```
#ifndef DISTANCE__H
#define DISTANCE__H

/////////////////////////////////////////////////////////////////////////////
// CDistance

class CDistance : public CObject
{
// Construction
public:
      CDistance();       // standard constructor
      ~CDistance();

// Attributes
protected:
      double m_Min;
      double m_Max;
      double m_Norm;
      double m_Sum;

      double* m_Matrix;
      const double* m_Long;
      const double* m_Lat;
      long m_nCases;
      long m_nMatSize;
      BOOL m_bNormalized;
      BOOL m_bEuclidian;
      BOOL m_bManhattan;

// Functions
public:
      void Empty();
      void Compute(long n, const double* m_Long, const double* m_Lat, CString Method = "Manhattan Metric");

      double Normalize(double);
      double UnNormalize(double);
      BOOL Normalize();
      BOOL UnNormalize();
      BOOL IsNormalized();

      double Get(long n1, long n2);
      double GetNorm();
      double GetMin();
```

© PNR and Associates, All rights reserved

```
	double GetMax();
	double GetSum();

	double FindNext(double value, BOOL higher = TRUE);
	double LatFromDist(double Lat, double Distance);
	double LongFromDist(double x1, double y1, double Distance);
	unsigned long GetLTIndex(long n1, long n2);

	double ComputeDistance(double x1, double y1, double x2, double y2);
};

#endif

// Distance.cpp : implementation file
//

#include "stdafx.h"
#include "Distance.h"

#include <stdio.h>
#include <math.h>

void PeekMsg()
{
	MSG Message;
	if (::PeekMessage(&Message, NULL, 0, 0, PM_REMOVE))
	{
		::TranslateMessage(&Message);
		::DispatchMessage(&Message);
	}
}

CDistance::CDistance()
{
	m_nCases = 0;
	m_Norm = 0;
	m_Min = 0;
	m_Max = 0;
	m_Sum = 0;

	m_Long = NULL;
	m_Lat = NULL;
	m_Matrix = NULL;
```

© PNR and Associates, All rights reserved

```
	m_bEuclidian = FALSE;
	m_bManhattan = TRUE;
	m_bNormalized = FALSE;
}

CDistance::~CDistance()
{
	if (m_Matrix)
		if (VirtualFree(m_Matrix, 0, MEM_RELEASE))
			m_Matrix = NULL;
}

void CDistance::Empty()
{
	// Initialize
	m_nCases = 0;
	m_Norm = 0;
	m_Min = 0;
	m_Max = 0;
	m_Sum = 0;

	m_Long = NULL;
	m_Lat = NULL;
	m_bEuclidian = FALSE;
	m_bManhattan = TRUE;
	m_bNormalized = FALSE;

	// Deallocate memory for the distance matrix
	if (m_Matrix)
		if (VirtualFree(m_Matrix, 0, MEM_RELEASE))
			m_Matrix = NULL;

	return;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////
//
//  Computational functions
//
//////////////////////////////////////////////////////////////////////////////////////////////////////

void CDistance::Compute(long n, const double* Long, const double* Lat, CString Method /* Manhattan Metric
*/)
{
```

© PNR and Associates, All rights reserved

```
Empty();

// Initialize
m_Long = Long;
m_Lat = Lat;
m_nCases = n;
m_nMatSize = ((m_nCases + 1) * m_nCases) / 2;
if (n <= 1)
      return;

if (Method == "Euclidian")
{
      m_bEuclidian = TRUE;
      m_bManhattan = FALSE;
}
else if (Method == "Manhattan Metric")
{
      m_bEuclidian = FALSE;
      m_bManhattan = TRUE;
}

// Allocate distance matrix based on free memory
m_Matrix = (double*)VirtualAlloc(NULL, m_nMatSize * sizeof(double), MEM_COMMIT, PAGE_READWRITE);

// Set distance matrix
if (m_Matrix)
{
      // Fill lower triangle of matrix
      unsigned long index;
      for (long i = 0; i < m_nCases; i++)
            for (long j = 0; j <= i; j++)
            {
                  // Calculate distances
                  index = GetLTIndex(i,j);
                  if (m_bEuclidian)
                        m_Matrix[index] = ComputeDistance(m_Long[i], m_Lat[i], m_Long[j], m_Lat[j]);
                  else if (m_bManhattan)
                  {
                        m_Matrix[index] = ComputeDistance(m_Long[i], m_Lat[i], m_Long[j], m_Lat[i]);
// x1, y1 to x2, y1
                        m_Matrix[index] += ComputeDistance(m_Long[j], m_Lat[j], m_Long[j], m_Lat[i]);
// x2, y2 to x2, y1
                  }
```

© PNR and Associates, All rights reserved

```
                        // Set attributes
                        if (m_Matrix[index] != 0.0)
                        {
                              m_Sum += m_Matrix[index];
                              if (m_Matrix[index] > m_Max)
                                    m_Max = m_Matrix[index];
                              if (m_Min == 0.0 || m_Matrix[index] < m_Min)
                                    m_Min = m_Matrix[index];
                        }
                  }
      }
      else
      {
            AfxMessageBox("\r\nWARNING:  Unable to allocate enough virtual memory for the distance
matrix.\r\n");

            // Fill lower triangle of matrix
            double dist;
            for (long i = 1; i < m_nCases; i++)
                  for (long j = 0; j < i; j++)
                  {
                        // Calculate distances
                        if (m_bEuclidian)
                              dist = ComputeDistance(m_Long[i], m_Lat[i], m_Long[j], m_Lat[j]);
                        else if (m_bManhattan)
                        {
                              dist = ComputeDistance(m_Long[i], m_Lat[i], m_Long[j], m_Lat[i]);   // x1, y1
to x2, y1
                              dist += ComputeDistance(m_Long[j], m_Lat[j], m_Long[j], m_Lat[i]);  // x2, y2
to x2, y1
                        }

                        // Set attributes
                        if (dist != 0.0)
                        {
                              m_Sum += dist;
                              if (dist > m_Max)
                                    m_Max = dist;
                              if (m_Min == 0.0 || dist < m_Min)
                                    m_Min = dist;
                        }
                  }
      }
```

© PNR and Associates, All rights reserved

```
	m_Norm = m_Sum / m_nCases;
	return;
}

double CDistance::ComputeDistance(double x1, double y1, double x2, double y2)
{
	// Compute linear euclidian distance
	double distance = 0.0;
	double radius = 3956.0; // Approximate radius of the earth in miles
	double pi = acos(-1.0);
	double radians = pi / 180.0;

	x1 *= radians;
	x2 *= radians;
	y1 *= radians;
	y2 *= radians;

	if (x1 != x2 || y1 != y2)
	{
		distance = pow(sin((y2 - y1) / 2.0), 2.0) + cos(y1) * cos(y2) * pow(sin((x2 - x1) / 2.0), 2.0);
		distance = (2 * asin(min(1, pow(distance, 0.5)))) * radius;
	}

	return distance;
}

unsigned long CDistance::GetLTIndex(long n1, long n2)
{
	if (n1 < n2)
		return ((n2 + 1) * n2 / 2 + (n1 + 1)) - 1;
	else
		return ((n1 + 1) * n1 / 2 + (n2 + 1)) - 1;
}

//////////////////////////////////////////////////////////////////////////////////////////////
//
//  Normalization functions
//
//////////////////////////////////////////////////////////////////////////////////////////////

BOOL CDistance::IsNormalized()
{
	return m_bNormalized;
}
```

© PNR and Associates, All rights reserved

```
double CDistance::Normalize(double num)
{
	return num / m_Norm;
}

double CDistance::UnNormalize(double num)
{
	return num * m_Norm;
}

BOOL CDistance::Normalize()
{
	if (!m_bNormalized)
	{
		for (long i = 0; i < m_nMatSize; i++)
			m_Matrix[i] /= m_Norm;

		m_bNormalized = TRUE;
		return TRUE;
	}
	else
		return FALSE;
}

BOOL CDistance::UnNormalize()
{
	if (m_bNormalized)
	{
		for (long i = 0; i < m_nMatSize; i++)
			m_Matrix[i] *= m_Norm;

		m_bNormalized = TRUE;
		return TRUE;
	}
	else
		return FALSE;
}

double CDistance::LatFromDist(double Lat, double Distance)
{
	double Radius = 3956.0;
	double pi = acos(-1.0);
	double Radians = pi / 180.0;
```

© PNR and Associates, All rights reserved

```
	return (Distance / Radius + Lat * Radians) / Radians;
}

double CDistance::LongFromDist(double x1, double y1, double Distance)
{
	double Radius = 3956.0;
	double pi = acos(-1.0);
	double Radians = pi / 180.0;

	x1 *= Radians;
	y1 *= Radians;

	return (x1 + 2 * asin(pow( pow(sin(Distance/(2*Radius)), 2.) / (cos(y1)*cos(y1)), .5))) / Radians;
}

//////////////////////////////////////////////////////////////////////////////////////////////
//
//  Attribute functions
//
//////////////////////////////////////////////////////////////////////////////////////////////

double CDistance::FindNext(double value, BOOL higher /*TRUE*/)
{
	double temp, next = -1.0;
	for (long i = 1; i < m_nCases; i++)
		for (long j = 0; j < i; j++)
		{
			temp = Get(i, j);
			if ((higher == TRUE && temp > value && (temp < next || next == -1.0)) ||
			   (higher == FALSE && temp < value && (temp > next || next == -1.0)))
				next = temp;
		}

	return next;
}

double CDistance::Get(long n1, long n2)
{
	if (m_Matrix)
		return m_Matrix[GetLTIndex(n1,n2)];
	else
	{
		double dist;
```

© PNR and Associates, All rights reserved

```
            if (m_bEuclidian)
                  dist = ComputeDistance(m_Long[n1], m_Lat[n1], m_Long[n2], m_Lat[n2]);
            else if (m_bManhattan)
            {
                  dist = ComputeDistance(m_Long[n1], m_Lat[n1], m_Long[n2], m_Lat[n1]);   // x1, y1 to x2,
y1
                  dist += ComputeDistance(m_Long[n2], m_Lat[n2], m_Long[n2], m_Lat[n1]);  // x2, y2 to x2,
y1
            }

            return dist;
      }
}

double CDistance::GetNorm()
{
      return m_Norm;
}

double CDistance::GetMin()
{
      if (m_bNormalized)
            return m_Min / m_Norm;
      else
            return m_Min;
}

double CDistance::GetMax()
{
      if (m_bNormalized)
            return m_Max / m_Norm;
      else
            return m_Max;
}

double CDistance::GetSum()
{
      if (m_bNormalized)
            return m_Sum / m_Norm;
      else
            return m_Sum;
}

#ifndef FINALIZE__H
```

© PNR and Associates, All rights reserved

```
#define FINALIZE__H

#include "Global.h"
#include "GrdClust.h"

class CFinalize : public CObject
{
public:
      CFinalize();
      DECLARE_DYNCREATE(CFinalize)

      ~CFinalize();

public:
      CClustComp* m_pFinal;
      CDblArray *m_Vertices;

//    CDblArray m_HWRatio;
      CDblArray m_Height;
      CDblArray m_Width;

protected:
      CDWordArray* m_ClusMembers;
      CDWordArray* m_ClusPerimMembers;

      // Size of number of points
      CDWordArray m_Perimeter;
      CDWordArray m_ClusPath;
      CStringArray m_ClusPathLabel;
      CByteArray m_OutlierFlag;

      // Size of number of clusters
      CDblArray m_MinLat;
      CDblArray m_MinLong;
      CDblArray m_MaxLat;
      CDblArray m_MaxLong;

      CDblArray m_LongCentroids;
      CDblArray m_LatCentroids;
      CDblArray m_ServeLongCentroids;
      CDblArray m_ServeLatCentroids;

      CDblArray m_ClusArea;
      CDblArray m_ClusDist;
```

© PNR and Associates, All rights reserved

```
        CDblArray m_ClusMaxPtDist;
        CStringArray m_ClusLabels;

        CDblArray m_GrpDist;
        CDblArray m_GrpMaxPtDist;
        CDWordArray m_GrpPath;
        CStringArray m_GrpPathLabel;

        CDWordArray m_ActualClusters;
        CDWordArray m_OutlierClusters;

public:
        void Run(CClustComp*, SPECS* Specs = NULL);
        BOOL IsComputed();

        long GetNClus();
        long GetNPts();

        long GetPtPerimiter(long pt);
        long GetPtPath(long pt);
        CString GetPtPathLabel(long pt);

        BYTE GetClusOutlierFlag(long clus);
        double GetClusMinLat(long clus);
        double GetClusMinLong(long clus);
        double GetClusMaxLat(long clus);
        double GetClusMaxLong(long clus);

        double GetClusLongCentroid(long clus);
        double GetClusLatCentroid(long clus);
        double GetClusServeLongCentroid(long clus);
        double GetClusServeLatCentroid(long clus);

        double GetClusArea(long clus);
        double GetClusDist(long clus);
        double GetClusMaxPtDist(long clus);
        CString GetClusLabel(long clus);

        double GetGrpDist(long clus);
        double GetGrpMaxPtDist(long clus);
        double GetGrpPath(long clus);
        CString GetGrpPathLabel(long clus);

        void GetClusMembers(long clus, CDWordArray& members);
```

© PNR and Associates, All rights reserved

```
      void GetClusPerimMembers(long clus, CDWordArray& members);

protected:
      void Alloc(void);
      void DeAlloc(void);

      void SetInfo(void);
      void ReduceOversized(CDWordArray &Members, CObArray &NewClusts);
      void ClusterCleanup(SPECS* Specs);
      void AspectRatio();

      void SetLabels(void);

//    void ComputePerimeter(void);
      void ComputeArea(void);
      void ComputeCentroids(CString Method = "OfMaxPtDist");
      void ComputePath(int, CString);
};


#endif


#include "stdafx.h"
#include "Finalize.h"

#include "Perimeter.h"
#include "Area.h"
#include "Path.h"
#include "Distance.h"

#include "string.h"
#include "math.h"


IMPLEMENT_DYNCREATE(CFinalize, CObject)

CFinalize::CFinalize()
{
      m_pFinal = NULL;
      m_ClusMembers = NULL;
      m_ClusPerimMembers = NULL;
//    m_Vertices = NULL;
}
```

© PNR and Associates, All rights reserved

```
CFinalize::~CFinalize()
{
      delete[] m_ClusMembers;
      delete[] m_ClusPerimMembers;
//    delete[] m_Vertices;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// CFinalize:     Allocation/DeAllocation Functions
//
////////////////////////////////////////////////////////////////////////////////////////////////////

void CFinalize::DeAlloc()
{
      // Size of number of points
      m_Perimeter.RemoveAll();
      m_ClusPath.RemoveAll();
      m_ClusPathLabel.RemoveAll();
      m_OutlierFlag.RemoveAll();

      // Size of number of clusters
      m_MinLat.RemoveAll();
      m_MinLong.RemoveAll();
      m_MaxLat.RemoveAll();
      m_MaxLong.RemoveAll();

      m_LongCentroids.RemoveAll();
      m_LatCentroids.RemoveAll();
      m_ServeLongCentroids.RemoveAll();
      m_ServeLatCentroids.RemoveAll();

      m_ClusArea.RemoveAll();
      m_ClusDist.RemoveAll();
      m_ClusMaxPtDist.RemoveAll();
      m_ClusLabels.RemoveAll();

      m_GrpDist.RemoveAll();
      m_GrpMaxPtDist.RemoveAll();
      m_GrpPath.RemoveAll();
      m_GrpPathLabel.RemoveAll();

      m_ActualClusters.RemoveAll();
```

© PNR and Associates, All rights reserved

```
        m_OutlierClusters.RemoveAll();

        m_Height.RemoveAll();
        m_Width.RemoveAll();

        if (m_ClusMembers)
        {
                delete[] m_ClusMembers;
                m_ClusMembers = NULL;
        }

        if (m_ClusPerimMembers)
        {
                delete[] m_ClusPerimMembers;
                m_ClusPerimMembers = NULL;
        }


//      delete m_Vertices;
//      m_Vertices = NULL;
}

void CFinalize::Alloc()
{
        DeAlloc();

        long nClus = GetNClus();
        long nPts = GetNPts();

        // Size of number of points
        m_Perimeter.SetSize(nPts);
        m_ClusPath.SetSize(nPts);
        m_ClusPathLabel.SetSize(nPts);

        // Size of number of clusters
        m_OutlierFlag.SetSize(nClus);
        m_MinLat.SetSize(nClus);
        m_MinLong.SetSize(nClus);
        m_MaxLat.SetSize(nClus);
        m_MaxLong.SetSize(nClus);
        m_LongCentroids.SetSize(nClus);
        m_LatCentroids.SetSize(nClus);
        m_ServeLongCentroids.SetSize(nClus);
        m_ServeLatCentroids.SetSize(nClus);
```

© PNR and Associates, All rights reserved

```
      m_ClusArea.SetSize(nClus);
      m_ClusDist.SetSize(nClus);
      m_ClusMaxPtDist.SetSize(nClus);
      m_ClusLabels.SetSize(nClus);
      m_GrpDist.SetSize(nClus);
      m_GrpMaxPtDist.SetSize(nClus);
      m_GrpPath.SetSize(nClus);
      m_GrpPathLabel.SetSize(nClus);

      m_Height.SetSize(nClus);
      m_Width.SetSize(nClus);

      m_ClusMembers = new CDWordArray[nClus];
      m_ClusPerimMembers = new CDWordArray[nClus];

//    m_Vertices = new CDblArray[nClus];
}

/////////////////////////////////////////////////////////////////////////////////////////////////////
//
// CFinalize:     Get Functions
//
/////////////////////////////////////////////////////////////////////////////////////////////////////

long CFinalize::GetNClus()
{
      return m_pFinal->m_CurClust->m_nClusters;
}

long CFinalize::GetNPts()
{
      return m_pFinal->m_nPoints;
}

long CFinalize::GetPtPerimiter(long pt)
{
      return m_Perimeter[pt];
}

long CFinalize::GetPtPath(long pt)
{
      return m_ClusPath[pt];
}
```

© PNR and Associates, All rights reserved

```
CString CFinalize::GetPtPathLabel(long pt)
{
      return m_ClusPathLabel[pt];
}

BYTE CFinalize::GetClusOutlierFlag(long clus)
{
      return m_OutlierFlag[clus];
}

double CFinalize::GetClusMinLat(long clus)
{
      return m_MinLat[clus];
}

double CFinalize::GetClusMinLong(long clus)
{
      return m_MinLong[clus];
}

double CFinalize::GetClusMaxLat(long clus)
{
      return m_MaxLat[clus];
}

double CFinalize::GetClusMaxLong(long clus)
{
      return m_MaxLong[clus];
}

double CFinalize::GetClusLongCentroid(long clus)
{
      return m_LongCentroids[clus];
}

double CFinalize::GetClusLatCentroid(long clus)
{
      return m_LatCentroids[clus];
}

double CFinalize::GetClusServeLongCentroid(long clus)
{
      return m_ServeLongCentroids[clus];
}
```

© PNR and Associates, All rights reserved

```
double CFinalize::GetClusServeLatCentroid(long clus)
{
      return m_ServeLatCentroids[clus];
}

double CFinalize::GetClusArea(long clus)
{
      return m_ClusArea[clus];
}

double CFinalize::GetClusDist(long clus)
{
      return m_ClusDist[clus];
}

double CFinalize::GetClusMaxPtDist(long clus)
{
      return m_ClusMaxPtDist[clus];
}

CString CFinalize::GetClusLabel(long clus)
{
      return m_ClusLabels[clus];
}

double CFinalize::GetGrpDist(long clus)
{
      return m_GrpDist[clus];
}

double CFinalize::GetGrpMaxPtDist(long clus)
{
      return m_GrpMaxPtDist[clus];
}

double CFinalize::GetGrpPath(long clus)
{
      return m_GrpPath[clus];
}

CString CFinalize::GetGrpPathLabel(long clus)
{
      return m_GrpPathLabel[clus];
```

© PNR and Associates, All rights reserved

```
}

void CFinalize::GetClusMembers(long clus, CDWordArray& members)
{
	long sz = m_ClusMembers[clus].GetSize();
	members.SetSize(sz);
	for (long i = 0; i < sz; i++)
		members[i] = m_ClusMembers[clus].GetAt(i);
	return;
}

void CFinalize::GetClusPerimMembers(long clus, CDWordArray& members)
{
	long sz = m_ClusPerimMembers[clus].GetSize();
	members.SetSize(sz);
	for (long i = 0; i < sz; i++)
		members[i] = m_ClusPerimMembers[clus].GetAt(i);
	return;
}

/////////////////////////////////////////////////////////////////////////////////////////////////
//
// CFinalize:	Set Functions
//
/////////////////////////////////////////////////////////////////////////////////////////////////

void CFinalize::SetInfo()
{
	// Define Cluster Membership
	for (long i=0; i<m_pFinal->m_nPoints; i++){
		m_ClusMembers[m_pFinal->m_CurClust->m_pCluster[i]-1].Add(i);
	}

	BOOL AllOutliers = TRUE;
	for (i = 0; i < m_pFinal->m_CurClust->m_nClusters; i++) {
		if (!m_pFinal->m_CurClust->IsOutlier((long)i)) {
			AllOutliers = FALSE;
			break;
		}
	}

	if (AllOutliers) {
		long pos = 0;
		float CurSize, MaxSize = 0;
```

© PNR and Associates, All rights reserved

```
            for (i = 0; i < m_pFinal->m_CurClust->m_nClusters; i++) {
                  CurSize = m_pFinal->m_CurClust->GetClustLines(m_pFinal->m_CurClust->GetCluster(i));
                  if (CurSize > MaxSize)
                        pos = i;
            }

            m_pFinal->m_CurClust->SetOutlierFlag(FALSE, pos);
      }

      // Assign cases that belong to each cluster, the outlier flags and actual and outlier clusters
      for (i = 0; i < m_pFinal->m_CurClust->m_nClusters; i++) {
            if (m_pFinal->m_CurClust->IsOutlier((long)i)) {
                  m_OutlierFlag[i] = 1;
                  m_OutlierClusters.Add(i);
            } else {
                  m_ActualClusters.Add(i);
            }
      }

      m_ActualClusters.FreeExtra();
      m_OutlierClusters.FreeExtra();

      return;
}

void CFinalize::SetLabels()
{
      char buff[5];
      CString clus;
      long i, j, actSz, outSz;
      CDistance* pDist = new CDistance();

      actSz = m_ActualClusters.GetSize();
      outSz = m_OutlierClusters.GetSize();

      // Label actual clusters
      for (i = 0; i < actSz; i++) {
            m_ClusLabels[m_ActualClusters[i]] = "c000";
            clus = itoa(i + 1, buff, 10);
            m_ClusLabels[m_ActualClusters[i]] = m_ClusLabels[m_ActualClusters[i]].Left(4 -
clus.GetLength());
            m_ClusLabels[m_ActualClusters[i]] += clus;
      }
```

© PNR and Associates, All rights reserved

```
	// Label outlier clusters to closest actual cluster
	if (actSz == 1) {
		for (i = 0; i < outSz; i++) {
			m_ClusLabels[m_OutlierClusters[i]] = m_ClusLabels[m_ActualClusters[0]];
		}
	} else {
		for (i = 0; i < outSz; i++) {
			long index = 0;
			double d = pDist->ComputeDistance(m_LongCentroids[m_ActualClusters[0]],
m_LatCentroids[m_ActualClusters[0]],
								m_LongCentroids[m_OutlierClusters[i]],
m_LatCentroids[m_OutlierClusters[i]]);

			for (j = 1; j < actSz; j++) {
				double check = pDist->ComputeDistance(m_LongCentroids[m_ActualClusters[j]],
m_LatCentroids[m_ActualClusters[j]],
								m_LongCentroids[m_OutlierClusters[i]],
m_LatCentroids[m_OutlierClusters[i]]);
				if (check < d) {
					d = check;
					index = m_ActualClusters[j];
				}
			}

			m_ClusLabels[m_OutlierClusters[i]] = m_ClusLabels[index];
		}
	}

	delete pDist;
	return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// CFinalize:	Computational Functions
//
////////////////////////////////////////////////////////////////////////////////////////////////////

BOOL CFinalize::IsComputed()
{
	BOOL bRet = TRUE;
	if (!m_pFinal)
		bRet = FALSE;
	return bRet;
```

© PNR and Associates, All rights reserved

```
}

void CFinalize::Run(CClustComp* pClus, SPECS* Specs)
{
	m_pFinal = pClus;
	ClusterCleanup(Specs);
	Alloc();

	// Compute perimeter, area and centroids of clusters
	SetInfo();
	AspectRatio();
//	ComputePerimeter();
	ComputeArea();
	ComputeCentroids();
	SetLabels();

	if (Specs)
	{
		if (Specs->bCPath)
			ComputePath(1, Specs->DistMeasure);

		if (Specs->bGPath)
			ComputePath(2, Specs->DistMeasure);
	}
}

void CFinalize::ReduceOversized(CDWordArray &Members, CObArray &NewClusts)
{
	CDistance pDist;
	double MinLat, MaxLat, MinLong, MaxLong;
	MinLat = MaxLat = MinLong = MaxLong = 0;

	for (long i = 0; i < (long)Members.GetSize(); i++)
	{
		long j = Members[i];

		if (m_pFinal->GetLat(j) < MinLat || !MinLat)
			MinLat = m_pFinal->GetLat(j);
		if (m_pFinal->GetLat(j) > MaxLat || !MaxLat)
			MaxLat = m_pFinal->GetLat(j);

		if (m_pFinal->GetLong(j) < MinLong || !MinLong)
			MinLong = m_pFinal->GetLong(j);
		if (m_pFinal->GetLong(j) > MaxLong || !MaxLong)
```

© PNR and Associates, All rights reserved

```
            MaxLong = m_pFinal->GetLong(j);
}

double MidLong = (MinLong+MaxLong) / 2.;
double height = pDist.ComputeDistance(MidLong, MinLat, MidLong, MaxLat);
double width = pDist.ComputeDistance(MinLong, MinLat, MaxLong, MinLat);

CDWordArray *MemSplit[2];
MemSplit[0] = new CDWordArray;
MemSplit[1] = new CDWordArray;

float Lines[2];
Lines[0] = Lines[1] = 0.;

if (height > width) {
      double MidLat = (MinLat+MaxLat) / 2.;

      for (i = 0; i < (long)Members.GetSize(); i++) {
            if (m_pFinal->GetLat(Members[i]) > MidLat) {
                  MemSplit[0]->Add(Members[i]);
                  Lines[0] += m_pFinal->m_pWeight[Members[i]];
            } else {
                  MemSplit[1]->Add(Members[i]);
                  Lines[1] += m_pFinal->m_pWeight[Members[i]];
            }
      }
} else {
      for (i = 0; i < (long)Members.GetSize(); i++) {
            if (m_pFinal->GetLong(Members[i]) > MidLong) {
                  MemSplit[0]->Add(Members[i]);
                  Lines[0] += m_pFinal->m_pWeight[Members[i]];
            } else {
                  MemSplit[1]->Add(Members[i]);
                  Lines[1] += m_pFinal->m_pWeight[Members[i]];
            }
      }
}

for (i = 0; i < 2; i++) {
      if (Lines[i] > m_pFinal->m_NM_Count && height*width > .0026149) {
            ReduceOversized(*MemSplit[i], NewClusts);
            delete MemSplit[i];
      } else {
            if (MemSplit[i]->GetSize())
```

© PNR and Associates, All rights reserved

```
                        NewClusts.Add(MemSplit[i]);
                    else
                        i = i;
                }
        }
}

void CFinalize::ClusterCleanup(SPECS* Specs)
{
        m_pFinal->m_CurClust->ClusConsolidate();

static BOOL SplitLarge=TRUE;
if (SplitLarge) {
        CObArray NewClusts;
        CDWordArray Oversized;
        CDWordArray StartPos;

        CDWordArray* ClusMembers;
        ClusMembers = new CDWordArray[m_pFinal->m_CurClust->m_nClusters];

        for (long i=0; i<m_pFinal->m_nPoints; i++) {
                ClusMembers[m_pFinal->m_CurClust->m_pCluster[i]-1].Add(i);
        }

        for (i=0; i<m_pFinal->m_CurClust->m_nClusters; i++) {
                float nLines = m_pFinal->m_CurClust->GetClustLines(m_pFinal->m_CurClust->GetCluster(i));
                if (nLines > m_pFinal->m_NM_Count) {
                        Oversized.Add(i);
                        StartPos.Add(NewClusts.GetSize());
                        ReduceOversized(ClusMembers[i], NewClusts);
                }
        }

        StartPos.Add(NewClusts.GetSize());

        m_pFinal->m_CurClust->AddSplits(NewClusts, Oversized, StartPos, m_pFinal->m_pWeight, m_pFinal-
>m_ClGrid, Specs->nMinN);

        delete[] ClusMembers;
        for (int n=0; n<NewClusts.GetSize(); n++)
                delete NewClusts[n];
}
}
```

© PNR and Associates, All rights reserved

```
void CFinalize::AspectRatio()
{
	long nClus = m_pFinal->m_CurClust->m_nClusters;

	for (long i=0; i<nClus; i++)
		m_MinLat[i] = m_MaxLat[i] = m_MinLong[i] = m_MaxLong[i] = 0.0;

	for (i=0; i<m_pFinal->m_nPoints; i++)
	{
		int Cluster = m_pFinal->m_CurClust->m_pCluster[i] - 1;

		if (m_pFinal->GetLat(i) < m_MinLat[Cluster] || !m_MinLat[Cluster])
			m_MinLat[Cluster] = m_pFinal->GetLat(i);
		if (m_pFinal->GetLat(i) > m_MaxLat[Cluster] || !m_MaxLat[Cluster])
			m_MaxLat[Cluster] = m_pFinal->GetLat(i);

		if (m_pFinal->GetLong(i) < m_MinLong[Cluster] || !m_MinLong[Cluster])
			m_MinLong[Cluster] = m_pFinal->GetLong(i);
		if (m_pFinal->GetLong(i) > m_MaxLong[Cluster] || !m_MaxLong[Cluster])
			m_MaxLong[Cluster] = m_pFinal->GetLong(i);
	}

	CDistance pDist;
	for (i=0; i<nClus; i++)
	{
		double MidLong = (m_MinLong[i] + m_MaxLong[i]) / 2.;
		double height = pDist.ComputeDistance(MidLong, m_MinLat[i], MidLong, m_MaxLat[i]);
		double width = pDist.ComputeDistance(m_MinLong[i], m_MinLat[i], m_MaxLong[i], m_MinLat[i]);
//		double Ratio = (width == 0) ? 0 : height / width;

		m_Height[i] = height;
		m_Width[i] = width;
	}

	return;
}

void CFinalize::ComputeArea()
{
	/* Calculate cluster areas  */
	CArea* pArea = new CArea;
	for (long i = 0; i < m_pFinal->m_CurClust->m_nClusters; i++)
	{
```

© PNR and Associates, All rights reserved

```
            m_ClusArea[i] = pArea->Compute(m_ClusMembers[i], m_pFinal->m_pLong, m_pFinal->m_pLat,
m_Vertices[i], m_Perimeter);
            pArea->GetPerimiterPts(m_ClusPerimMembers[i]);
      }

      delete pArea;
}

void CFinalize::ComputeCentroids(CString Method /*OfMaxPtDist*/)
{
      // Compute centroids
      long i, j, k, N;
      long nClus = m_pFinal->m_CurClust->m_nClusters;
      CDistance* pDist = new CDistance();

//Method = "OfAllPts";

      if (Method == "OfMaxPtDist") {
            for (i = 0; i < nClus; i++) {
                  double MaxPtDist = 0.0, dist = 0.0, maxLong = 0.0, maxLat = 0.0;

                  CDWordArray List;
                  for (j=0; j<m_ClusMembers[i].GetSize(); j++)
                        if (m_Perimeter[m_ClusMembers[i].GetAt(j)])
                              List.Add(m_ClusMembers[i].GetAt(j));

                  m_LongCentroids[i] = m_pFinal->GetLong(List[0]);
                  m_LatCentroids[i] = m_pFinal->GetLat(List[0]);

                  for (j=1; j<List.GetSize(); j++) {
                        for (k = 0; k < j; k++) {
                              dist = pDist->ComputeDistance(m_pFinal->GetLong(List[j]),
                                                      m_pFinal->GetLat(List[j]),
                                                        m_pFinal->GetLong(List[k]),
                                                        m_pFinal->GetLat(List[k]));

                              if (dist > MaxPtDist) {
                                    MaxPtDist = dist;
                                    m_LongCentroids[i] = (m_pFinal->GetLong(List[j]) + m_pFinal-
>GetLong(List[k])) / 2;
                                    m_LatCentroids[i] = (m_pFinal->GetLat(List[j]) + m_pFinal-
>GetLat(List[k])) / 2;
                              }
                        }
```

© PNR and Associates, All rights reserved