only to guide efficient search through the leaf structures. Unlike the node regions in k-d trees or oct-trees, sibling regions in balltrees are allowed to intersect and need not partition the entire space. These two features are critical for the applications and give balltrees their representational power. Figure 1 shows an example of a two-dimensional balltree.

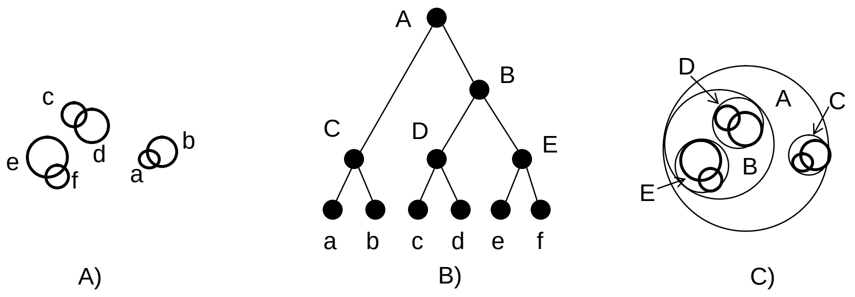


Figure 1. A) A set of balls in the plane. B) A binary tree over these balls. C) The balls in the resulting balltree.

In this report we study the problem of building a balltree from given leaf balls. Once the tree structure is specified, the internal balls are determined bottom up from the leaf balls. We would like to choose the tree structure to most efficiently support the queries needed in practical usage. Efficiency will depend both on the distribution of samples and queries and on the type of retrieval required. We first discuss several important queries and then describe a cost function which appears to adequately approximate the costs for practical applications. We then compare five different construction algorithms in terms of both the cost of the resulting tree and the construction time. The algorithms are compared on samples drawn from several probability distributions which are meant to be representative of those that arise in practice.

Implementation

Our implementation is in the object oriented language Eiffel [Meyer, 1989]. There are classes for balls, balltrees, and balltree nodes. “BALL” objects consist of a vector “ctr” which holds the center of the ball and a real value “r” containing the radius. “BLT_ND” objects consist of a BALL “bl”, and pointers “par, lt,rt” to the node’s parents and children. “BALLTREE” objects have a pointer “tree” to the underlying tree and a variety other slots to enhance retrievals (such as local priority queues and caches). All reported times are in seconds on a Sun Sparcstation 1 with 16 Megabytes of memory, running Eiffel Version 2.2 from Interactive Software Engineering. Compilation was done with assertion checking off and garbage collection, global class optimization, and C optimization on. As with any object oriented language, there is some extra overhead associated with dynamic dispatching, but this overhead should affect the different algorithms similarly.

Queries which use Simple Pruning

There are two broad classes of query which are efficiently supported by the balltree structure. The first class employs a search with simple pruning of irrelevant branches. The second class requires the use of branch and bound during search. This section presents some simple pruning queries and the next gives examples of the more complex variety.

Given a query ball, we might require a list of all leaf balls which contain the query ball. We implement this as a recursive search down the balltree in which we prune away recursive calls at internal nodes whose ball does not contain the query ball. If an internal node's ball doesn't contain the query ball, it is not possible for any of the leaf balls beneath it to contain it either. In the balltree node class "BLT_ND" we may define:

```
push_leaves_containing_ball(b:BALL,l:LLIST[BLT_ND]) is
      -- Add the leaves under Current which contain b to l.
   do
      if leaf then
         if bl.contains_ball(b) then l.push(Current) end
      else
         if bl.contains_ball(b) then
            lt.push_leaves_containing_ball(b,l);
            rt.push_leaves_containing_ball(b,l);
         end; -- if
      end; -- if
   end;
```

Similarly, we might ask for all leaf balls which intersect a query ball. We prune away recursive calls from internal nodes whose ball doesn't intersect the query ball.

```
push_leaves_intersecting_ball(b:BALL,l:LLIST[BLT_ND]) is
      -- Add the leaves under Current which intersect b to l.
   do
      if leaf then
         if bl.intersects_ball(b) then l.push(Current) end
      else
         if bl.intersects_ball(b) then
            lt.push_leaves_intersecting_ball(b,l);
            rt.push_leaves_intersecting_ball(b,l);
         end; -- if
      end; -- if
   end;
```

Finally, we might ask for all leaf balls which are contained in the query ball. Here we must continue to search beneath any internal node whose ball intersects the query ball because some descendant ball might be contained in it.

```
push_leaves_contained_in_ball(b:BALL,l:LLIST[BLT_ND]) is
      -- Add the leaves under Current which are contained in b to l.
   do
      if leaf then
         if b.contains_ball(bl) then l.push(Current) end
      else
         if bl.intersects_ball(b) then
            lt.push_leaves_contained_in_ball(b,l);
            rt.push_leaves_contained_in_ball(b,l);
         end; -- if
      end; -- if
   end;
```

A point is just a degenerate ball. Two important special cases of these queries in which some of the balls are points are the tasks of returning all leaf balls which contain a given point and returning all point leaves which are contained in a given ball.

Queries which use Branch and Bound

More complex queries require a branch and bound search. An important example for balltrees whose leaves hold points is to retrieve the *m* nearest neighbor points of a query point. Using balltrees we may use a similar approach to that discussed in [Friedman, *et. al.*, 1977] for *k-d* trees. Again we recursively search the tree. Throughout the search we maintain the smallest ball "bball" centered at the query point which contains the *m* nearest leaf points seen in the search so far. In this algorithm we prune away recursive searches which start at internal nodes whose ball doesn't intersect the bball. This pruning is likely to happen most effectively if at each internal node we first search the child which is nearer the query point and then the other child. Because balltree nodes can intersect we cannot stop the search when the bball lies inside node ball as is possible in k-d trees. Because node regions are tighter around the sample points, however, balltrees may be able to prune nodes in situations where a k-d tree could not.

To retrieve the *m* nearest neighbors we maintain a priority queue of the best leaves seen ordered by distance from the query point. We will only show the function for finding the nearest neighbor, but the extension to the *m* nearest neighbors should be clear.

In this case we assume that nn_search is defined in a class which has "bball" as an attribute and that its center "bball.ctr" has been set to the query point and its radius "bball.r" to a large enough value that it contains the root ball. "nn" is another attribute which will hold the result when the routine returns. "near_dist_to_vector" is a routine in the BALL class which returns the distance from the closest point in the ball to a given vector.

```
nn_search(n:T) is
      -- Replace nn by closest leaf under n to bball.ctr, if closer than bball.r
   local d,ld,rd:REAL;
   do
      if n.leaf then
         d:=bball.ctr.dist_to_vector(n.bl.ctr);
         if d < bball.r then bball.set_r(d); nn:=n end; -- reset best
      else -- at internal node
         ld:=n.lt.bl.near_dist_to_vector(bball.ctr);
         rd:=n.rt.bl.near_dist_to_vector(bball.ctr);
         if ld > bball.r and rd > bball.r then -- no sense looking here
         else
```

```
            if ld<=rd then -- search nearer node first
                nn_search(n.lt);
                if rd < bball.r then nn_search(n.rt) end; -- check if still worth searching
            else
                nn_search(n.rt);
                if ld < bball.r then nn_search(n.lt) end;  -- check if still worth searching
            end; -- if
        end; -- if
    end; -- if
end;
```

There are several natural generalizations of this query to ones involving balls. Distance between points may be replaced by distance between ball centers, minimum distance between balls, or maximum distance between balls. In each case the minimum distance to an ancestor node is a lower bound on the distance to a leaf and so may be exactly as above to prune the search.

A query which arises in one of the construction algorithms we will describe below must return the leaf ball which minimizes the volume of the smallest ball containing it and a query ball. The search proceeds as above, but with the pruning value equal to the volume of the smallest ball which contains the query ball and a single point of the interior node ball.

Statistical Nature of the Data

The criterion for a good balltree structure depends on both the type of query it must support and on the nature of the data it must store and access. For most of the applications we have in mind, it is appropriate to take a statistical view of the stored data and queries. We assume that the leaf balls are drawn randomly from an underlying probability distribution and that the queries are drawn from the same distribution. We would like systems to perform well on average with respect to this underlying distribution. Unfortunately, we expect distributions of very different types in different situations. A very powerful *nonparametric* approach to performance analysis has begun to appear (e.g. [Friedman, *et. al.* 1977] and [Noga and Allison, 1985]) which gives provably good results for a wide variety of distributions in the asymptotic limit of large sample size. In both of these references the underlying distribution is required to be non-singular and in the large sample limit each small region becomes densely sampled and locally looks like a uniform distribution. If an algorithm behaves well on the uniform distribution and adjusts itself to the local sample density, then it will have good asymptotic performance on non-singular distributions. Unfortunately, some of the most important applications have data of a very different character. Instead of being smooth, the data itself may be hierarchically clustered or has its

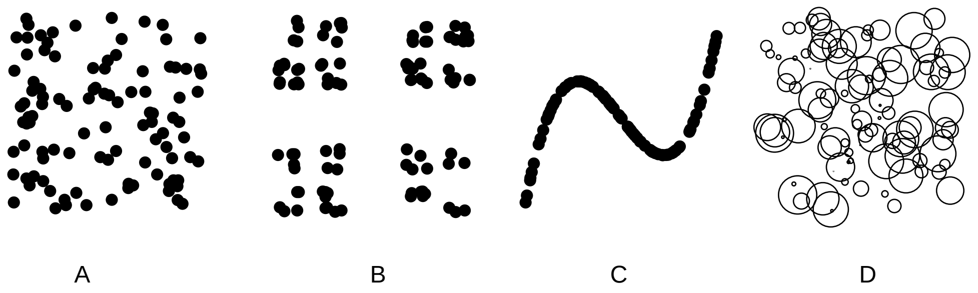


Figure 2. 101 leaves from the four distribution types. A) Uniform B) Cantor C) Curve D) Uniform balls.

support on lower dimensional surfaces. Part of the motivation for the development of the balltree structure was that it should deal well with these cases.

For any particular statistical model, possibly singular, one may hope to perform an asymptotic analysis similar to that in [Friedman, *et. al.,* 1977]. Because we are interested in the performance on small data sets for a variety of distributions, we have taken an empirical approach to comparing the different construction algorithms. We studied each algorithm in 8 situations corresponding to 4 different probability distributions in 2-dimensions and similar distributions in 5-dimensions. Samples from the 2-dimensional versions of these distributions are shown in figure 2. Because the case in which the leaves are actually points is very important, we have emphasized it in the tests.

The first distribution is the uniform distribution in the unit cube. For the reasons discussed above, behavior on this should be characteristic of general smooth distributions. For the second distribution we wanted to study a case with intrinsic hierarchical clustering. We chose a distribution which is uniform on a fractal, the Cantor set. The one-dimensional Cantor set may be formed by starting with the unit interval, removing the middle third, and then recursively repeating the construction on the two portions of the interval which are left. If points on the interval are expressed in ternary notation, then the Cantor set points are those that have no "1's" in their representation. In higher dimensions we just take the product of Cantor sets along each axis. The third distribution is meant to study the situation in which sample points are restricted to a lower dimensional nonlinear submanifold. We draw points from a polynomial curve which is embedded in such a way that it doesn't lie in any affine subspaces. In the final distribution, the leaves are balls instead of just points. The centers are drawn uniformly from the unit cube and the radii uniformly from the interval [0,.1].

The Volume Criterion

It might appear that these different distributions would require the use of different balltree construction criteria in order to lead to good performance. We have found, however, that a simple efficiency measure is sufficient for most applications. The most basic query is to return all leaf regions which contain a given point. A natural quantity to consider for this query is the average number of nodes that must be examined during the processing of a uniformly distributed query point. As described above, to process a point, we descend the tree, pruning away subtrees when their root region does not contain the query point. This process, therefore, examines the internal nodes whose regions contain the point and their children. We may minimize the number of nodes examined by minimizing the number of internal balls which contain the point. Under the uniform distribution, the probability that a ball will contain a point is proportional to its volume. We minimize the average query time by minimizing the total volume of the regions associated with the internal nodes. While this argument does not directly apply to the other distributions, small volume trees generally adapt themselves to any hierarchical structure in the leaf balls and so maximize the amount of pruning that may occur during any of the searches. We therefore use the total volume of all balls in a balltree as a measure of its quality. All reported ball volumes were actually computed by taking the ball radius to a power equal to the dimension of the space and so are only proportional to the true volume.

Unfortunately, it appears to be a difficult optimization problem to find the binary tree over a given set of leaf regions which minimizes the total tree node volume. Instead, we will compare several heuristic approaches to the construction. The next few sections describe the five studied construction algorithms. In each case we give the idea of the algorithm and a key Eiffel function for its implementation. So as not to obscure the structure, we have eliminated all housekeeping and bulletproofing code used in the actual implementation. Operations whose implementation is straightforward are not defined, the meaning should be clear from context.

K-d Construction Algorithm

We will call the simplest algorithm the *k-d construction algorithm* because it is similar to the method described in [Friedman, *et. al.*, 1977] for the construction of *k-d* trees. It is an off-line top down algorithm. By this we mean that all of the leaf balls must be available before construction and that the tree is constructed from the root down to the leaves. At each stage the algorithm splits the leaf balls into 2 sets from which balltrees are recursively built. These trees become the left and right children of the root node. The balls are split by choosing a dimension and a value and splitting the balls into those whose center has a coordinate in the given dimension which is less than the given value and those in which it is greater.

The dimension to split on is chosen to be the one in which the balls are most extended and the splitting value is chosen to be the median. Because median finding is linear in the number of samples, and there are $\log N$ stages, the whole construction is $O(N \log N)$.

Introducing a class whose instances are arrays of balls yields a clean implementation. The key function is "select_on_coord" which moves the balls around so that the balls associated with a node are contiguous in the ball array. The entire construction algorithm then looks very much like quicksort, except that different dimensions are manipulated on different recursive calls.

In the class BALL_ARRAY we define a fairly conventional selection algorithm whose runtime is linear in the length of the range.

```
select_on_coord (c,k,li,ui:INTEGER) is
        -- Move elts around between li and ui, so that the kth element ctr
        -- is >= those below, <= those above, in the coordinate c.
    local l,u,r,m,i:INTEGER; t,s:T;
    do
        from l:=li; u:=ui until not (l<u) loop
            r := rnd.integer_rng_rnd(l,u); -- random integer in [l,u]
            t := get(r); set(r,get(l)); set(l,t); -- swap
            m := l;
            from i:=l+1 until i>u loop
                if get(i).ctr.get(c) < t.ctr.get(c) then
                    m:=m+1;
                    s:=get(m); set(m,get(i)); set(i,s); -- swap
                end; -- if
                i:=i+1
            end; -- loop
            s:=get(l); set(l,get(m)); set(m,s); -- swap
            if m<=k then l:=m+1 end;
            if m>=k then u:=m-1 end
        end -- loop
    end;
```

To actually construct the tree, we initialize a BALL_ARRAY "bls" with the desired leaf balls. The construction can then proceed in a natural recursive manner:

```
build_blt_for_range(l,u:INTEGER):BLT_ND is
        -- Builds a blt for the balls in the range [l,u] of bls.
    local c,m:INTEGER; bl:BALL
    do
        if u=l then -- make a leaf
            Result.Create; Result.set_bl(bls.get(u));
        else
            c := bls.most_spread_coord(l,u); m := int_div((l+u),2); bls.select_on_coord(c,m,l,u); -- split left and right
            Result.Create; Result.set_lt(build_blt_for_range(l,m)); Result.lt.set_par(Result); -- do left side
            Result.set_rt(build_blt_for_range(m+1,u)); Result.rt.set_par(Result); -- do right side
            bl.Create(bls.get(0).dim); bl.to_bound_balls(Result.lt.bl, Result.rt.bl); Result.set_bl(bl); -- fill in ball
        end -- if
    end;
```

The tree that is produced is perfectly balanced, but may not adapt itself well to any hierarchical structure in the leaf balls.

Top Down Construction Algorithm

The k-d construction algorithm doesn't explicitly try to minimize the volume of the resulting tree. For uniformly distributed data, it is not hard to see that it should asymptotically do a good job. It is natural, however, to think that using an explicit volume minimization heuristic to choose the dimension to cut and the value at which to cut it might improve the performance of the algorithm. We refer to this approach as the "top down construction algorithm". As in the k-d approach we work recursively from the top down. At each stage we choose the split dimension and the splitting value along that dimension so as to minimize the total volume of the two bounding balls of the two sets of balls. To find this optimal dimension and split value, we sort the balls along each dimension and construct a cost array which gives the cost at each split location. This is filled in by first making a sequential pass from left to right expanding a test ball to contain each successive entry and inserting its volume in the cost array. While the exact volume of the bounding ball of a set of balls depends on the order in which they are inserted, this approach gives a good approximation to the actual parent ball volume. Next a sequential pass is made from right to left and the bounding ball's volume is added in to the array. In this manner the best dimension and split location are found in $O(N \log N)$ time and the whole algorithm should take $O(N(\log N)^2)$.

We implement this approach in a similar manner to the k-d approach. The array "cst" is used to hold the costs of the different cutting locations.

```
fill_in_cst(l,u:INTEGER) is
      -- Fill in the cost array between l and u. Split is above pt.
   local i:INTEGER
   do
      bl.to(bls.get(l));
      from i:=l until i>=u loop -- do left side
         bl.expand_to_ball(bls.get(i));
         cst.set(i,bl.pvol);
         i:=i+1
      end; -- loop
      bl.to(bls.get(u));
      from i:=u until i<=l loop -- do right side
         bl.expand_to_ball(bls.get(i)); -- info relevant to i-1
         cst.set(i-1,cst.get(i-1)+bl.pvol);
         i:=i-1
      end; -- loop
   end;
```

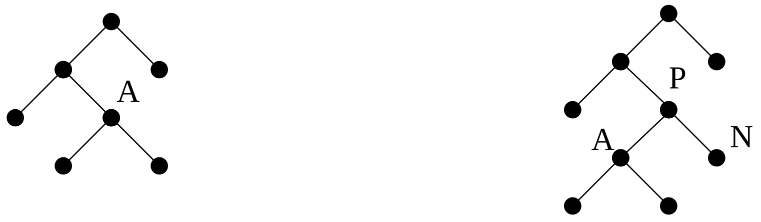
The tree construction proceeds by filling in the cost array for each of the dimensions and picking the best one to recursively proceed with.

```
build_blt_for_range(l,u:INTEGER):BLT_ND is
      -- Builds a blt for the balls in the range [l,u] of bls.
   local i,j,c,m, bdim,bloc:INTEGER; bcst:REAL; nbl:BALL
   do
      if u=l then -- make a leaf
         Result.Create;
         Result.set_bl(bls.get(u));
```

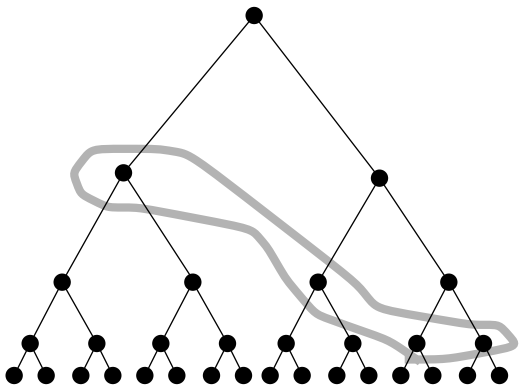
```
        else
            bdim:=0;bloc:=l;
            from i:=0 until i=bls.dim loop
                bls.sort_on_coord(i,l,u);
                fill_in_cst(l,u);
                if i=0 then bcst:=cst.get(l) end; -- initial value
                from j:=l until j>=u loop
                    if cst.get(j)<bcst then bcst:=cst.get(j); bdim:=i; bloc:=j; end;
                    j:=j+1
                end; -- loop
                i:=i+1
            end; -- loop
            bls.sort_on_coord(bdim,l,u); -- sort on best dim
            Result.Create;
            Result.set_lt(build_blt_for_range(l,bloc)); Result.lt.set_par(Result);
            Result.set_rt(build_blt_for_range(bloc+1,u)); Result.rt.set_par(Result);
            nbl.Create(bls.dim); nbl.to_bound_balls(Result.lt.bl, Result.rt.bl); Result.set_bl(nbl);
        end -- if
    end;
```

On-line Insertion Algorithm

The next algorithm builds up the tree incrementally. We will allow a new node N to become the sibling of any node in an existing balltree. The diagram shows the new node N



becoming the sibling of the old node A under the new parent P. The algorithm tries to find the insertion location which causes the total tree volume to increase by the smallest amount. In addition to the volume of the new leaf, there are two contributions to the volume increase: the volume of the new parent node and the amount of volume expansion in the ancestor balls above it in the tree. As we descend the tree, the total ancestor expansion almost always increases while the parent volume decreases. As the search for the best insertion location proceeds, we maintain the nodes at the “fringe” of the search in a priority queue ordered by their ancestor expansion. We also keep track of the best insertion point



found so far and the volume expansion it would entail. When the smallest ancestor expansion of any node in the queue is greater than the entire expansion of the best node, the search is terminated. Nodes may be deleted by simply removing them and their parent and adjusting the volumes of all higher ancestors. Notice that because of the properties of bounding balls, in rare circumstances the expansion of an ancestor node due to a smaller node may be larger than for a smaller node (this doesn't happen if boxes are used instead of balls).

The new volume in internal balls that is created by this operation consists of the entire volume of P plus the amount of expansion created in all the ancestors of P. Choosing the insertion point according to the criterion of trying to minimize this new volume leads to several nice properties. New balls which are large compared to the rest of the tree tend to get put near the top, while small boxes which lie inside of existing balls end up near the bottom. New balls which are far from existing balls also end up near the top. In this way the tree structure tends to reflect the clustering structure of leaf balls.

This routine in the BALLTREE class returns a pointer to the best sibling in the tree. "tb" is a test ball which is a global attribute of the class. "frng" is the priority queue which holds the fringe nodes.

```
best_sib (nl:BLT_ND):BLT_ND is
      -- The best sibling node when inserting new leaf nl.
   local bcost:REAL; -- best cost = node vol + ancestor expansion
         tf,tf2:BLT_FRNG[BLT_ND]; -- test fringe elements
         done:BOOLEAN; v,e:REAL;
   do
      if tree.Void then -- Result.Void means tree is void
      else
         frng.clr;
         Result:=tree; tb.to_bound_balls(tree.bl, nl.bl);
         bcost:=tb.pvol;
         if not Result.leaf then
            tf.Create; tf.set_aexp(0.); -- no ancestors
            tf.set_ndvol(bcost); tf.set_nd(Result);
            frng.ins(tf); -- start out the queue
         end; -- if
         from until frng.empty or done loop
            tf := frng.pop;-- best candidate
            if tf.aexp >= bcost then -- no way to get better than bnd
                done := true -- this is the bound in branch and bound
            else
               e := tf.aexp + tf.ndvol - tf.nd.pvol; --new ancestor expans
               -- do left node
               tb.to_bound_balls(tf.nd.lt.bl,nl.bl); v := tb.pvol;
               if v+e < bcost then bcost := v+e; Result := tf.nd.lt end;
               if not tf.nd.lt.leaf then
                  tf2.Create; tf2.set_aexp(e); tf2.set_ndvol(v);
                  tf2.set_nd(tf.nd.lt); frng.ins(tf2);
               end; -- if
               -- now do right node
               tb.to_bound_balls(tf.nd.rt.bl,nl.bl); v := tb.pvol;
               if v+e < bcost then bcost := v+e; Result := tf.nd.rt end;
               if not tf.nd.rt.leaf then
                  tf2.Create; tf2.set_aexp(e); tf2.set_ndvol(v);
                  tf2.set_nd(tf.nd.rt); frng.ins(tf2);
               end; -- if
            end; -- if
         end; -- loop
         frng.clr;
      end; -- if
   end; -- best_sib
```

Once the sibling of the node is determined, the following routine will create a parent and insert it and the new leaf into the tree. “repair_parents” recursively adjusts the bounding balls in the parents of a node with a changed ball.

```
ins_at_node (nl,n:BLT_ND) is
      -- Make nl be n’s sibling. n.Void if first node.
   local nbl:BALL; npar,nd:BLT_ND;
    do
      nl.set_par(npar); -- just in case something is there
      nl.set_lt(npar);nl.set_rt(npar);
      if tree.Void-- if nothing there, just insert nl
      then tree := nl;
      else
         npar.Create; npar.set_par(n.par);
         if n.par.Void then tree := npar;-- insert at top
         elsif n.par.lt=n then n.par.set_lt(npar)
         else n.par.set_rt(npar)
         end; -- if
         npar.set_lt(n); npar.set_rt(nl);
         nl.set_par(npar); n.set_par(npar);
         nbl.Create(dim);
         nbl.to_bound_balls(nl.bl,n.bl);
         npar.set_bl(nbl);
         repair_parents(npar);
      end; -- if
   end; -- ins_at_node
```

To remove a node, we remove its parent and fix up the bounding balls of the ancestors.

```
rem(n:T) is
      -- Remove leaf n from the tree. Forget its parent
   local np,ns,vdt:BLT_ND;
```

```
 do
    if n.Void then-- do nothing if empty
    elsif n.par.Void then tree.forget -- last node in tree
    else
       np:=n.par;
       if n=np.lt then ns:=np.rt else ns:=np.lt end; -- sibling
       ns.set_par(np.par);
       if np.par.Void then tree:=ns
       elsif np.par.lt=np then np.par.set_lt(ns)
       else np.par.set_rt(ns) end;
       np.Forget; n.set_par(np); -- just in case someone asks for it
       from np:=ns.par until np.Void loop
          np.bl.to_bound_balls(np.lt.bl,np.rt.bl); -- adjust balls
          np := np.par
       end; -- loop
       n.set_par(vdt);
    end; -- if
 end; -- rem
```

Cheaper On-line Algorithm

We also investigated a cheaper version of the insertion algorithm in which no priority queue is maintained and only the cheaper of the two child nodes at any point is further explored. Again the search is terminated when the ancestor expansion exceeds the best total expansion.

```
cheap_best_sib (nl:BLT_ND):BLT_ND is
    -- A cheap guess at the best sibling node for inserting new leaf nl.
 local bcost:REAL;-- best cost = node vol + ancestor expansion
       ae:REAL;-- accumulated ancestor expansion
       nd:BLT_ND; done:BOOLEAN; lv,rv,wv:REAL;
 do
    if tree.Void then-- Result.Void means tree is void
    else
       Result:=tree; tb.to_bound_balls(tree.bl,nl.bl); wv:=tb.pvol;
       bcost := wv;
       ae:=0.;-- ancestor expansion starts at zero.
       from nd:=tree until nd.leaf or done loop
          ae:=ae+wv-nd.pvol; -- correct for both children
          if ae>=bcost then done:=true -- can't do any better
          else
             tb.to_bound_balls(nd.lt.bl,nl.bl); lv:=tb.pvol;
             tb.to_bound_balls(nd.rt.bl,nl.bl); rv:=tb.pvol;
             if ae+lv<=bcost then Result:=nd.lt; bcost:=ae+lv; end;
             if ae+rv<=bcost then Result:=nd.rt; bcost:=ae+rv; end;
             if lv-nd.lt.pvol<=rv-nd.rt.pvol -- left expands less
             then wv:=lv; nd:=nd.lt;
             else wv:=rv; nd:=nd.rt; end;
          end; -- if
       end; -- loop
    end; -- if
 end; -- cheap_best_sib
```

Bottom Up Construction Algorithm

The bottom up heuristic repeatedly finds the two balls whose bounding ball has the smallest volume, makes them siblings, and inserts the parent ball back into the pool. In some ways this is similar to the Huffman algorithm for finding efficient codes. Here, though, the cost depends on the combination of the two nodes being combined and so the choice becomes more expensive. The simplest, brute-force implementation maintains the current candidates in an array and on each iteration checks the volume of the bounding ball of each pair to find the best. A straightforward implementation of this approach requires N passes most of which are of size $O(N^2)$, for a total construction time of $O(N^3)$.

Improved Bottom Up Algorithm

Two observations allow us to substantially reduce the cost of this algorithm. If each node kept track of the other node such that the volume of their joint bounding ball was minimized and the volume of that ball, then the node with the minimal stored cost and its stored mate would be the best pair to join. Secondly, most of the balls keep the same mate when a pair is formed and when one's mate is paired elsewhere the best cost can only increase. As described in the last section, the balltree is an ideal structure for determining each ball's best mate. We therefore maintain a dynamic balltree using one of the insertion algorithms for holding the unfinished pieces of the bottom-up balltree. An initial pass determines the best mate for each node. The nodes are kept in a priority queue ordered by the volume of the bounding ball with their best mate. As the algorithm proceeds some of these mates will become obsolete, but the best bounding volume can only increase. We therefore iterate removing the best node from the priority queue and if it has not already been paired, we recompute its best mate using the insertion balltree. If the recomputed cost is less than the top of the queue, then we remove it and its mate from the insertion balltree, form a parent node above them, compute the parent's best mate and reinsert the parent into the insertion balltree and the priority queue. When there is only one node left in the insertion balltree, the construction is complete. We present the routines for finding the best pair and for merging them. "pq" is the priority queue of pending nodes and the variables "b1" and "b2" will hold the best pair to merge. "has_leaf" tests whether a balltree has a given node as a leaf.

```
find_best_pair is
        -- Put best two to merge in b1,b2, and remove from pq and blt.
    local done:BOOLEAN; btm:BLT_FPND
    do
        b1.Forget; b2.Forget;
        from until done loop
            if pq.empty then done:=true -- returns Void when done
            else
                btm := pq.pop;
                if blt.has_leaf(btm) then -- if not there then keep on
                    blt.rem(btm);-- take it out of the tree
                    btm.set_bvol(blt.best_vol_to_ball(btm.bl)); --recomp
                    if pq.empty or else
                        btm.bvol <= pq.top.bvol then done:=true
                    else
                        pq.ins(btm); blt.cheap_ins(btm); -- try again
                    end; -- if
                end; -- if
            end; -- if
        end; -- loop
        if not btm.Void then
            b1:=btm; b2:=blt.best_node_to_ball(b1.bl); blt.rem(b2);
        end; -- if
    end;
```

```
merge_best_pair is
      -- Combine the best two, replace combo in pq and blt.
   local bn:BLT_ND; bl:BALL; vbf:BLT_FPND
   do
      if (not b1.Void) and (not b2.Void) then
         bn.Create;
         bn.set_lt(b1.tree); bn.lt.set_par(bn);
         bn.set_rt(b2.tree); bn.rt.set_par(bn);
         bl.Create(blt.dim); bl.to_bound_balls(bn.lt.bl,bn.rt.bl);
         bn.set_bl(bl); b1.Create;
         b1.set_tree(bn); b1.set_bl(bl); -- never resize so can share
         b1.set_bvol(blt.best_vol_to_ball(bl));
         pq.ins(b1); blt.cheap_ins(b1);
         b1:=vbf; b2:=vbf;
      end; -- if
   end;
```

The figures compare the construction time of the brute force approach against the algorithmic one for uniform data in 2, 5, and 10 dimensions and show that the speedup is substantial.

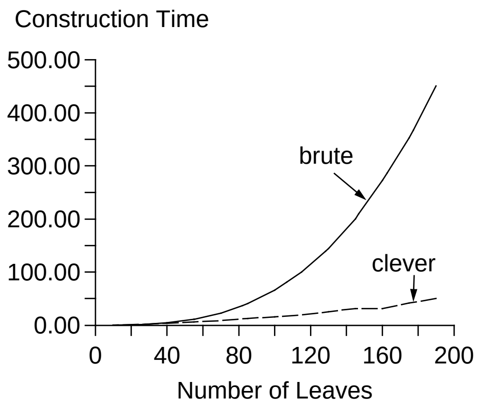


Figure 3. Construction time vs. size for bottum up construction method. Brute force approach is compared with clever one for 2-dimensional uniformly distributed point leaves.

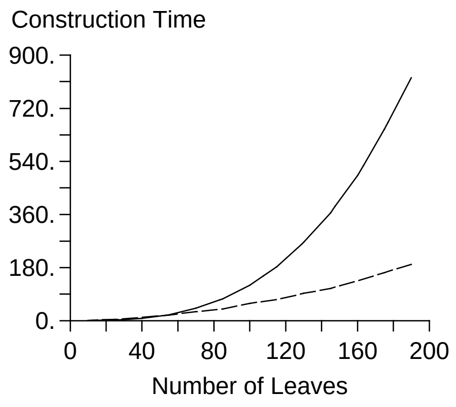


Figure 4. Construction time vs. size for bottum up construction method. Brute force approach is compared with clever one for 5-dimensional uniformly distributed point leaves.

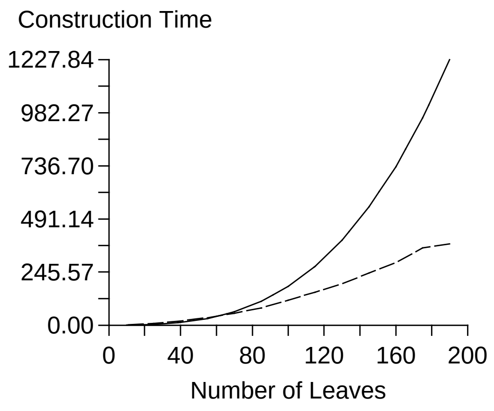


Figure 5. Construction time vs. size for bottum up construction method. Brute force approach is compared with clever one for 10-dimensional uniformly distributed point leaves.

Construction Data

In this section we present the experimental results giving the volume of the constructed balltree and the construction time as a function of the number of leaves for each of the algorithms and each of the distributions discussed above. In each graph a different dashing pattern is used to denote each of the algorithms. The first figures label the curves and the usage is the same in the others.

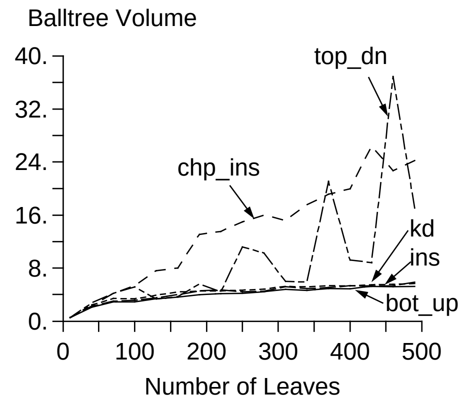


Figure 7. Balltree volume vs. size for 2-dimensional uniformly distributed point leaves.

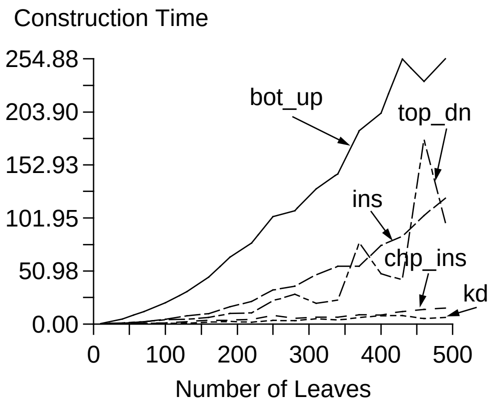


Figure 8. Balltree construction time vs. size for 2-dimensional uniformly distributed point leaves.

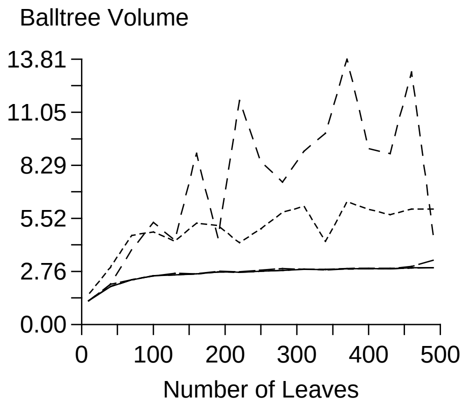


Figure 9. Balltree volume vs. size for 2-dimensional Cantor set distributed point leaves.

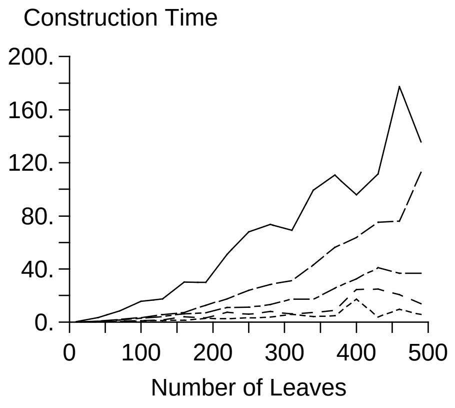


Figure 10. Balltree construction time vs. size for 2-dimensional Cantor set distributed point leaves.

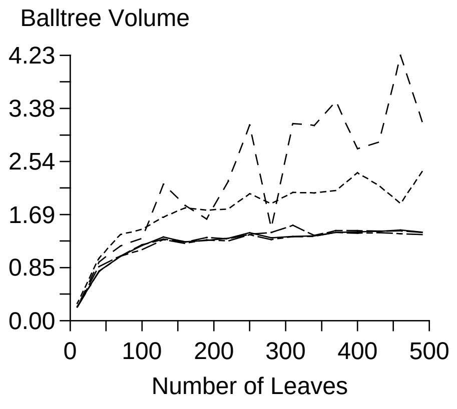


Figure11.Balltreevolumevs.sizefor2-dimensionalpoint leaves distributed on a curve.

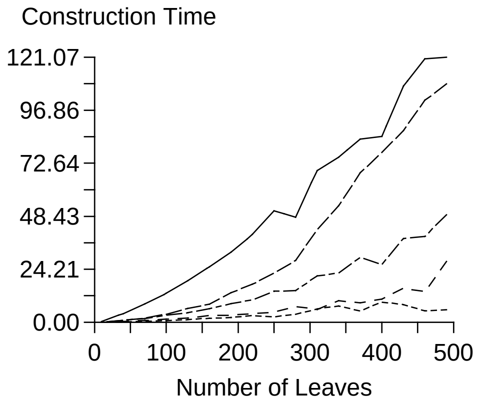


Figure 12. Balltree construction time vs. size for 2-dimensional point leaves distributed on a curve.

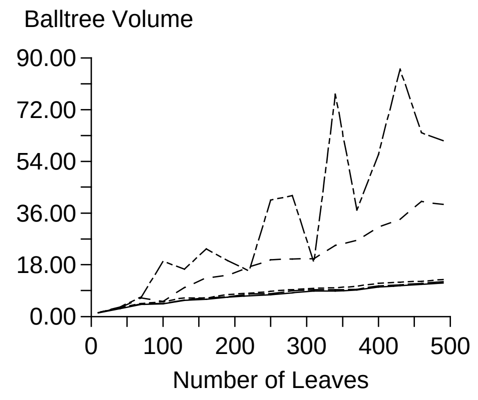


Figure 13. Balltree volume vs. size for 2-dimensional uniformly distributed leaf balls with radii uniformly distributed below .1.

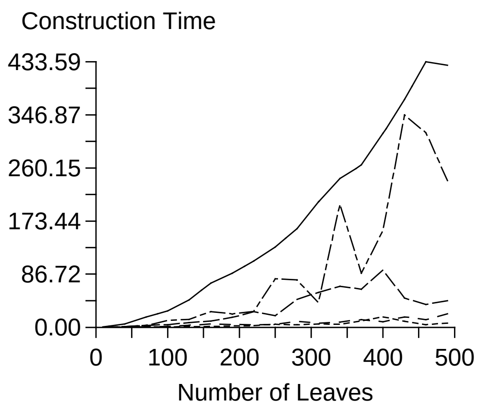


Figure 14. Balltree construction time vs. size for 2-dimensional uniformly distributed leaf balls with radii uniformly distributed less than .1.

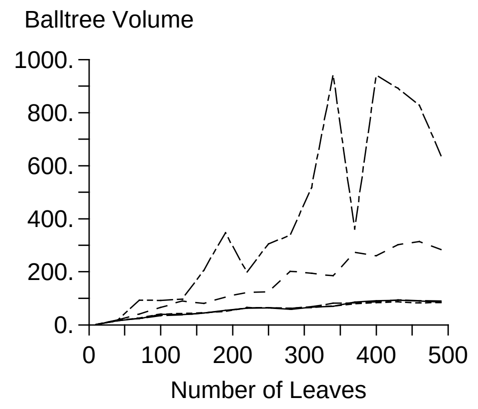


Figure 15. Balltree volume vs. size for 5-dimensional uniformly distributed point leaves.

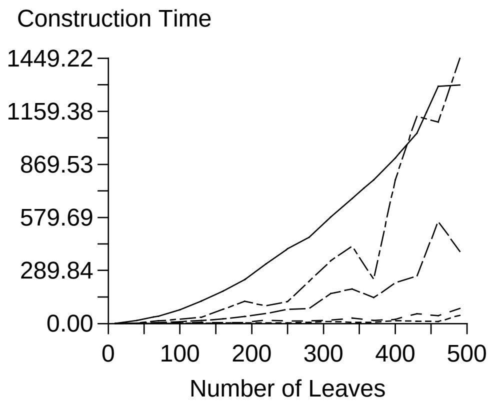


Figure 16. Balltree construction time vs. size for 5-dimensional uniformly distributed point leaves.

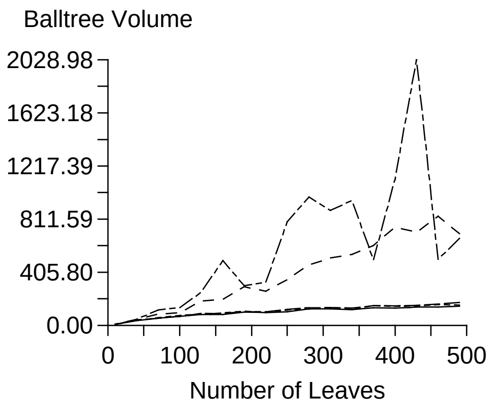


Figure 17. Balltree volume vs. size for 5-dimensional Cantor set distributed point leaves.

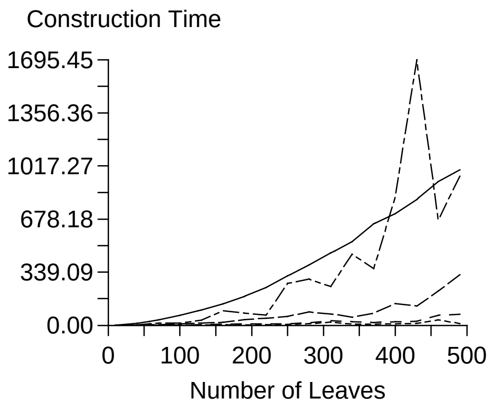


Figure 18. Balltree construction time vs. size for 5-dimensional Cantor set distributed point leaves.

Balltree Volume

Construction Time

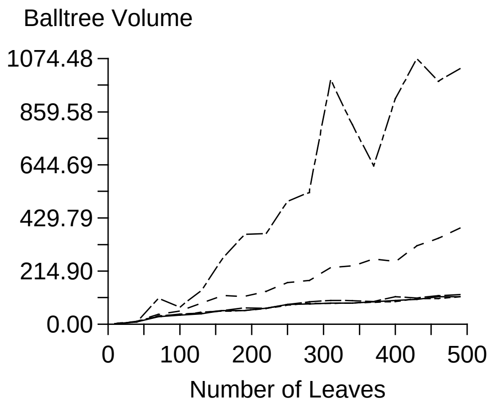


Figure 21. Balltree volume vs. size for 5-dimensional uniformly distributed leaf balls with radii uniformly distributed below .1.

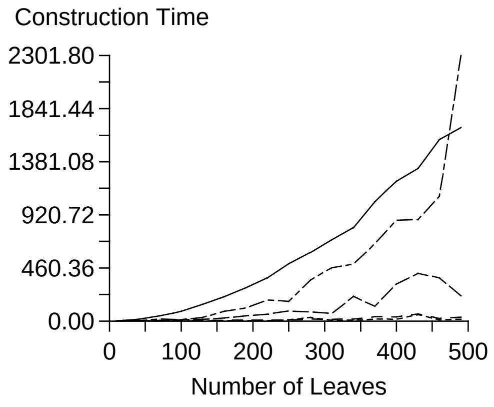


Figure 22. Balltree construction time vs. size for 5-dimensional uniformly distributed leaf balls with radii uniformly distributed less than .1.

Let us now discuss this data. The rankings of the different algorithms for cost of construction were almost identical in all the tests. The bottom up algorithm was virtually always the most expensive (being eclipsed only occasionally by the top down algorithm). This is perhaps to be expected since it must use one of the others during its construction. The *k-d* algorithm in each case had the smallest construction time followed closely by the cheap insertion algorithm. It is heartening that in each case the cost appears to be growing only slightly faster than linear (with the possible exception of the top down algorithm).

The bottom up algorithm consistently produced the best trees followed closely by the insertion algorithm. The *k-d* algorithm did very well on the uniform data (as expected) but rather poorly on the curve and Cantor data. In the next section we will see that this is because it doesn't adapt well to any small-scale structure in the data. On the uniform data, the top-down algorithm was the worst, followed by the cheap insertion algorithm. It is perhaps surprising that the top-down approach did worse than the *k-d* approach on the uniform data. The top-down approach appears very sensitive to the exact structure of the data as evidenced by the wild fluctuations in tree quality. It appears that it must make choices which affect the whole structure and quality of the tree before their true impact is clear. All of the algorithms did quite well on the curve data, particularly in 5 dimensions.

Conclusions

To give further insight into the nature of the trees constructed, figure 23 shows the interior balls and tree structure produced by the five algorithms on 30 Cantor distributed points in the plane. The *k-d* algorithm blindly slices the points in half taking no account of the hierarchical structure. This allows it to produce a perfectly balanced tree but at the expense of missing the structure in the data. The two insertion algorithms produced exactly the same tree in this case. It appears to have early on made a decision which forced the final tree to have a very large ball near the root. This is typical of the cost of using an on-line algorithm. The top-down and bottom up algorithms found very similar trees of essentially equivalent quality.

In conclusion, if the data is smooth and there is lots of it, the *k-d* approach is fast and simple and has much to recommend it. If the data is clustered or sparse or has extra structure, the *k-d* approach tends not to reflect that structure in its hierarchy. The bottom up approach in all cases does an excellent job at finding any structure and aside from its construction cost is the preferred approach. In situations where on-line insertion is necessary, the full insertion algorithm approaches the bottom up algorithm in quality. The cheaper insertion approach does significantly worse but leads to construction times nearing those of the k-d approach. A balltree constructed by any means may be further modified using the insertion or cheaper insertion algorithms.

Bibliography

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel, “An Algorithm for Finding Best Matches in Logarithmic Expected Time,” ACM Transactions on Mathematical Software}, 3:3 (1977) 209-226.

Bertrand Meyer, *Eiffel: The Language*, Interactive Software Engineering, Goleta, CA, 1989.

Stephen M. Omohundro, “Efficient Algorithms with Neural Network Behavior,” Complex Systems, 1 (1987) 273-347.

Stephen M. Omohundro, “Foundations of Geometric Learning,” University of Illinois Department of Computer Science Technical Report No. UIUCDCS-R-88-1408 (1988).

Stephen M. Omohundro, “Geometric Learning Algorithms”, Proceedings of the Los Alamos Conference on Emergent Computation, (1989). (ICSI Technical Report No. TR-89-041)

M. T. Noga and D. C. S. Allison, “Sorting in Linear Expected Time,” *Bit* **25** (1985) 451-465.

Hanan Samet, “The quadtree and related hierarchical data structures,” ACM Computing Surveys 16:2 June (1984) 187-260.6

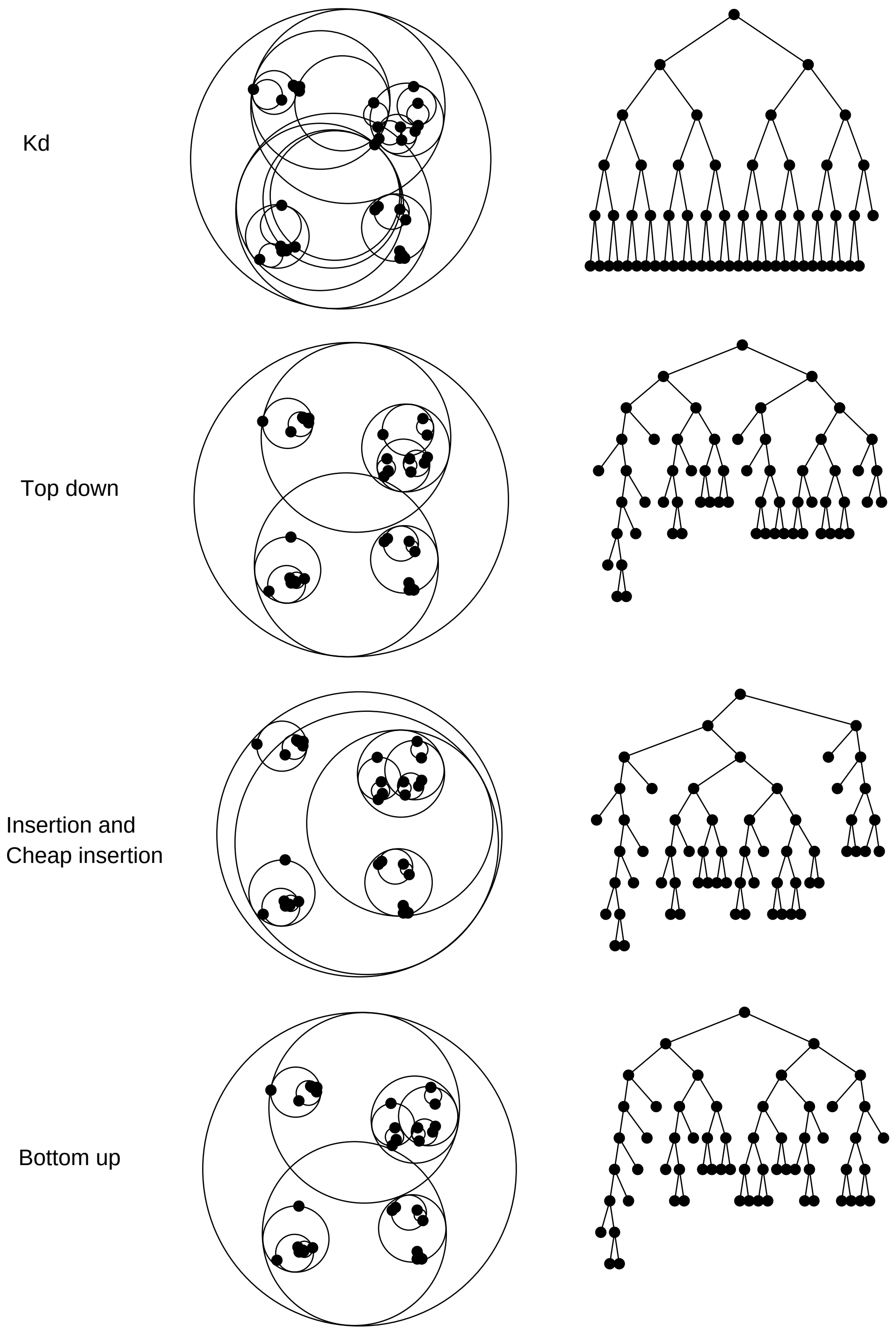


Figure 23. The ball and tree structure created by the five algorithms on Cantor random data.