ABSTRACT

IMPLEMENTATION OF A GEOMETRIC HASHING TECHNIQUE AND ITS APPLICATION TO 3D MOLECULAR STRUCTURE SEARCH

by
Jennifer Lynn Cerequas

This paper proposes the application of a geometric hash technique to the searching of 3D chemical structures. Chemical structures are represented in global XYZ coordinate format. An algorithm is applied to the substructures within the existing chemical structures to hash them into hash tables on disk. A query substructure is then hashed to find matches (hits) of the existing hash tables. The entries in the matching hash tables are compared to the query substructure to find the existing substructures that are an "approximate" match.

The result is a technique which allows existing substructures to be compared to a (new) query substructure and matches found in the case of rotation, atom insertion/deletion, and small differences in precision of the XYZ coordinates.

IMPLEMENTATION OF A GEOMETRIC HASHING TECHNIQUE AND ITS APPLICATION TO 3D MOLECULAR STRUCTURE SEARCH

by
Jennifer Lynn Cerequas

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

May 1998

APPROVAL PAGE

IMPLEMENTATION OF A GEOMETRIC HASHING TECHNIQUE AND ITS APPLICATION TO 3D MOLECULAR STRUCTURE SEARCH

Jennifer Cerequas

5/7/1998

Dr. Jason T.L. Wang, Thesis Adviser — Date
Associate Professor of Computer and Information Science, NJIT

5/7/1998

Dr. James Calvin, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

5-7-98

Dr. Franz Kurfess, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Jennifer Lynn Cerequas

Degree: Master of Science in Computer Science

Date: May 1998

Undergraduate and Graduate Education:

- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 1998
- Bachelor of Engineering (Major in Computer Science)
  Stevens Institute of Technology, Hoboken, NJ, 1986

Major: Computer Science

This thesis is dedicated to
my husband, Fred R. Cerequas IV, for all his love and support.

ACKNOWLEDGMENT

The author wishes to give special thanks to her advisor, Professor Jason T.L. Wang for all his guidance, encouragement, and patience throughout this project. Thanks also to Professors James Calvin and Franz Kurfess for serving as committee members.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

There are many cases where 3D graphs are used to represent physical entities. This includes chemical compounds and proteins. It is desired to search existing 3D molecular databases, for example, to find compounds similar to a new compound in drug discovery [Charifson and Leach 1998]. Similar compounds should be found in the case of rotation, and adding/deleting an atom. It is also desired that the search be as efficient as possible.

This thesis presents the application of geometric hashing technique [Lamden and Wolfson 1998] to 3D molecular structures [Wang and Wang 1997]. In the first step, existing structures (which have been divided into substructures) are converted from their global XYZ coordinates to a local XYZ coordinate frame and then hashed into 3D hash tables on disk. The lowest numbered node in the substructure is at the origin of the local coordinate frame.

The substructure is processed (hashed) three nodes (atoms) at a time. Three points is enough to be uniquely defined in three dimensions. The lengths of the three legs of the triangle formed by the set of three nodes are used to determine the hash table address. The local coordinates of the three nodes, with respect to the basis points of the local coordinate frame, are stored in the appropriate hash table (file).

In the second step, a query substructure can be compared against the existing substructures in the hash tables on disk. Again, the query substructure is processed three nodes at a time. Once the hash table address is determined, the existing entries are

compared against the global coordinates of the current three nodes. The result is used to maintain a counter for each existing substructure. Each time the result for an existing substructure is the same, the counter is incremented. It can be shown that if the counter is large enough relative to the number of nodes in the query substructure, the existing substructure should be considered a match.

CHAPTER 2

APPLICATION DEVELOPMENT

2.1 Environment

The current programs were developed on a Pentium PC with Windows95 OS using the Borland C/C++ compiler and programming environment. The input was 3D substructures in a modified MOLfile format.

2.2 Algorithms

2.2.1 Preprocessing Phase

This phase processes the existing input substructures from Ascii files and outputs each into one or more Ascii hash files. Each input file will contain a substructure in a modified MOLfile [Dalby and Nourse 1992] format. The first line in the file contains the structure number (1-M), and the substructure number (0-N) within the structure. The following lines contain the XYZ global coordinates for each node (atom) in the substructure.

The lowest numbered (first) node (atom) in the substructure will be used to the determine the local XYZ coordinate frame for the substructure. This node will be the origin of the local coordinate frame. The local coordinate frame is represented by three basis points:

PB1 $(X0,Y0,Z0)$ PB2 $(X0+1,Y0,Z0)$ PB3 $(X0,Y0+1,Z0)$

Using these basis points, subtract their coordinates from the global coordinates of each node in the substructure to determine the local coordinates of each node:

$X_J - X0,\ \ Y_J - Y0,\ \ Z_J - Z0$

The hash address is determined for each combination [Knuth 1973] of three nodes in the substructure. For each combination of three nodes, determine the distance between each pair of nodes as follows:

$$L1 = ((X_i - X_j)^2 + (Y_i - Y_j)^2 + (Z_i - Z_j)^2)$$

$$L2 = ((X_i - X_k)^2 + (Y_i - Y_k)^2 + (Z_i - Z_k)^2)$$

$$L3 = ((X_k - X_j)^2 + (Y_k - Y_j)^2 + (Z_k - Z_j)^2)$$

Use these lengths to sort the three nodes so the pair with the shortest distance is stored first. This is done to maintain consistency with the on-line, search phase.

Each length is multiplied by a large number (e.g. 10,000) so it can be treated as an integer and small differences in precision are ignored. This value is then divided by a prime number (e.g. 1009), and the remainder divided by the number of desired hash entries for each of three dimensions (e.g. 62). The hash address of the three node combination (i,j,k) will therefore be:

$h[l1][l2][l3]$

The entry in the hash table will consist of the 3D graph number (1-m), the substructure number (0-n), and the local coordinates of the three nodes with respect to the

basis points of the substructure. The vectors $V_{i,bn}$ are used to represent another local coordinate frame for the three nodes. The coordinates of the three basis points with respect to this local coordinate frame form a 3 x 3 matrix which is calculated as follows:

$$SF_0[i,j,k] = \begin{bmatrix} V_{i,b1} \\ V_{i,b2} \\ V_{i,b3} \end{bmatrix} \quad X \quad A^{-1}$$

where

$$A = \begin{bmatrix} V_{i,j} \\ V_{i,k} \\ V_{i,j} \times V_{i,k} \end{bmatrix}$$

($V_{i,b1}$ is the vector from node *i* to basis point 1; $V_{i,j}$ is the vector from node *i* to node *j.)*

The hash table entry (at address *h[I1][I2][I3])* for the three node combination (i,j,k) is therefore:

D, S, SFs$[i,j,k]$

where D is the structure number (1-M), and S is the substructure number (0-N) within D.

2.2.2 On-Line Phase

This phase processes a query substructure Q from an Ascii file and outputs the existing structure and substructure numbers (D,S) that are considered to match Q.

Q is hashed using the same techniques described in section 2.2.1. For each combination of three nodes (u,v,w) in Q, the hash address is calculated. Then the coordinates of the three basis points of each existing entry in the hash table with respect to the global coordinate frame of the query substructure are calculated as follows:

$$SF_Q = SFs[i,j,k] \quad X \begin{bmatrix} V_{u,v} \\ V_{u,v} \\ V_{u,v} \; x \; V_{u,v} \end{bmatrix} \quad + \quad \begin{bmatrix} P_u \\ P_u \\ P_u \end{bmatrix}$$

where P_u is the global coordinate of node u.

For each match (hit) in the hash table for substructure s that yields the same SF_Q, increment a counter for s. Then, for each counter that is greater than:

(n-1) x (n-2) x (n-3) / 6

where n is the number of nodes in Q, display the structure number D and substructure number S since this substructure is considered an "approximate" match.

CHAPTER 3

RESULTS AND DISCUSSION

3.1 Pre-Processing Phase

The two input files each contained one substructure from the input structure.

The input data for substructure 0 was as follows:

```
1   0
7
  1.0178   1.0048   2.5101
  1.2021   2.0410   2.0020
  1.3960   2.9864   2.0006
  0.7126   2.0490   3.1921
  0.7610   2.7125   3.0124
  1.0097   3.6478   2.2660
  1.1329   4.5002   2.2024
```

The input data for substructure 1 was as follows:

```
1   1
6
  1.0097   3.6478   2.2660
  1.1329   4.5002   2.2024
  1.5309   5.2026   1.7191
  1.4529   6.1015   1.5712
  1.0356   6.0030   2.2820
  0.7359   5.0571   2.6857
```

When the pre-processing program is run for substructure 0, thirty-five hash files are created (there are thirty-five combinations of seven nodes taken three at a time). When the program is run for substructure 1, twenty hash files are created (there are twenty combinations of six nodes taken three at a time).

3.2 On-Line Phase

The input file for the query substructure was as follows:

```
0   -0.020300  2.964012  2.777921
1   -0.269000  4.153153  2.911494
2   -0.317400  4.749386  3.253592
3    0.172100  3.913515  4.100777
4    0.366000  3.244026  3.433268
```

The output to the screen showed a match for structure 1, substructure 0. The number of hits to the hash files was ten (there are ten combinations of five nodes taken three at a time). The counter was ten.

The results show that a match is found even though the global coordinates of the query substructure are different from those of the input substructure, and the query structure had one less node than the input substructure.

CHAPTER 4

CONCLUSIONS

In this thesis, we show that geometric hashing technique can be used for 3D molecular substructure searching and approximate matching. Minor differences in precision do not affect the match. A match will be found in the case of rotation and differences such as one node (atom) not present in search substructure versus input substructure.

4.1 Future Enhancements

4.1.1 Data Storage

A relational database could be used to store the existing 3D structures. The information would include the type of structure (eg. Chemical, biological, protein). Alternatively the structures could be grouped based on chemical or biological properties. A technique could then be used to find structures "similar" to the query structure. A metric would have to be defined for what is "similar".

4.1.2 User Interface

The application could use a third party tool (eg. ChemDraw) for input of the data structures and the query structures. The tool would translate the structures into the necessary format for use by the program (eg. MOLfile format). The matching structure(s) would then appear graphically within the tool.

APPENDIX

PROGRAM LISTINGS

This appendix contains the programs used to implement the algorithms presented in this thesis.

```
/*****************************************************************/
/* Program: subhash.c                                            */
/* Author: Jennifer L. Cerequas                                  */
/*                                                               */
/* This program reads the existing substructures and stores      */
/* them in hash files on disk.                                   */
/*****************************************************************/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "thesis.h"

main()
{
int status,i,j,k,l,m,n,t,s,st;
int BOUND;
int savnode;
double PB1[3], PB2[3], PB3[3], N[100][3];
double M[3][3],MI[3][3],MIB[3][3];
float mf;
double x[3],y[3],z[3];
struct nl {
 double length;
 int fnode;
 int snode;
 struct nl *lptr;
} ;
struct nl *lenptr, *savptr, *ptr1, *ptr2, *sav3ptr;
long int l1,l2,l3;
long int prime = 1009;
long int nrow = 62;
int dn,sn,nsnode;
```

```
int nc[3];
FILE *hbuck;
char hfname[13] = "hb      .dat";
char ifname[81];

nsnode = 0;

/* This program assumes the input file is in Ctab format;
with the following exceptions:
each input file contains one substructure;
the first line contains the datagraph number (1 to 9999) and the
substructure number within the datagraph(0-9999); */


do
{
printf("Enter filename containing datagraph/substructure global coordinates:\n");
gets(ifname);


hbuck = fopen(ifname,"r");

}
while (hbuck == NULL);

fscanf(hbuck,"%4d",&dn);        /* read datagraph number */
fscanf(hbuck," %4d",&sn);       /* read substructure number */
fscanf(hbuck,"%3d",&nsnode);    /* read number of atoms in substructure */
for (m=0; m<nsnode;m++)
{
      for (n=0; n<3;n++)    /* read x,y,z global coordinates for each*/
         {                  /* atom in substructure */
          if(fscanf(hbuck," %10f",&mf)==EOF) goto rend;
         N[m][n] = mf;
         }
}
rend:;

fclose(hbuck);



        /*once have read in an entire substructure with global coordinates...*/
PB1[0] = N[0][0];  /* coordinate x of lowest numbered node in substructure*/
PB1[1] = N[0][1];  /* coordinate Y of lowest numbered node in substructure*/
```

```
PB1[2] = N[0][2];  /* coordinate Z of lowest numbered node in substructure*/
PB2[0] = N[0][0] +1; /* coordinate x of lowest numbered node in substructure*/
PB2[1] = N[0][1];  /* coordinate Y of lowest numbered node in substructure*/
PB2[2] = N[0][2];  /* coordinate Z of lowest numbered node in substructure*/
PB3[0] = N[0][0];  /* coordinate x of lowest numbered node in substructure*/
PB3[1] = N[0][1] +1; /* coordinate Y of lowest numbered node in substructure*/
PB3[2] = N[0][2];  /* coordinate Z of lowest numbered node in substructure*/

        for (k=1; k<=nsnode-2; k++)
                {
                for (j=k+1; j<=nsnode -1; j++)
                 {
                 for (i=j+1; i <=nsnode; i++)
                        {
                        nc[0] = k-1; /*N array index*/
                        nc[1] = j-1;
                        nc[2] = i-1;



                        lenptr = (struct nl *)
                                        malloc (sizeof(struct nl) );
                        savptr = lenptr;
                        for (l=1;l<3;l++)
                        { lenptr->lptr = (struct nl *)
                                        malloc (sizeof(struct nl) );
                          lenptr = lenptr->lptr;
                        }
                        lenptr->lptr = savptr;

                        lenptr= savptr;

                        lenptr->length = sqrt (pow((N[nc[1]][0] - N[nc[0]][0]),2) +
                                                pow((N[nc[1]][1] - N[nc[0]][1]),2) +
                                                pow((N[nc[1]][2] - N[nc[0]][2]),2));
                        lenptr->fnode = nc[0];
                        lenptr->snode = nc[1];

                        lenptr = lenptr->lptr;

                        lenptr->length = sqrt (pow((N[nc[2]][0] - N[nc[1]][0]),2) +
                                                pow((N[nc[2]][1] - N[nc[1]][1]),2) +
                                                pow((N[nc[2]][2] - N[nc[1]][2]),2));
                        lenptr->fnode = nc[1];
                        lenptr->snode = nc[2];
```

```
                    lenptr = lenptr->lptr;

                    lenptr->length = sqrt (pow((N[nc[2]][0] - N[nc[0]][0]),2) +
                                           pow((N[nc[2]][1] - N[nc[0]][1]),2) +
                                           pow((N[nc[2]][2] - N[nc[0]][2]),2));
                    lenptr->fnode = nc[2];
                    lenptr->snode = nc[0];


                    sav3ptr = lenptr;
                    lenptr = savptr;
/* use bubble sort to sort 3 atoms from shortest to longest connecting
        distance */
                    BOUND = 3;
 bsort:;
                    t=0;
                    ptr1 = lenptr;
                    savptr = lenptr;
                    for (l=1;l<BOUND;l++)
                    {
                     ptr2 = ptr1->lptr;
                     if (ptr1->length > ptr2->length)
                     {
                             ptr1->lptr = ptr2->lptr;
                             ptr2->lptr = ptr1;
                             if (l > 1)
                             { sav3ptr = ptr1;
                             savptr->lptr = ptr2; }
                             else
                             { sav3ptr->lptr = ptr2;
                                     lenptr = ptr2; }
                             savptr = ptr2;
                             t = l;
                     }
                    else
                     {
                     savptr = ptr1;
                     ptr1=ptr2;
                     }
                    }
                    if (t > 0)
                     { BOUND = t;
                             goto bsort; }
```

```
savptr = lenptr;

for (st=1;st<=2;st++)
{
 for (s=1;s<=3;s++)
 {

 if (lenptr->snode != lenptr->lptr->fnode)
         {
          savnode = lenptr->lptr->fnode;
          lenptr->lptr->fnode = lenptr->lptr->snode;
          lenptr->lptr->snode = savnode;
         }
 lenptr=lenptr->lptr;
 }
}

lenptr = savptr;
nc[0] = lenptr->fnode;
nc[1] = lenptr->snode;
lenptr = lenptr->lptr;
 nc[2] = lenptr->snode;

for (s=1; s<=3;s++)
{
lenptr = savptr->lptr;
free (savptr);
savptr = lenptr;
}



x[0] = N[nc[0]][0];
x[1] = N[nc[1]][0];
x[2] = N[nc[2]][0];
y[0] = N[nc[0]][1];
y[1] = N[nc[1]][1];
y[2] = N[nc[2]][1];
z[0] = N[nc[0]][2];
z[1] = N[nc[1]][2];
z[2] = N[nc[2]][2];

status = hash_address(x,y,z,prime,nrow, &l1,&l2,&l3);

status = calc_vector(N[nc[0]],PB1,MIB[0]);
```

```
                    status = calc_vector(N[nc[0]],PB2,MIB[1]);
                    status = calc_vector(N[nc[0]],PB3,MIB[2]);

                    status = calc_vector(N[nc[0]],N[nc[1]],M[0]);
                    status = calc_vector(N[nc[0]],N[nc[2]],M[1]);
                    status = crossprod_vector(M[0],M[1],M[2]);

                    status = inverse_matrix(M,MI);

                    status = crossprod_matrix(MIB, MI, M);

                    sprintf(&hfname[2],"%2.2d",l1);
                    sprintf(&hfname[4],"%2.2d",l2);
                    sprintf(&hfname[6],"%2.2d",l3);
                    strcpy(&hfname[8],".dat");

                    hbuck = fopen(hfname,"a");
                    for (m=0; m<3;m++)
                     {
                      fprintf(hbuck,"%4d %4d",dn,sn);
                      for (n=0; n<3;n++)
                            fprintf(hbuck," %10.4f",M[m][n]);
                      fprintf(hbuck,"\n");
                      }


                    fclose(hbuck);
                    }
                  }
                 }

printf("done...\n");
}

int inverse_matrix(A, AI)
double A[3][3],AI[3][3];
{
double AP[2][2];
double D2,D3; /*Determinate*/
double none;
int i,j,k,l,row,col;

D3=(A[0][0]*A[1][1]*A[2][2]) + (A[0][1]*A[1][2]*A[2][0]) +
  (A[0][2]*A[1][0]*A[2][1]) -
```

```
(A[0][2]*A[1][1]*A[2][0]) - (A[0][0]*A[1][2]*A[2][1]) -
(A[0][1]*A[1][0]*A[2][2]);


for(i=0;i<3;i++)
        for (j=0;j<3;j++)
        {
         row=0; col=0;
         for (k=0;k<3;k++)  /*find AP matrix: A without ith row and jth column*/
          for (l=0;l<3;l++)
                if (k != i && l != j)
                 {
                  AP[row][col] = A[k][l];

                  col++;
                  if (col == 2)
                          {row++;
                           col = 0;
                          }
                 }


         D2 = AP[0][0]*AP[1][1] - AP[0][1]*AP[1][0];


         none = pow (-1,(i+j+2));

         AI[j][i]=(none*D2)/D3;
        }


return(0);
}


int crossprod_matrix(A, B, C)
double A[3][3],B[3][3],C[3][3];
{
int i,j,k;

for (i=0;i<3;i++)
 for (k=0;k<3;k++)
  {
  C[i][k] = 0.0;
  for (j=0;j<3;j++)
```

```
            C[i][k]+=A[i][j]*B[j][k];
    }

return(0);
}

int crossprod_vector(A, B, C)
double A[3], B[3], C[3];
{
C[0] = A[1]*B[2] - A[2]*B[1];
C[1] = A[2]*B[0] - A[0]*B[2];
C[2] = A[0]*B[1] - A[1]*B[0];

return(0);
}

int calc_vector(A, B, C)
double A[3], B[3], C[3];
{
C[0] = B[0] - A[0];
C[1] = B[1] - A[1];
C[2] = B[2] - A[2];

return(0);
}


int hash_address (x,y,z,prime,nrow,l1,l2,l3)
double x[3],y[3],z[3];
long int prime,nrow,*l1,*l2,*l3;
{
double reall;


reall = (pow((x[0]-x[1]),2) + pow((y[0]-y[1]),2) + pow((z[0]-z[1]),2))
                              * 10000.0;
*l1 = reall;
*l1 = *l1 % prime % nrow;

reall = (pow((x[0]-x[2]),2) + pow((y[0]-y[2]),2) + pow((z[0]-z[2]),2))
                              * 10000.0;
*l2 = reall;
*l2 = *l2 % prime % nrow;
```

```
reall = (pow((x[2]-x[1]),2) + pow((y[2]-y[1]),2) + pow((z[2]-z[1]),2))
                        * 10000.0;
*l3 = reall;
*l3 = *l3 % prime % nrow;

return(0);
}
```

```
/*****************************************************************/
/* Program: subsearch.c                                          */
/* Author: Jennifer L. Cerequas                                  */
/*                                                               */
/* This program reads the query substructure and searches the    */
/* hash files for matching substructures.                        */
/*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "thesis.h"

int matrix_cmp (double [3][3],double [3][3]);
int matrix_cpy (double [3][3],double [3][3]);



main()
{
int status,i,j,k,l,m,n,t,s,st;
int BOUND;
int savnode;
double N[100][3];
double M[3][3],MI[3][3],MIB[3][3];
float mf;
double x[3],y[3],z[3];
struct nl {
 double length;
 int fnode;
 int snode;
 struct nl *lptr;
} ;
struct nl *lenptr, *savptr, *ptr1, *ptr2, *sav3ptr;


struct sfqs {
 int counter;
 double sfq [3][3];
 struct sfqs *lptr;
};
struct sfqs *sptr, *savsptr;

struct ds_head {
```

```
 int dnum;
 int snum;
 int nsnodes;
 struct ds_head *dptr;
 struct sfqs *sptr;
 };
struct ds_head *dsptr, *savdptr, *dshead_ptr;


long int l1,l2,l3;
long int prime = 1009;
long int nrow = 62;
int dn,sn,nsnode;
int nhit;
int counter,nmatch;
int nc[3];
FILE *hbuck;
char hfname[13] = "hb      .dat";
char ifname[81];

nsnode = 0;
nhit = 0;

do
{
printf("Enter filename containing Query substructure global coordinates:\n");
gets(ifname);

hbuck = fopen(ifname,"r");

}
while (hbuck == NULL);

  while(fscanf(hbuck,"%4d",&m)!= EOF)  /* read x,y,z global coordinates*/
  {                                    /* of atoms in query substructure*/
                                for (n=0; n<3;n++)
                                {
                                        fscanf(hbuck," %10f",&mf);
                                        N[m][n] = mf;
                                }
                                nsnode++;
  }

                               fclose(hbuck);
```

```
printf("starting loop\n");


dshead_ptr = (struct ds_head *) 0;
for (k=1; k<=nsnode-2; k++)
        {
        for (j=k+1; j<=nsnode -1; j++)
         {
         for (i=j+1; i <=nsnode; i++)
                {
                nc[0] = k-1; /*N array index*/
                nc[1] = j-1;
                nc[2] = i-1;



                lenptr = (struct nl *)
                                        malloc (sizeof(struct nl) );
                savptr = lenptr;
                for (l=1;l<3;l++)
                {
                 lenptr->lptr = (struct nl *)
                                        malloc (sizeof(struct nl) );
                 lenptr = lenptr->lptr;
                }
                                        lenptr->lptr = savptr;

                lenptr= savptr;

                lenptr->length = sqrt (pow((N[nc[1]][0] - N[nc[0]][0]),2) +
                                        pow((N[nc[1]][1] - N[nc[0]][1]),2) +
                                        pow((N[nc[1]][2] - N[nc[0]][2]),2));
                lenptr->fnode = nc[0];
                lenptr->snode = nc[1];
                lenptr = lenptr->lptr;

                lenptr->length = sqrt (pow((N[nc[2]][0] - N[nc[1]][0]),2) +
                                        pow((N[nc[2]][1] - N[nc[1]][1]),2) +
                                        pow((N[nc[2]][2] - N[nc[1]][2]),2));
                lenptr->fnode = nc[1];
                lenptr->snode = nc[2];
                lenptr = lenptr->lptr;

                lenptr->length = sqrt (pow((N[nc[2]][0] - N[nc[0]][0]),2) +
                                        pow((N[nc[2]][1] - N[nc[0]][1]),2) +
```

```
                                        pow((N[nc[2]][2] - N[nc[0]][2]),2));
                    lenptr->fnode = nc[2];
                    lenptr->snode = nc[0];


                     sav3ptr = lenptr;
                     lenptr = savptr;


/* use bubble sort to sort 3 nodes from shortest to longest connecting
       distance */
                    BOUND = 3;
bsort:;
                    t=0;
                    ptr1 = lenptr;
                    savptr = lenptr;
                    for (l=1;l<BOUND;l++)
                    {
                     ptr2 = ptr1->lptr;
                     if (ptr1->length > ptr2->length)
                     {
                            ptr1->lptr = ptr2->lptr;
                            ptr2->lptr = ptr1;
                            if (l > 1)
                            {sav3ptr = ptr1;
                             savptr->lptr = ptr2; }
                            else
                            { sav3ptr->lptr = ptr2;
                             lenptr = ptr2;}
                            savptr = ptr2;
                            t = l;
                     }
                    else
                     {
                      savptr = ptr1;
                      ptr1=ptr2;
                     }
                    }
                    if (t > 0)
                     { BOUND = t;
                            goto bsort; }

                    savptr = lenptr;
```

```
for (st=1;st<=2;st++)
{
 for (s=1;s<=3;s++)
 {
 if (lenptr->snode != lenptr->lptr->fnode)
          {
           savnode = lenptr->lptr->fnode;
           lenptr->lptr->fnode = lenptr->lptr->snode;
           lenptr->lptr->snode = savnode;

          }
 lenptr=lenptr->lptr;
 }
}
/*       while (!sorted);*/

lenptr=savptr;

nc[0] = lenptr->fnode;
nc[1] = lenptr->snode;
lenptr = lenptr->lptr;
nc[2] = lenptr->snode;


for (s=1;s<=3;s++)
{
lenptr = savptr->lptr;
free (savptr);
savptr = lenptr;
}


x[0] = N[nc[0]][0];
x[1] = N[nc[1]][0];
x[2] = N[nc[2]][0];
y[0] = N[nc[0]][1];
y[1] = N[nc[1]][1];
y[2] = N[nc[2]][1];
z[0] = N[nc[0]][2];
z[1] = N[nc[1]][2];
z[2] = N[nc[2]][2];
```

```
status = hash_address(x,y,z,prime,nrow, &l1,&l2,&l3);

sprintf(&hfname[2],"%2.2d",l1);
sprintf(&hfname[4],"%2.2d",l2);
sprintf(&hfname[6],"%2.2d",l3);
strcpy(&hfname[8],".dat");
hbuck = fopen(hfname,"r");


if (hbuck == NULL) continue;
nhit++;
do
{
for (m=0; m<3;m++)
 {
  if(fscanf(hbuck,"%4d %4d",&dn,&sn)== EOF) goto efile;
  for (n=0; n<3;n++)
          { fscanf(hbuck," %10f",&mf);
           MIB[m][n] = mf;}

 }


status = calc_vector(N[nc[0]],N[nc[1]],M[0]);
status = calc_vector(N[nc[0]],N[nc[2]],M[1]);
status = crossprod_vector(M[0],M[1],M[2]);

status = crossprod_matrix(MIB, M, MI);

for (m=0; m<3;m++)  /* fill matrix with Pu*/
 {
          M[m][0] = x[0];
          M[m][1] = y[0];
          M[m][2] = z[0];
 }

status = add_matrix(MI, M, M);

dsptr = dshead_ptr;
while (dsptr != (struct ds_head *) 0)
 {
 savdptr = dsptr;
 if (dsptr->dnum == dn && dsptr->snum == sn)
  {
          sptr = dsptr->sptr;
```

```
        while (sptr != (struct sfqs *) 0)
         {
                savsptr = sptr;
                if (matrix_cmp(M, sptr->sfq))  /* found a
                                        matching SFQ */
                 {

                 sptr->counter++;
                 break;
                 }
                else
                 sptr = sptr->lptr;
                }
        if (sptr == (struct sfqs *) 0) /* setup a new SFQ */
                 {
                 sptr = (struct sfqs *)
                                malloc (sizeof(struct sfqs) );
                 savsptr->lptr = sptr;
                 sptr->lptr = (struct sfqs *) 0;
                 status = matrix_cpy(sptr->sfq,M);
                 sptr->counter=1;
                 }
       break;
 }
else
 dsptr = dsptr->dptr;
}
if (dsptr == (struct ds_head *) 0) /* setup a new header for
                                        graph D, substructure S */
{
                dsptr = (struct ds_head *)
                malloc (sizeof(struct ds_head) );
                 dsptr->dptr = (struct ds_head *) 0;
                 dsptr->dnum = dn;
                 dsptr->snum = sn;

                 sptr = (struct sfqs *)        /* setup first SFQ */
                                malloc (sizeof(struct sfqs) );
                dsptr->sptr = sptr;
                sptr->lptr = (struct sfqs *) 0;
                status = matrix_cpy(sptr->sfq,M);
                sptr->counter=1;

                if (dshead_ptr == (struct ds_head *) 0)
                                dshead_ptr = dsptr;
```

```
                                        else
                                                savdptr->dptr = dsptr;


                                        }

                        }
                        while(1);
                        efile:;
                        fclose(hbuck);
                        }
                }
                }

nmatch = (nsnode - 1) * (nsnode - 2) * (nsnode -3) /6;

printf("Number of hits = %d\n", nhit);
while (dshead_ptr != (struct ds_head *) 0)
{
 counter = 0;
 sptr = dshead_ptr->sptr;
 while (sptr != (struct sfqs *) 0)
        { if (sptr->counter > counter ) counter = sptr->counter;
          sptr = sptr->lptr;
        }
 if (counter > nmatch)
 {      printf("Counter = %d\n", counter);
        printf("Query substructure matches Data Graph %d, Substructure %d\n",
                        dshead_ptr->dnum, dshead_ptr->snum); }
 dshead_ptr = dshead_ptr->dptr;
}

printf("done...\n");
}

int inverse_matrix(A, AI)
double A[3][3],AI[3][3];
{
double AP[2][2];
double D2,D3; /*Determinate*/
double none;
int i,j,k,l,row,col;

D3=(A[0][0]*A[1][1]*A[2][2]) + (A[0][1]*A[1][2]*A[2][0]) +
  (A[0][2]*A[1][0]*A[2][1]) -
```

```
(A[0][2]*A[1][1]*A[2][0]) - (A[0][0]*A[1][2]*A[2][1]) -
(A[0][1]*A[1][0]*A[2][2]);

for(i=0;i<3;i++)
        for (j=0;j<3;j++)
        {
         row=0; col=0;
         for (k=0;k<3;k++) /*find AP matrix: A without ith row and jth column*/
          for (l=0;l<3;l++)
                if (k != i && l != j)
                 {
                  AP[row][col] = A[k][l];
                  col++;
                  if (col == 2)
                        {row++;
                         col = 0;
                        }
                 }


         D2 = AP[0][0]*AP[1][1] - AP[0][1]*AP[1][0];

         none = pow (-1,(i+j+2));
         AI[j][i]=(none*D2)/D3;
        }


return(0);
}


int crossprod_matrix(A, B, C)
double A[3][3],B[3][3],C[3][3];
{
int i,j,k;

for (i=0;i<3;i++)
 for (k=0;k<3;k++)
  {
  C[i][k] = 0.0;
  for (j=0;j<3;j++)
        C[i][k]+=A[i][j]*B[j][k];
  }
```

```
return(0);
}

int add_matrix(A, B, C)
double A[3][3],B[3][3],C[3][3];
{
int i,k;

for (i=0;i<3;i++)
 {
 for (k=0;k<3;k++)
        C[i][k] = A[i][k] + B[i][k];
 }

return(0);
}

int matrix_cmp(A, B)
double A[3][3],B[3][3];
{
double diff;
int i,k;

for (i=0;i<3;i++)
 {
 for (k=0;k<3;k++)
  { diff = A[i][k] - B[i][k];
        if (diff < 0.0) diff = -diff;
        if(diff >= 0.00020) return(0); }
 }

return(1);
}

int matrix_cpy(A, B)
double A[3][3],B[3][3];
{
int i,k;

for (i=0;i<3;i++)
 {
 for (k=0;k<3;k++)
        A[i][k] = B[i][k];
 }
```

```
return(0);
}

int crossprod_vector(A, B, C)
double A[3], B[3], C[3];
{
C[0] = A[1]*B[2] - A[2]*B[1];
C[1] = A[2]*B[0] - A[0]*B[2];
C[2] = A[0]*B[1] - A[1]*B[0];

return(0);
}

int calc_vector(A, B, C)
double A[3], B[3], C[3];
{
C[0] = B[0] - A[0];
C[1] = B[1] - A[1];
C[2] = B[2] - A[2];

return(0);
}


int hash_address (x,y,z,prime,nrow,l1,l2,l3)
double x[3],y[3],z[3];
long int prime,nrow,*l1,*l2,*l3;
{
double reall;


reall = (pow((x[0]-x[1]),2) + pow((y[0]-y[1]),2) + pow((z[0]-z[1]),2))
                        * 10000.0;
*l1 = reall;
*l1 = *l1 % prime % nrow;

reall = (pow((x[0]-x[2]),2) + pow((y[0]-y[2]),2) + pow((z[0]-z[2]),2))
                        * 10000.0;
*l2 = reall;
*l2 = *l2 % prime % nrow;


reall = (pow((x[2]-x[1]),2) + pow((y[2]-y[1]),2) + pow((z[2]-z[1]),2))
                        * 10000.0;
*l3 = reall;
```

```
*l3 = *l3 % prime % nrow;

return(0);
}
```

```
/*   Thesis.h */

int calc_vector (double [3], double[3], double[3]);
int inverse_matrix (double [3][3],double [3][3]);
int crossprod_matrix (double [3][3],double [3][3],double [3][3]);
int crossprod_vector (double [3], double [3], double [3]);
int add_matrix (double [3][3],double [3][3],double [3][3]);
int hash_address (double [3],double [3],double [3],
          long int,long int,long int *,long int *,long int *);
```

REFERENCES

Dalby, Arthur and James G. Nourse, "Description of Several Chemical Structure File Formats Used by Computer Programs at Molecular Design Limited", J. Chem. Inf. Comput. Sci, 32(1992):244-255.

Charifson, Paul S. and Andrew R. Leach, "The Generation and Use of Large 3D Databases in Drug Discovery" [On-line]. Available HTTP: www.awod.com/netsci/Science/Cheminform/feature03.htlm, January 1998.

Kazmierczak, Marcus, "Java Script Linear Algebra Calculator" [On-line]. Available HTTP: www.mkaz.com/math/js_calc.html, January 1998.

Knuth, Donald E., The Art of Computer Programming, Vol. 1, Second Edition, Addision-Wesley, Reading, MA, 1973, pp. 51,52.

Kochan, Stephen, Programming in C, Hayden Books, Indianapolis, Indiana, 1988.

Lamdan, Y. and H. Wolfson, "Geometric Hashing: A General and Efficient Model-Based Recognition Scheme", Proc. Inter. Conf. on Computer Vision, (1998):237-249.

Wang, Jason and G.W. Chirn and T.G. Marr, "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results", Proc. ACM SIGMOD Conf, (1994):115-125.

Wang, Xiong and Jason T.L. Wang, "Approximate Substructure Search in a Database of 3D Graphs", Proceedings of the Third International Conference of Information Sciences, Research Triangle Park, North Carolina, March 1997, pp. 12-15 .

_____. Handbook of Mathematical, Scientific, and Engineering Formulas, Research and Education Association, New York, NY, 1984, pp. 13-17, 291-294.