Larry Andrews

A Template for the Nearest Neighbor Problem

Introduction

The NNP (nearest neighbor problem) is central to the solution of many practical applications. Also known as the "Post Office Problem," it describes the need to find among a group of known positions, the point closest to some randomly chosen probe position. Obvious uses for such an algorithm are various map-lookup problems, cluster analysis, finite-element problems, string lookup, and computing intermolecular contacts between two protein molecules.

It is usually assumed that lots of lookups will be done against a static dataset, so the cost of building the lookup structure is usually less important than the cost of the lookup itself. There are many methods in the literature for solving the NNP: kd-trees, quadtrees, and scan-line-based methods. In 1983, MacDonald and Kalantari [1] published an overlooked algorithm, based on what you would now call partition trees. Their algorithm may have been overlooked since it used a complicated double-linked tree, but simple recursion accomplishes the same result. It is optimal (in most cases) and fast; it has the advantage that storage is nearly linear in the number of objects in the database. Time to build the tree is proportional to `(n log n)` for `n` points in the data set, and the time to retrieve the nearest neighbor to some probe point is proportional to `(log n)(log m)`, where `m` is the dimensionality of the data.

Over the years, I have used this algorithm to solve many problems from my own work. Rewritten in five computer languages, it has allowed me to do searches in large datasets, typically in 6, 7, 10, or up to 36 dimensions. Computing the intermolecular contacts between protein molecules in crystal was simple to implement and extremely fast to execute. Just for fun, I wrote a program to draw a line from a screen cursor to the nearest point among 20,000 random points on the screen; it worked in real time. A simple modification allows the method to return all points within a specified radius of the probe point. Joining the set of near points in real time to the cursor on the screen makes it look like a spider is crawling on the monitor.

In translating the algorithm into C++, I needed to handle cases in several dimensions, sometimes within a single program, so a generic solution was important. Since the algorithm itself is independent of dimension, the creation of a template class seemed the obvious solution. That solution is presented here.

Partitioning the Space

The algorithm begins by selecting points to add to a tree (Neartree). The points are added one by one, and it helps if they are selected in a somewhat random way. The method works for any selection order, but in the worst case, the search time is `O(n)`.

For the first two points, the space is divided by a manifold bisect-

ing the line between the two points. (For points in a plane, the bisector is a line.) From then on, each added point is examined to see which half-plane it is in, and it is added to the Neartree node that defined that half-plane. Another way to say the same thing is that every point after the first two is attached in the tree to the nearer of those two points. When two points have been added to one of the original half-planes, the bisector of the line between those two points is used to divide that half-plane. Figure 1 illustrates the initial steps.

Building the Tree

Implementing the partitioning rule is simple. There are only three cases for the condition of a Neartree node when a new point is to be added:

1. The node is empty (just created); the new point is made the left object.
2. The node is half full (only one point has been added, by convention, on the left); the new point is made the right object.
3. The node is full (both the right and left points have objects in them); the new point is added to the Neartree below the nearer of the two points.

In the third case, there is one piece of bookkeeping to be done. The distance from the object in the node to the object below in the tree that is farther away from the node point needs to be updated. If the stored maximum distance is less than the distance from the node point to the new point, then the maximum distance is updated to be the new value. This will become crucial in speeding the search for nearest neighbors. Listing 1 lists the template and includes the class constructor and the code for `m_fnInsert`, which places the points into the Neartree.

Finding the Nearest Neighbor

The Neartree isn't much good without a search algorithm for it. Every node in the Neartree has at least one node point in it, and many have two. And many of the node points have Neartrees descending from them. For a given probe point (`p`) and initial search radius (`dRadius`):

1. Examine the left node object; if it is within the minimum search radius, remember the object and decrease the search radius to the new minimum distance.
2. Examine the right object (if it exists) and repeat the update if appropriate.
3. For each of the Neartrees descending from the node (there may be none, one, or two), use the triangle rule to decide whether to search more deeply. If the rule indicates that there could be closer objects below in the Neartree, then apply steps 1 and 2 to the node below.
4. When there is no more searching to perform, return the object nearest to the probe point.

Listing 1 includes the code for functions that implement the search for the nearest neighbor search. The code is divided into a public and a private portion so that the user can input a `const` value for the search radius.

The Triangle Inequality

The power of this algorithm derives mostly from a simple rule that you learned in high school geometry: the triangle inequality. This simply states that no side of a triangle can be larger than the sum of the lengths of the other two sides. The NNP algorithm uses

this rule to truncate searches earlier than they would be otherwise (at least most of the time). The truncation reduces what would be a linear search to `O(log n)`.

In this algorithm, I use the rule in the following way. At any given node, for some search point and for some search radius, the distance from the node object to the farthest object below it in the Neartree is known (stored during Neartree construction). Looking at Figure 2, you see that any possible objects that are closer than the current search radius lie only within the intersection of the circle around the probe point (with radius, `dRadius`) and the circle around the node object (with radius, `dMax`). If the two circles do not intersect, then no object exists in the Neartree that could form a triangle with the probe and the node object. In other words,

```
D - dRadius >= dMax,
```

where `D` is the distance between the probe point and the node object. The inequality must be true if any further solutions are possible. If there are no solutions, then I do not need to search the tree any deeper.

This NNP algorithm would not be as powerful if it did not use the triangle inequality. The ability to prune branches as searches become infeasible is what speeds the search. Reducing the search radius as new, closer points are found further prunes the search. In the end, only a few percent (or less) of the points in the tree are usually examined.

Changing the Distance Measure

In the example code, the distance measure is the normal Euclidean (L2) measure. But any measure that obeys the triangle inequality could be used, and sometimes there are advantages to other measures. I sometimes use city block measure (L1) because it is faster to compute, especially since it does not require a square root. For many computations, city block measure is completely adequate. For example, if you have a large dataset and want to know if the probe point exists within the dataset, the minimum distance will be zero if the point is in the dataset. Its distance from the probe will be zero by any of several measures, including city block measure.

Another measure that can be used is Hamming distance (zero for when the coordinate values are the same and one for when they are different). So for the vectors {1,2,3} and {1,3,2}, the Hamming distance is two. It's simple, but it would suffice to determine if a point were within a dataset.

Finally, another simple measure is the maximum value (L-infinity) measure. The distance is the absolute value of the largest of the coordinate value differences found. For instance, for the vectors {1,2,3} and {4,4,4}, the L-infinity distance between them is three.

Points within a Sphere

The very same Neartree that is used for finding nearest neighbors can be used to find the contents of a spherical region. If the search radius is not updated as new objects near the probe object are found, then all node objects within the search radius of the probe will be visited. In the accompanying code, those objects are returned in a data structure `std::vector<?>` so that they can all be examined.

Farthest Neighbor, an Extra Benefit

While I have never needed to find the element of a dataset that is farthest from some probe, it could be useful in some computer

graphics applications. The code to do this search can use the identical Neartree as the NNP, however, the code is subtly different. The trick is that instead of requiring that the triangle inequality be true, now the inequality must be false for there to be new points to be tested. The search is truncated because the search radius grows as the search progresses. In Figure 3, only the little shaded region could possibly contain any new points to be tested. In other words, once `dRadius` exceeds the distance from the probe point to the current object plus the distance to the farthest object below, the search on that branch is stopped.

Limitations

The first obvious limitation of this algorithm is that it requires a metric space with a measure that obeys the triangle inequality. While there are lots of cases where that is not a particular problem, there are many database uses where no particular metric is known. So the NNP mostly applies to geometrical problems where inexact matching is appropriate.

A second, more subtle limitation is that ordering of the input while building the Neartree can have large effects on the retrieval speed. As an example of worst-case behavior, consider a one-dimensional problem where integers are input in ascending order. In the whole tree, no left object will have a descending tree; if the probe is closest to the last object that was added to the Neartree, every object in the tree will have been visited to find it.

In most cases, there is some mitigating factor. For example, consider the case of a protein molecule. The entire molecule may be a single chain of amino acid residues, and they will usually be input in serial order so that each residue input is close to the previous, similar to the worst case scenario above. However, the whole molecule folds up, and as the molecule snakes around, the near contact with itself makes the Neartree better conditioned. Experiments of adding atoms randomly or of trying to choose initial atoms widely separated did not noticeably or consistently affect the speed of retrieval.

One consideration about using this NNP algorithm is that you cannot delete or update nodes without rebuilding the tree. Because the distance from any node object to the farthest object below is stored, moving any object would require updating all `dMax`'s up to the root. One possible way around this problem for small perturbations is to save the magnitude of the shift vector and add it to the search radius; then after searching, it would be necessary to reconfirm the matches.

Implementation

The class the `CNearTree` template acts upon must provide a few methods: `operator double` and `operator-` are required. I usually include a constructor, a copy constructor, and a destructor. I have included a sample program using a simple class in the online source (available at <www.cuj.com/code>). A complete program that uses `double` as the parameterized type is:

```
#include "TNear.h"
#include <cstdio>
void main()
{ CNearTree< double > dT;
  double dNear;
// store 1.5 in the tree
  dT.m_fnInsert( 1.5 );
// find the nearest point to 2.0
  if ( dT.m_bfnNearestNeighbor(
```

```
      10000.0,   dNear,  2.0 )) }
  printf("%f\n",double(dNear-2.0); }
```

and it should print 0.5 (that's how far 2.0 is from 1.5)

Internal vs. External Storage

As written, the template code stores the node objects in the tree itself. Copying the objects into the Neartree causes a small time penalty, but if the objects are also being stored outside in another data structure, then there is also a space penalty. One large advantage to copying the objects into the tree is that they are then protected from modification or deletion. But the template can easily be modified to only store pointers to the data, or the objects could be modified to be pointers and the template left unchanged. ❑

Reference

[1] Iraj Kalantari and Gerard McDonald. "A Data Structure and an Algorithm for the Nearest Point," *IEEE Transactions on Software Engineering*, September 1983.

Larry Andrews has a Ph.D. in chemistry from the University of Washington in Seattle. He can be reached at andrewsl@ix.netcom.com.

Listing 1: ***Template for the nearest neighbor search***

```
#if !defined(TNEAR_H_INCLUDED)
#define TNEAR_H_INCLUDED


#include <limits.h>
#include <float.h>
#include <math.h>
#include <vector>

//============================================================
template <typename T> class CNearTree
{
    T * m_ptLeft; // left object stored in this node
    T * m_ptRight;// right object stored in this node
    double m_dMaxLeft; //longest distance from the left object
                       // to anything below it in the tree
    double m_dMaxRight;// longest distance from the right object
                       // to anything below it in the tree
    CNearTree * m_pLeftBranch;  //tree descending from the left
    CNearTree * m_pRightBranch; //tree descending from the right

public:

//============================================================
   CNearTree(void)  // constructor
   {
      m_ptLeft       = 0;
      m_ptRight      = 0;
      m_pLeftBranch  = 0;
      m_pRightBranch = 0;
      m_dMaxLeft     = DBL_MIN;
      m_dMaxRight    = DBL_MIN;
   }  //  CNearTree constructor

//============================================================
   ~CNearTree(void)  // destructor
   {
      delete m_pLeftBranch  ;  m_pLeftBranch  =0;
      delete m_pRightBranch ;  m_pRightBranch =0;
      delete m_ptLeft       ;  m_ptLeft       =0;
      delete m_ptRight      ;  m_ptRight      =0;
```

Listing 1: ***continued***

```
      m_dMaxLeft     = DBL_MIN;
      m_dMaxRight    = DBL_MIN;
   }  //  ~CNearTree

//=============================================================
   void m_fnInsert( const T& t )
   {
      // do a bit of precomputing if possible so that we can
      // reduce the number of calls to operator 'double' as much
      // as possible; 'double' might use square roots
      double dTempRight =  0;
      double dTempLeft  =  0;
      if ( m_ptRight  != 0 )
      {
         dTempRight  = fabs( double( t - *m_ptRight ));
         dTempLeft   = fabs( double( t - *m_ptLeft  ));
      }
      if ( m_ptLeft == 0 )
      {
         m_ptLeft = new T( t );
      }
      else if ( m_ptRight == 0 )
      {
         m_ptRight   = new T( t );
      }
      else if ( dTempLeft > dTempRight )
      {
         if (m_pRightBranch==0) m_pRightBranch=new CNearTree;
         // note that the next line assumes that m_dMaxRight
         // is negative for a new node
         if (m_dMaxRight<dTempRight) m_dMaxRight=dTempRight;
         m_pRightBranch->m_fnInsert( t );
      }
      else
      {
         if (m_pLeftBranch ==0) m_pLeftBranch=new CNearTree;
         // note that the next line assumes that m_dMaxLeft
         // is negative for a new node
         if (m_dMaxLeft<dTempLeft) m_dMaxLeft=dTempLeft;
         m_pLeftBranch->m_fnInsert( t );
      }
   }  //  m_fnInsert

//=============================================================
   bool m_bfnNearestNeighbor ( const double& dRadius,
                               T& tClosest,   const T& t ) const
   {
      double dSearchRadius = dRadius;
      return ( m_bfnNearest ( dSearchRadius, tClosest, t ) );
   }  //  m_bfnNearestNeighbor

//=============================================================
   bool m_bfnFarthestNeighbor ( T& tFarthest, const T& t ) const
   {
      double dSearchRadius = DBL_MIN;
      return ( m_bfnFindFarthest (dSearchRadius, tFarthest, t));
   }  //  m_bfnFarthestNeighbor

//=============================================================
   long m_lfnFindInSphere ( const double& dRadius,
               std::vector<  T >& tClosest,   const T& t ) const
   {   // t is the probe point, tClosest is a vector of contacts
      // clear the contents of the return vector so that
      // things don't accidentally accumulate
      tClosest.clear( );
      return ( m_lfnInSphere( dRadius, tClosest, t ) );
   }  //  m_lfnFindInSphere

private:

//=============================================================
   bool m_bfnNearest ( double& dRadius,
                       T& tClosest,   const T& t ) const
   {
      double   dTempRadius;
      bool  bRet = false;
      // first test each of the left and right positions to see
      // if one holds a point nearer than the nearest so far.
      if (( m_ptLeft!=0 ) &&
        ((dTempRadius = fabs(double(t-*m_ptLeft))) <= dRadius))
```

Listing 1: ***continued***

```
      {
         dRadius  = dTempRadius;
         tClosest = *m_ptLeft;
         bRet     = true;
      }
      if (( m_ptRight!=0 ) &&
        (( dTempRadius = fabs(double(t-*m_ptRight)))<=dRadius))
      {
         dRadius  = dTempRadius;
         tClosest = *m_ptRight;
         bRet     = true;
      }
      // Now we test to see if the branches below might hold an
      // object nearer than the best so far found. The triangle
      // rule is used to test whether it's even necessary to
      // descend.
      if (( m_pLeftBranch  != 0 )  &&
        ((dRadius+m_dMaxLeft) >= fabs(double(t-*m_ptLeft))))
      {
         bRet|=m_pLeftBranch->m_bfnNearest(dRadius,tClosest,t);
      }

      if (( m_pRightBranch != 0 )  &&
        ((dRadius+m_dMaxRight) >= fabs(double(t-*m_ptRight))))
      {
         bRet|=m_pRightBranch->m_bfnNearest(dRadius,tClosest,t);
      }
      return ( bRet );
   };   // m_bfnNearest

//=============================================================
   long m_lfnInSphere ( const double& dRadius,
               std::vector<  T >& tClosest,   const T& t ) const
   {   // t is the probe point, tClosest is a vector of contacts
      long lReturn = 0;
      // first test each of the left and right positions to see
      // if one holds a point nearer than the search radius.
      if ((m_ptLeft!=0) && (fabs(double(t-*m_ptLeft))<=dRadius))
      {
         tClosest.push_back( *m_ptLeft ); // It's a keeper
         lReturn++ ;
      }
      if ((m_ptRight!=0)&&(fabs(double(t-*m_ptRight))<=dRadius))
      {
         tClosest.push_back( *m_ptRight ); // It's a keeper
         lReturn++ ;
      }
      //
      // Now we test to see if the branches below might hold an
      // object nearer than the search radius. The triangle rule
      // is used to test whether it's even necessary to descend.
      //
      if (( m_pLeftBranch  != 0 )  &&
         ((dRadius+m_dMaxLeft) >= fabs(double(t-*m_ptLeft))))
      {
         lReturn +=
           m_pLeftBranch->m_lfnInSphere( dRadius, tClosest, t );
      }
      if (( m_pRightBranch != 0 )  &&
         ((dRadius+m_dMaxRight) >= fabs(double(t-*m_ptRight))))
      {
         lReturn +=
           m_pRightBranch->m_lfnInSphere( dRadius, tClosest, t );
      }
      return ( lReturn );
   }  //  m_lfnInSphere

//=============================================================
   bool m_bfnFindFarthest ( double& dRad,
                            T& tFarthest,   const T& t ) const
   {
//   deleted from the journal listing since it is quite similar
//   to nearest
        return ( false );
   };   // m_bfnFindFarthest


}; // template class TNear

#endif // !defined(TNEAR_H_INCLUDED)
```

— End of Listing —

Figure 1: ***Initial steps in building the Neartree by partitioning the space. The numbers indicate the order in which the objects were inserted***

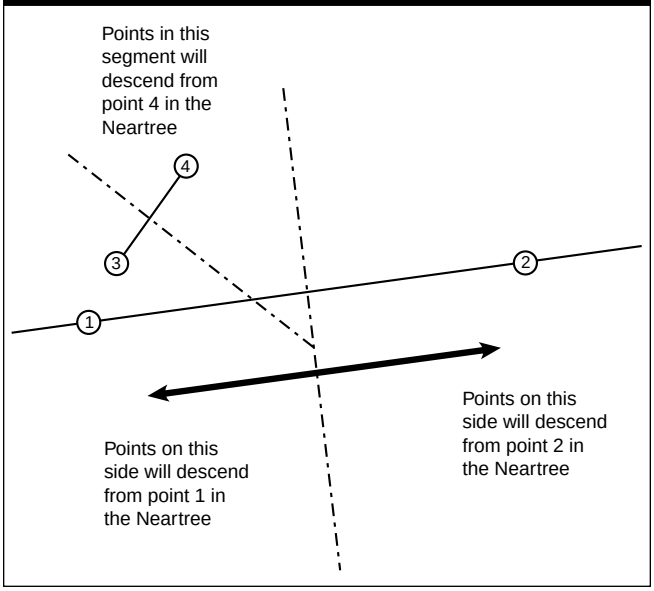

Figure 2: ***A 2-dimensional example of one step in the Nearest Neighbor Algorithm***

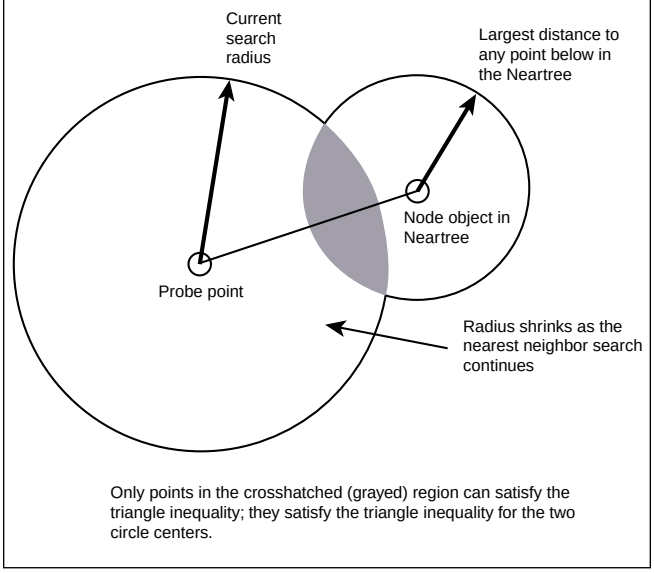

Figure 3: ***A 2-dimensional example of one step in the Farthest Neighbor Algorithm***