计算几何课程实验报告

题目：直线排布的最短路径问题

Title：Finding shortest paths in arrangements of lines

指导教师：邓俊辉

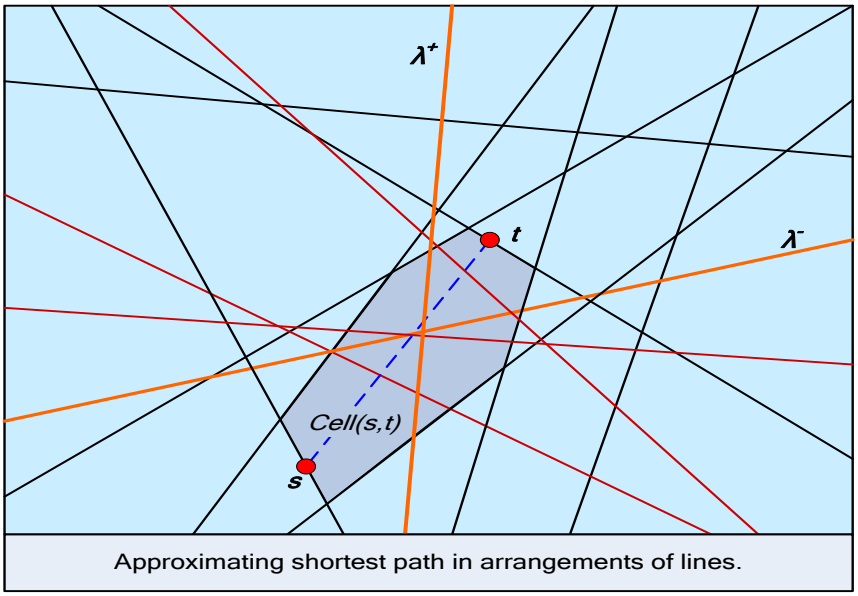
小组成员：闵贤龙 邹嘉 司天歌

清华大学计算机系

2004 年 1 月 11 日

问题的背景

在平面上有 n 条直线组成集合 L，对于给定的 L 上的任意 s 和 t 两个点，要求在集合 L 上找出 s 与 t 之间的最短路径。（如下图所示）



图一. 问题的描述

可以解决此问题的算法有：

1).在集合 L 构成的图上运用 Dijkstra 算法[1]，其时间复杂度为 $O(n^2 \log n)$；显然这种算法完全没有利用直线排布（Arrangements）的规律性，因此并不是最优的。

2).近似算法[2]，即在集合 L 构成的图上，计算得到一个子图（图一中的灰色区域），并在此子图上运用 Dijkstra 算法，其时间复杂度为 $O(n^2 \log n)$；这种算法得到的近似最短路径至多为精确值的两倍。

本次实验主要研究、实现了以上两种算法，并对测试结果进行了分析和比较。

算法及原理

1. Dijkstra 算法

该算法是 Dijkstra 于 1959 年提出的。首先它要求问题满足如下三个假设：

1).存在 N 个节点；

2).每一对节点间至少有一条路径连接着两个节点；

3).所有的边（branches）都是有长度的。

该算法的主要思路是：点 s 到点 t 的最短路径，暗含了该路径上子路径也一定是最短的，因为如果某一条子路径不是最短，那在 s 和 t 之间一定存在一条包含最短子路径的最短路径。

算法中定义了如下几个集合：

- **点集合**

SET A: 到点 s 的最短路径已知的点的集合；

SET B: 可能被添加入 SET A 的潜在节点的集合；

SETC: 剩余的点的集合；

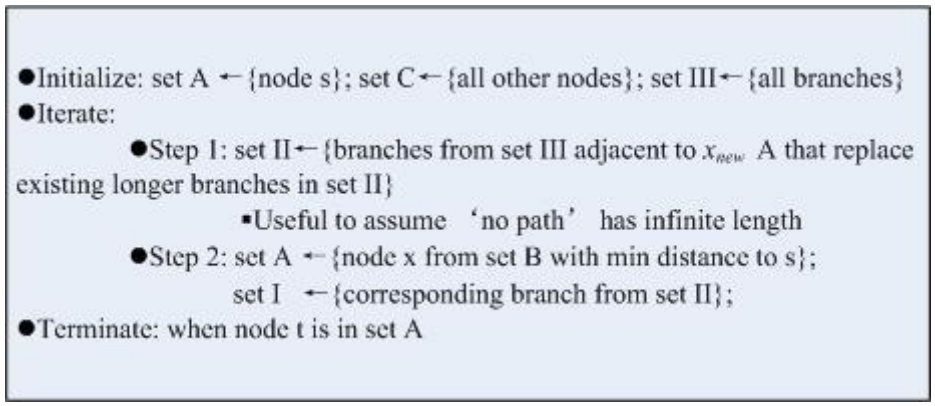
- **边集合**

SET I: 连接 SET A 中点的边的集合；

SET II: 可能被添加入 SET I 的潜在边的集合；

SET III: 其余的边的集合；

根据以上定义，算法可描述如下：



图二. Dijkstra 算法

2. 近似算法

该算法由 Prosenjit Bose 等于 1996 年提出，是我们研究和实现的重点。其主要思路是根据直线排布（Arrangements）的几何特征，求出一个最多包含 n 个节点的子图，并在这个子图上运用 Dijkstra 算法，从而找到一条近似最短路径。Bose 等人严格证明了通过近似算法找到的近似最短路径至多是精确值的两倍。

1） 算法的描述

算法首先定义了如下几个概念：

Cross lines：与 s,t 点连线相交的直线的集合，即图一中的细红线。

$\lambda+$：cross lines 中与线段 st (假设线段 st 是有方向的，s —> t，cross lines 也是有方向的，从与 st 的交点指向直线在 st 的上方部分)夹角最小的直线。

$\lambda-$：cross lines 中与直线 st 夹角最大的直线。

算法的主要过程为：

- 确定 Cross lines 以及 $\lambda+$，$\lambda-$；
- 将 Cross lines 中那些既不过点 s，也不过点 t 的直线剔除出 L，得到 L'，根据 L'中的直线与 st 连线的相对位置，将 L'中的直线转化为半平面，并将这些半平面相交，得到一个包含 st 连线的凸区域 Cell(s,t)，即图一中的灰色区域；

  半平面相交的 Divide & Conquer 算法如下：

```
Algorithm DC_HalfplaneIntersection (_setOfHalfplane H)
{
 if card(H) == 1
 then just return this halfplane as the answer ;
 else
 {
     Split H evenly into subsets H1 and H2;  //of sizes n/2 and n/2 respectively
    C1 := DC_HalfplaneIntersection (H1);
    C2 := DC_HalfplaneIntersection (H2);
     C := ConvexIntersection (C1, C2);
     Return C;
  }
}
```

图三 半平面相交算法

其中，ConvexIntersection(C_1, C_2)的算法，即两个凸区域求交的算法有多种，下面给出最常用的 Edge Chasing 的算法，此算法由 O'Rourke 等[3]于 1982 年提出，其时间复杂度为 $O(n)$。算法基本如下：

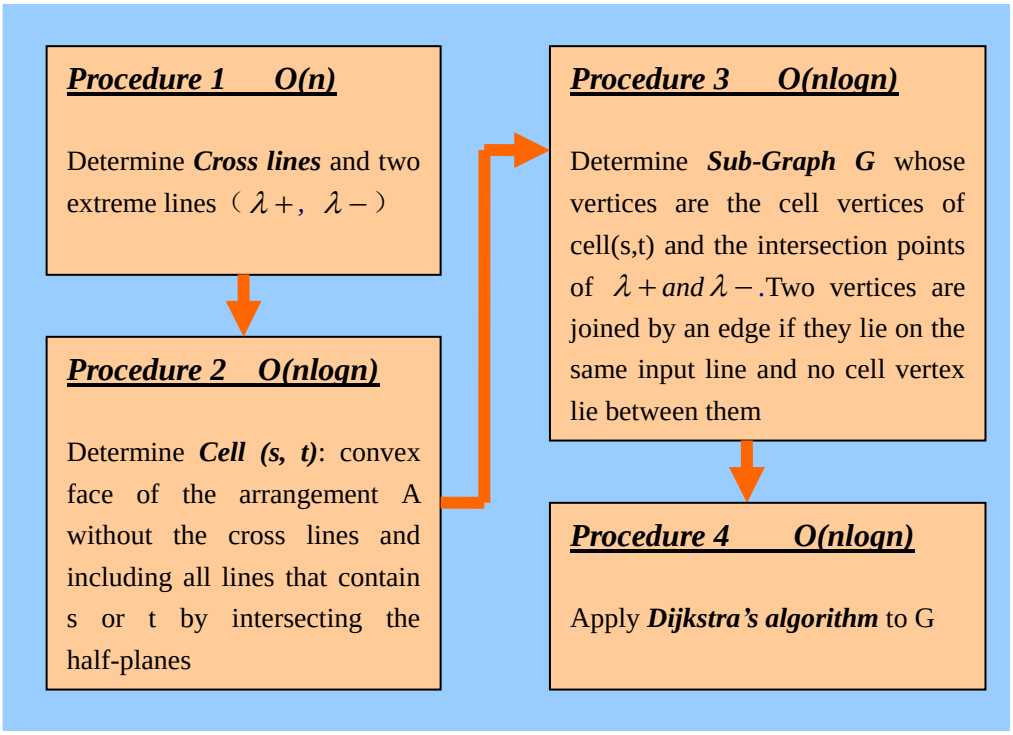
```
Algorithm ConvexIntersection(P, Q)
{
    Perform a (e.g. CCW) traversal of the boundaries of P and Q
    Maintain a pair of edges: e ∈ ∂P and f ∈ ∂Q.
    From a consideration of the relative positions of these edges ,
    advance e or f (e.g. CCW) along ∂P or ∂Q.
}
```

图四. 凸区域相交算法

- 将 Cross lines 与 Cell(s,t)的边界相交，得到一系列点，将这些点加入到 Cell(s,t)中；
- 两条极值直线 $\lambda+$，$\lambda-$ 在 Cell（s,t）内的部分（包括这两条极值直线的交点，如

果交点落在 Cell(s,t)内部的话）与 Cell(s,t)边界上的点和边一起组成了子图 G；

- 在子图上运用 Dijkstra 算法，得到的最短路径就是问题所要求的最短路径的一个近似值。



图五.算法的描述

2） 算法的复杂度分析

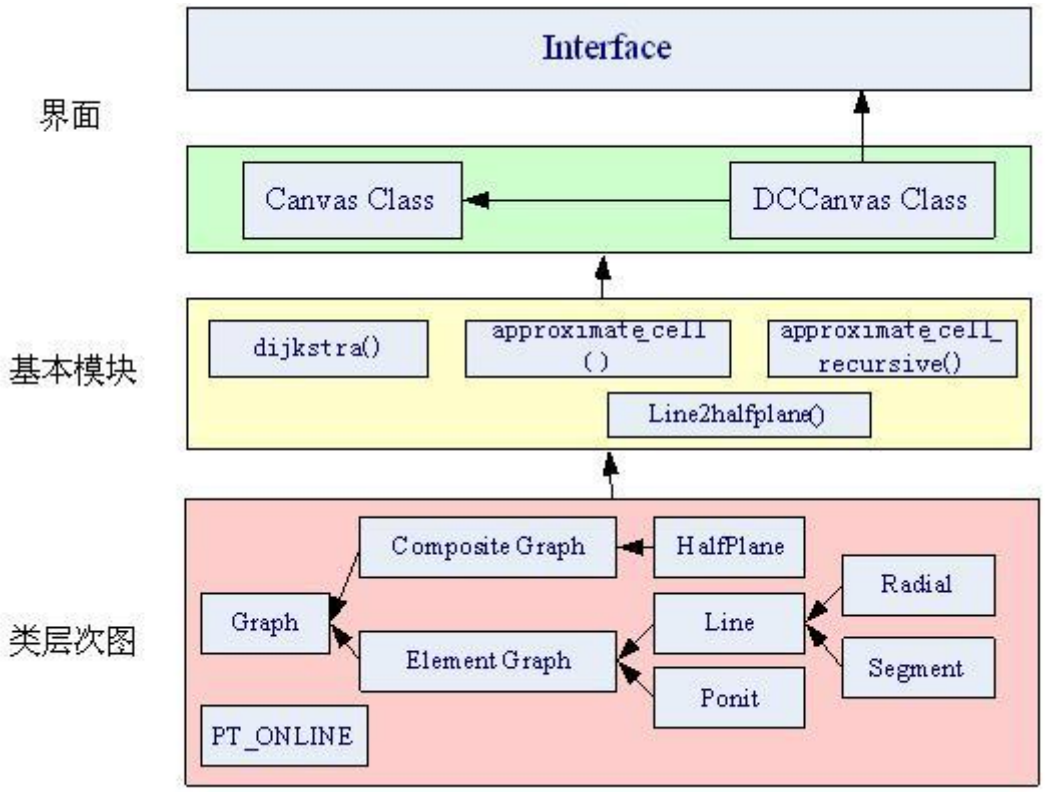
由于确定 Cross lines 以及极值直线的时间是线性的，为 *O*(n)；确定 Cell(s,t)是通过半平面相交得到的，半平面相交问题的复杂度下界为 *O*(nlogn)，而 Divide & Conquer 方法达到了此下界；将 Cross lines 与 Cell(s,t)的边界相交并将交点归入 Cell(s,t)的过程的下界也为 *O*(nlogn)；由于最终得到的 Graph G 至多只有 n 个顶点，因此在其上运用 Dijkstra 算法的时间为 *O*(nlogn)。因此整个算法的复杂度为 *O*(nlogn)。

系统设计与数据结构

本次开发的目的是实现 Dijkstra 算法和近似算法，求得平面上任意给出的 n 条直线上的两点间的最短路径，并用直观的方式分别给出两种算法所求得的结果。

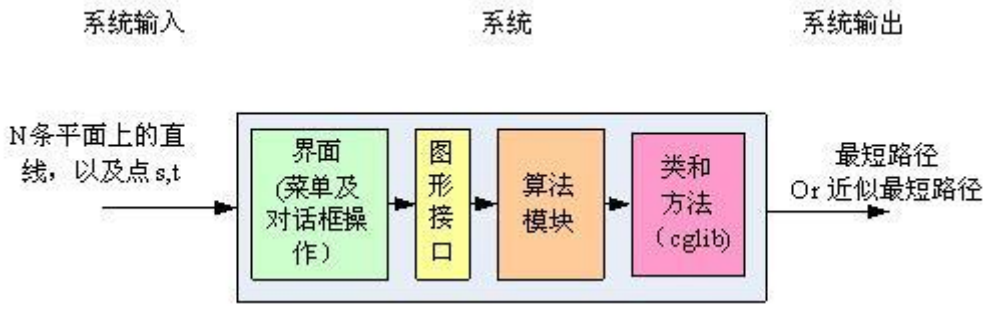
本次开发是以 Windows 操作系统为平台，VC6.0 为开发环境，WinCvs1.3 为版本管理工具，采用 Composite 的设计模式，以 C++为编程语言进行的面向对象系统的开发。系统设计按如下几方面进行说明。

1. 系统体系结构



图六.系统体系结构图

2.基本设计概念和处理流程



图七.系统处理流程

3.类、模块和接口设计

1）类的设计

在本次开发中主要采用 Composite 模式[4]来定义包含基本对象和组合对象的类层次结

构，使得用户对单个对象和组合对象的使用具有一致性，从而可以简化客户代码，并使设计更加一般化。附录一的 cglib 是本次开发过程中制作的 html 帮助文档，里面包含了算法实现所用到的各个类的完整定义，以及它们之间的关系介绍。

2）模块和接口，以及数据结构

- bool approximate_cell(std::vector< Halfplane* > **planes*, HalfPlane * *new_plane*)

描述：输入一组半平面，用循环相交的方法得到这些半平面相交所得的凸区域；

Parameters：

planes 输入的半平面数组

new_plane 输出的指向最后得到的凸区域 Cell 的指针

- bool approximate_cell_recursive(std::vector< Halfplane* > **planes*, HalfPlane * *new_plane*)

描述：输入一组半平面，用二分递归的方法得到这些半平面相交所得的凸区域；

Parameters：

planes 输入的半平面数组

new_plane 输出的指向最后得到的凸区域 Cell 的指针

- bool dijkstra (EdgeMatrix * *edge*, double *dist*[ ], int *path*[ ])

描述：输入包含顶点和边信息的邻接矩阵，然后用 dijkstra 算法求得两点间的最短路径；

Parameters：

edge：输入的边矩阵

dist：输出的距离数组

path：输出的路径数组

- void line2HalfPlane (Line * *ln*, HalfPlane * *plane*, Point * *ref_pt*)

描述：将输入的直线根据其在平面上的位置转化为半平面；

Parameters:

ln： 输入的 line 对象

plane： 输出的 halfplane 指针指向一个半平面的存储位置

ref_pt： 参照点，通过直线与该点的相对位置来判断所得半平面的方向

4.主要算法设计

- approximate_cell()

```
48 {
49    if( !new_plane || ( planes->size() <= 0 ) )
50       return false;
51    int i, n;
52    n = planes->size();
53    HalfPlane plane, *p;
54    plane.copy( (*planes)[0] );
55    for( i = 1; i < n; ++ i )
56    {
57       p = (*planes)[i];
58       HalfPlane h_plane;
59       if( !plane.intersect( p, &h_plane ) )
60          return false;
61       plane.copy( &h_plane );
62    }
63    new_plane->copy( &plane );
64    return true;
65 }
```

- approximate_cell_recursive()

```
68 {
69    if( !new_plane || ( planes->size() <= 0 ) )
70       return false;
71
72    if( planes->size() == 1 )
73    {
74       new_plane->copy( (*planes)[0] );
75       return true;
76    }
77    else
78    {
79       int i, j, m;
80       i = 0;
81       j = planes->size();
82       m = j / 2;
83       std::vector< HalfPlane* > planes1;
84       std::vector< HalfPlane* > planes2;
85
86       for( int k = 0; k < j; ++ k )
87       {
88          if( k < m )
89             planes1.push_back( (*planes)[k] );
90          else
```

```
91              planes2.push_back( (*planes)[k] );
92        }
93
94        HalfPlane new_plane1, new_plane2;
95        bool b1 = approximate_cell_recursive( &planes1, &new_plane1 );
96        bool b2 = approximate_cell_recursive( &planes2, &new_plane2 );
97        if( b1 && b2 )
98        {
99           if( new_plane1.intersect( &new_plane2, new_plane ) )
100              return true;
101        }
102        return false;
103     }
104 }
```

- dijkstra()

```
205 {
206     int i, n;
207
208     if( !edge )
209         return false;
210
211     n = edge->size();
212     std::vector<int> S( n, 0 );
213
214     for( i = 0; i < n; ++ i )
215     {
216         dist[i] = (*edge)[ n - 1 ][ i ];
217         S[i] = 0;
218         if( ( i != ( n - 1 ) ) &&
219             ( dist[i] < MAX_DISTANCE ) )
220             path[i]= n - 1;
221         else
222             path[i]= -1;
223     }
224
225     S[ n - 1 ] = 1;
226     dist[ n - 1 ] = 0;
227
228     for( i = 0; i < n - 1; ++ i )
229     {
230         int min = MAX_DISTANCE;
231         int u = n - 1;
232         for( int j = 0; j < ( n-1 ); ++ j )
233         {
```

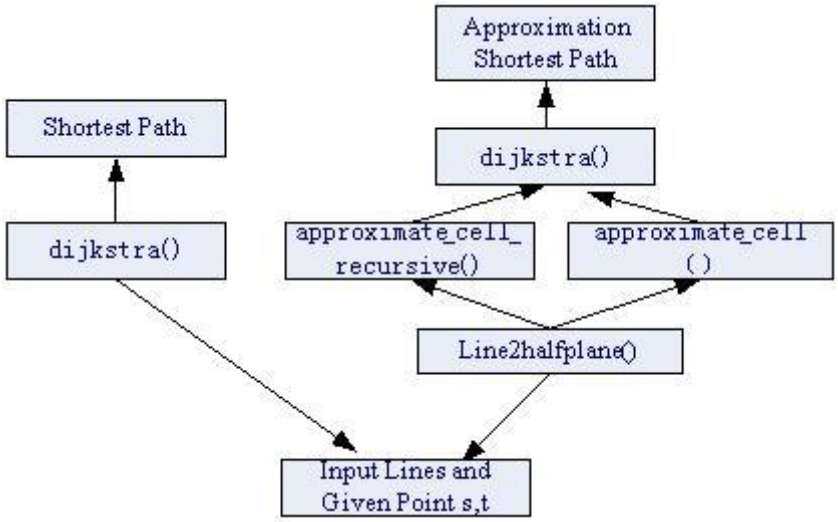
```
234          if( !S[j] && ( dist[j] < min ) )
235          {
236             u = j;
237             min = dist[j];
238          }
239       }
240       S[u] = 1;
241       for( int k = 0; k < n - 1; ++ k )
242       {
243          if( !S[k] && dist[u] + (*edge)[u][k] < dist[k] )
244          {
245             dist[k] = dist[u] + (*edge)[u][k];
246             path[k] = u;
247          }
248       }
249    }
250    return true;
251 }
```

- line2HalfPlane()

```
9 {
10    if( !plane || !ln || !ref_pt )
11       return ;
12    HalfPlane p, rectPlane;
13    int r = lineDirection( ln->getStart(), ln->getEnd(), ref_pt );
14    if( r <= 0 )
15    {
16       p.add( ln->getStart() );
17       p.add( ln->getEnd() );
18    }
19    else if( r > 0 )
20    {
21       p.add( ln->getEnd() );
22       p.add( ln->getStart() );
23    }
24
25    if( ( width == 0 ) || ( height == 0 ) )
26    {
27       plane->copy( &p );
28       return ;
29    }
30    Point pt;
```

```
31    pt.setX( 0 - border );pt.setY( 0 - border );
32    rectPlane.add( &pt );
33    pt.setX( width + border );pt.setY( 0 - border );
34    rectPlane.add( &pt );
35    pt.setX( width + border );pt.setY( height + border );
36    rectPlane.add( &pt );
37    pt.setX( 0 - border );pt.setY( height + border );
38    rectPlane.add( &pt );
39    if( !p.intersect( &rectPlane, plane ) )
40    {
41 #ifdef _DEBUG
42        assert( 0 );
43 #endif
44    }
45 }
```

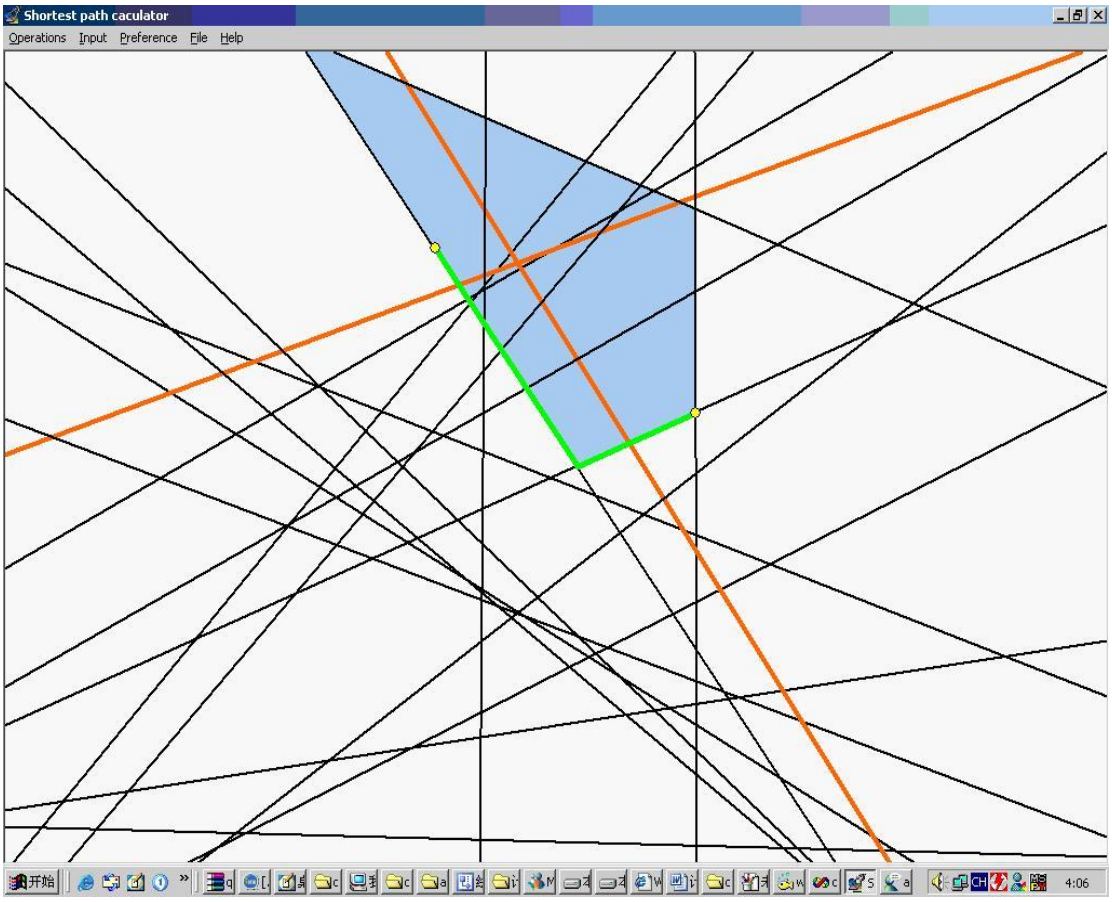
以上各个模块间的关系如下图所示：



图八. 模块间的关系

5.界面设计

本套系统的界面设计遵循直观和便捷两大原则，采用菜单和对话框的操作方式。界面大致如图九所示。



图九.系统界面

1） Operations 下拉菜单

表一.Operations 下拉菜单功能设计

选项	功能
Clear	清空窗口
Refresh	刷新，去除上一次操作的结果但保留原先创建的直线和点的信息。
Dijkstra	利用 Dijkstra 算法求最短路径
Appr.	利用半平面循环相交的近似算法求近似最短路径
Appr.ani	上面算法的逐步演示过程
Appr.Recur	利用半平面递归相交的近似算法求近似最短路径
Appr.Recur.ani	上面算法的逐步演示过程
Exit	退出程序

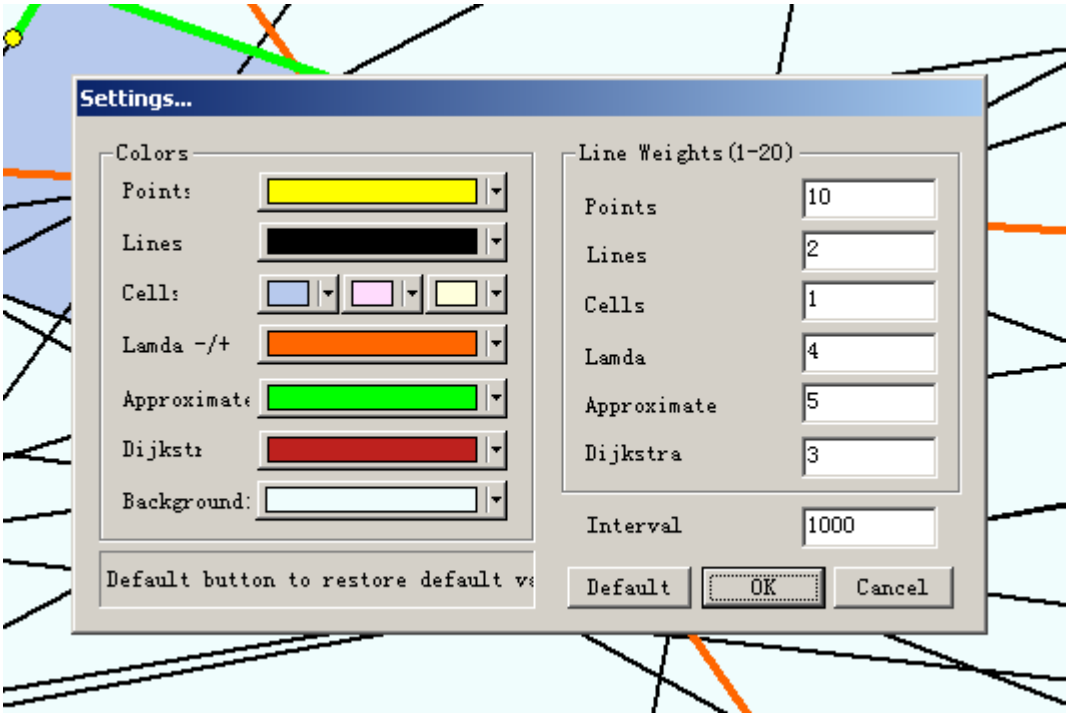
2）Input 下拉菜单

表二.Input 下拉菜单功能设计

选项	功能
Auto generate lines	自动生成直线，每点击一次，生成 10 条
Draw lines	通过在屏幕上点击来生成直线，点击两点，生成一条直线，且这条直线是这两点的连线
Pick Point	

3) Preference 下拉菜单

Preference 中只有一个选项 Settings,点击后出现如下对话框，可自定义各种几何对象的颜色、线条等。



图十 Settings 选项

4) File 下拉菜单

表三 File 下拉菜单功能设计

选项	功能
Open	打开一个文件（一个直线集和点 s,t 信息）
Save	保存当前文件

5）Help 下拉菜单

主要介绍版本信息及作者的联系方式。

6) 计时功能

当运算完一次算法后，都会弹出对话框，告知该算法此次运算所花费的时间。

6.文件清单

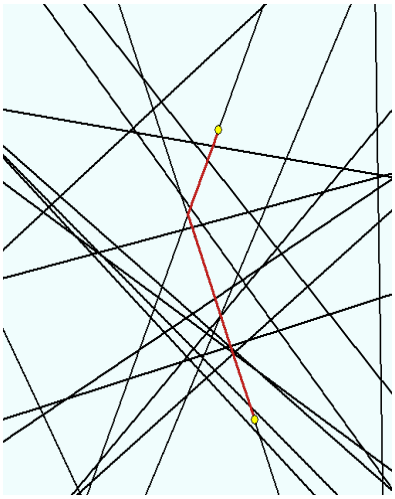
表四. 文件清单

approximate.h	dijkstra.cpp	point.cpp	colorbtn.cpp
approximate.cpp	elementgraph.h	radial.h	Global.h
canvas.h	elementgraph.cpp	Radial.cpp	Global.cpp
canvas.cpp	graph.h	AboutDialog.h	Resourse.h
comdefs.h	graph.cpp	AboutDialog.cpp	SettingDlg.h
compositegraph.h	halfplane.h	cg.h	SettingDlg.cpp
Compositegraph.cpp	halfplane.cpp	cg.cpp	StdAfx.h
dccanvas.h	line.h	cgDlg.h	StdAfx.cpp
Dccanvas.cpp	line.cpp	cgDlg.cpp	
dijkstra.h	point.h	colorbtn.h	

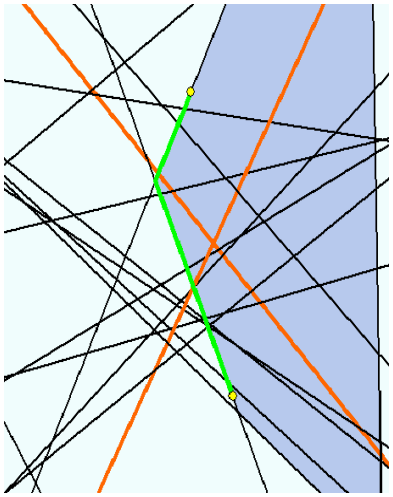
测试、对比结果

由于循环半平面相交的近似算法（approximate_cell（））和递归半平面相交的近似算法(approximate_cell_recursive（）)，只是在半平面相交的实现环节上有差异（具体可以参看上文中主要算法设计部分），而其结果相同，因此在测试结果描述过程中，为省略篇幅，只给出了两幅图，如图十一所示。

1. 一般情形



(a) Dijkstra 算法



(b)近似算法

图十一.一般情况的测试结果

测试中，还利用了开发的计时功能对三种算法的运算效率进行了统计，部分结果如下表所示。

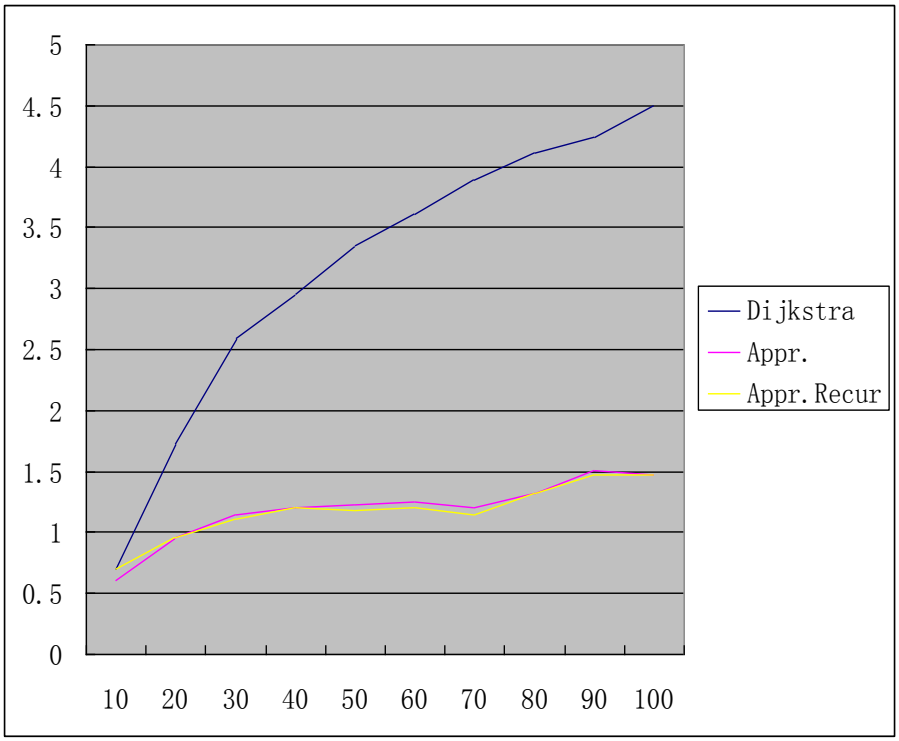
表五.算法的性能比较

测试机型：CPU: Athlon XP 2000+(Thorton)MEM: 512MB(DDR333)

单位：秒

直线数目	Dijkstra 算法	Appr(循环半平面相交)	Appr.Recur(递归半平面相交)
10	0.0005	0.0004	0.0005
20	0.0053	0.0009	0.0009
30	0.0392	0.0014	0.0013
40	0.0911	0.0016	0.0016
50	0.2253	0.0017	0.0015
60	0.41	0.0018	0.0016
70	0.7749	0.0016	0.0014
80	1.2841	0.0021	0.0021
90	1.7333	0.0032	0.003
100	3.1167	0.003	0.003

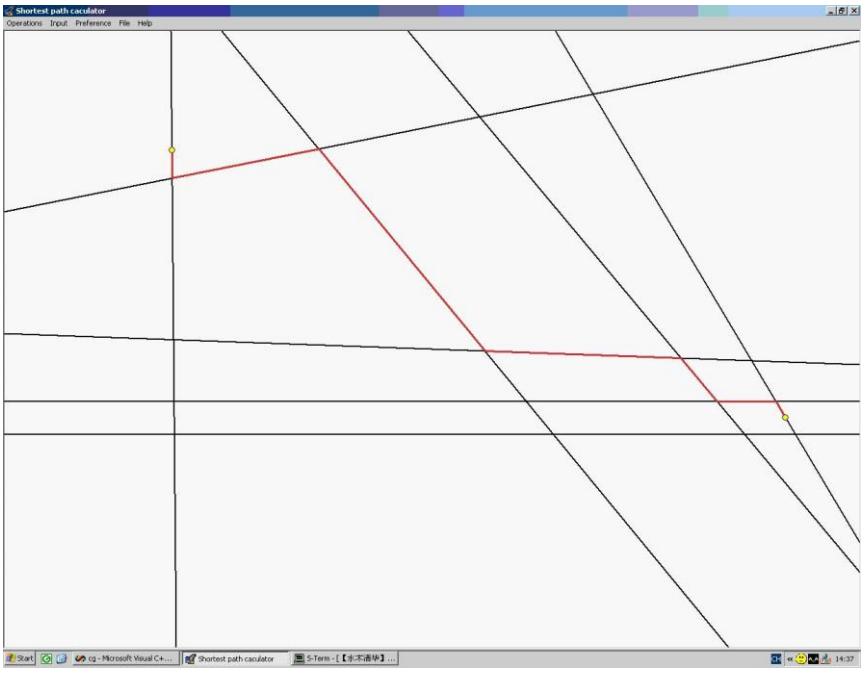
上表的曲线图如下所示：纵坐标作了 log(10000*x)的处理



图十二 三种算法运算时间比较

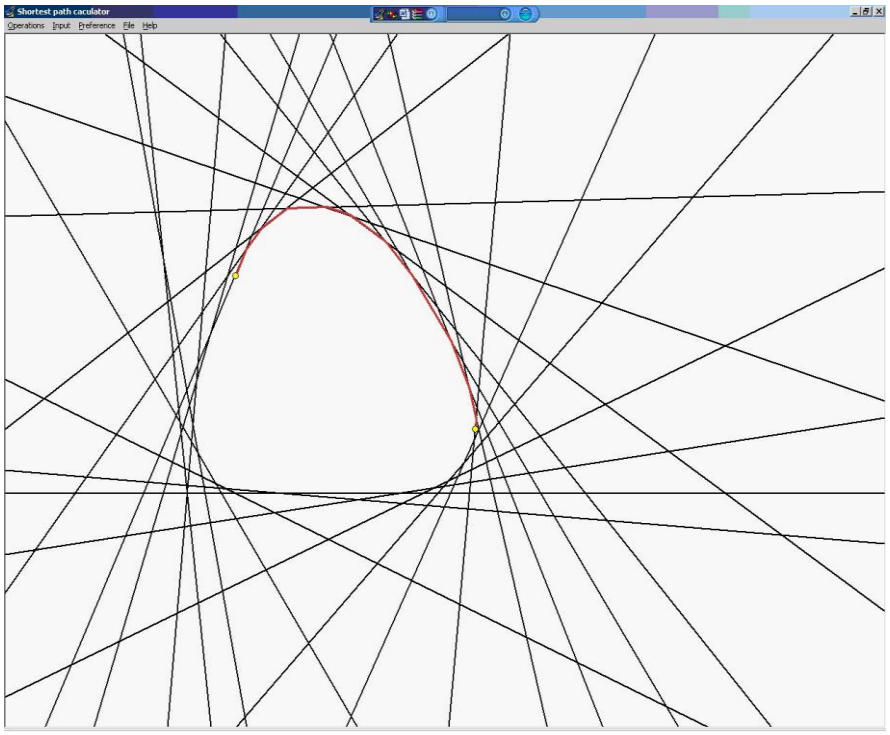
另外还通过理论分析，找到了几种生成较特殊的最短路径的大致办法，如下面两例所示。

2.ZigZag 情形



图十三.Zigzag 最短路径情形

3.最短路径由 n 段线段构成



图十四.较复杂的最短路径

以上 2，3 两种特殊情况都已保存在相应文件夹中，需要演示或观察这两种情况时，直接在 File 菜单下选择 Open 即可打开。

4.结果分析与结论

通过测试我们发现近似算法和 Dijkstra 算法所求得的最短路径在绝大部分时候（90%）都是重合的。而根据表五，近似算法的运行速度远高于 Dijkstra 算法。且其前者对内存的占用也远低于后者（在随后部分还会对此专门讨论）。因此，我们得出结论，近似算法优于 Dijkstra 算法。

此外，在测试中，系统表现出了较好的健壮性和稳定性，极少出现死机的情况。

实现过程中遇到的问题及解决对策

1.Dijkstra 算法实现中静态数组上限问题

- 问题描述

在最初的实现中，在输入直线的数目比较多的情况下，会出现数组下标溢出的情况。

- 问题分析和解决

设直线数目为 n，Dijkstra 算法使用的邻接矩阵的距离存储数量级为 $O(n^4)$。在该版本的 Dijkstra 算法实现过程中，使用的变量为长度固定的静态数组，因此导致出现溢出的问题。

为此采用动态数组存储点与点之间的距离，根据直线的交点数目确定其存储空间。

2.Edge Chasing 算法的异化情况处理

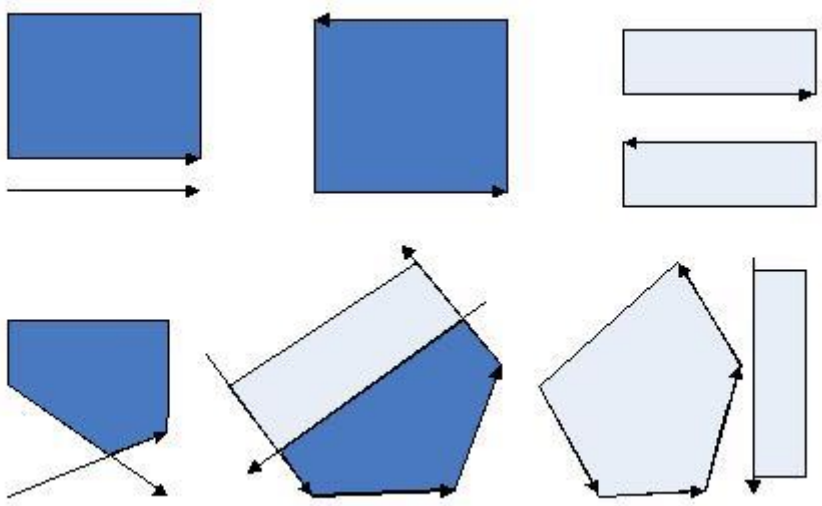
- 问题描述

Edge Chasing 算法是用来求凸包相交的算法。由于我们未找到 O'Rourke 关于该算法的原文，而讲义中未给出一些特殊的异化情况的处理方法。因此，在实现过程中，我们对该算法的异化情况作出了分析和处理。

- 问题分析和解决

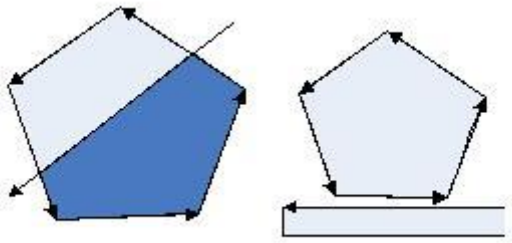
异化情况分成如下的几类情况：

a. 开放的半平面相交



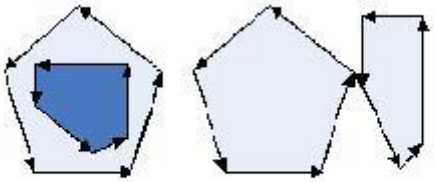
图十五 开放的半平面相交

b. 开放的半平面和封闭的半平面相交



图十六 开放的半平面和封闭的半平面相交

c. 封闭的半平面相交（凸包相交）



图十七 封闭的半平面相交

为了简化处理，我们将所有开放的半平面与一个足够大的矩形求交得出封闭的凸包，然后再进行通常的 Edge Chasing 算法处理。

在近似算法中，由于输入的参数是直线集合，因此利用 *st* 的中点为输入的所有直线生成开放的半平面，然后再将开放的半平面转化成封闭的凸包，最后进行凸包集合的求交运算。

对于情况 c，Edge Chasing 算法运算的过程中，如果 edge chasing 已经结束，但是还没有找到新的凸包上的点，必须要处理一下。

此外，在 Edge Chasing 算法的通常情况下，也可能出现两个边(edge)平行、共线等情况。这些情况之中，当出现以下两种情况认为凸包的交集为空。

图十八 凸包交集为空的情况

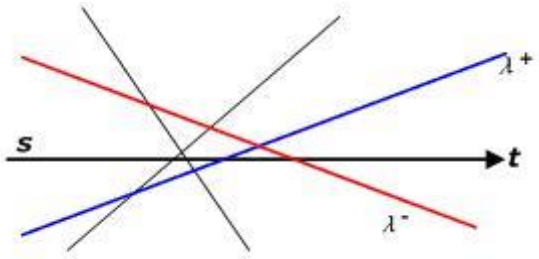
3. $\lambda\pm$选取问题

- 问题描述

在近似算法中，$\lambda\pm$的选取和计算先后出现了错误。

◆ 问题分析和解决

根据近似算法的描述，分别定义为$\lambda\pm$与输入向量 *st* 相交的直线中，正的斜率最小斜率和负的斜率最大的两条直线。如图：



图十九 $\lambda\pm$选取问题

在确定直线与向量 st 的夹角时，采用如下的步骤：

a. 把直线 ln 向量化为 *l*；

规定向量 *l* 方向时，要求 *l* 的目标点在向量 *st* 左边。即使 l 与 st 的夹角在 0～180 度之间。该步骤利用向量叉积运算来完成。

b. 利用点积，计算 *l* 与 *st* 的夹角的余弦值 *d*；

$$d = \frac{l \cdot st}{|l| \cdot |st|}$$

由于余弦函数在 0～180 度之间是单调递减的，所以比较利于选择$\lambda\pm$。

c. 选取 d 最大和 d 最小的两条直线，对应的分别是$\lambda+$和$\lambda-$。

4.选点问题

◆ 问题描述

在选取屏幕上的点时，很难及时把鼠标的指针精确地指定在某直线上。

◆ 问题分析和解决

屏幕上的坐标都是整数，然而在计算点在直线上的过程中为了精确使用的是浮点数，因此很难选中输入点。

定义了 PIXEL_THREADHOLD，在判断鼠标位置是否在某直线上时，根据直线的斜率做出了校正(Snap to Geometry)。

具体算法如下：

```
if ( line.verticle() || float_compare( line.slope(), 1 ) > 0
)
{
        double x = line.getX( pt.getY() );
        pt.setX( x );
        if( line.onLine( &pt ) )
                add_point();
}
Else
{
        double y = line.getY( pt.getX() );
        pt.setY( y );
if( line.onLine( &pt ) )
                add_point();
}
```

图二十 调整点位置的算法

5.显示顺序和屏幕刷新问题

- 问题描述

  在处理屏幕上的元素显示的过程中，刷新和显示有时不够理想或者不正常。表现如下：

  a. 程序切换出去后，有些画好的图形就不见了；

  b. 有时输入点被其他的元素覆盖住，虽然不影响结果，但是影响美观；

  c. 另外进行计算和重新输入时，没有及时地清空屏幕上的前一次计算的结果；

- 问题分析和解决

  a. 有关的画图的处理应该统一由 CCgDlg::OnPaint()函数调用，在需要更新的时候，应该调用 Invalidate()函数通知程序执行 CCgDlg::OnPaint()进行画图。从而解决程序切换出去后，有些图形不见的情况；

  b. 在 CCgDlg::OnPaint()函数中，规定各个元素的画图顺序，如下：

```
cell_.draw( canvas_ );  /* draw Cell (s,t). */
canvas_->clear();    /* draw background. */
lines_graph_.draw( canvas_ );  /* draw input lines. */
ap_segments_.draw( canvas_ ); /* draw approximate result segments . */
dj_segments_.draw( canvas_ ); /* draw dijkstra result segments . */
points_.draw( canvas_ );     /* draw input points ( s, t ). */
```

图二十一 元素画图顺序

c. 在进行其它计算前，及时清空前一次运算结果；

6.延时函数处理

- 问题描述

  在进行近似算法的演示时，在程序延时处理时，应用程序无法响应用户输入。

- 问题分析和解决

  程序利用 Sleep()函数实现延时处理，Sleep()函数在执行时挂起了当前的线程（主线程），使得用户无法在应用程序中进行其它的处理。

  利用 Windows 提供的 MsgWaitForMultipleObjects()函数以及相关的消息处理函数，实现了 msgLoopWait()延时函数，该函数的延时过程中支持对 Windows 消息的处理。

  但是也带来了一定的副作用，即演示会由于用户的鼠标移动、打开对话框等操作而暂停。

7.Dijkstra 算法实现中内存释放问题

- 问题描述

  Dijkstra 算法运算时，消耗内存巨大，仅在下一次计算时释放。

- 问题分析和解决

  Dijkstra 算法实现中使用的邻接矩阵占用了巨大的内存空间，并且，该矩阵属于运算中间结果，在计算完毕后应该及时释放，从而减少系统内存使用消耗。

  因此，在进行 Dijkstra 运算结束后，及时释放邻接矩阵占用的内存。

改进方向

1.工程上

1）目前只能显示直线在当前固定窗口中的部分，可以将窗口改进成可移动观看的形式；

2）由于Dijkstra算法在输入直线较多的情况下，运算会比较复杂，较耗时，因此可以考虑改进为后台计算模式，即在后台开一个线程计算，这样前台就可以继续操作，不会失去响应。

2. 理论研究上

1）文献[2]只给出了一个近似上界的证明，但在我们的实验中发现从来没有达到过这个上界，可以考虑能否降低这个上界

2）可以进一步考察其精确度的具体分布，从而加深对近似算法和问题几何本质的理解。

参考文献

[1] E.W. Dijkstra. A note on two problems in connexion with graphs. Numberische Mathematik, 1:269-271, 1959

[2] P. Bose, W. Evans, D. Kirkpatrik, M. McAllister, and J.Snoeyink. Approximating shortest paths in arrangements of lines. In Proc. 8th Canad. Conf. Computational Geometry, pp. 143-148, 1996.

[3] [OCON82] J. O'Rourke, C. Chien, T. Olson, and D

. Naddor. A New Linear Algorithm for Intersecting Convex Polygons. Comput. Graph

. Image Process., 19:384-391, 1982.

[4] Design Patterns Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vissides. 1994

致谢

感谢邓老师的悉心、生动的授课使我们三人在短短一学期时间内对计算几何、算法、数据结构有了深入的理解。而最后的课程作业不仅使我们所学的知识得到了进一步巩固，还极大地锻炼了我们的动手能力，设计能力以及团结协作能力。1 月 9 日，检查实验过程中，邓老师的和蔼和耐心也促使了我们在之后的两天内又将程序和文档进行了完善。感谢邓老师给了我们这样一个机会，让我们作为一个 Team,一起创造了一份值得回味的精彩！