北京大学

数学科学学院

算法与数据结构课程

上机报告

基于网络的景点平面图系统

2011/12/20

陈 霄 泓

指导老师：孙 猛

目　录

问题描述与需求分析

原始问题描述

这里我们要设计一个具有简单的导游功能的程序，程序应当能提供以下功能：

1.查询各景点的相关信息
2.查询任意两个景点之间的最短路径
3.对于给出的路径长度上界 M，查询任意两个景点之间长度不大于 M 的所有路径
4.增加、删除、更新有关景点和道路的信息
5.对于多个景点，给出最佳游览路径

需求分析

上面的需求可以说还是很“原始”的。因此，解决问题的第一步是要对模糊的原始需求进行精化，形成一个定义明确的需求。经过分析，我认为要求实现的程序应当能够提供如下的核心功能：

1. **查询功能**
   a) 查询景点信息
      i. 查询单个景点的简介
   b) 查询路径信息
      i. 查询通过两个景点之间的一条最短路径
      ii. 查询通过多个景点之间的一条最短路径
      iii. 查询通过两个景点之间的全部限长路径
2. **编辑功能**
   a) 编辑景点信息
      i. 增加一个景点
      ii. 删除一个景点
      iii. 修改一个景点的信息
   b) 编辑道路信息
      i. 增加一条道路
      ii. 删除一条道路
      iii. 修改一条道路的长度

抽象数据类型的设计

在形成明确的需求之后，便要着手设计抽象数据类型。我认为，程序中核心的抽象数据类型有四个，它们分别是：景点（Plot）、道路（Road）、路径（Path）、图（Graph）。

下面对每一个抽象数据类型进行分析。

景点（Plot）

景点是程序中最基本的逻辑单位，一切操作最终都应归结于对景点的操作，由此可见对景点的操作将会是使用频繁的。从实际情况上分析，景点不应该同名，因此将景点名字作为关键码是合理的[1]。由于景点本身属性十分简单，因此可以比较容易的写出它的抽象数据类型的定义。

```
ADT Plot is
    operations:
    /*根据名字和简介创建景点*/
    Plot creat_Plot(char *name, Introduction intro);
    /*获得景点的名字*/
    char *getName_Plot(Plot plot);
    /*介绍景点*/
    void introduce_Plot(Plot plot);
    /*重命名景点*/
    void renewName_Plot(Plot plot, char *name);
    /*更新景点的简介*/
    void renewIntrduction_Plot(Plot plot, Introduction intro);
end ADT Plot
```

道路（Road）

道路是基于景点之上，用来表示景点之间关系的抽象数据类型。它同样是图（Graph）的基本元素之一。根据需求，这里的道路是有正权值无向的道路。一条道路可以由其两个端点唯一的指定，因此道路的关键码可以看成是景点关键码的一个二元组。道路包含的信息即为道路的长度。综上述，可将道路抽象数据类型定义为如下形式：

```
ADT Raod is
    operations:
    /*创建一条道路*/
    Raod creat_Road(double length, Plot plot1, Plot plot2);
    /*获得道路的两个端点*/
    void getEndpoint_Road(Road road, Plot *p1, Plot *p2);
    /*获得道路的长度*/
    double getLength_Road(Road road);
    /*更改道路的长度*/
    void renewLength_Road(double newlength);
end ADT Road
```

路径（Path）

程序中相当重要的功能是对路径的查询操作。从需求来看，路径中有可能会出现环（例如，当计算通过多个景点的最短路径时，可能需要多次经过一个景点），但是路径是无向的。一条路径的关键码可以看成是沿路径上各个景点组成的有序序列，不妨称其为路径的景点序

[1] 事实上在程序中，图中的一个景点拥有两个关键码，分别是景点的名字和景点在图中景点表中出现的位置。使用这两个关键码都可以完成对景点的检索和其他的进一步操作，程序也确实提供了这些功能。但应注意的是，景点在景点表中的位置不是永远固定的，如果图中的景点是动态变化、处于不断被添加和删除的过程中的，那么景点在景点表中的位置也将随之变化。即景点名字为主关键码，景点在表中的位置为次关键码。

列。路径的长度或者权定义为顺次经过路径时，相邻两个景点之间道路长度的代数和。

从本质上说，路径不是程序内部需要的数据类型，而是作为结果呈现给用户的。因此对路径很重要的一个操作便是展示（display）一条路径。

路径的抽象数据类型如下定义：

```
ADT Path is
    operations:
    /*获得路径的权*/
    double getLength_Path(Path path);
    /*获得路径中景点的个数(景点序列的长度)*/
    int getPlotsNumber_Path(Path path);
    /*获得沿路径上的景点(景点序列)*/
    Plot *getPlotsAlong_Path(Path path);/*return an array of Plot*/
    /*展示一条路径*/
    void display_Path(Path path);
end ADT Path
```

图（Graph）

图能提供的操作非常丰富，程序的一切功能都应在图的操作中有体现。从需求分析，这里的图是带权的无向图，亦即是网络。特别的，图中每条道路的权都是一个正实数。

图的抽象数据类型可以定义如下：

```
ADT Graph is
    operations:
    /*查询景点*/
    int findPlot(Graph graph, char *plotname, Plot *p);
    /*查询路径*/
    int findPath(Graph graph, Plot plot1, Plot plot2, Path *p);
    /*查询最短路径*/
    int findShortestPath(Graph graph, Plot plot1, Plot plot2, Path *p);
    /*查询所有路径*/
    Path *findAllPath(Graph graph, Plot plot1, Plot plot2);
    /*查询限长路径*/
    Path findAllPathWithLimitedLength(Graph graph, Plot plot1, Plot
plot2, double M);
    /*判断两景点之间是否有路径连结*/
    int isConnected(Graph graph, Plot plot1, Plot plot2);
    /*查询通过多个景点的最短路径*/
    Path findShortestPathAlong(Graph graph, int plotnumber, Plot
plotlist[]);
    void addPlot(Graph graph, Plot plot);
    void removePlot(Graph graph, Plot plot);
    void renewPlot(Graph graph, Plot plot);
    void addRoad(Graph graph, Road road);
    void removeRoad(Graph graph, Road road);
    void renewRoad(Graph graph, Road road);
    /*展示一张图*/
    void display_Graph(Graph graph);
end ADT Graph
```

存储结构设计

这里来着重讨论图的存储结构。

从之前的抽象数据类型定义中可以看出，程序中的图是典型的网络，并且操作大多都与路径的检索和查找有关，因此采用抽象数据类型图来实现程序中的图是合理的选择。

抽象数据类型图最常用的两种存储结构为邻接表表示和邻接矩阵表示两种方法。考虑到实际情况中，游览图中，景点与景点之间的道路连结可能会很密集，同时在程序中还提供了丰富的对路径的检索操作，而路径的操作又是基于道路的查找的，因此存储实现必须能支持对道路信息的快速查询。综上所述，采用邻接矩阵表示法来表示程序中的图类型是比较合理的[2]。

此外，程序中还提供了对景点的检索功能。当图中包含的景点比较多的时候，合理的组织图中景点表的结构对于程序整体的运行效率有很大影响，因此我使用了散列表[3]来组织景点表。当然对于此次大作业，图中保存的是北京大学的景点游览图，景点个数不会很多，此时简单的用线性表来组织景点图也是可以接受的。

模块划分与抽象数据类型的实现

基于分模块开发大程序的思想，我在对需求进行彻底分析的基础上，决定将程序分解成两大模块。一方面是景点图模块，支持各种具体的查询、检索、编辑操作。另一方面是命令模块，用来控制程序流程，交互式接受用户命令，返回相应操作的结果。下面具体讨论各个模块各自的具体实现方式。

命令模块

考虑到程序是交互式的，通过用户的命令来完成各种操作，因此设计出命令模块来完成对主函数中流程的控制。加入命令模块后，主函数的主体部分便成为如下的形式：

```
while(commander())/*交互式读取命令*/
       ;
```

commander 函数

commander 函数的工作是从用户处获取一组命令，完成相应的工作，然后将结果返回给用户。当然这一切工作都是通过调用其他函数来完成。函数 commander 的返回值说明程序应该等待用户的下一条命令还是结束程序。

分析 commander 函数的工作可以发现，commander 函数完成的工作可以分解为两个子工作：

```
1.从用户处获取命令，返回相应的命令名称
2.根据命令名称，调用相关的函数完成工作，返回结果
```

从而自然的，可以设计两个子函数来完成上面两个子工作，这样 commander 函数可以定义成下面这个形式：

---

[2] 参见本文第 13 页，存储结构的设计与选择。

[3] 参见本文第 8 页，散列表。

```
int commander(Graph *pgraph)
{
    CMDTYPE cmdtype = get_explainCommand();
    int executeResault = executeCommand(pgraph, cmdtype);
    return executeResault;
}
```

其中 CMDTYPE 是一个整型类型，其中的元素取值为一些事先定义好的枚举常量，用来指示 executeCommander 函数完成不同的工作。

get_explainCommand 函数

这个函数从用户处读入命令，然后解释这些命令，返回命令相应的编号。这里实际上就需要对命令的格式做出一些约定。首先为了降低程序的复杂度和实现起来的难度，我假设景点的简介中间没有换行，景点的名字中间没有空白字符。

提供的命令见下表所示：

命令编号	命令格式	说明
EXIT	exit	保存并退出程序
DISPLAY	display	展示图中所有景点的名字及其编号[4]
ADD_PLOT	add_plot	添加景点
DELETE_PLOT	delete_plot 景点[5]	删除景点
MODIFY_PLOT_N	modify_plot_n 景点	修改景点名字
MODIFY_PLOT_I	modify_plot_i 景点	修改景点简介
SEARCH_PLOT	search_plot 景点	查询景点的全部信息
ADD_ROAD	add_road 长度[6] 景点 1 景点 2	添加道路
DELETE_ROAD	delete_road 景点 1 景点 2	删除道路
MODIFY_ROAD	modify_road 长度 景点 1 景点 2	修改道路长度
SEARCH_PATH_L	search_path_l 长度 景点 1 景点 2	查询全部限长路径
SEARCH_PATH_S	search_path_s 景点 1 景点 2 ... 景点 k	查询一条最短路径

executeCommand 函数

这个函数根据命令名称，执行命令然后将结果返回。函数的主题部分是一个 switch 语句，对不同的命令名称分类处理。执行命令时，调用图模块中提供的函数完成操作。

图模块

可以说上面的命令模块仅仅是一个框架，一切交互式的程序中都可以使用命令模块来控

[4] 此处景点的编号意即为景点的次关键码，下文中凡出现景点编号一词均按如上理解。
[5] 这里的景点参数可以使用主关键码—名字，也可以使用次关键码—景点表中的位置来指定。下面凡是要求指定一个景点而又没有指明指定方式的，均按如上理解。
[6] 这里的长度参数应能够被程序理解为一个浮点数。例如可以提供一个整数或者带小数点的浮点数等。

制函数流程[7]，而程序功能的具体实现则是由图模块来完成的。一方面，图模块要能够实现命令模块中要求实现的各项功能。另一方面，图模块要提供合适的函数借口，以便于命令模块调用。最后，图模块要能将共享的数据和私有的数据区分开，以防止外部函数不经过提供的接口函数直接访问图中的数据。

为了达到上面所述的目的，我采用了多个源文件开发程序的方法。将图模块对外提供的接口定义在头文件 graph.h 中，而将其实现在源文件 graph.c 中，并且将图模块的私有数据定义为 static 存储类型，使其只具有文件作用域，以防止其他源文件越界访问数据。

算法设计及分析

下面着重来讨论程序中重要的算法。

散列表

前面已经说过，考虑到程序中提供了对景点的查询操作，因此程序使用了字典结构来组织景点线性表。具体地说，即是将线性表组织成一张散列表，通过散列函数来快速定位景点在景点表中出现的位置，从而希望将查询景点的时间代价减少为$O(1)$。因此，设计一个好的散列函数是十分重要的。而一个好的散列函数应该满足简单一致散列的假设，也就是说散列地址能独立地等概率的全覆盖到基本区域。

此处的关键字是景点的名字，即一个字符串。特别的，这个字符串中可能存放着一个中文的字符串。第一步便是要将这个字符串解释为一个自然数。这里当然有许多解释方法，我不假思索的选择了其中比较容易实现的一个。其次便是将这个自然数散列到基本区域的地址集合中去，我同样简单地采用了取模[8]的方法。从而有如下的 h 函数的定义：

```
static int h(const Graph *pgraph, const char *s)
{
    unsigned int n = 0;
    while(*s != '\0')
        n += *(s++);
    return n % pgraph->MAXNUM;/*MAXNUM是个素数*/
}
```

定义为 static 存储类型是因为 h 函数是图模块的私有函数，不允许其他源文件中的函数调用它。

此处这个 h 函数的选择当然不一定是最好的，事实上，我并没有考察过这个函数的散列性质是否令人满意。不过从实际效果上来看，景点查询和检索的速度还是很不错的。

除了 h 函数的设计，还有检索算法的设计。考虑到可能会不断地有景点插入和删除，因此需要设计一个特殊的量来表示删除景点后形成的空位。这个空位必须区别于空指针值 NULL，因为检索算法一旦检索到 NULL 就认定检索失败了。

最终我使用的指针值是景点指针表所在的图结构的指针值 pgraph，这个值满足上面的所有要求。实际上采用景点指针表的地址也是可行的。

---

[7] 参见第 12 页，模块与模块之间的耦合性。

[8] 这里忽略了很多细节上的处理。例如如果图容量 MAXNUM 不是一个素数，那么直接取模其实并不恰当。

最后，检索算法采用开地址线性探查法，每次探查的步长间隔是一个单位长度。

限长路径的检索

这是整个程序的核心算法之一。首先来明确问题需求：已知景点图和两个景点，以及一个限制长度 M，这两个景点间的全部不超过 M 的所有路径。

先对问题进行分析，首先，从实际情况考虑，找到的路径不应该是有环的，并且路径的两端点就是给出的两个景点。其次，问题要求找出全部的路径，因此整个问题本质上是一个遍历的问题，即寻找所有从景点 1 到景点 2 的长度不超过 M 的简单路径。最后注意到景点图是一个权重为整数的图，因此任何一条满足条件的路径，其前缀路径的长度也不超过 M。根据以上特点，可以在深度优先周游算法的基础上给出限长路径搜索的算法：

```
1.建立一个路径，置景点1为路径的起点
2.寻找第一个从景点1可达的，且使新路径长度不超过M的下一个景点
3.将找到的景点放入路径，并判断此景点是不是景点2
4.如果是，则找到一条符合要求的路径
5.将此景点移出路径，放入下一个景点，循环第2步
6.当栈为空时，遍历结束

说明：算法本质上就是按照深度优先策略搜索整个图，只不过在此基础上增加了两个操作。一是当栈顶元素就是目的地景点2时，将存在栈中的路径保存起来。二是当当前路径长度已经大于限长M时，停止对此时栈顶元素的展开（即认为对此元素已经搜索完毕），从而直接进入下一个景点的搜索。
```

由于路径是无环的，因此可以用一个数组简单的实现上面的算法。具体的实现在源文件相应函数的头部有注明。

最短路径的检索

枚举法

同样，先来定义问题：在图 $G=(V,E)$ 中给出一个景点集合 $S=\{P_1,P_2,\ldots,P_k\}\subseteq V$，求一条路径 L，使得 L 的景点序列是 S 的子集。分析发现，对于给定的图 G 和景点集合 S，路径 L 有可能是有环的。事实上在某些情况下，L 必须是有环路径才能保证遍历全部的 S 中的全部景点。但是显然，L 的景点序列中的第一个和最后一个景点一定是 S 中的互异景点，并且 L 一定会按某个次序依次访问 S 中的景点。即对于任何一个 L，一定存在某个 $1,2,\ldots,k$ 的排列 $i_1,i_2,\ldots,i_k$ 使得 L 可以被看成 $k-1$ 条子路径的叠加，即：

$$L：P_{i_1}\to\cdots\to P_{i_2}\to\cdots\to P_{i_j}\to\cdots\to P_{i_k}，$$

其中第 j 条子路径 $L_j=P_j\to\cdots\to P_{j+1}$ 是从 P_j 到 P_{j+1} 的最短路径（其中可能包括其他 S 中的点）。因为如果 L_j 不是最短的，那么把 L_j 换成从 P_j 到 P_{j+1} 的最短路径之后，得到的新的

总路径L'的长度会更短，并且L'同样遍历了S中的景点，这与L是最短路径矛盾。

因此可以遍历$1,2,\ldots,k$的所有排列，对每一种排列$i_1,i_2,\ldots,i_k$计算$k-1$条最短路径长度的和。而子路径的最短路径可以由 Floyd 算法在$O(n^3)$的时间内完成，因此总的算法时间代价为$O(k!n^3)$。

此算法的时间代价是指数型的[9]。在实际测试中，对于规模为 n=23，k≤10 的输入，程序可以迅速查找到最短路径。当 k=11 时，程序需要花费大约 15 秒来查询最短路径，而当 k=13 时，程序查找路径的时间大约为 90 分钟。可以说算法的时间代价并不令人满意，我一直希望能寻找一个多项式时间的算法，但是一直没能找到。

退而求其次，可以在不改变整个算法本质的前提下，对算法的一些细节做一点优化处理，值得算法开销估计中前面的那个系数能稍微小一点。比如说利用图是无向的这一个特点，在枚举时不用考虑所有的排列，而只需要考虑其中一半的排列即可。例如，如果考虑了排列$i_1,i_2,\ldots,i_k$，就不用考虑排列$i_k,i_{k-1},\ldots,i_1$了。

进一步的思考

上面已经对程序中使用的算法进行了陈述，并且指出算法的时间代价是$O(k!n^3)$。对于这样一个算法，在实际应用中会受到极大地限制。可以说采用枚举这种蛮力算法当给定的必经景点集合S逐渐变大时，几乎是不可行的，因此寻找一个渐进意义下更好的算法是十分必要的。关于这个问题，我有下面的一些想法。

问题的定义和前一小节相同。首先我发现这个问题可以简化[10]成下面这个问题：设无向带权图$\tilde{G}=(S,\tilde{E})$，$S=\{P_1,P_2,\ldots,P_k\}$。对于任意的互异景点P_i和P_j之间都有道路直接相连，道路的权值（用$|P_iP_j|$表示）是一个正实数。并且$\tilde{G}$满足三角不等式，即对任意三个景点$P_i,P_j,P_k\in S$，恒有$|P_iP_j|\le|P_iP_k|+|P_kP_j|$。现在需要求一条通过$S$中所有点的最短路径$\tilde{L}$。

首先来说明一下为什么可以将原问题转化为上面这种形式。正如上一小节陈述的，原问题所要求的那条路径L一定会按照某种顺序依次访问S中的各个点。不妨设L是按下面所示的方式访问的：

$$L:\ P_{i_1}\to\cdots\to P_{i_2}\to\cdots\to P_{i_j}\to\cdots\to P_{i_k},$$

那么子路径$L_j=P_j\to\cdots\to P_{j+1}$是从$P_j$到$P_{j+1}$的最短路径。因此可以从从原始的图$G=(V,E)$出发来定义一个新的图$\tilde{G}=(S,\tilde{E})$，其中$S=\{P_1,P_2,\ldots,P_k\}$是必经景点集合，

---

[9] 由于$k!\sim\sqrt{2\pi k}\left(\frac{k}{e}\right)^k$，因此这个时间代价从渐进意义上说比指数级别更差。

[10] 仅仅从形式上看，问题的叙述更简单，条件似乎也加强了。

$\widetilde{E}=\{(P_i,P_j)\mid$在图$G$中存在从$P_i$到$P_j$的路径$\}$是边集合，并且图$\widetilde{G}$中两景点$P_i$和$P_j$之间的道路长度就定义为图$G$中景点$P_i$和$P_j$之间的最短路径长度。按此定义可知，如果图$G$是连通的，那么图$\widetilde{G}$自然就满足任意两个景点之间有道路相连。并且由于图$\widetilde{G}$中两景点之间的权定义为图$G$中的最短路径长度，因此图$\widetilde{G}$中自然成立三角不等式。这样我们就把原问题转化为一个最短路径遍历图$\widetilde{G}$的问题了。

其次是来思考有没有合适的算法解决这个问题。图$\widetilde{G}$具有一般的图没有的性质，它既是满足三角不等式的，同时也是一个充满边的稠密图。如果把$\widetilde{L}$看成是一个满足某些最短性质的$\widetilde{G}$一个子图，则很自然的会想到去使用类似于求最小生成树的方法来求$\widetilde{L}$。但很可惜这样是行不通的，因为求最小生成树的两个有效算法 Prim 算法和 Kruskal 算法都是基于最小生成树的 MST 性质，通过贪心法或者动态规划法来实现的[11]，但是仅仅满足 MST 性质并不足以保证所求得的就是一条路径，因为最小生成树有可能有分叉[12]。因此，我们如果可以寻找到一条题目所要求的最短路径的合适（递归）性质的话，就有可能利用动态规划甚至是贪心算法，从小到大来构造最短路径了。

在这样的想法的基础上，我开始寻找最短路径的性质。首先我得到的结论是最短路径$\widetilde{L}$不能是有环的，即$\widetilde{L}$可以由$1,2,\ldots,k$的排列$i_1,i_2,\ldots,i_k$来决定。证明可以用反证法，假设$\widetilde{L}$有环，例如：

$$\widetilde{L}:P_{i_1}\to\cdots\to\underline{P_{i_j}\to P_{i_{j+1}}\to\cdots\to P_{i_t}\to P_{i_j}}\to P_{i_l}\to\cdots P_{i_k},$$

那么下面这条路径：

$$\widetilde{L}_1:P_{i_1}\to\cdots\to\underline{P_{i_j}\to P_{i_{j+1}}\to\cdots\to P_{i_t}}\to P_{i_l}\to\cdots P_{i_k}$$

将会比$\widetilde{L}$的权值更小（至少一样大）[13]。

但是接下来我就遇到了困难。尽管可以像上面一样不断地使用反证法和三角不等式来证明下面一系列的结论：

设$\widetilde{L}:P_{i_1}\to\cdots\to P_{i_{s-1}}\to P_{i_s}\to P_{i_{s+1}}\cdots\to P_{i_{t-1}}\to P_{i_t}\to P_{i_{t+1}}\cdots P_{i_k}$是最短路径，则：

a) $\left|P_{i_1}P_{i_k}\right|\geq\left|P_{i_l}P_{i_{l+1}}\right|,(1\leq l\leq k-1)$，

b) $\left|P_{i_1}P_{i_s}\right|\geq\left|P_{i_{s-1}}P_{i_s}\right|,(2\leq s\leq k)$，

---

[11] 尤其是 MST 性质保证了贪心法能获得最优解。

[12] $\widetilde{L}$一般不是图$\widetilde{G}$的最小生成树

[13] 利用三角不等式，结论是显然的。

c) $\left|P_{i_s}P_{i_k}\right| \geq \left|P_{i_s}P_{i_{s+1}}\right|, (1 \leq s \leq k-1)$，

d) $\left|P_{i_1}P_{i_t}\right| + \left|P_{i_s}P_{i_k}\right| \geq \left|P_{i_{s-1}}P_{i_s}\right| + \left|P_{i_t}P_{i_{t+1}}\right|, (2 \leq s \leq t \leq k-1)$；

但是这些结论都不足以构建一个足以生成 $\tilde{L}$ 的递归定义式，因此没能形成一个有效的算法。注意到性质 a)似乎暗示我们可以选择使得 $|P_{i_s}P_{i_t}| = \max_{1\leq j,l\leq k}\left\{|P_{i_j}P_{i_l}|\right\}$ 的点来作为最短路径的两个端点，但这只是一种合情的猜测，并没有得到证实。

调试、修改与体会

上机实习的收获与感想

吸取了第一次大作业的经验，我在动手写这次的大作业之前，先花了近一个星期来分析问题，明确需求，进而来设计抽象数据类型及其存储结构。在完成核心的数据类型的定义之后，我开始对程序进行功能划分，最终确定下来两个模块，最后再分别开发两个模块。而一些具体函数，比如路径查询函数的设计，则是在整个程序的框架已经基本实现，程序已经能够运行之后，再开始研究设计算法，实现函数功能，因此开发过程总的来说还是比较顺利的。此外在这一次的程序中，我第一次尝试使用多个源文件来开发程序的技术。由于在大一的计算概论课程中并没有全面涉及这方面内容，仅仅介绍了 C 语言用于支持程序分块开发的一些语言特性，因此我可以说是在边学边写中完成作业的。从此次的实习中，我第一次感受到头文件在多个源文件之间传递信息的重要作用，第一次理解为什么要用 static 去限制函数的作用域，也第一次体会到预处理命令给程序调试带来的极大的方便。

程序中用到的许多技术基本上都是在课本中出现过的。它们使我能够更高效、更清晰的设计出具有更好可移植性的程序需要的函数来。例如使用数组来构建一个简单的链接栈（search_Path_1 函数）、使用循环来控制函数流程（commander 函数、next 函数）、使用指针实现编译时不定长多维数组的参数传递（search_path_s 函数及其子函数等），通过对课本的学习可以知道这些技术的存在，但是当自己遇到实际问题，并且在解决问题的过程中使用了这些技术之后，我发现自己对于这些技术的理解加深了，不再像以前仅仅停留在感性的认识上，而是对其在实践中具体怎样起作用有了新的认识。

程序中的进一步修改与调试

此次的程序中仍然存在着不少问题。以下简单的陈述一下我的想法。

模块与模块之间的耦合性

我认为在程序的所有问题中，最严重的一个问题就是不同模块与模块之间的耦合性还是太强，即 graph.c 和 command.c 之间的联系还是太紧密。按照我的设想，command.c 应当是一个通用的交互式获取命令的命令模块，任何对于图模块内部的修改，例如将修改图的存

储形式修改为邻接表表示，或者给图中的景点简介部分添加图片、音频、视频等新的信息，或者修改查询检索算法等等，都仅仅局限在 graph.c 中而不必对 command.c 进行任何改动。同样的，对于读入的命令格式，命令参数的形式等等规定的改动也只需要相应的改动 command.c 而不需要修改图模块。但是在这个程序中，command.c 仿佛长了一双上帝的眼睛，它能清清楚楚的看到 graph.c 中的每一步操作是怎样实现的，而且还时不时插上两手，这使得几乎不可能将 command.c 搬到其他工程中去重复使用，除非对其进行进一步的封装和加工。而与此同时，为了让 command.c 完全不关心图模块的内部构成，graph.h 中提供的接口数目比现在要大大增加，所提供的操作数目也会大大增加。但是由于提供的操作数目增加的原因在于操作变得更加细致，以前要负责两三件甚至五六件事情的函数现在只需要完成一项工作，因此函数的实现变得容易了，动作的描述、流程的控制都因为操作更加细腻而变得容易，并且写一个小（子）程序出错的概率要比写一个同时完成好多件事的大函数出错的概率小得多，因此总的来说还是划得来的。这相当于把用于调试排错的时间花在了前期的设计分析上，同时还获得了良好的模块化。

当然对于此次的大作业，现在的程序也就能够使用了。但是如果以后碰到新的实际问题也出现了图模块或者命令模块的话，我会选择重复使用现在已经写好的模块，增加更多的操作，完成更好的封装，然后用到新的环境中去。

空间资源的利用率

此外程序还有一个问题就是对空间的需求量相对比较大，这个问题最集中体现在查询最短路径的时候。因为最短路径需要使用两个矩阵，以一个容量为 100 的图为例，那么这两个矩阵所含的元素的个数都是 100^2=10000 个。在一般情况下，仅仅这两个矩阵就需要占用掉将近 250 千字节的内存空间。如果算上整个程序可能会占用的全部空间，包括图本身的信息，景点及其简介的信息，道路的信息等等，占用的空间大小大约是 750 至 1250 千字节，而实际上这个空间在大多数情况下是被浪费了的[14]。首先考虑到图是无向图，因此相应的矩阵可以只保存一个上三角矩阵，从而将空间占用量减少一半。另一方面，如果图的容量很大但是图中景点个数比较少时，可以用一个更小的矩阵来只保存景点与景点之间的道路连结信息，而不必真的去建立一个和图本身一样大的矩阵来存放道路信息。

存储结构的设计与选择

其实在一开始设计存储结构的过程中，我在邻接矩阵表示法和邻接表表示法之间确实曾犹豫过选择哪一种。但是由于算法的具体实现在邻接矩阵表示法下会更加直观容易理解，并且管理一个矩阵要比管理一张链接表所花费的心思少得多，因此我实际上不停地在劝说自己选择邻接矩阵表示法来表示景点图。现在在程序已经完成之后回过头来看，可能选择临接矩阵来表示景点图并不是一个正确的选择。其中一个主要的缘故是，景点图并不像我之前所认为的那样是一个密集图，正相反，景点图是一个典型的稀疏图，平均一个景点只和 4~5 个景点有道路直接相连，而景点的数目可能是数十甚至数百个，两者差了两个数量级。在这种情况下，邻接表表示法不但能节省大量的空间，而且并不会拖慢查询道路信息的操作。这也给了我一个教训，在今后设计存储结构时不能想当然去选择“看上去”最简单，最直观的方法。尽管直观的方法容易理解，书写程序也更加容易，但是如果存储结构并不能很好的贴合

[14] 除了下面要说的几个原因，还有一个原因是景点图实际上是一个稀疏图。具体阐述见第 13 页，存储结构的设计与选择。

实际的需求，就应当果断放弃直观性，去选择一个更适合问题所需要算法的数据存储结构。