

## 实验三: 导航软件实验报告

舒文炫 PB18000029

2021 年 12 月 21 日

# 目录

<b>1</b>	<b>实验要求</b>	<b>2</b>
<b>2</b>	<b>设计思路</b>	<b>3</b>
<b>3</b>	<b>关键代码讲解</b>	<b>5</b>
3.1	参数及数据结构 . . . . .	5
3.2	文件预处理模块 . . . . .	6
3.3	图操作模块 . . . . .	8
3.4	队列操作模块 . . . . .	10
3.5	生成所需二进制文件 . . . . .	12
3.6	寻找最短路径模块 . . . . .	12
3.7	顶层模块 . . . . .	14
<b>4</b>	<b>调试分析</b>	<b>15</b>
4.1	时空复杂度分析 . . . . .	15
4.2	实验中遇到的问题及解决 . . . . .	15
<b>5</b>	<b>代码测试</b>	<b>16</b>
<b>6</b>	<b>实验总结</b>	<b>18</b>
<b>7</b>	<b>附录</b>	<b>19</b>

# Chapter 1

## 实验要求

导航软件主要是进行最短路径算法的应用，基本要求是在给定数据集上建立图结构，并在建立的图结构上实现 Dijkstra 算法求解任意两点的最短路径，这里只需要朴素的 Dijkstra 算法 (时间复杂度为  $O(|V|^2)$ ), 且正确预处理数据为二进制文件并读入，并且通过助教验收时给出的测试样例

除了基础要求，我在此基础上实现了更多的要求，将 dijkstra 算法时间复杂度降低到  $O(|E|\log|V|)$ , 而且我这里的平均运行时间远小于 100s，达到了后面的 25s 以内，同时我将预处理的二进制文件压缩为  $O(2|E| + |V|)$  大小。

## Chapter 2

# 设计思路

为了实现更好的文件管理和功能的模块化，我将这个实验代码拆分成实现不同功能的各个文件，这也方便进行 debug。

首先是一些全局变量的定义，这里实际上就是定义一下预处理产生的文件的名称，存放的位置，将其集中到一个地方，方便管理，这一部分放在 `config.h` 中

然后是数据结构定义模块，这里需要定义图结构，这里考虑到文件很大，结点很多，如果使用邻接矩阵大概率直接爆炸，而且一般而言这种都是稀疏图，这里按照邻接链表来设计，由于需要输出最短路径，所以这里也需要对路径定义一个结构体，用来保存到达某个顶点路径长度，这一部分放在 `graph.h` 中

由于我们不能直接用原文件操作，这样会产生大量的 IO，很花时间，所以这里我需要定义一些关于文件预处理的函数，这一部分放在 `fileop.h` 中

定义完这个数据结构，就能在其基础上实现图的一系列操作了，比如读入一般的二进制文件的图的创建，创建弧结点，插入弧结点，打印图。考虑到这里的压缩二进制文件，需要大小  $O(2|E| + |V|)$ ，感觉可以直接把这个图转为二进制文件保存下来，大小肯定是满足的，所以我还增添了将图转为二进制文件的功能，然后配套的读入这种文件的图的创建这一部分放在 `graphop.h` 中

然后考虑到在进行 dijkstra 算法时，有大量的对顶点队列的操作，所以这里需要补充一下，从而让算法显得更简洁，队列的数据结构也在这里定义了，需要设计插入，删除等基本操作，同时我的改进 dijkstra 算法是基于不同的队列操作策略，这里设计了两套操作，一套就是普通的有序队列，一条就是使用二叉堆实现的优先队列，将其放在 `queueop.h` 中

设计完这一系列操作，可以着手更高层的操作了

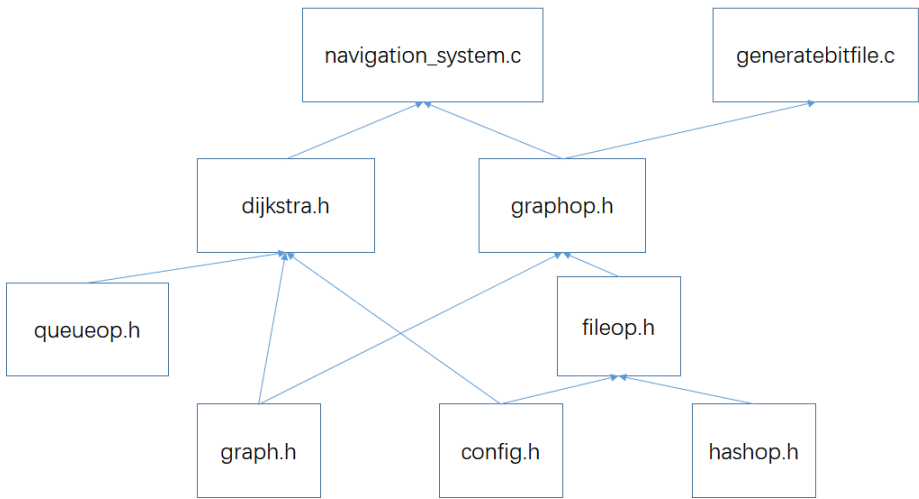
首先是预处理模块，这一模块我准备将其独立出来，不放在导航软件这个大系统里面，而是成为一个单独的函数，主要是这种函数只需要运行一次，得到二进制文件就可以了，没必要每次跑导航软件直接还去询问是否运行，这一函数我将其命名为 `generatebitfile.c`

然后是最核心的 dijkstra 算法实现了，这里需要一个朴素的 dijkstra 算法，一个优化的 dijkstra 算法，将其放到 `dijkstra.h` 中

最后就是导航软件的综合封装，通过命令行输入进行交互，这一模块就是 `navigation_system.c`

上面的文字用一张图表示就是

箭头表示依赖关系



# Chapter 3

## 关键代码讲解

这一部分将具体展示我的代码

### 3.1 参数及数据结构

```
1  #ifndef CONFIG
2  #define CONFIG
3  char rawbitfile[30]="bitstream/bit.dat";
4  char usedbitfile[30]="bitstream/ubit.dat";
5  char cpedbitfile[30]="bitstream/cbit.dat";
6  #endif // CONFIG
7
```

这一部分就定义了我会将产生的二进制文件存放的位置，这里会将所有文件放在 bitstream 文件夹，rawbitfile 是第一步产生的二进制文件，是通过原始文件直接产生的二进制文件，usedbitfile 是对第一步二进制文件稍微处理之后的文件，这样可以方便我创建图，cpedbitfile 是压缩之后的二进制文件

```
1  #ifndef GRAPH
2  #define GRAPH
3  typedef struct ArcNode { //结点的组成域
4      int adjvex; //边指向的顶点位置
5      struct ArcNode* nextarc; //指向下一条弧的指针
6      int info; //该弧相关信息指针（长度）
7  }ArcNode;
8  typedef struct VNode { //表头结点
9      ArcNode* firstarc; //指向第一条依附该顶点的边的指针
10     VNode, * AdjList;
11 }VNode;
12 typedef struct { //图
13     AdjList vertices; //可以间接使用数组
14     int vexnum, arcnum; //图的当前顶点数和弧数
15 }ALGraph;
16 typedef struct {
17     int vex;
18     int length;
19 }path;
20
21
22
23
24
25
26
27 #endif // GRAPH
```

这一部分定义了我使用的数据结构，和课本上的邻接链表数据结构定义基本一致，不多赘述，这里主要还定义了路径这一结构体，用来储存到对应顶点路径长度

```

#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include"config.h"
#include"hashop.h"
int preprocfile()//预处理原文件
int procbitfile()//处理原二进制文件
int countfilevex()//数文件中顶点的个数

int preprocfile(){
FILE *fp1,*fp2;
int i,count1=0;
char src[20],dst[20],distance[20];
int buffer1[3];
char filename[100];
printf("input the filename\n");
scanf("%s",filename);
fp1=fopen(filename,"r");
fp2=fopen(rawbitfile,"wb");
if(fp1==NULL||fp2==NULL){
printf("\nCannot open file!");
exit(-1);
}
while(!feof(fp1)){//每次读出三个整型数，分别是源点，目的点，距离，将其按照二进制写入既定文件
fscanf(fp1,"%s%s",src,dst,distance);
buffer1[0]=atoi(src);
buffer1[1]=atoi(dst);
buffer1[2]=atoi(distance);
fwrite(buffer1,sizeof(int),3,fp2);
count1++;
}
printf("--%d\n",count1);
fclose(fp1);
fclose(fp2);
return 0;
}

```

## 3.2 文件预处理模块

这一部分进行原始文件的预处理，将其转为二进制文件，主要的思想就是按照原始文件的格式读入，然后转为整型数，然后按照二进制的形式写入给定的文件即可

```

int procbitfile(){
FILE *fp,*fp1;
int maxvex=0;
int i;
int buffer[3];
fp=fopen(rawbitfile,"rb");
fp1=fopen(usedbitfile,"wb");
if(fp==NULL||fp1==NULL){
printf("file does not exist,please create it first\n");
exit(-1);
}
while(1){//这一部分主要是看数值最大的顶点是多少，这个就是创建图的顶点数
fread(buffer,sizeof(int),3,fp);
if(feof(fp))
break;
if(maxvex<buffer[0])
maxvex=buffer[0];
if(maxvex<buffer[1])
maxvex=buffer[1];
}

rewind(fp);
fwrite(&maxvex,sizeof(int),1,fp1);
while(1){
fread(buffer,sizeof(int),3,fp);
if(feof(fp))
break;
fwrite(buffer,sizeof(int),3,fp1);
}
fclose(fp);
fclose(fp1);
return 0;
}

```

这一部分进行原始二进制文件的处理，这一部分主要是为了方便创建图，我需要顶点的个数才能创建顶点列表，所以在这里我猜测这里面最大的数值，就是顶点个数

```

int countfilevex() {
    FILE *fp;
    int vexnum=0;
    int* vexlist;
    int* hashlist;
    int buffer[3];
    vexlist=(int*)malloc(30000000*sizeof(int));
    memset(vexlist,0,30000000);
    hashlist=(int*)malloc(40000000*sizeof(int));
    memset(hashlist,0,40000000);
    fp=fopen(rawbitfile,"rb");
    if(fp==NULL){
        printf("file does not exist,please create it first\n");
        exit(-1);
    }
    while(1){//使用哈希查找，快速统计出有多少个顶点
        fread(buffer,sizeof(int),3,fp);
        if(feof(fp))
            break;
        if(!hashfind(hashlist,buffer[0])){
            vexlist[vexnum]=buffer[0];
            vexnum+=int countfilevex::vexnum;
        }
        if(!hashfind(hashlist,buffer[1])){
            vexlist[vexnum]=buffer[1];
            vexnum++;
        }
    }
    fclose(fp);
    printf("the vexnum is %d\n",vexnum);
    return 0;
}

```

这一部分是验证我的猜想，统计出现的不重复的顶点个数，这里我就把不重复的顶点存到 hash 表中，这样会加快速度，这一函数也确实验证了我的猜想

```

1  #include<stdio.h>
2
3  int hashfind(int* vexlist,int a){//find a in vexlist,if not found insert it;
4
5
6  int hashfind(int* hashlist,int a){
7      int hashvalue;
8      hashvalue=(a+7857649)%40000000;
9      if(hashlist[hashvalue]==0){
10         hashlist[hashvalue]=a;
11         return 0;
12     }
13     else{
14         while(hashlist[hashvalue]!=a&&hashlist[hashvalue]!=0){
15             hashvalue=(hashvalue+7860767)%40000000;
16         }
17         if(hashlist[hashvalue]==a)
18             return 1;
19         else{
20             hashlist[hashvalue]=a;
21             return 0;
22         }
23     }
24 }

```

这个就是 hash 方法，就是简单的线性探测



### 3.3 图操作模块

```
#include "fileop.h"
#include <malloc.h>
#include "graph.h"

ALGraph* Create_Graph() //创建图
ArcNode* create_node(int a, int b) //创建结点
int insert_node(ALGraph* G, int i, ArcNode* p) //将弧结点插入到图中
int Print_Graph(ALGraph* G) //打印图
int turnbitfile(ALGraph* G) //将图转为二进制文件
ALGraph* Create_Graph_by_compress() //通过压缩后文件创建图

ArcNode* create_node(int a, int b) {
    ArcNode* p;
    p = (ArcNode*) malloc(sizeof(ArcNode));
    p->adjvex = a;
    p->info = b;
    p->nextarc = NULL;
    return p;
}

int insert_node(ALGraph* G, int i, ArcNode* p) //头插
{
    p->nextarc = G->vertices[i].firstarc;
    G->vertices[i].firstarc = p;
    return 0;
}
```

这里就是创建弧结点和插入弧结点，为了简便使用头插法

```
ALGraph* Create_Graph() {
    int vexnum, arcnum, vex1;
    ALGraph* G;
    int buffer[3];
    int i=0;
    ArcNode* p;
    FILE* fp;
    G = (ALGraph*) malloc(sizeof(ALGraph));
    fp = fopen("usedbitfile", "rb");
    if (fp == NULL) {
        printf("file does not exist, please create it first\n");
        exit(-1);
    }
    fread(&vexnum, sizeof(int), 1, fp);
    G->vexnum = vexnum;
    G->vertices = (AdjList) malloc((vexnum+1)*sizeof(VNode));
    for (i=0; i<=vexnum; i++)
        G->vertices[i].firstarc = NULL;
    while (1) //read the file to get the graph
    {
        fread(buffer, sizeof(int), 3, fp);
        if (feof(fp))
            break;
        p = create_node(buffer[1], buffer[2]);
        insert_node(G, buffer[0], p);
        arcnum++;
    }
    G->arcnum = arcnum;
    fclose(fp);
    return G;
}
```

这里是读入稍微处理之后的二进制文件，创建图，先读入一个整型数这个是顶点个数，然后就能创建顶点列表，之后就是把弧一个一个的填进去就好了

```

int turnbitfile(ALGraph* G) {
    FILE* fp;
    int i, j, k;
    int* arcarray;
    ArcNode* p;
    fp=fopen(cpedbitfile, "wb");
    if (fp==NULL) {
        printf("failed\n");
        exit(-1);
    }
    arcarray=(int*)malloc((G->vexnum+1)*sizeof(int));
    arcarray[0]=0;
    for (i=1; i<=G->vexnum; i++) {
        j=0;
        p=G->vertices[i].firstarc;
        while (p) {
            j++;
            p=p->nextarc;
        }
        arcarray[i]=j;
    }
    fwrite(&G->vexnum, sizeof(int), 1, fp);
    fwrite(arcarray, sizeof(int), G->vexnum+1, fp);
    for (i=1; i<=G->vexnum; i++) {
        p=G->vertices[i].firstarc;
        while (p) {
            fwrite(&p->adjvex, sizeof(int), 1, fp);
            fwrite(&p->info, sizeof(int), 1, fp);
            p=p->nextarc;
        }
    }
    fclose(fp);
    return 0;
}

```

这里是将创建好的图转为二进制文件，通过这个可以实现二进制文件的压缩，因为这在某情况下是构建一个图所需最基本的条件了，思想就是先统计每个顶点有几条弧，这样可以构成一个数组，存的时候先把这个数组存到二进制文件中，大小就是  $O(V)$  的，然后直接把所有的弧按顶点顺序存进去，大小就是  $O(E)$ ，这样总体来看就是  $O(V+E)$  也就实现了压缩的要求

```

ALGraph* Create_Graph_by_compress() {
    FILE* fp;
    int vexnum, arcnum, vexl;
    int adj, info;
    int* arcarray;
    ALGraph* G;
    int i=0, j;
    ArcNode* p;
    G=(ALGraph*)malloc(sizeof(ALGraph));
    fp=fopen(cpedbitfile, "rb");
    if (fp==NULL) {
        printf("file does not exist, please create it first\n");
        exit(-1);
    }
    fread(&vexnum, sizeof(int), 1, fp);
    G->vexnum=vexnum;
    G->vertices=(AdjList)malloc((vexnum+1)*sizeof(VNode));
    arcarray=(int*)malloc((vexnum+1)*sizeof(int));
    fread(arcarray, sizeof(int), vexnum+1, fp);
    for (i=0; i<=vexnum; i++)
        G->vertices[i].firstarc=NULL;
    for (i=1; i<=vexnum; i++) {
        for (j=0; j<arcarray[i]; j++) {
            fread(&adj, sizeof(int), 1, fp);
            fread(&info, sizeof(int), 1, fp);
            p=create_node(adj, info);
            insert_node(G, i, p);
            arcnum++;
        }
    }
    G->arcnum=arcnum;
    fclose(fp);
    return G;
}

```

这个就是用压缩后的文件创建图，根据我的压缩算法去反过来就可以了，先读出每个顶点的弧个数数组，然后依次读出弧，插入到顶点中即可

### 3.4 队列操作模块

这一模块是我导航算法的核心，这里使用了两套队列操作，下面依次讲解

```
typedef struct qwq{
    path* queuelist;
    int maxlen;
    int currlen;
}my_queue;

my_queue* initqueue(int maxlen)://初始化队列
//常规方法
my_queue* enqueue(my_queue *q,path p)://常规入队列
int dequeue(my_queue *q,path* p)://常规出队列
int printqueue(my_queue *q)://打印队列
//二叉堆
my_queue* enheapqueue(my_queue *q,path p)://入二叉堆
int deheapqueue(my_queue* q,path* p)://拿出二叉堆根节点
```

首先是我对队列的定义，由于需要把距离作为权，每次找距离最小的，所以每个结点除了需要结点本身还需要到这个点的路径长度，这样整合为一个结构体，作为队列元素。后面的就是两套队列的操作

```
my_queue* initqueue(int maxlen){
    my_queue* q;
    q=(my_queue*)malloc(sizeof(my_queue));
    q->maxlen=maxlen;
    q->currlen=0;
    q->queuelist=(path*)malloc((maxlen+1)*sizeof(path));
    return q;
}
```

这是对队列初始化，构建这样一个空的优先队列

```
my_queue* enqueue(my_queue* q,path p){
    int j,k;
    if(q->currlen>=q->maxlen){
        printf("overflow!\n");
        exit(-1);
    }
    for(j=0;j<q->currlen;j++){
        if(q->queuelist[j].length>p.length)
            break;
    }
    if(j>=q->currlen){
        q->queuelist[j].vex=p.vex;
        q->queuelist[j].length=p.length;
    }
    else{
        for(k=q->currlen;k>j;k--){
            q->queuelist[k].vex=q->queuelist[k-1].vex;
            q->queuelist[k].length=q->queuelist[k-1].length;
        }
        q->queuelist[j].vex=p.vex;
        q->queuelist[j].length=p.length;
    }
    q->currlen++;
    return q;
}
```

这一函数是普通的队列插入，由于是需要队列的头为距离最小的，这样出队列只要把第一个拿出去就好，其实这里叫优先队列好一点，方法就每次插入结点时，依次比较距离找到位置，然后空出这个位置，然后插入即可

```

int dequeue(my_queue* q, path* p) {
    int k;
    if (q->currlen==0) {
        return (-1);
    }
    p->length=q->queuelist[0].length;
    p->vex=q->queuelist[0].vex;
    for (k=0; k<q->currlen-1; k++) {
        q->queuelist[k].length=q->queuelist[k+1].length;
        q->queuelist[k].vex=q->queuelist[k+1].vex;
    }
    q->currlen--;
    return 0;
}

int printqueue(my_queue *q) {
    int i;
    for (i=0; i<q->currlen; i++) {
        printf("%d,%d->", q->queuelist[i].vex, q->queuelist[i].length);
    }
    printf("\n");
}

```

由于插入保证了队列有序，出队列直接拿走第一个结点就可以了，注意到不存在最短路径的判断，就是如果这个优先队列空了，那说明找不到通往所需顶点的路径，返回-1。后面的打印队列也比较简单，就是简单遍历，不多赘述

```

my_queue* enheapqueue(my_queue *q, path p) {
    int j, k;
    if (q->currlen>=q->maxlen) {
        printf("overflow!\n");
        exit(-1);
    }
    for (j=q->currlen+1; j>1; j=j/2) {
        if (q->queuelist[j/2].length>p.length) {
            q->queuelist[j].length=q->queuelist[j/2].length;
            q->queuelist[j].vex=q->queuelist[j/2].vex;
        }
        else {
            break;
        }
    }
    q->queuelist[j].length=p.length;
    q->queuelist[j].vex=p.vex;
    q->currlen++;
    return q;
}

```

这个是优化算法的优先队列实现，使用二叉堆，每次插入就是先将其放在队尾，然后一路向上交换，不过为了减少交换的次数，这里我稍微优化了一下，就是先不把要插入的元素放进去，而是先去自下而上挪位置，找到了位置后，把要插入的元素放进去就可以了，这样可以减少一些常数因子，稍微加快算法

```

int deheapqueue(my_queue* q, path* p) {
    int k;
    if (q->currlen==0) {
        return (-1);
    }
    p->length=q->queuelist[1].length;
    p->vex=q->queuelist[1].vex;
    q->currlen--;
    for (k=2; k<=q->currlen; k=k*2) {
        if (k<=q->currlen&&(q->queuelist[k].length>q->queuelist[k+1].length))
            k++;
        if (k<=q->currlen&&(q->queuelist[q->currlen+1].length>q->queuelist[k].length)) {
            q->queuelist[k/2].length=q->queuelist[k].length;
            q->queuelist[k/2].vex=q->queuelist[k].vex;
        }
        else
            break;
    }
    q->queuelist[k/2].length=q->queuelist[q->currlen+1].length;
    q->queuelist[k/2].vex=q->queuelist[q->currlen+1].vex;
    return 0;
}

```

由于要维护二叉堆，所以删除算法就变得复杂了一些，这里维护就是自上而下维护，同样的为了减少常数因子，也就是交换次序，我先不把最后一个放到第一个，而是先去挪位置，把位置空好再插入，这里判断最短路径不存在也是用相同的方法

## 3.5 生成所需二进制文件

```
#include "graphop.h"

int main() {
    int start, compress_en;
    ALGraph* G;
    printf("if you have the bitfile already, please input 0, or input 1 to generate it\n");
    scanf("%d", &start);
    if (start) {
        preprocfile();
        procbitfile();
    }
    printf("do you want to compress it? if so input 1 or input 0\n");
    scanf("%d", &compress_en);
    if (compress_en) {
        G = Create_Graph();
        turnbitfile(G);
    }
    printf("you have got the bitfile, please close the window\n");
    return 0;
}
```

这个函数需要调用 graphop.h, 顺便也调用了 fileop.h 里面定义了需要的函数, 这里逻辑比较简单, 就是输入要转为二进制文件的文件名, 转完之后问一下是否需要压缩, 如果要压缩就进一步操作一下

## 3.6 寻找最短路径模块

这一模块进行指定起始点到目的地的最短路径的寻找, 分为两个, 一个朴素方法, 一个优化方法, 他们最基本的区别就是优先队列的实现不同

```
int naiveshortestPath(ALGraph *G, int v0, int v1) {
    int w = v0;
    int i, vex1, findflag;
    double start_time, finish_time;
    my_queue* q;
    int* d;

    int* vfinal = (int*)malloc((G->vexnum + 1) * sizeof(int)); // 判断结点, 是否已经被使用了, 使用了为1, 没使用为0
    d = (int*)malloc((G->vexnum + 1) * sizeof(int)); // 该结点在该条路径上的前一个顶点, 用来反向输出路径
    path temp_path;
    path* pa = (path*)malloc((G->vexnum + 1) * sizeof(path)); // 该顶点是否在路径上
    ArcNode* pt;
    start_time = clock();
    q = initqueue(G->vexnum);
    for (i = 0; i < G->vexnum + 1; i++) { // 初始化
        vfinal[i] = 0;
        pa[i].vex = i;
        pa[i].length = -1;
        d[i] = -1;
    }
    pa[v0].length = 0;
    vfinal[v0] = 1;
    for (i = 0; i < G->vexnum + 1; i++) {
        pt = G->vertices[w].firstarc;
        while (pt) {
            if (pa[pt->adjvex].length < 0 || (pa[pt->adjvex].length > pa[w].length + pt->info)) { // 如果下一个结点还没加入, 或者到下一个结点长度比当前
                d[pt->adjvex] = w; // 更改到pt->adjvex的最小路径
                pa[pt->adjvex].length = pa[w].length + pt->info; // 更新最小路径值
                q = enqueue(q, pa[pt->adjvex]); // 更改的结点加入优先队列
                // printqueue(q);
            }
            pt = pt->nextarc; // 去看与w相连的下一个顶点
        }
        if (dequeue(q, &temp_path) != -1) { // 将距离最小的出队列
            // printqueue(q);
            w = temp_path.vex;
            vfinal[w] = 1; // 结点被使用
        }
        if (w == v1) {
            findflag = 1;
            finish_time = clock();
            break;
        }
    }
    else {
        findflag = 0;
        finish_time = clock();
        printf("the path does not exist!\n");
        break;
    }
    printf("the time consumed is %lf s\n", (double)(finish_time - start_time) / CLOCKS_PER_SEC);
    if (findflag) {
        printf("the length of the path is %d\n", pa[v1].length);
        printf("the path is below\n");
        vex1 = v1;
        while (vex1 != v0) {
            printf("%d<-", vex1);
            vex1 = d[vex1];
        }
        printf("%d\n", vex1);
        printf("have fun for your journey!\n");
    }
    return 0;
}
```

这里我定义了一些变量，注释里面也写了其功能，我就讲解一下我的实现思路，Dijkstra 算法每次选出一个顶点，所以最多需要循环顶点数量次，所以这里循环中止条件就是循环了超过顶点数量次，不过如果提前找到了所要的目的顶点，会通过 break 直接中止循环。每一次循环回去把当前队列第一个顶点拿出来，这里需要注意的就是队列会不会空，空了说明不存在路径第一次循环就是起始点，然后将该顶点的所有没有加入队列的邻接顶点加入到优先队列中，这里如果顶点还没加入对应的路径长度域值为-1，可以通过这个判断，然后如果加入这个顶点使得一些最短路径得到更新，就去更新，更新后的值存到对应顶点路径长度域里面。结束是 vfinal 数组中对该节点标记为 1，表示该节点以后不讨论了，运行这个算法之前和之后我都计时函数，记下开始运行时间和结束时间，两者做差然后把结果转为秒，就能得到程序运行时间了，最后如果没找到路径就说路径不存在，找到了就把路径长度以及具体路径给出来

```
int bettershortestPath(ALGraph *G, int v0, int v1) {
    int w = v0;
    int i, vex1, findflag;
    double start_time, finish_time;
    my_queue *q;
    int *d;

    int *vfinal = (int *)malloc((G->vexnum + 1) * sizeof(int)); // 判断结点v是否已经被使用了，使用了为1，没使用为0
    d = (int *)malloc((G->vexnum + 1) * sizeof(int)); // 该顶点在该条路径上的前一个顶点，用来反向输出路径
    path temp_path;
    path *pa = (path *)malloc((G->vexnum + 1) * sizeof(path)); // 该顶点是否在路径上
    ArcNode *pt;
    start_time = clock();
    q = initqueue(G->vexnum);
    for (i = 0; i < G->vexnum; i++) { // 初始化
        vfinal[i] = 0;
        pa[i].vex = i;
        pa[i].length = -1;
        d[i] = -1;
    }
    pa[v0].length = 0;
    vfinal[v0] = 1;
    for (i = 0; i < G->vexnum; i++) {
        pt = G->vertices[i].firstarc;
        while (pt) {
            if (pa[pt->adjvex].length < 0 || (pa[pt->adjvex].length > pa[w].length + pt->info)) { // 如果下一个结点还没加入，或者到下一个结点长度比当前短
                d[pt->adjvex] = w; // 更改到pt->adjvex的最短路径
                pa[pt->adjvex].length = pa[w].length + pt->info; // 更新最小路径值
                q = enheapqueue(q, pa[pt->adjvex]); // 更改的结点加入优先队列
                // printf("%d\n", q);
            }
            pt = pt->nextarc; // 去看与w相连的下一个顶点
        }
        if (deheapqueue(q, &temp_path) != -1) { // 将距离最小的出队列
            // printf("%d\n", q);
            w = temp_path.vex;
            vfinal[w] = 1; // 结点被使用
            if (w == v1) {
                findflag = 1;
                finish_time = clock();
                break;
            }
        }
        else {
            findflag = 0;
            finish_time = clock();
            printf("the path does not exist!\n");
            break;
        }
    }
    printf("the time consumed is %lf s\n", (double)(finish_time - start_time) / CLOCKS_PER_SEC);
    if (findflag) {
        printf("the length of the path is %d\n", pa[v1].length);
        printf("the path is below\n");
        vex1 = v1;
        while (vex1 != v0) {
            printf("%d<-", vex1);
            vex1 = d[vex1];
        }
        printf("%d\n", vex1);
        printf("have fun for your journey!\n");
    }
    return 0;
}
```

只需要稍作比较就可以发现，这与前一个的差别仅仅只在优先队列的实现上面，这里使用了二叉堆来实现，所有运行速度更快，其他的由于和上面一样，不做赘述

## 3.7 顶层模块

```
#include "graphop.h"
#include "dijkstra.h"
int main() {
    int i, flag=1, method, filetype;
    int src, dst;
    ALGraph* G;
    printf("欢迎进入导航系统, 输入1读入原始二进制文件, 输入0读入压缩二进制文件\n");
    scanf("%d", &filetype);
    printf("读入并创建中, 这可能会花一些时间.....\n");
    if (filetype)
        G = Create_Graph();
    else
        G = Create_Graph_by_compress();
    printf("文件读入完成, 地图已经创建, 缺德地图持续为您导航\n");
    //Print_Graph(G);
    while (flag) {
        printf("please input the src and dst\n");
        scanf("%d %d", &src, &dst);
        printf("input 1 for naive method, 2 for better method\n");
        scanf("%d", &method);
        printf("正在为您规划路径, 请稍等\n");
        if (method == 1)
            naiveshortestPath(G, src, dst);
        else
            bettershortestPath(G, src, dst);
        printf("input 1 to continue, 0 to finish\n");
        scanf("%d", &flag);
    }
    return 0;
}
```

经过前面的准备，这里的顶层导航软件框架就很清晰了，应该不需要我多做解释，一目了然

# Chapter 4

## 调试分析

### 4.1 时空复杂度分析

这一部分我只分析关键代码的时空复杂度，主要是很多都很容易，全分析的话报告实在是太长了，而且没必要 orz(第一次实验报告太详细了，写吐了)。在 fileop.h 中处理原文件，原二进制文件由于都是把整个文件读一遍，时间复杂度就说里面的行数，也就是弧的数目。在 queueop.h 中，对应一般的优先队列，插入需要  $O(V)$  的时间，因为最坏的情况可能需要遍历整个队列才能找到位置，删除对应也需要  $O(V)$  因为不光要删除结点，还要把后面的往前移。对于二叉堆，插入需要  $O(\log V)$  时间，删除也是  $O(\log V)$  时间。最后来到朴素的 dijkstra 算法，循环次数最多顶点数目次，然后里面插入删除结点最多也是需要顶点数目时间，所有时间复杂度  $O(V^2)$ ，优化后的算法，循环次数改不了，但是插入删除结点时间变为  $O(\log V)$  插入次数最多为  $O(E)$ ，因为可能每看一条边都要变一次，删除次数最多为  $O(V)$ ，所以总时间复杂度变为  $O((E + V)\log V)$ ，这里如果所有结点都可从源节点到达，时间复杂度变为  $O(E\log V)$

### 4.2 实验中遇到的问题及解决

- 文件预处理那一块，所给初始文件只有每条弧的信息，没有给有多少顶点，这使得没有办法方便的构造图，解决方法就是堆初始文件进行处理，统计顶点个数，但是一个一个统计又太慢，这里我猜想是从一开始到最大值，中间没有漏的，也就是顶点数值最大值就是顶点个数，同时为了验证我的猜想，我使用哈希查找的办法加快统计速度，结果验证了我的猜想，不过还是跑了将近一个小时 orz。

```
the vexnum is 23947347
Process returned 0 (0x0)   execution time : 3190.272 s
```

```
the max is 23947347
```

- 文件压缩那一块，一开始并没有什么压缩的思路，Huffman 压缩是不可能的，这辈子都不可能的 (越压越大.jpg)，但是实验报告里面要求的空间大小提供了一些提示，就是把所有的顶点以及其对于弧的数目存下来，外加上每个顶点的弧按顺序存好就行
- 内存空间的问题，书上给的算法是在邻接矩阵下实现的，但是对于这个如果使用邻接矩阵，电脑内存远远不够，所以我考虑使用邻接表，这样也就对应需要修改算法使用邻接表实现



## Chapter 5

# 代码测试

首先是对预处理文件的测试，将测试文件放到导航软件相同目录下，运行 generatebitfile 程序，按照程序提示依次输入，得到下面的结果

```
if you have the bitfile already, please input 0, or input 1 to generate it
1
input the filename
distance_info.txt
--58333345
do you want to compress it? if so input 1 or input 0
1
you have got the bitfile, please close the window
Process returned 0 (0x0)   execution time : 164.132 s
Press any key to continue.
```

然后打开 bitstream 文件夹，可以看到生成了三个二进制文件，第三个是普通的二进制文件，第二个是压缩后的二进制文件

名称	日期时间	文件类型	大小
bit.dat	2021/12/21 9:46	DAT 文件	683,594 KB
cbit.dat	2021/12/21 9:47	DAT 文件	549,274 KB
ubit.dat	2021/12/21 9:47	DAT 文件	683,594 KB

然后就是运行导航软件了，运行 navigation\_system.exe, 开始会让你选择读入的文件，这里我选择读入压缩后的文件

```
欢迎进入导航系统, 输入1读入原始二进制文件, 输入0读入压缩二进制文件
0
读入并创建中, 这可能会花一些时间.....
文件读入完成, 地图已经创建, 缺德地图持续为您导航
please input the src and dst
```

然后就是输入起点和终点，选择使用的寻路算法  
首先是朴素方法

```
please input the src and dst
1000000 2000000
input 1 for naive method, 2 for better method
2
正在为您规划路径, 请稍等
the time consumed is 9.624000 s
the length of the path is 3693584
the path is below
20000000<-20000001<-951471<-20000079<-1999724<-951117<-951118<-1999725<-1999726<-951119<-951120<-951457<-951400<-2487705<-9
51460<-2000068<-2000072<-951463<-2000073<-1999678<-951071<-951406<-951405<-950819<-1999427<-1999429<-950821<-2487467<-19
99430<-2487454<-1999397<-950833<-950834<-1999442<-2487407<-1999289<-950829<-950830<-1999387<-1999386<-950778<-2487451<-9
50856<-950626<-950625<-1999234<-1999233<-950624<-1999375<-950672<-950671<-950670<-1999280<-2500076<-981574<-2500077<-203
0162<-2030161<-981573<-2030167<-981584<-981583<-981591<-981589<-981588<-2500083<-2030176<-2030175<-981587<-984745<-98162
8<-2030214<-2030213<-981626<-981625<-984337<-2032926<-984408<-2498072<-982259<-2500339<-982206<-984489<-2501215<-984333<-
2033191<-2033190<-984603<-984490<-984025<-2501087<-2032613<-2025351<-983956<-2501058<-2032545<-2032544<-983955<-2027037
<-2027036<-983957<-983958<-2501057<-976696<-2025354<-2025291<-2025292<-2031806<-976730<-976731<-2025327<-976732<-2031948
<-984469<-981838<-2025373<-2025372<-976777<-2025370<-976775<-2025374<-1028087<-2519167<-2076687<-2076686<-2519173<-10280
90<-2076673<-2076672<-2076703<-2083877<-2076702<-1028117<-1028118<-1020492<-2069080<-1020491<-1020512<-2069093<-2069094<-
2069105<-2069106<-1020518<-2069102<-1020515<-2069114<-1020540<-2516073<-2069138<-1020556<-2069145<-2069142<-2516076<-10
20562<-2069148<-2069156<-2516081<-2069160<-1020573<-1020596<-2069181<-1020594<-2069172<-1020585<-2069198<-1020612<-10206
29<-1022432<-2516553<-2070297<-1021713<-1022435<-1022434<-2083287<-2519154<-2076643<-1034699<-1022442<-1022441<-2071029<-
2071028<-2071037<-2071043<-2071045<-1022461<-2071047<-2516856<-1022463<-1022464<-2070287<-2070286<-1021702<-2070314<-10
21730<-1022103<-2516713<-2070688<-2070683<-2070750<-2070697<-2070696<-2516740<-1022163<-2516734<-2070945<-2070946<-25213
15<-1022349<-2516812<-2070948<-2070947<-2516820<-1022392<-1022386<-1022385<-2516829<-2070969<-2517142<-1023075<-1023074<-
2071658<-2071661<-1023077<-1023076<-2071713<-2071714<-1023129<-1023257<-2071727<-2071728<-2519988<-2517045<-2071501<-2071
```

这里用了 9.6s

然后相同的起点和终点，使用优化算法

```
please input the src and dst
1000000 2000000
input 1 for naive method, 2 for better method
2
正在为您规划路径, 请稍等
the time consumed is 0.639000 s
the length of the path is 3693584
the path is below
20000000<-20000001<-951471<-20000079<-1999724<-951117<-951118<-1999725<-1999726<-951119<-951120<-951457<-951400<-2487705<-9
51460<-2000068<-2000072<-951463<-2000073<-1999678<-951071<-951406<-951405<-950819<-1999427<-1999429<-950821<-2487467<-19
99430<-2487454<-1999397<-950833<-950834<-1999442<-2487407<-1999289<-950829<-950830<-1999387<-1999386<-950778<-2487451<-9
50856<-950626<-950625<-1999234<-1999233<-950624<-1999375<-950672<-950671<-950670<-1999280<-2500076<-981574<-2500077<-203
0162<-2030161<-981573<-2030167<-981584<-981583<-981591<-981589<-981588<-2500083<-2030176<-2030175<-981587<-984745<-98162
8<-2030214<-2030213<-981626<-981625<-984337<-2032926<-984408<-2498072<-982259<-2500339<-982206<-984489<-2501215<-984333<-
2033191<-2033190<-984603<-984490<-984025<-2501087<-2032613<-2025351<-983956<-2501058<-2032545<-2032544<-983955<-2027037
<-2027036<-983957<-983958<-2501057<-976696<-2025354<-2025291<-2025292<-2031806<-976730<-976731<-2025327<-976732<-2031948
<-984469<-981838<-2025373<-2025372<-976777<-2025370<-976775<-2025374<-1028087<-2519167<-2076687<-2076686<-2519173<-10280
90<-2076673<-2076672<-2076703<-2083877<-2076702<-1028117<-1028118<-1020492<-2069080<-1020491<-1020512<-2069093<-2069094<-
2069105<-2069106<-1020518<-2069102<-1020515<-2069114<-1020540<-2516073<-2069138<-1020556<-2069145<-2069142<-2516076<-10
20562<-2069148<-2069156<-2516081<-2069160<-1020573<-1020596<-2069181<-1020594<-2069172<-1020585<-2069198<-1020612<-10206
```

这里用了 0.63s

然后换一个例子，这个例子如果使用朴素算法，时间巨长无比，无法忍受，这里我等了十几分钟遂放弃，但是用优化算法

```
please input the src and dst
3141592 2718281
input 1 for naive method, 2 for better method
2
正在为您规划路径, 请稍等
the time consumed is 14.121000 s
the length of the path is 23688021
the path is below
2718281<-4459479<-2723672<-3469395<-2718284<-3469397<-4459485<-4459488<-3469400<-3469459<-4459490<-3469403<-3469405<-271
8290<-4459492<-2718294<-4459495<-4459517<-3469437<-2724937<-4468723<-3469476<-2718331<-2718301<-4459503<-2723965<-272494
1<-2718326<-4459540<-4459541<-3469436<-3480242<-4467809<-4461597<-3471972<-3471977<-4468542<-3480972<-3471986<-4461608<-34
71988<-4461610<-3471989<-4461611<-4461616<-3471996<-2724275<-4467798<-3472349<-2720032<-4461898<-4467801<-3479548<-347
2359<-2720037<-3472361<-2720039<-4461914<-3472366<-4461918<-2720042<-3472370<-2720044<-4461923<-3472374<-3472376<-272005
0<-3472382<-2720051<-3472387<-4461936<-3472388<-3472389<-4461940<-3472404<-2720063<-2720061<-3472401<-4461949<-4461974<-34
72439<-2720081<-4461975<-4461981<-3472442<-3472445<-4461983<-3472449<-2720097<-4462002<-3472473<-2720102<-4462135<-347
2633<-2720200<-3472640<-2720206<-4462161<-3472664<-2720217<-4462162<-3472668<-3472673<-4462168<-4462186<-3472699<-446218
9<-3472715<-3472720<-2720248<-4462207<-3472721<-3472725<-4462210<-3472730<-2720253<-3472738<-3472764<-4462241<-4462242<-27
270273<-3472768<-4462249<-4462251<-3472787<-4462261<-2720312<-3472829<-4462296<-3472832<-3474704<-2721421<-2721423<-272
1425<-2721431<-4463860<-2721436<-2721440<-2721441<-3474784<-3474785<-3469428<-2718304<-4463902<-4463906<-3474794<-272147
8<-4463921<-2721483<-3474816<-4463926<-4464023<-2721555<-3474939<-3474940<-2721556<-2721558<-3474942<-2721559<-4464028<-
2721560<-3474944<-2721569<-3474960<-3474961<-4464045<-4464046<-2721600<-4464085<-2721601<-2721604<-3475022<-3475041<-347
4610<-4463756<-3475028<-3475043<-3475045<-4464107<-3475046<-3475047<-4464114<-4464562<-3475595<-3475598<-3475599<-347560
1<-2721954<-3475615<-3475617<-3475622<-2721960<-3475673<-3475674<-4464629<-3475690<-4464643<-2722002<-4464641<-3475799<-
2722066<-3475806<-4464736<-3475827<-2722080<-2722081<-3475831<-4464756<-3475833<-2722111<-3475889<-2722117<-3475890<-347
7258<-3477259<-2722952<-4465969<-2723042<-3477459<-4466088<-3477464<-4466092<-3477467<-3477487<-2723065<-3477496<-347749
7<-3477498<-4466121<-3477499<-4466123<-4466208<-3477605<-3477607<-2723129<-3477611<-2723133<-3477614<-4466217<-3477616<-34
77633<-2723143<-2724523<-4466261<-3477670<-3477671<-3477672<-3479604<-2724311<-3479603<-2724310<-2719030<-3470656<-271
```

只用 14.12s! 非常之快，可以看到我这个优化算法很好，并且常数因子也不是很大

## Chapter 6

# 实验总结

本次实验主要是实现 Dijkstra 算法。通过这次实验，我更深入的理解了 Dijkstra 算法，也学到了如何去优化一个算法的时间复杂度，可以通过一些比较高级的数据结构，这里就是二叉堆，实际上还有斐波那契堆，不过实现起来比较麻烦，我就不考虑了，这让我切实体会到了算法的魅力。还有算法也得建立在物理基础上，实际落地时，空间有时也会成为制约因素，这里用邻接矩阵是比较困难的，但是邻接表就很好。

# Chapter 7

## 附录

关于程序运行的说明，在测试部分已经基本给出，这里不再赘述  
提交的文件清单

- bitstream 文件夹存放二进制文件
- config.h 指示二进制文件存放位置
- graph.h 数据结构定义
- hashop.h 哈希操作
- fileop.h 文件操作
- queue.h 优先队列操作
- dijkstra.h 导航算法
- graphop.h 图操作
- generatebitfile.c 生成二进制文件
- navigation\_\_system.c 导航系统
- generatebitfile.exe 生成二进制文件程序
- navigation\_\_system.exe 导航系统程序