动物园看守问题的实现

杜　炤 [024856]
陈　汐 [035405]
袁进辉 [035404]

1. 问题描述

1.1 背景

我们的朋友 Mike 是动物园的看守，其平面示意图如图 1 所示。大多数时间他都坐在图中 S 处的椅子上。当然，作为工作人员的他不可能整天都那么悠闲。每过一段时间，他都要给动物们喂食。动物们被关在依墙而建的笼子里面，所以他需要从 S 点出发，依次走到每个笼子跟前，把食物放进笼子中，最后再回到 S 点。

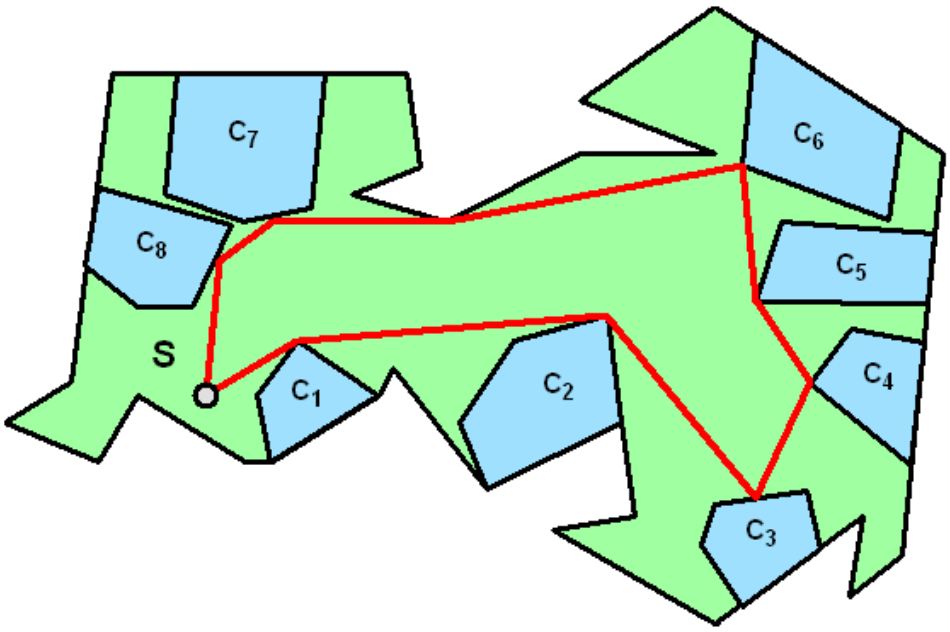


图 1 动物园平面示意图

但是动物园是如此之大，以至于勤劳的 Mike 也开始发愁。他希望我们可以帮助他找到一条最佳路径，让他喂动物所走的距离最短。

1.2 描述

在 **CG** 中，Mike 的问题被称为 Zoo-Keeper's Problem，最早由 Chin 和 Ntafos 于 1992 年提出，参见文献**[1]**,其数学描述如下

1） 动物园的围墙不自交，构成一个简单多边形，记为 P

2） 笼子是凸多边形，记为 C_i，每个 C_i 与 P 至少有一条公共边

3） 可达区域由动物园围墙和笼子的边缘所围成，构成简单多边形 Q

4） 问题：在 Q 内寻找一条从 S 点出发，能接触各个笼子的最短封闭路径

这是我们将要实现的算法所能解决的。事实上，这个算法的能力要更强一些，它解决的是一个比 Zoo-Keeper's Problem 更一般的问题，我们把它称为 Zoo-Keeper's Queries。

同 Zoo-Keeper's Problem 相比，Zoo-Keeper's Queries 的区别在于我们寻找的不再是一条封闭路径。由于动物园描述以及起始位置 S 都是长时间保持不变的，我们希望对它们进行预处理，使得任意给出一个查询点 F，都能比较快速的找到 S 和 F 两点间的最佳路径。我们的 Mike 可以从 S 出发，沿着这条最佳路径给每个动物喂食，并最终到达目的地 F。相信有了这个算法，他的工作会轻松很多。

2. 算法及原理

2.1 光束的概念

算法中最重要的一个概念是（Floodlight）光束。从结构上来看，它由一个顶点及从这点出发的两条射线组成，图 2 中给出了光束的例子及其照射的区域。具体如何在我们的问题中使用光束这个概念呢？我们希望从下面这个简单的问题中给出比较直观的解释。

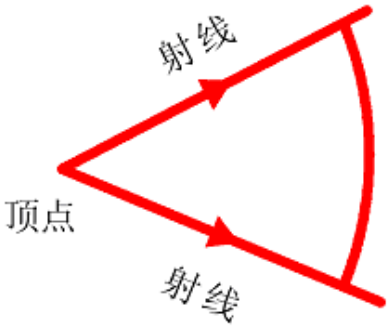


图 2 光束示意图

如图 3 所示，对起点 S 我们希望对笼子 C_1 边界上的任意一点 E，都能快速的得到从 S 到 E 的最短路径。事实上，从图中不难看出，从 S 到 C_1 各点的最短路径构成了一个像（Funnel）漏斗一样的形状，它由一些光束合并而成。这让我们可以用有限多个光束来表示所有从 S 到笼子 C_1 上各点 E 的最短路径。

更具体的，我们希望建立如图 4 所示的一个树型结构，其中树的节点就是图 3 中的这些光束。其中，光束 F_2 以另一个光束 F_1 为父节点，当且仅当 F_2 的顶点为光束 F_1 所照射。比如说，从图 3 中的各个光束可以创建如图 4 所示的树型结构。获得这个树型结构后，为了得到从 S 到 E 的最短路径，我们只需要先判断 E 被哪个光束所照射，从而得到树型结构中相应的叶子节点。然后，通过树的上溯就可以得到最短路径。比如说，图 3 中的 E 点被光束 1 所照射。通过图 4 中树的上溯，可知

E ---- 光束 1 的顶点 ---- 光束 2 的顶点 ---- 光束 3 的顶点即 S

即为从 E 到 S 的最短路径。事实上，我们下面给出的算法中，预处理部分的主要任务就是构造非常类似的一个树型结构。 而路径查询算法也就是利用该树型结构，通过上溯得到最短路径。

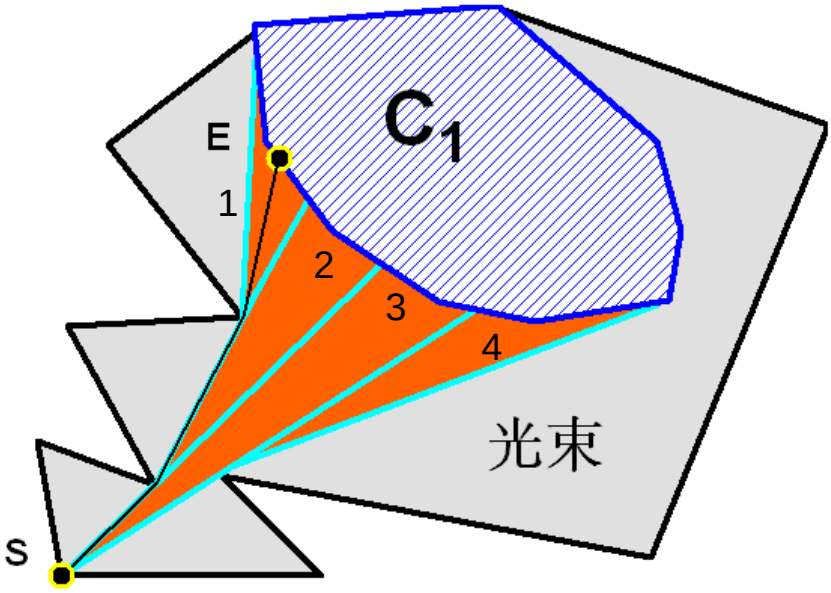


图 3 光束概念示例

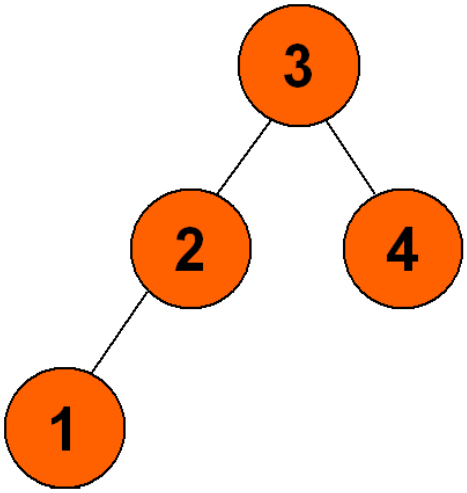


图 4 树型结构示例

2.2 算法简述

在上一节中，我们指出算法预处理部分的主要任务是构造一个树型结构。更精确的，我们称它为 Ftree，它以光束作为自己的节点。光束 F_2 以另一个光束 F_1 为父节点，当且仅当 F_2 的顶点为 F_1 所照射。那么，哪些光束对我们后面路径的查询是有帮助的，必须被加入到 Ftree 中去呢？

事实上，我们将始终维护一个光束序列 Flist。它是一组光束，并且总是按照射角度的逆时针方向排列。我们将不断更新 Flist，对它进行各种操作，而每次得到的新光束则都会被加入到 Ftree 中去。更具体的，在得到最初的光束序列 Flist，即从出发点 S 照射到第一个笼子 C_1 的光束序列（如图 3 所示）后，我们要让它在下一个笼子 C_2 上反射，然后照射到第二个笼子 C_3 上去。接下来再让 Flist 在 C_3 上反射，并照射到下一个笼子 C_4 上去。以此类推，最后从笼子 C_{k-1} 上反射出来的 Flist 会被照射到最后一个笼子 C_k 上去并在 C_k 上被反射。由于最后这个光束序列的特殊重要性，我们称它为 FinalList。

由此，通过上述不断更新的过程，我们得到了 Ftree 和 FinalList。利用 FinalList 中的光束，可以对多边形 Q 进行分割，并对分割结果进行预处理。在路径查询部分中，对每个输入的终点 q，我们只需要通过 Point Location 操作得到 FinalList 中照射点 q 的那个光束，就可以获得 Ftree 中相应的那个叶子节点。同 **2.1** 中类似，由此叶子节点进行上溯就可以得到最短路径。由于预处理后 Point Location 的操作需要时间 O(logn)，树的上溯时间为 O(K)，这里 K 为最短路径的点数，亦即上溯的步数。因此，路径查询的时间复杂

度为 O(logn+K)，这是非常令人满意的。但在 **2.6** 中，我们将指出，Ftree 的空间复杂度是无法令人满意的，最坏情况可能达到 n^2 的数量级，因此还需要一些技巧来加以改进。但算法的主要思想就是上面描述的整个过程。

下面，我们首先对光束序列 Flist 的反射和照射操作给出更细致的描述，然后指出数据结构 Ftree 在空间复杂度上的缺陷以及相应的解决措施。最后，给出预处理和路径查询的完整描述和时间复杂度分析。

2.3 镜面反射原理

对线段 pq 以及同侧两点 a 和 b，如何得到从 a 到 b 并且接触线段 pq 的最短路径呢？方法由下面的镜面反射原理给出。我们所需做的就是将点 a 对线段 pq 做镜面反射操作，得到点 a'，然后将点 a' 同 b 相连接，得到 a'b 同直线 pq 的交点 r。根据 r 的位置可以分为如下的三种情况

1）点 r 在线段 pq 上，如图 5 所示
2）点 r 不在线段 pq 上且离 p 点更近，如图 6 所示
3）点 r 不在线段 pq 上且离 q 点更近，处理方法同 **2）**类似

如果换一个角度来看镜面反射原理，我们会有更大的收获。设从 a 点出发，到达了笼子 C 某边 pq 上一点 r 那么路径中的下一点 b 会有哪些可能呢？同样，我们可以分成如下的三种情况

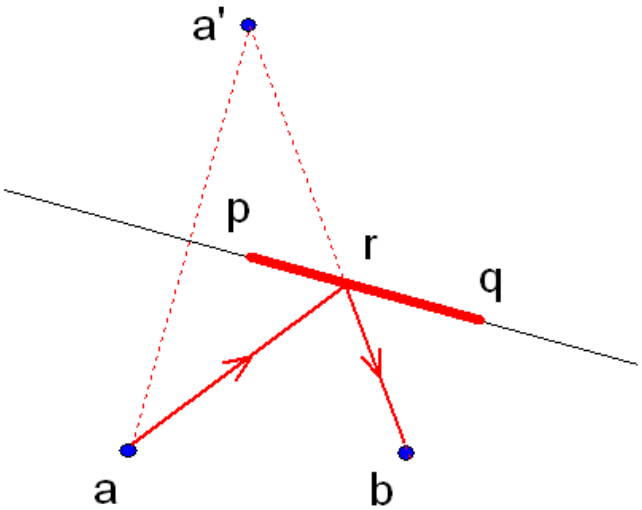


图 5 镜面反射原理示例

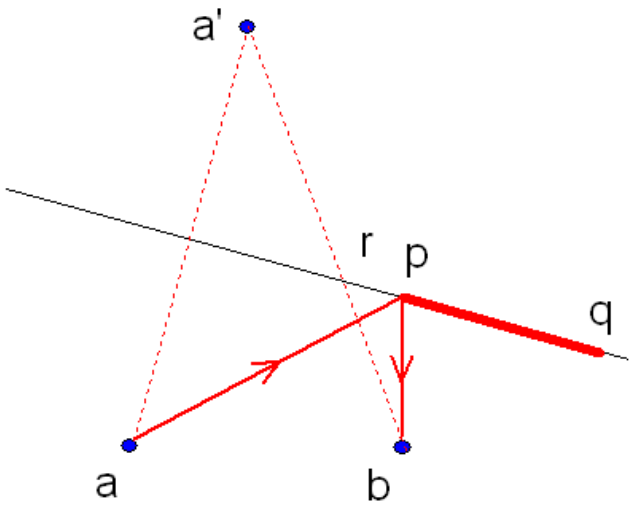


图 6 镜面反射原理示例

1）点 r 在线段 pq 内，则下一条路径可能的方向被完全确定，如图 7(1) 所示

2）点 r 与顶点 p 重合，设 p 的前一点为 w，则可细分为如下的四种情况

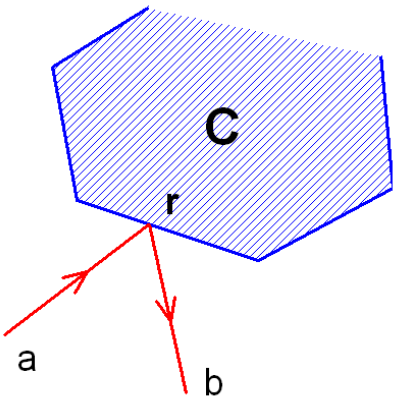
2.1）点 w 在直线 ap 右侧且 wp 是笼子 C 的边界。情况如图 7(2) 所示，此时下一条路径可能的方向由以 p 为顶点，$r_1 r_2$ 两条射线所夹成的光束所确定。其中的射线 r_1 是射线 ap 在线段 pq 上的反射结果

2.2）点 w 在直线 ap 左侧且 wp 是笼子 C 的边界。情况如图 7(3) 所示，此时下一条路径可能的方向由以 p 为顶点，$r_1 r_2$ 两条射线所夹成的光束所确定。其中的射线 $r_1 r_2$ 分别是射线 ap 在线段 pq 和 pw 上反射的结果

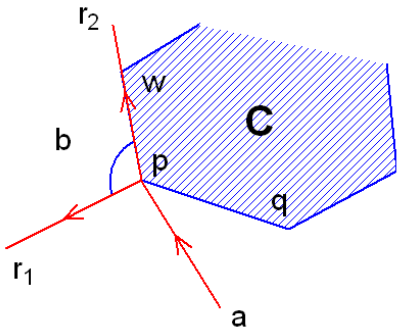
2.3）线段 wp 不是笼子的边界，且 w 在 r_1 左侧。这里的 r_1 是射线 ap 在线段 pq 上的反射结果，如图 7(4) 所示。这种情况下，当前路径是死路，无论往哪里走，都不可能是满足条件的最短路径

2.4）线段 wp 不是笼子的边界，且 w 在 r_1 右侧，如图 7(5) 所示。此时下一条路径可能的方向由以 p 为顶点，$r_1 r_2$ 两条射线所夹成的光束所确定。其中的 r_1 是射线 ap 在线段 pq 上反射的结果，而 r_2 就是 pw 射线

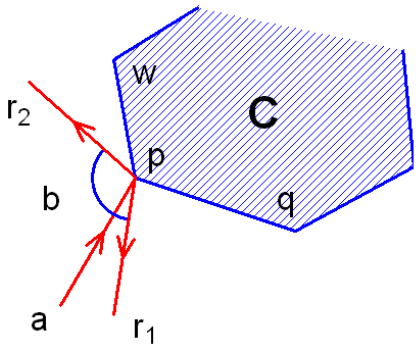
3）点 r 与顶点 q 重合，处理方法同 2）类似



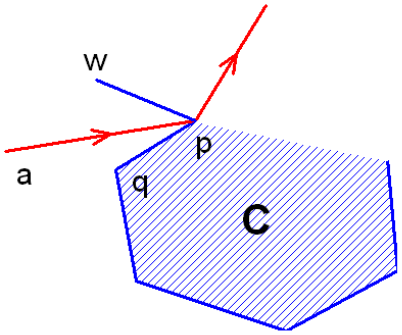
(1)



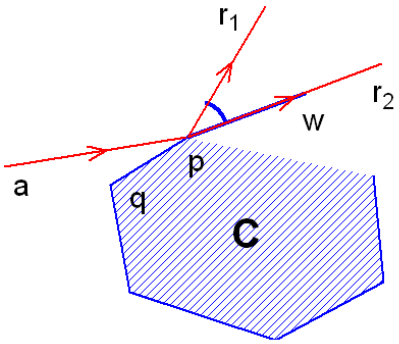
(2)



(3)



(4)



(5)

图 7 镜面反射原理示例

得到上述版本的镜面反射原理后，我们就可以对光束的反射操作展开讨论了。

2.4 光束的反射

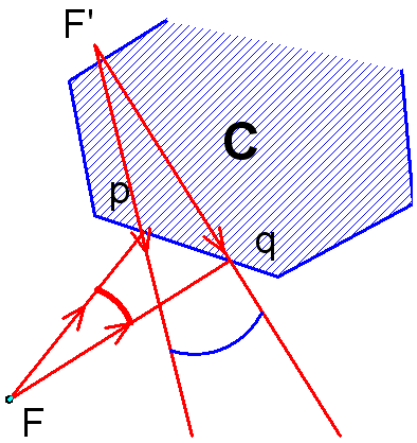
对于照射到笼子 C 的光束 F 而言，它代表了从起点 S 到笼子 C 且接触前面各个笼子的一组最短路径。那么，这些路径到达笼子 C 后，下一步会如何前进？从 **2.3** 的镜面反射原理来看，这些路径下一步的可能性在某种程度上是完全确定的。事实上，从 F 我们可以得到后续的一组光束，它们描述了这组路径下一步所有可能的前进方向。我们把这些光束称为 F 在笼子 C 上的反射光束，旧的 Flist 会被这些光束所替代，同时它们也将被加入到 Ptree 中去，并且以产生它们的那个光束为父节点。使用上面的镜面反射原理，我们需要考虑如下的四种情况

1）光束 F 完全照射在笼子 C 的某边 pq 内，如图 8(1) 所示。此时 F 产生的反射光束只有 F' 一个。F' 的顶点和射线由 F 的顶点和射线对 pq 反射而得

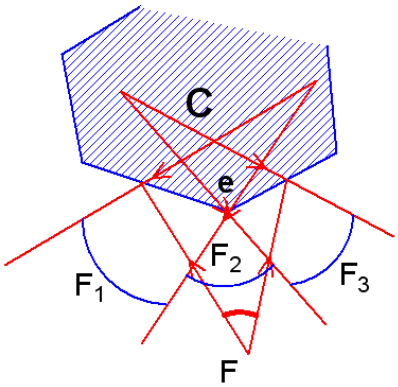
2）笼子 C 的某个顶点 e 落在光束 F 照射范围内，如图 8(2) 所示。此时，F 产生的反射光束有三个。除了像 **1）**中那样对 e 两侧线段反射得到两个光束 F_1 和 F_3，还有以 e 为顶点的光束 F_2，它为 F_1 和 F_3 所包夹

3）逆时针方向来看，光束 F 是照射到笼子 C 的最后一个光束，如图 8(3) 所示。此时，除了像 **1）**中那样对线段发射得到光束 F1 外，对笼子 C 未被照射的各个顶点也要加上相应的光束，从前面的镜面反射定律来看，它们也都是可能的前进方向

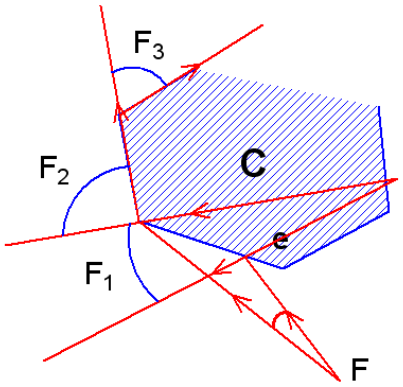
4）逆时针方向来看，光束 F 是照射到笼子 C 的第一个光束，同 **3）**类似处理



(1)



(2)



(3)

图 8 光束反射情况

把上述三种情况综合起来，不难得到定理 **I**：设笼子 C_i 被 m 个光束组成的光束序列所照射，那么经过反射操作后，新光束序列的大小最多为 $m+2n_i$，这里的 n_i 是笼子 C_i 的顶点数目。由于所有笼子总的顶点数目线性，因此无论哪个时刻，Flist 的大小都是至多线性的。相应的，不难证明每一次反射操作，时间复杂度都至多是线性的。

2.5 光束的照射

在上面一节中，我们知道了如何把射向 C_i 的光束序列更新为反射光束序列。那么是否可以马上对这些光束进行下一步的反射操作呢？这里的一个问题在于，其中的很多光束并没有照射到笼子 C_{i+1} 上。为此，我们必须把这些无效光束删除。此外，还有一些新的光束会被创建并插入到 Flist 中去。算法中我们把 Flist 的上述更新过程称为光束序列的照射。为了实现照射，我们需要 Bounding Path 的帮助。

这里的 Bounding Path 由一组多边形 Q 顶点间的最短路径组成。按顺时针顺序，设笼子 C_i 的第一个顶点为 p_i 最后一个顶点为 q_i，则我们需要 pp_1、pq_1、p_iq_{i+1} 和 q_ip_{i+1} 这些最短路径，所有的这些最短路径就被称为 Bounding Path。其中，从 pp_1 和 pq_1 可以得到最初从 p 射向第一个笼子 C_1 的 Flist。而 p_iq_{i+1} 和 q_ip_{i+1} 则被用来进行 C_i 到 C_{i+1} 的照射操作。称它们为 Bounding Path 的原因在于，对 C_i 上任意一点 a 及 C_{i+1} 上任意一点 b，从 a 到 b 的最短路径一定被 p_iq_{i+1} 和 q_ip_{i+1} 所包夹，如图 9 所示。

对 Bounding Path，文献[2]中给出了如下的定理 **II**：所有 Bounding Path 的总长度

是线性的。由于多边形两个顶点之间最短路径可在 O(logn+K) 时间内获得，这里 K 为最短路径的长度。所以获得所有的 Bounding Path 只需要 O(nlogn) 的时间复杂度就可以全部计算出来。

在照射过程中，我们需要删除那些没有照射到笼子的光束。有了 Bounding Path 的支持后，我们只需要删除那些完全被 Bounding Path 所遮挡的光束就可以了。图 9 给出了照射操作的例子。从笼子 C_i 反射出了 7 个光束，其中的光束 1、2、5、6、7 由于完全被从 C_i 到 C_{i+1} 的 Bounding Path 即 p_iq_{i+1} 和 q_ip_{i+1} 所遮挡，所以被删除。而剩下的光束 3 和 4 部分照射到了笼子 C_{i+1}，所以其射线会被修正。同时，两个新的光束也会被加入到光束序列中去。

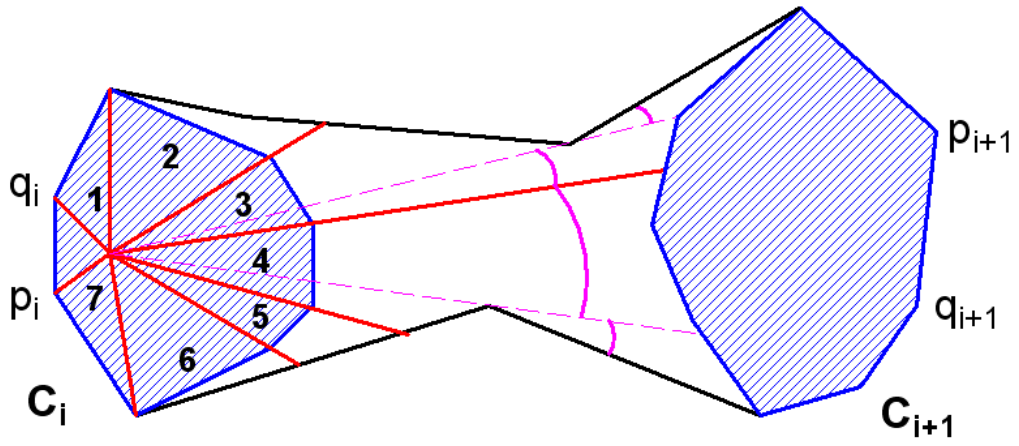


图 9 照射操作示例

2.6 空间复杂度及其对策

在 **2.3** 中，我们把 Ftree 和对多边形 Q 的分割作为预处理的结果，使得路径查询可以在 O(logn+K) 的时间复杂度内完成。但这里存在的一个重要问题在于，Ftree 的大小一般情况下不是线性的。简单来看，在 **2.4** 中，我们指出每个时刻的光束序列 Flist 大小至多线性，但实际上它们也是可以达到线性的。另一方面，笼子个数也可以达到线性，因此对某些特殊的 Zoo，Ftree 中光束总数可以达到 n^2 的数量级，这显然是不能令人满意的。

为了解决这个问题，我们需要对 Ftree 进行精简。简单的说，我们将删除其中的一些节点，把 Ftree 精简为另一个树型结构 Ptree。与此同时，输出另一个数据结构 Itree。同前面的 Ftree 不同，后两个数据结构都是线性的，从而使空间复杂度降为线性。

这里 Itree 的全称是 Interval Tree。对 n 条 R 轴上的线段，我们可以进行预处理得到相应的 Itree，其大小线性。对于 R 轴上一点 x，可以用 Itree 进行查询，得到所有包含点 x 的线段，其时间复杂度为 O(logn+K)。这里 K 为返回的线段数目。具体 Itree 的实现方法可以参见文献[4]。

这里 Ptree 的全称是 Generating Point Tree，它只包含 Ftree 中的部分节点，是后者的精简版本。具体地说，所有那些顶点不是多边形顶点的光束被删除了。事实上，这些被删除的光束就是那些照射在笼子边上反射出来的光束，即图 8(1) 中的 F'，图 8(2) 中的 F_1 和 F_3 以及图 8(3) 中的 F_1。不难证明，Ptree 中剩下的那些光束的顶点要么是笼子的顶点，要么是 Bounding Path 的顶点，总数保持线性。相应的，Ptree 的空间复杂度自然也就是线性的了。

剩下的问题在于，我们如何通过这两个线性大小的数据结构，同样以 O(logn+K) 时间复杂度得到最短路径呢？同 **2.2** 中类似，我们首先进行 Point Location，然后通过相应的叶子节点对 Ptree 进行上溯，得到相应的点列。但同 **2.2** 中不同，这个点列并不是完整的最短路径，它缺少了在笼子边界上反射的那些接触点。如果只知道点 a 经过笼子 C_i 的反射得到点 b，我们是无法确定接触点的位置的。为此，我们需要 Itree 来告诉我们这个反射是在 C_i 的哪一条边上进行的。

由此，我们就知道 Itree 的作用了。在得到 Ftree 和 FinalList 后，我们可以给后者的每一条光束加上编号 $F_1 \ldots F_N$。利用 Ftree，对每个笼子 C_i 的第 j 条边 e_{ij} 都可以得到线段 $[b_{ij}\ c_{ij}]$。这条线段的意义在于，对其中的每个整数 m，从光束 F_m 对 Ftree 进行上溯得到的那个序列在 C_i 的第 j 条边即 e_{ij} 上进行了反射。由于笼子的边数线性，线段总数是线性的。通过预处理可以得到相应的 Itree 数据结构，其空间复杂度线性。

2.7 算法描述和分析

通过上面的解释，我们可以得到预处理和路径查询的算法描述如下

```
预处理算法
{
    计算所有的 Bounding Path
    初始化 Ftree
    利用 Bounding Path 初始化 Flist
```

```
    for ( i = 1 … k )
    {
        将 Flist 在笼子 Ci 上进行反射操作
        将 Flist 照向笼子 Ci+1
    }

    利用 Flist 中的光束对多边形 Q 进行分割，并预处理
    利用 Ftree 和 Flist 得到 Ptree 和 Itree
    返回 （ 多边形 Q 分割的预处理结果、Ptree 和 Itree ）
}

路径查询算法
{
    通过 Point Location 找到照射终点 q 的光束 F
    由 F 得到 Ptree 的叶子节点，由此节点上溯得到点列 Plist
    使用 F 的编号对 Itree 进行查询，得到线段的集合 Segs
    从 Plist 和 Segs 得到从 p 到 q 的最短路径
}
```

最后，对算法的时间复杂度给出简单的分析。在 **2.6** 中我们已经指出，通过 Ptree 和 Itree 的使用，路径查询的时间复杂度保持为 O(logn+K)。那么预处理的时间复杂度呢？不难看出，其中最消耗时间的步骤是反射操作。光束序列的大小是线性的，而笼子的数目也是线性的。因此，光束的反射操作需要 $O(n^2)$ 的时间复杂度，这也就是算法预处理部分的时间复杂度。

在后面“为解决的问题”中，我们将介绍文献[2]中作者使用的一种数据结构，它可以使预处理的时间复杂度有效得降低为 O(nlogn)。

3. 系统设计

程序从结构上分成前端显示部分和后端处理部分，前端显示负责测试数据的生成和结果的显示，后端程序主要负责数据的处理。程序设计语言采用 C++, 编译器为 Microsoft 的 Visual Studio C++ 6.0.

基本几何元素(点、线段、射线、方向等)的数据结构采用 CGAL 库的定义，同时对这些数据结构的基本操作也利用了 CGAL 库提供的函数。

3.1 程序前端主要模块

程序前端部分的组成及其关系，如图 10 所示。本模块的设计目标是界面友好，负责生成合理的测试数据，并在数据处理完后，能够清晰的显示结果。程序最终界面如图 11 所示。

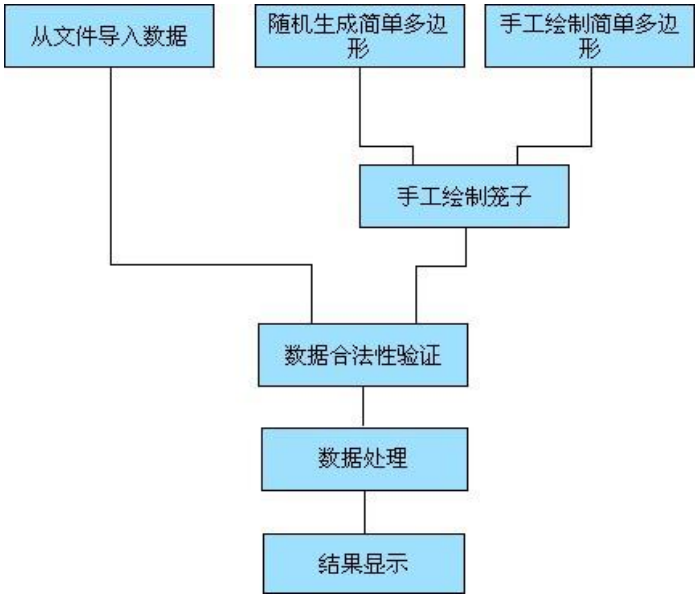


图 10 程序前端结构示意图

3.2 程序后端流程

根据算法原理部分的介绍，我们不难得到后端数据处理的流程，如图 11 所示。这部分是程序的核心，设计目标是正确、稳定的处理前端送来的测试数据并返回结果。

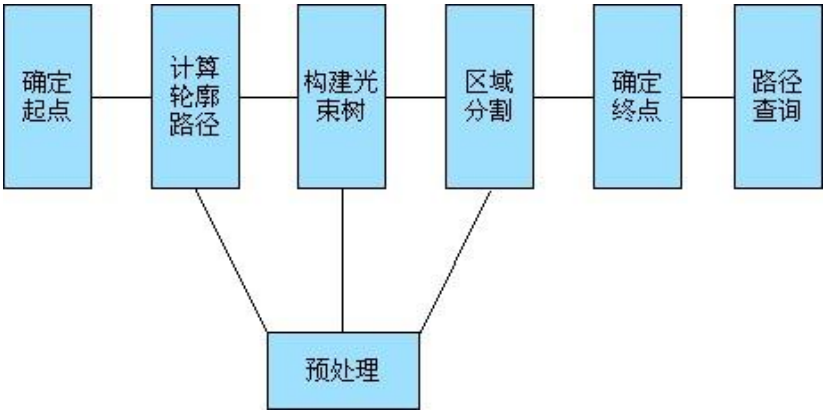
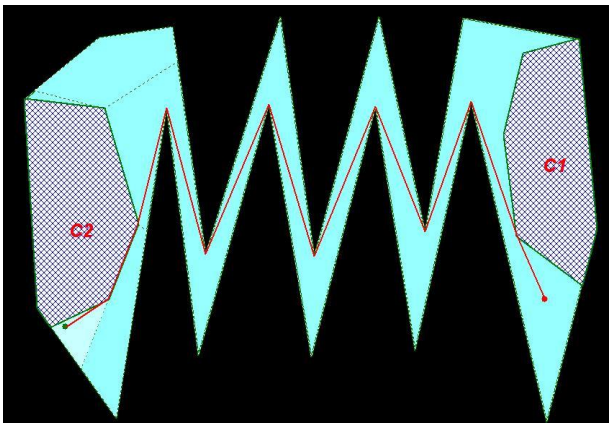


图 12 数据处理流程

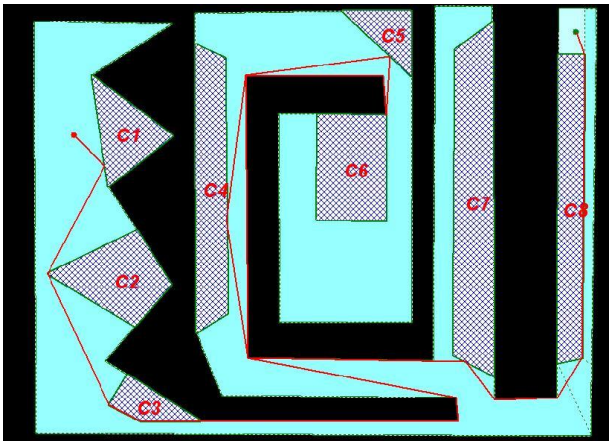
4. 系统测试及结果

4.1 正确性

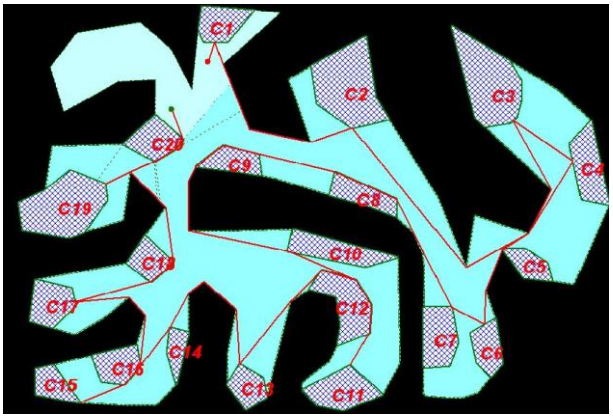
测试了大量的例子，没有发现计算错误的情况，图 13 列举几个示例。



(1)鳄鱼的嘴巴



（2）北京动物园



(3)Mike 工作的动物园

图 13 求解例子举例

4.2 效率

若干例子的运行时间统计如图 14 所示，机器主频 1.51GHz,内存 256M。随着顶点数的增加，预处理所需要的时间增长很快，这是因为在实现的过程中，有些步骤并没有达到理想的效果，这在文章的第 6 部分由介绍。

顶点数	27	37	41	44	51	54	72	84	91	140
预处理时间/s	0.1	0.3	0.4	0.6	0.9	1.1	1.5	3.5	4.3	17.0

图 14 运行效率

5. 问题及对策

1) 在实现该程序的过程中，很多地方都遇到了精度的问题，为了避免该问题，我们保存多边形的时候，并没有直接保存点的坐标，而是保存了点的序号，这样就能保证在判断是否同一个点时达到 100%的准确性。

2) 算法的问题，在求轮廓最短路径(BoundingPath)时,采用了求解简单多边形内任意两点最短路径的算法，但是当起始点可以在多边形边界上时，该算法有时会出错，这是因为：原算法中当 funnel 的 tailchain 的最后一个端点可以“直接看到”终点时，就不必再利用 funnel 的生长过程来求取路径了，直接把终点和 tailchain 连接起来就可以了，如图 15 所示，此处 A 点可以 “直接看到”b,意思就是线段 ab 不与多边形的任何一条边相交。但当起点和终点是多边形的顶点时，如果仍使用上述“直接看到”的定义，当 tailchain 走到图 16 所示的位置时，就会直接连接到终点，显然符合可以直接看到的定义，但不是我们需要的，这样我们需要对“直接看到”新的定义，约束两点连线必须在多边形内部，这可以通过判断线段的中点是否在多边形内部来做到。

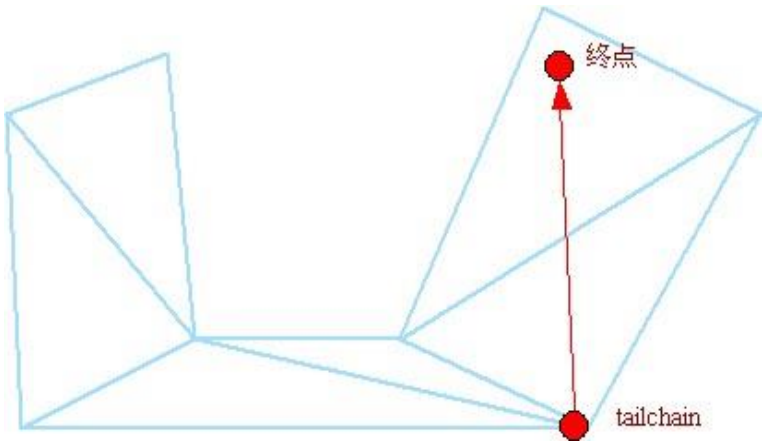


图 15 当 tailchain 能直接看到终点，就不必生长 tailchain

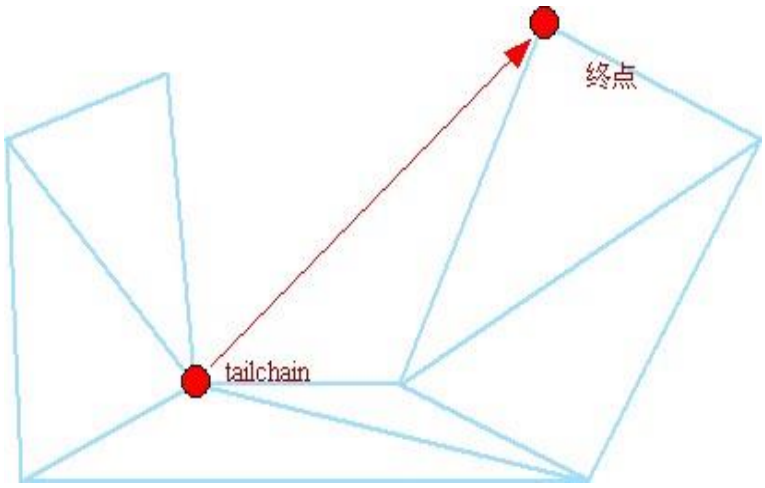


图 16 起始点在多边形边界时，须重新定义“直接看到”

6. 未解决的问题

在算法描述部分中指出，我们的预处理算法需要 $O(n^2)$ 的时间复杂度。虽然在一般的测试数据中，预处理还是很快的，但随着输入数据规模的增加，预处理所需要的时间会上升得很快，这一点在测试部分已经给出了很好的说明。为此，寻找更为快速的预处理算法是非常必要的。

导致 $O(n^2)$ 时间复杂度的主要原因就是反射操作。在文献[2]中，作者将我们使用的那种简单的光束序列 Flist 替换为一个类似于平衡二叉树的结构，即 Floodlight Tree，从而可以在 O(nlogn) 的时间复杂度内完成同样的预处理任务，把它简称为 FT。我们希望对这个数据结构进行简单的介绍，然后指出它是如何在 O(nlogn) 时间内完成所有反射操作的。

首先，FT 是一棵平衡二叉树，在叶子节点上存放了当前的光束。比较特殊的地方在于它的内部节点，每个节点都存放了一个平面变换（包括平移、旋转和反射操作）。假设某个叶子节点存放的光束是 v，从叶子节点上溯直到根节点，可以顺序得到各个内部节点存放的平面变换 m_1、m_2 … m_r。重要的一点在于，这个叶子节点实际代表的光束并不是 v，而是光束 v 顺序经过 m_1、m_2…m_r 变换后得到的结果。为什么要这样麻烦的去存放光束呢？原因在于，为了对某个光束进行反射操作，并不一定要对叶子节点上的 v 进行，而只需要对从 v 到根节点中的某一个内部节点进行反射操作就可以了。

具体反射操作的方法在于，我们不是对光束逐个进行反射操作，而是对笼子的每条边逐个进行反射操作。由于光束在 FT 中是按照逆时针顺序排列的，所以其中在笼子 C 的某条边 pq 上反射的光束必然是其中连续的一段。特别的，我们可以用 O(logn) 时间查找到端点 p 和 q 所在的光束 F_p 和 F_q，则某个光束在 pq 上反射当且仅当它位于 F_p 和 F_q 之间。比如说，图 9 给出了一棵简单的 FT。假设 F_p 是其中的第 2 个光束，F_q 是其中的

第 7 个光束，那么我们需要将 2 – 7 共六个光束针对线段 pq 进行反射操作。关键的地方在于我们不需要对 2 – 7 这些叶子节点上的光束逐个进行反射操作。而只需要对其中一些子树的根节点进行反射操作就可以了。比如说，为了对 3 – 6 这 4 个光束进行反射操作，只需要对图中那两个标为 R 的内部节点进行反射操作就可以完成。事实上，这里使用的技巧在课上介绍的各种算法中也曾经用到过很多次。不难证明，通过这种方法，对笼子的每条边，只需要 O(logn) 即树的深度就可以完成反射操作。而笼子的总边数当然是线性的，因此完成所有的反射操作也只需要 O(nlogn) 的时间复杂度就可以了。

由于我们时间有限，再加上 Floodlight Tree 这个数据结构也比较复杂，因此并没有在我们的程序中得到实现。

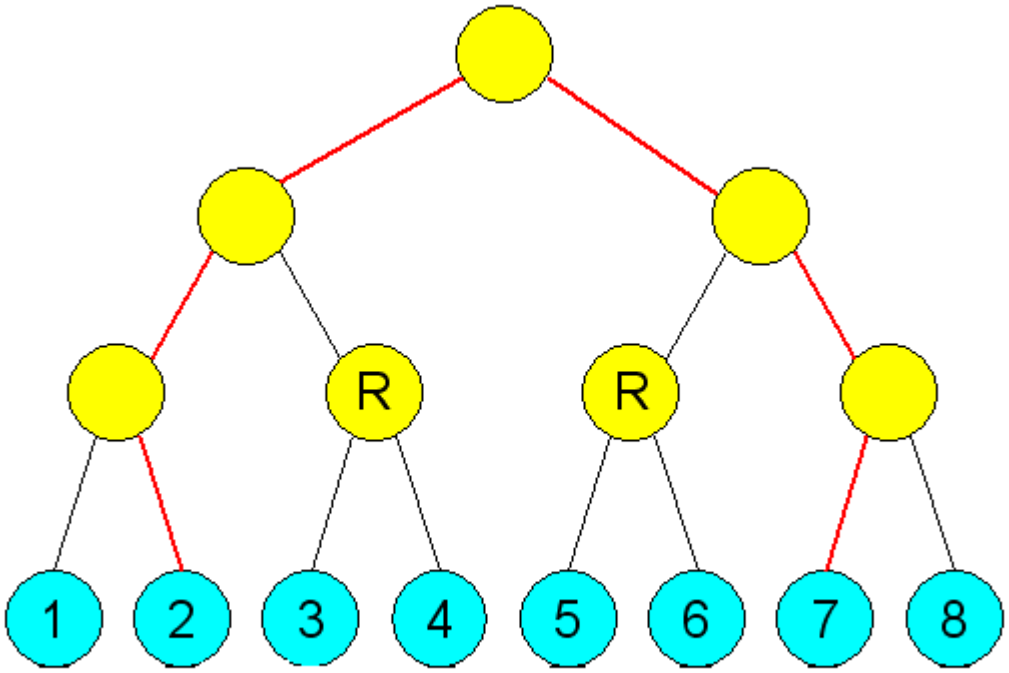


图 17 反射操作示例

7. 文献及代码来源

[1]W.Chin,S.Ntafos,The zoo-keeper route problem,Inform.Sci.63(1992)245-259.
[2]Sergei Bespamyatnikh. An O(nlogn) algorithm for the zoo-keeper's problem, Computational Geometry 24(2003)63-74.
[3]L.J.Guibas,J.Hershberger.Optimal shortest path queries in a simple polygon, J.Comput.Syst.Sci39(2)(1989)126-152

[4]H.Edelsbrunner,Dynamic data structures for orthogonal intersection queries,Report F59,Inst.Informationsverarb.Tech.Univ.Graz,Austria,1980.

代码来源：

1) 基本数据结构，以及对这些结构的基本操作来源于 CGAL。
2) 三角形 Earcutting 剖分，来自 Computational Geometry in C (Second Edition)的附录代码。