Exhibit 34

```
h47185
s 00006/00024/00234
d D 1.2 00/07/26 15:47:12 erling 2 1
c cleanup
e
s 00258/00000/00000
d D 1.1 00/07/26 15:36:30 erling 1 0
c date and time created 00/07/26 15:36:30 by erling
e
u
U
f e 0
t
T
I 1
#include "KDx.h"
#include "kd_search.h"                    // kd-search declarations

//----------------------------------------------------------------------
// Approximate nearest neighbor searching by kd-tree search
//        The kd-tree is searched for an approximate nearest neighbor.
//        The point is returned through one of the arguments, and the
//        distance returned is the squared distance to this point.
//
//        The method used for searching the kd-tree is an approximate
//        adaptation of the search algorithm described by Friedman,
//        Bentley, and Finkel, ``An algorithm for finding best matches
//        in logarithmic expected time," ACM Transactions on Mathematical
//        Software, 3(3):209-226, 1977).
//
//        The algorithm operates recursively. When first encountering a
//        node of the kd-tree we first visit the child which is closest to
//        the query point. On return, we decide whether we want to visit
//        the other child. If the box containing the other child exceeds
//        1/(1+eps) times the current best distance, then we skip it (since
//        any point found in this child cannot be closer to the query point
//        by more than this factor.) Otherwise, we visit it recursively.
//        The distance between a box and the query point is computed exactly
//        (not approximated as is often done in kd-tree), using incremental
//        distance updates, as described by Arya and Mount in ``Algorithms
//        for fast vector quantization," Proc. of DCC '93: Data Compression
//        Conference, eds. J. A. Storer and M. Cohn, IEEE Press, 1993,
//        381-390.
//
//        The main entry points is annkSearch() which sets things up and
//        then call the recursive routine ann_search(). This is a recursive
//        routine which performs the processing for one node in the kd-tree.
//        There are two versions of this virtual procedure, one for splitting
//        nodes and one for leaves. When a splitting node is visited, we
```

DfB Exhibit 11
Date: 9/4/19 Name: WOLD

```
//        determine which child to visit first (the closer one), and visit
//        the other child on return.  When a leaf is visited, we compute
//        the distances to the points in the buckets, and update information
//        on the closest points.
//
//        Some trickery is used to incrementally update the distance from
//        a kd-tree rectangle to the query point.  This comes about from
//        the fact that which each successive split, only one component
//        (along the dimension that is split) of the squared distance to
//        the child rectangle is different from the squared distance to
//        the parent rectangle.
//----------------------------------------------------------------------

//----------------------------------------------------------------------
//        To keep argument lists short, a number of global variables
//        are maintained which are common to all the recursive calls.
//        These are given below.
//----------------------------------------------------------------------

int             KDkdDim;            // dimension of space
KDpoint         KDkdQ;                      // query point
double          KDkdMaxErr;         // max tolerable squared error
KDpointArray    KDkdPts;            // the points
KDmin_k         *KDkdPointMK;               // set of k closest points

//----------------------------------------------------------------------
// annkSearch - search for the k nearest neighbors
//----------------------------------------------------------------------


  D 2
MFError annkSearch(KDPointSet* pPS, KDpoint q, int k,
 KDidxArray nn_idx, KDdistArray dd, double eps)
  E 2
  I 2
MFError annkSearch(KDTree* kdTree, KDpoint q, int k, KDidxArray nn_idx,
 KDdistArray dd, double eps)
  E 2
{
   MFError err = MF_SUCCESS;

  D 2
   if (pPS == 0)
  E 2
  I 2
   if (kdTree == NULL)
  E 2
        return MF_INTERNAL_PROGRAM_ERROR;
  D 2
```

```
    if (pPS->utype == BFS)
    {
        if ((err = annkBFS(&pPS->uval.bfs, q, k, nn_idx,  dd,  eps)) !=
          MF_SUCCESS)
            return err;
    }
    else if (pPS->utype == KDT)
    {
        if ((err = annkTreeSearch(&pPS->uval.kdTree, q, k, nn_idx,  dd,  eps)) !=
          MF_SUCCESS)
            return err;
    }
    else
        return MF_INTERNAL_PROGRAM_ERROR;
    E 2

    I 2
    if ((err = annkTreeSearch(kdTree, q, k, nn_idx,  dd,  eps)) != MF_SUCCESS)
        return err;

    E 2
    return MF_SUCCESS;
}

    D 2
MFError annkBFS(KDbruteForce* pBF, KDpoint q, int k,
  KDidxArray nn_idx, KDdistArray dd, double eps)
{
    /*TRACE("In annkBFS stub\n");*/
    return MF_FAILURE;
}

    E 2
MFError annkTreeSearch(KDTree* pKDT, KDpoint q, int k,
  KDidxArray nn_idx, KDdistArray dd, double eps)
{
    int i;

    KDkdDim = pKDT->dim;    /* copy arguments to static equivs */
    KDkdQ = q;
    KDkdPts = pKDT->pts;
    KDptsVisited = 0;   /* initialize count of points visited */

    if (k > pKDT->n_pts)
        return MF_FAILURE; /* return error code */

    KDkdMaxErr = KD_POW(1.0 + eps);

    KDkdPointMK = KDminKCreate(k);         /* create set for closest k points */
```

```
    if (pKDT->root->uType == KDLEAF)
    {
        kdLeafSearch(pKDT->root, annBoxDistance(q,
     pKDT->bnd_box_lo, pKDT->bnd_box_hi, pKDT->dim));

        /* replace KDkd_leaf::ann_search below w kdLeafSearch() */
    }
    else if (pKDT->root->uType == KDSPLIT)
    {
        kdSplitSearch(pKDT->root, annBoxDistance(q,
     pKDT->bnd_box_lo, pKDT->bnd_box_hi, pKDT->dim));

        /* replace KDkd_split::ann_search below w kdSplitSearch() */
    }


    for (i = 0; i < k; i++)
    {   /* extract the k-th closest points */
        dd[i] = ith_smallest_key(KDkdPointMK, i);
        nn_idx[i] = ith_smallest_info(KDkdPointMK,i);
    }
    Free(KDkdPointMK);              /* deallocate closest point set */

    return MF_SUCCESS;
}

//----------------------------------------------------------------------
//  kd_split::ann_search - search a splitting node
//----------------------------------------------------------------------

void kdSplitSearch(KDkd_node* pN, KDdist box_dist)
{
    KDcoord cut_diff;
    KDcoord box_diff1;
    KDcoord box_diff2;

    // check dist calc termination condition
    if (KDmaxPtsVisited && KDptsVisited > KDmaxPtsVisited)
        return;

                                    // distance to cutting plane
    cut_diff = KDkdQ[pN->cut_dim] - pN->cut_val;

    if (cut_diff < 0)
    {                       // left of cutting plane
        if (pN->child[LO]->uType == KDLEAF)
            kdLeafSearch(pN->child[LO], box_dist);
        else
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.