导航

PB20000096 潘廷岳

【实验要求】

①实现存储文件的压缩。

②实现图的构建和存储。

③实现Dijkstra算法计算给定点的最短路径，输出其长度及具体路径；在此基础上，实现Dijkstra算法的优化。

④拓展：实现地名检索功能（点不再以编号呈现），同时支持该类文件的压缩

【设计思路】

Part_1 文件压缩

对于文件压缩，考虑把边按照起点归类：同一起点的边存入一个vector容器中，并保存两个信息：终点以及边长。压缩信息主要由下列三个部分构成：

第1个信息：节点个数VNUM

第2~(VNUM+1)条信息中，第i条信息又包含三部分：

①以i号节点为起点的边条数ENUM（i）

②连续ENUM（i）条子信息反映了每条边的终点及权值。

最终，我们用了4 + 4*（2E+V）个byte对图文件的信息进行了存储。

Part_2 最短路求解

①解压后利用邻接表存图

②利用Dijkstra算法进行最短路求解

③考虑用小根堆维护每轮未被标记的dist最小点的寻找与更新

④用last_node保存每个点到起点最短路上的前驱点编号，并在Dijkstra算法中直接更新

⑤当最小未被标记的dist点为终点时，结束算法，并通过访问dist数组来输出最短路

【关键代码讲解】

Part_1 文件压缩代码

从txt中读取数据，并转成int类型

```
int GetNumber(FILE *fp)//char to int
{
    string str;
    char ch=0;
    int dec;
    while((ch!=' ')&&(ch != 10))
```

```
    {
        fread(&ch,1,1,fp);
        str += ch;
        if(feof(fp))
            break;
    }
    str.erase(str.length()-1);//Delete the bin-char
    int num = atoi(str.c_str());
    return num;
}
```

读取出边信息后按照起点压入容器中，每连续两个存储空间表示一条边的信息

```
while (!feof(fin))
{
    u = GetNumber(fin);
    v = GetNumber(fin);
    w = GetNumber(fin);
    node[u].push_back(v);
    node[u].push_back(w);
}
```

将压缩信息写入文件中

```
input = MAXVEX;//最大点编号
fwrite(&input,sizeof(int),1,fout);//写入总点数
for(i=1;i<=MAXVEX;i++)
{
    length = node[i].size()/2;
    fwrite(&length,sizeof(int),1,fout);
    for(auto j=node[i].begin();j<node[i].end();j++)
    {
        input = *j;
        fwrite(&input,sizeof(int),1,fout);
    }
}
```

Part_2 Dijkstra算法代码

结构体定义

```
struct VNode {
    int dist = 0, last_node = 0;//到起点的最短距离距离和最短路径前驱节点的编号
    bool vis = false;//是否被标记，Dijkstra算法要求
}node[MAXVEX];

typedef struct Vitality_node {
    int pos;
    int dis;
    bool operator <(const Vitality_node& x)const {
        return x.dis < dis;
    }
};//定义临时结构体变量，便于排序
```

初始化

```
priority_queue<Vitality_node>q;//记录节点编号
for (int i = 1; i <= vex_num; i++) {
    node[i].dist = INF; node[i].vis = false;
}//初始化点信息
node[S].dist = 0;
Vitality_node p = { S,0 };
q.push(p);//加入起点
```

Dijstra算法实现

```
while (!q.empty()) {
    Vitality_node t = q.top();
    q.pop();
    int u = t.pos;

    if (node[u].vis) continue;
    if (node[u].dist == INF || u == T) break;//若终点的dist已经是最短，则结束算法
    node[u].vis = true;//u是目前到达的最短处，从这个点向外松弛
    for (int i = head[u]; i; i = edge[i].next) {//枚举这个点的所有邻边
        int now = edge[i].v;

        if ((node[now].dist > node[u].dist + edge[i].w) && !node[now].vis) {
            node[now].dist = node[u].dist + edge[i].w;
            node[now].last_node = u;//记录前驱节点，
            p = { now, node[now].dist };
            q.push(p);//入队
        }
    }
}
```

堆函数实现（基础功能）

```
long long get()
{
    long long res=h[1].num;
    h[1].num=h[op].num;
    h[1].sum=h[op--].sum;
    long long now=1,nex1;
    while((now<<1)<=op)
    {
        nex1=now<<1;
        if(h[nex1].sum>h[nex1+1].sum&&nex1+1<=op)
        nex1++;
        if(h[nex1].sum>=h[now].sum)
        return res;
        swap(h[nex1].sum,h[now].sum);
        swap(h[nex1].num,h[now].num);
        now=nex1;
    }
    return res;
}
void put(long long x,long long w)
{
    h[++op].num=x;
    h[op].sum=w;
    long long nex1,now=op;
```

```
    while(now!=1)
    {
        nex1=now>>1;
        if(h[nex1].sum<=h[now].sum)
        break;
        swap(h[nex1].sum,h[now].sum);
        swap(h[nex1].num,h[now].num);
        now=nex1;
    }
    return;
}
```

结果输出

```
if (node[T].dist == INF) return false;//说明不存在最短路

printf_s("The shortest path's length from Src to Dst is: %d\n", node[T].dist);

int t = T;
printf_s("The shortest path from Dst to Src is:\n");
while (t != S) {
    printf_s("%d<-", t);//倒序输出最短路径
    t = node[t].last_node;
}
printf_s("%d\n", t);
```

地名映射：

```
map<string, int>::iterator iter;
if (Decode_ch == 32) {
    iter = VNode_construct.find(Message_que);

    if (iter != VNode_construct.end()) e[flag] = iter->second;//说明查找成功
    else {
        VNode_construct.insert(pair<string, int>(Message_que, ++vex_num));//否则给新顶点编号
        e[flag] = vex_num;
        strcpy_s(node[vex_num].V_name, Message_que);
    }

    for (int i = 0; i < Message_len; i++) Message_que[i] = '\0';
    Message_len = 0; flag++;
}//说明是空格
else {
    ...
}
```

这里使用了std::map实现地名映射

拓展文件的压缩

```
67      char s_code[MAX_CODE_LEN] = { '\0' };//压缩编码
68  };
69
70
71  void DFS_huffman_tree(Huffman_node* p, Char_array* CH, char* CH_array ,int dep){ ... }
85
86  void Build_Huffman(Char_array* CH){ ... }
115
116 void DOC_COMPRESS(const char* sIn,const char* sOut){ ... }
198
199   Huffman_node* Rebuild_tree_node[256];
200   //为树的建立提前开辟空间，防止局部变量的销毁
201   int Rebuild_tree_node_code = 0;
202 void Rebuild_huffman_tree(Huffman_node* R,char *s_code,char ch){ ... }
238
239 char Decode_bit_queue(queue <char> &bit_stack,Huffman_node * R){ ... }
251
252   int cnt = 0;
253
254
255 int count_dist(char* CH_que){ ... }
```

这里使用了Huffman压缩方式，这里给出了使用函数截图，压缩效果之后展示。

【调试分析】

Part_1 时空复杂度分析

显然，对于压缩程序及最短路求解程序，其空间复杂度均为O（|E|）

对于时间复杂度，由于Dijkstra将每条边、每个点都会访问一次，同时对于小根堆的一次维护操作大致需要 $\log_2^{|V|}$，故时间复杂度应为O（|V|+|E|）* $\log_2^{|V|}$

Part_2 遭遇的问题

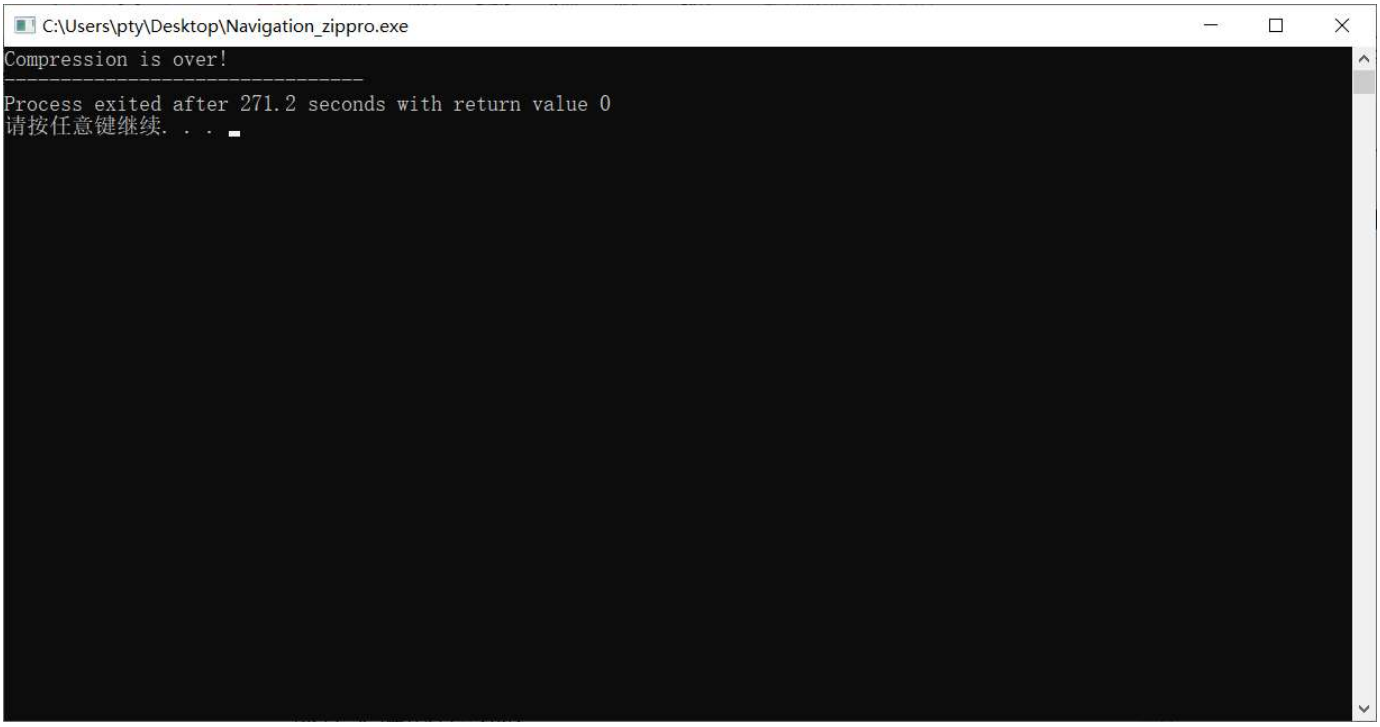
在本次实验中，本人遭遇的最大问题就是fscanf_s和fread用法没弄清，导致压缩时将有效信息当作文件结束符处理，导致解压缩失败。在查阅资料后方才明白原因，以下是对于其的说明：

> 文本方式不能完全读取，而二进制方式能的原因- 文本方式读取文件，最主要的用处是一次读取一整句(以换行符’/n’，即二进制的换行标志”/r/n”结束)，方便用于特殊用处ReadString、fscanf(…,”%s”,…)之类，每次读取的内容长度是不定的；而二进制读取方式Read、fread等，都是读取固定长度 所以文本方式读取对EOF的判定，是一个文件尾结束标志，如果是文本文件，则这个文件尾肯定不会出现在文件内容中(因为是不可打印字符构成的结束标志，人可读的文本文件不会包括它)，这样以结束标志为文件尾则是可以的；二进制文件内容可以是任意字节，如果把它当文本文件来读，以文件尾为结束，当然可能出现把文件内容判定为文件尾的情况；二进制读取方式由于每次读取固定字节，所以只需要用总文件长度(这个数值是系统管理的数值，不是计算得出来的)减去每次读取的长度(或根据Seek的位置计算长度)，就可以知道是否到文件尾，不需要定义结束标志；所以用二进制方式打开任何文件都是合理的。

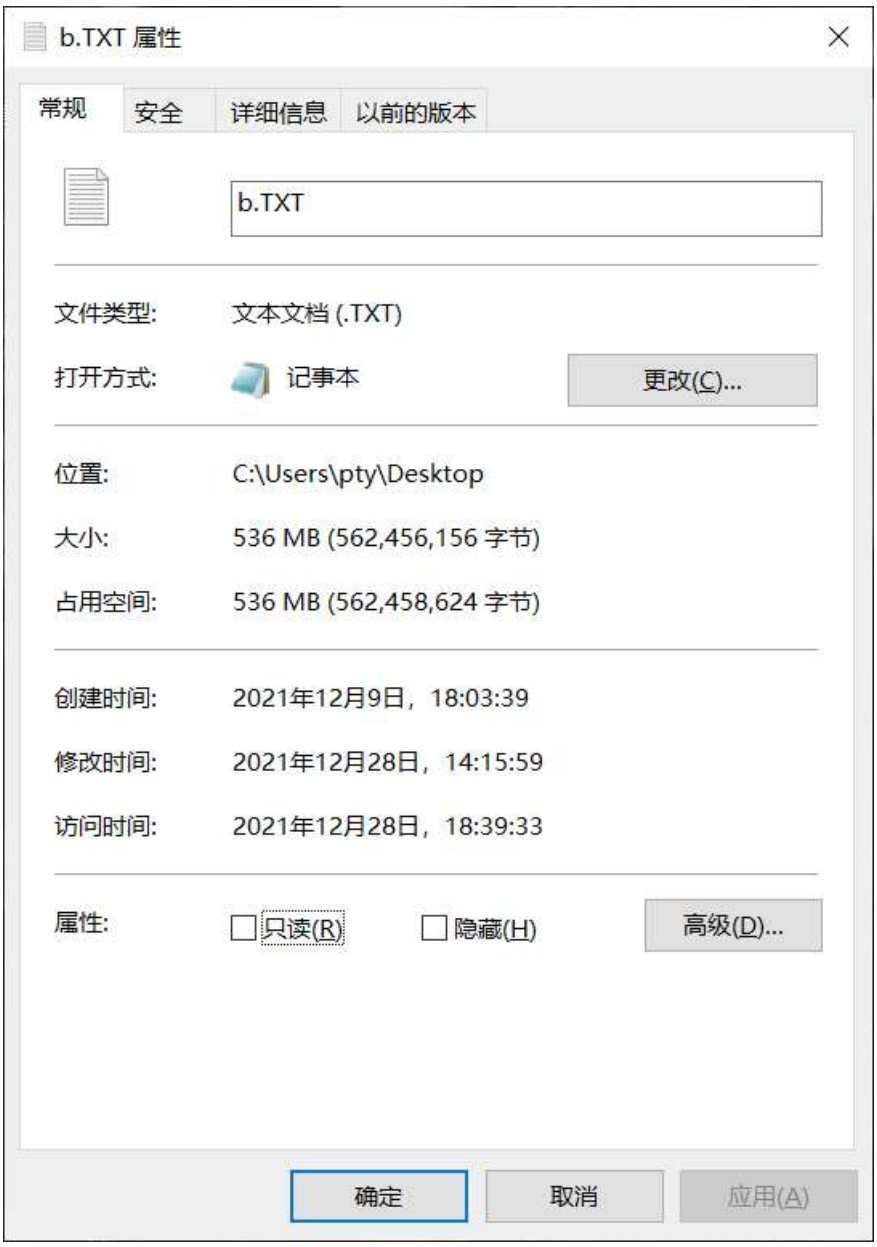
其次，对编写程序时结构化、规范化的缺乏，导致本人在调试阶段遭遇了很多不必要的麻烦，这促使本人认识到程序规范化、结构化的重要性。例如，在实现Huffman压缩的过程中，初始时由于未使用结构体，导致变量过多而经常无法迅速找到需要使用的变量，这使我耗费了很多不必要的时间。

【代码测试】

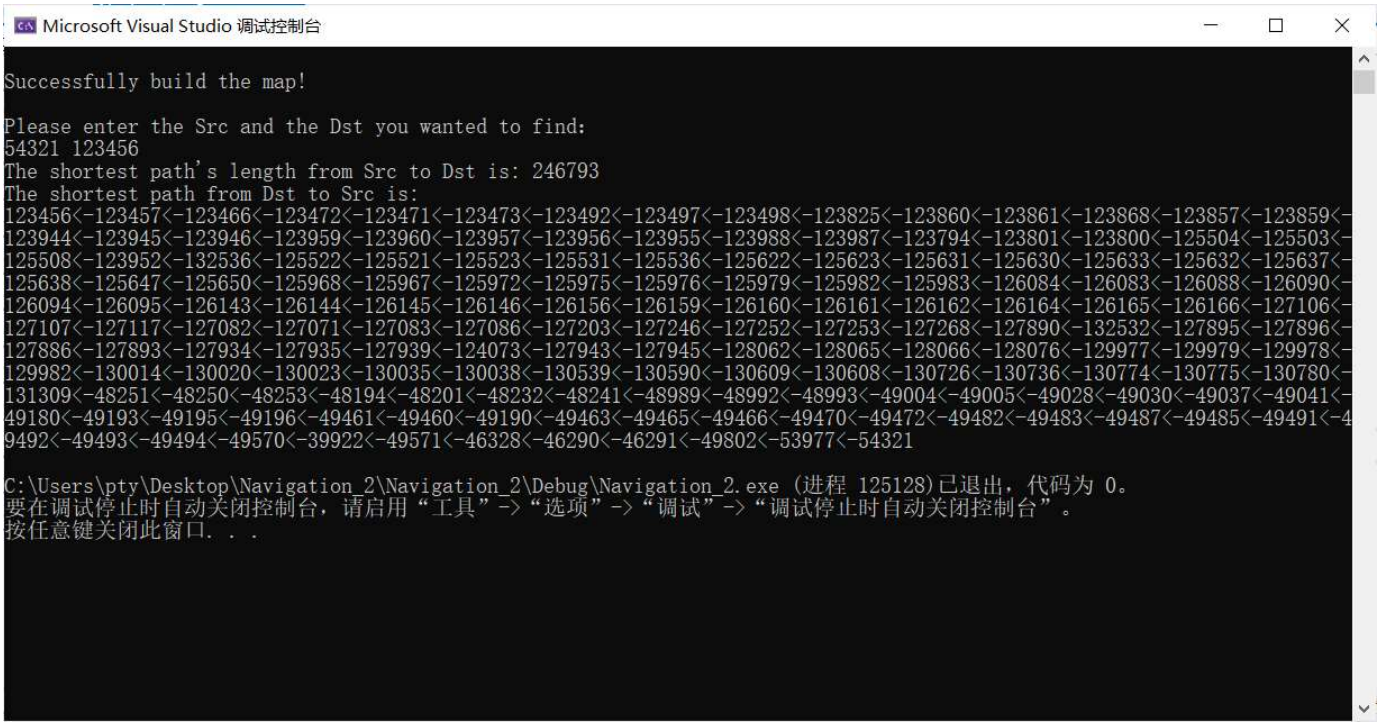
大文件压缩时间



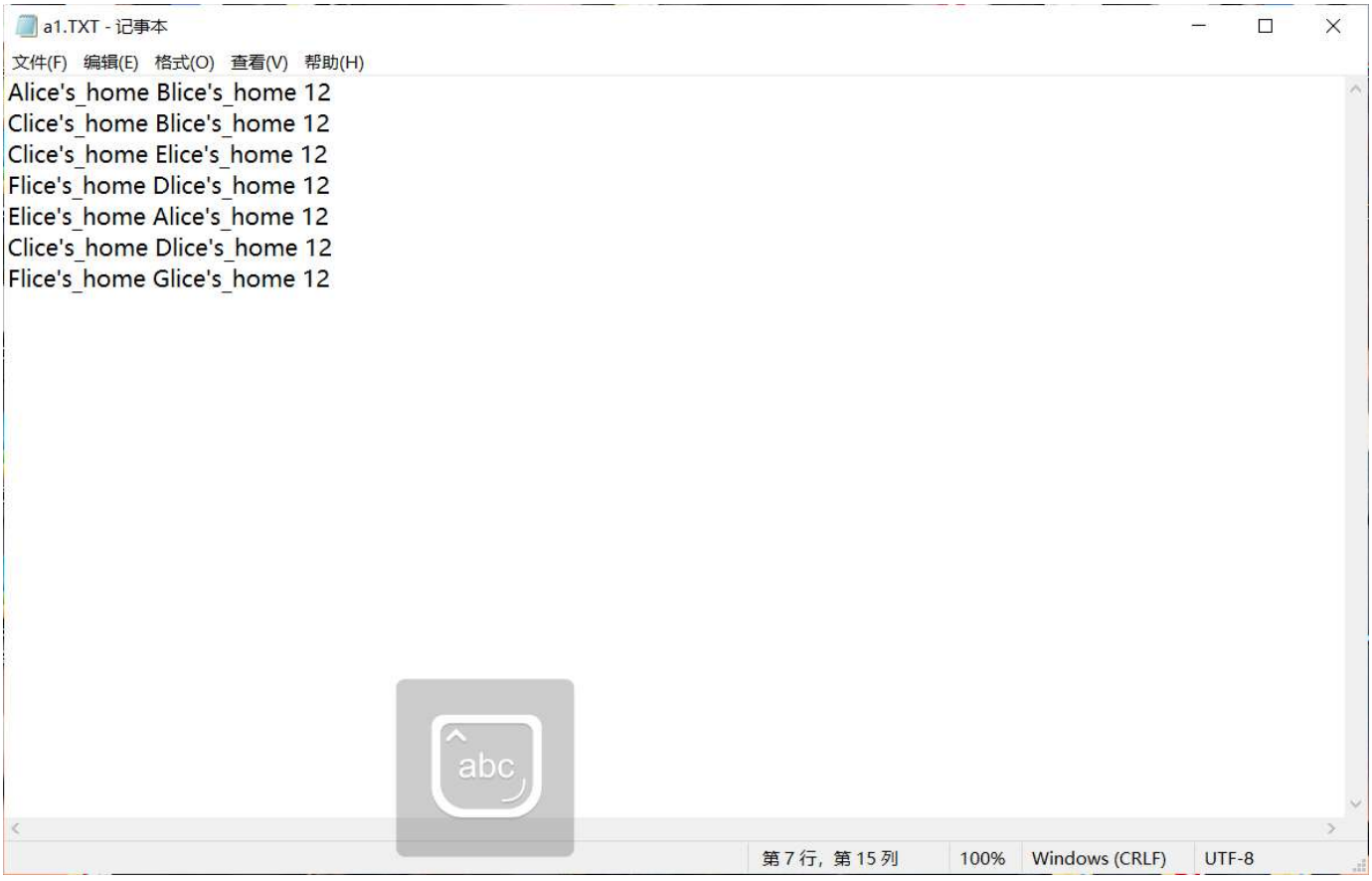
大文件压缩大小


b.TXT 属性
常规
安全
详细信息
以前的版本
b.TXT
文件类型:
文本文档 (.TXT)
打开方式:
记事本
更改(C)...
位置:
C:\Users\pty\Desktop
大小:
536 MB (562,456,156 字节)
占用空间:
536 MB (562,458,624 字节)
创建时间:
2021年12月9日，18:03:39
修改时间:
2021年12月28日，14:15:59
访问时间:
2021年12月28日，18:39:33
属性:
只读(R)
隐藏(H)
高级(D)...
确定
取消
应用(A)

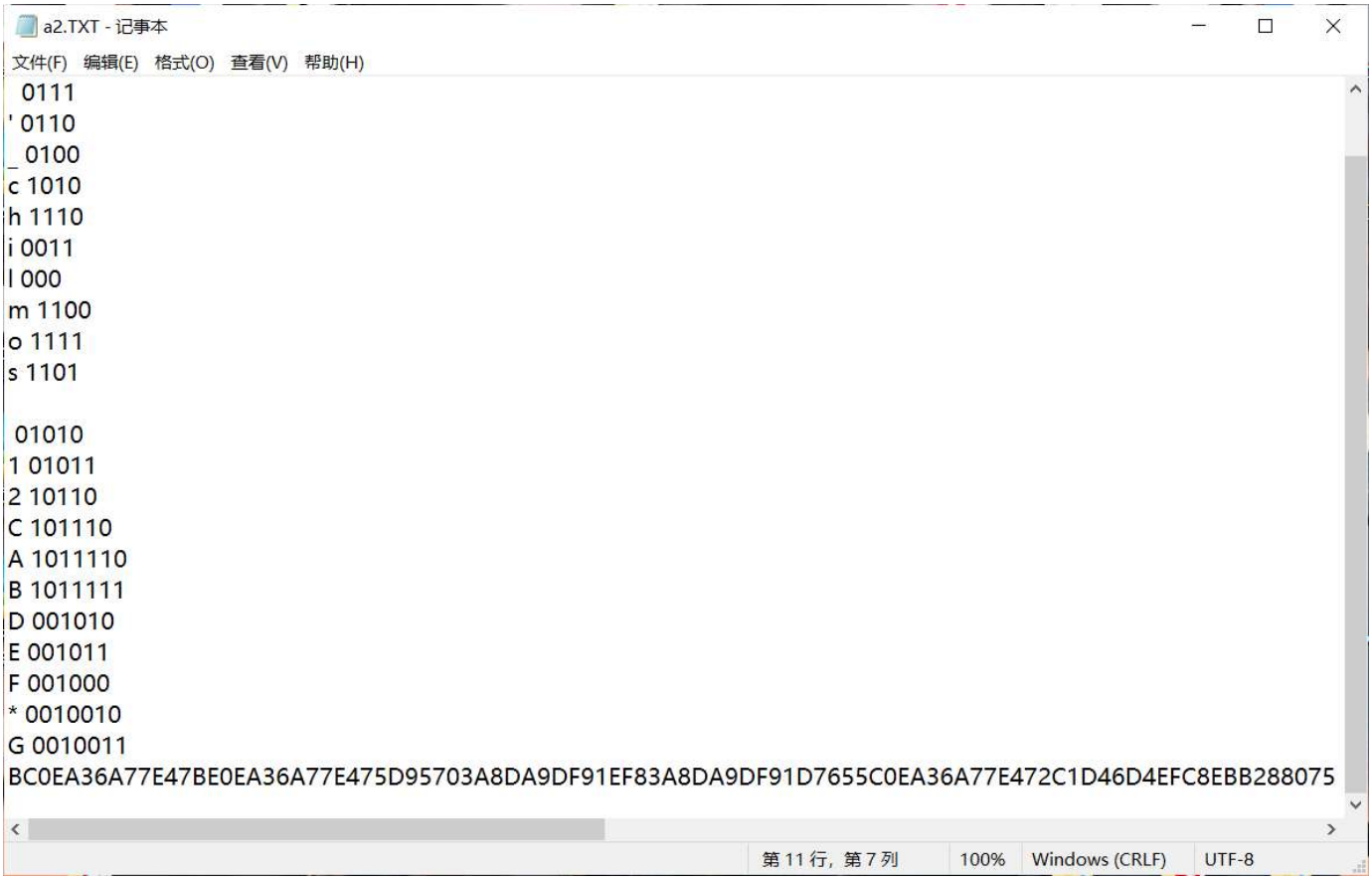
小文件运算结果



拓展构图展示

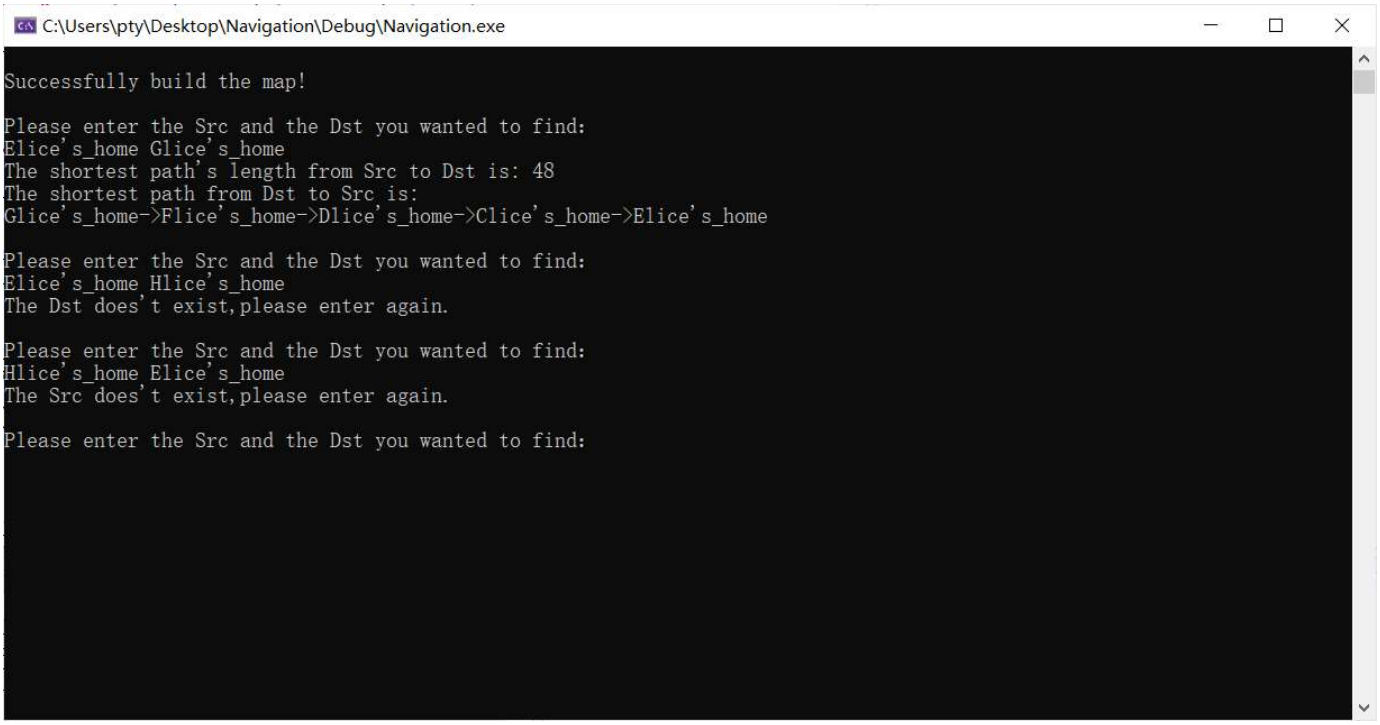


拓展压缩结果展示



***注：这里使用了16进制压缩；对于小文件来说，或许Huffman压缩效果并不明显，但对于大文件来说，其效果有了显著提高。**

拓展运行结果展示



【实验总结】

本人大约共耗费了12h来完成本次实验，先后实现了三个版本的导航系统。在实现Huffman压缩的过程中，本人充分认识到了程序结构性、命名规范性的重要性；在学习二进制压缩的过程中，本人对电脑存储、处理文件的概念和方式有了更加充分的认识，同时还对不同文件读取方式有了更深刻的了解；在书写Dijkstra算法的过程中，本人学习了优先队列的应用和实现，并亲自上手进行实践，收获巨大。

总的来说，此次实验让本人对“编程”有了更深刻的认识。

【附录】

①Navigation.cpp 自定义拓展项目文件，支持地名映射及相关文件压缩。

②lab03源代码.cpp 要求实现的非自定义拓展项目源文件，支持文件的压缩，及最短路算法堆优化、进一步的常数优化。

③Dijkstra*handwriting*heap.cpp 手动实现堆优化的Dijkstra算法。

④Navigation_zippro.cpp 文件压缩程序，可以将文件压缩至O（2|E|+|V|）大小

⑤Instruction.txt 拓展内容文件及手写堆版Dijkstra输入输出说明