哈尔滨工业大学

**<<数据结构与算法>>**

**实验报告**

**(2024年秋季学期)**

|  |  |
| --- | --- |
| **姓名：** | **郭屹鸣** |
| **学号：** | **2023111958** |
| **学院：** | **计算机学院** |
| **教师：** | **翟德明** |

实验一 线性结构及其应用

## 一、实验内容

实验内容：

一个迷宫可以看成是由 m×n 个房间组成的矩形，迷宫内部的每个房间有 8

个方向，每个方向或者有障碍（如墙）而不能通过，或者无障碍（如有门）而

能通过。入口为左上角房间，出口为右下角房间，问是否有简单路径从入口到

出口，若有则输出一条这样的路径；否则，提示“迷宫无入口到出口路经”。 实验要求：

1. 设计一个迷宫及其障碍的表示方式，并能随机或手动生成迷宫，并以适当方

式展示。

2. 设计并实现一个非递归的算法，输出从入口到出口的一条路径（如存在）。

3. 设计并实现一个递归的算法，找出从入口到出口的一条路径（如存在）。

4. 如果有多条路径，设计并实现一个算法找到步数最少的路径（捷径）。

5. 如果有多条路径，设计并实现一个算法找到所有路径。

6. 以适当的方式展示迷宫和所走路径等

我完成了以上6个任务

(1)用深搜和随机扰动生成了8方向的迷宫并且进行了可视化，支持输入到txt文本中，也可以选择手动输入（使用txt文本中现有的迷宫，不再随机生成）

(2)使用非递归栈进行DFS搜索，输出了一条路径，支持输入到txt文本中

(3)使用递归算法进行DFS深搜，输出了全部路径（也可以输出一条），支持输入到txt文本中

(4)使用基于队列的BFS广搜，输出的最短路路径，支持输入到txt文本中

(5)可视化了迷宫，并且为迷宫的房间标号，输出了对应的标号顺序表示路径

## 二、实验过程及结果

实验采用了模块化编程，采用了面向对象编程的思想理念，完成了自用栈的设计和算法的实现，完成了本次实验的全部任务。

（1）**自用栈的实现**

使用链表实现了栈的基本功能，压入，弹出，获得栈顶元素，判断空。还有自用的功能如展示栈元素，随机选取并且弹出栈中的元素(和随机扰动相关)，清空栈等。

（2）**随机生成迷宫**

* 结构存储

我从DFS联想到图论中两点之间寻路的问题，同时考虑到房间序号和房间在迷宫中的坐标存在一一映射的关系。所以把房间抽象为图的节点，利用邻接矩阵来存储整个图(也是为了方便后续随机扰动联通功能的实现)。

* DFS实现

使用自用栈来进行DFS随机生成主路，利用栈的压入弹出来实现回溯。并且将所生成的路径用邻接矩阵的形式表示(相当于把图上孤立的点建立连接关系)

* 随机扰动

将随机算法应用于邻接矩阵，联通某些节点(联通的对数和迷宫规模有关)，增加迷宫的复杂性。

* 可视化

基于邻接矩阵可以获取房间之间的联通关系，结合房间序号和房间的位置坐标将整个迷宫可视化。

（3）**迷宫寻路**

* 类的设计

寻路类Maze\_Operator继承了随机生成迷宫类Graze\_Maze，并且添加了读取txt中迷宫和走迷宫的功能。（本来想的是读取迷宫后再将迷宫抽象为图并且用邻接矩阵存储，相当于把生成过程反过来。但是想尝试一下探测路径的方法，就没有抽象成图）

* 栈实现非递归DFS

设计函数Stack\_DFS\_Only\_Pate()，函数使用栈来模拟递归过程。初始时将起点压入栈中，并不断通过遍历可能的方向查找相邻的未访问节点。当找到合适的路径时，将路径保存并输出。如果遇到终点，计数路径并保存结果。

* 递归DFS输出全部路径

设计函数Recursion\_DFS\_All\_Path(),函数通过递归实现深度优先搜索，查找并输出从起点到终点的所有可能路径。递归函数 DFS() 会在每次找到终点时保存当前路径，并在退出时将当前节点标记为未访问，以便继续查找其他路径。路径保存后会输出到文件中并在控制台显示

* 栈实现BFS

设计函数BFS\_Shortest\_Distance(),函数通过广度优先搜索（BFS）寻找从起点到终点的最短路径。使用队列来记录当前节点的相邻节点，并通过追踪每个节点的父节点来重构最短路径。当搜索到终点时，将路径保存并输出。如果没有找到路径，则输出“无路径”。

输出结果如图1至图3

背景图案

描述已自动生成 文本

描述已自动生成

图1：随机生成迷宫 图2：最短路径与单条路径

背景图案

描述已自动生成

背景图案

描述已自动生成

## 图片包含 文本 描述已自动生成

图3：全部路径(共112条)

## 三、实验心得

* **遇到的问题和解决方法**

（1）构思问题:迷宫可以走八个方向。我最开始以为就是简单的四边形走斜线。但是我发现根据题目的描述，应当八个方向对应不同的门，一个房间关联的两个斜角的连通性可以不同。这样就不能单独把迷宫看成简单的四边形，而是应该抽象成八边形。我找到了对应房间序号和坐标的关系，为房间建立了结构体，确定了坐标位置并可视化。

（1）生成迷宫：我想使用DFS深搜随机生成路径，但是在生成之后发现由于DFS的性质，生成的迷宫主路径过于单一。由于我是用栈实现生成迷宫的，所以随机取消终点的visited值来获得多条路径的方法不方便操作(如果用递归，就和我走迷宫的算法重复了)，所以就引入随机扰动，随机打通路径。

（3）走迷宫的实现：走迷宫直接继承了生成迷宫的类，但是编写完基本的函数和功能之后我发现代码无法运行。在经过漫长的debug后我发现栈溢出了。溯源后发现我的父类中存储邻接矩阵和迷宫的数组开的太大了。于是改小MVNUM并且弃用栈，改用C++中的动态数组vector

（4）DFS递归法输出所有路径：写完功能函数之后，我成功输出了单条路径，随后我略加改动过代码逻辑想输出全部路径，但是程序卡死。又是经过漫长的debug之后我发现自己实现用于存储路径栈(实验要求自己实现吧应该)又溢出了。同一天内两次溢出，我怒而改用标准库的stack，用堆存储解决了问题。

* **体会和感受**

我大概花了两半天的时间写完了实验。半天构思，两实现(两次栈溢出耗了快一天)。我有很多开心的瞬间。当看到迷宫生成出来的时候，当看到寻路可行的时候，解决栈的bug时候，最后实现功能程序全跑通的时候。同时，这次实验也大大提高了我的抽象建模和debug能力，我更加理解C++的编码规范和面向对象的理念。进步很大，很开心。

1. **源代码**
2. M\_Stack.h
3. #include <iostream>
4. #include <ctime>
5. #include <random>
6. #include <stdexcept>
7. template <typename T>
8. class M\_Stack
9. {
10. public:
11. typedef struct link
12. {
13. T data;
14. struct link \*next;
15. }LINK;
16. private:
17. LINK \*head;
18. int num;
19. public:
20. M\_Stack(){
21. head = new LINK;
22. head->next = nullptr;
23. this->num = 0;
24. }
26. ~M\_Stack()
27. {
28. LINK \*p = head,\*pr = nullptr;
29. while(p!=nullptr)
30. {
31. pr = p->next;
32. delete p;
33. p = pr;
34. }
35. }
36. void Push(T value)
37. {
38. LINK \*t = head;
39. LINK \*p = new LINK;
40. while(t->next!=nullptr) t = t->next;
41. t->next = p;
42. p->next = nullptr;
43. p->data = value;
44. this->num ++;
45. }
46. T Pop()
47. {
48. if(IsEmpty())
49. {
50. throw std::runtime\_error("The stack is Empty");
51. return T();
52. }
53. LINK \*p = head;
54. while(p->next->next != nullptr) p = p->next;
55. T value = p->next->data;
56. delete p->next;
57. p->next = nullptr;
58. this->num--;
59. return value;
60. }
61. T Get\_Top()
62. {
63. if(IsEmpty())
64. {
65. throw std::runtime\_error("The stack is Empty");
66. return T();
67. }
68. LINK \*p = head;
69. while(p->next!= nullptr) p = p->next;
70. return p->data;
71. }
72. bool IsEmpty()
73. {
74. return head->next == nullptr ? true : false;
75. }
76. T Random\_Select()
77. {
79. if(IsEmpty()) throw std::runtime\_error("The stack is Empty");
80. std::mt19937 generator(static\_cast<unsigned int>(std::time(nullptr)));
81. std::uniform\_int\_distribution<int> distribution(1,num);
82. int random\_number = distribution(generator);
83. int count = 1;
85. LINK \*p = head,\*pr = p->next;
86. while(pr!=nullptr)
87. {
88. if(count == random\_number)
89. {
90. p->next = pr->next;
91. T value = pr->data;
92. delete pr;
93. pr = p->next;
94. this->num--;
95. return value;
96. }
97. else
98. {
99. p = p->next;
100. pr = p->next;
101. count ++;
102. }
103. }
104. return T();
105. }
106. void Display()
107. {
108. LINK \*p = head->next;
109. while(p!=nullptr)
110. {
111. std::cout<<p->data<<" ";
112. p = p->next;
113. }
114. std::cout<<std::endl;
115. }
116. void Clear()
117. {
118. LINK \*p = head->next,\*pr = nullptr;
119. while(p!=nullptr)
120. {
121. pr = p->next;
122. delete p;
123. p = pr;
124. }
125. }
126. };
127. Generate\_Maze.h
128. #include "M\_Stack.h"
129. #include <utility>
130. #include <vector>
131. #include <fstream>
132. constexpr int MVNUM = 50;
133. class Graze\_Maze
134. {
135. public:
136. typedef struct
137. {
138. std::pair<int, int> Position;
139. bool visited;
140. int label;
141. } Vertix;
142. int row, col;  *// 将 row 和 col 放到 vexs 之前*
143. std::vector<Vertix> vexs;
145. std::vector<std::vector<bool>> arcs;
146. std::vector<std::vector<char>> Maze;
147. *//bool arcs[MVNUM][MVNUM];*
148. *//char Maze[MVNUM][MVNUM];*
149. int vexnum;
150. typedef struct
151. {
152. std::pair<int, int> Position;
153. int distance;
154. }direction;
155. std::vector<direction> Direction= {
156. {{3,0},1},
157. {{-3,0},-1},
158. {{0,3},col},
159. {{0,-3},-col},
160. {{3,3},col+1},
161. {{-3,-3},-col-1},
162. {{3,-3},-col+1},
163. {{-3,3},col-1}};
164. public:
165. Graze\_Maze(int \_row, int \_col): row(\_row), col(\_col), vexs(\_col\*\_row+1)
166. {
167. arcs.resize(MVNUM, std::vector<bool>(MVNUM, false));
168. Maze.resize(3 \* row, std::vector<char>(3 \* col, ' '));
170. Inital\_Maze();
171. }
172. ~Graze\_Maze(){};
173. void Inital\_Maze()
174. {
175. vexnum = row \* col;
176. for (int i = 0; i <= vexnum; i++)
177. {
178. vexs[i].label = i;
179. vexs[i].visited = false;
180. vexs[i].Position.first = ((i-1) / col) \* 3 + 1;
181. vexs[i].Position.second = ((i-1) % col) \* 3 + 1;
182. }
183. for (int i = 1; i <= vexnum; i++)
184. {
185. for (int j = 1; j <= i; j++)
186. {
187. i == j?arcs[i][j] = true : arcs[i][j] = false;
188. arcs[j][i] = arcs[i][j];
189. }
190. }
191. }
193. void Random\_Generate\_Maze()
194. {
195. M\_Stack<Vertix> S;
196. vexs[1].visited = true;*//下标从1开始*
197. S.Push(vexs[1]);
199. while(!S.IsEmpty())
200. {
201. int count = 0;
202. std::vector<int> record\_nearby;*//下标从0开始*
204. Vertix locate = S.Pop();
205. for( std::vector<direction>::iterator it = Direction.begin();it!=Direction.end();it++)
206. {
207. int x = locate.Position.first + it->Position.first;
208. int y = locate.Position.second + it->Position.second;
209. int new\_label = 0;
210. if(Locate\_Position(x,y,new\_label)&&(!vexs[new\_label].visited))
211. {
212. count++;
213. if(count == 1) S.Push(locate);
214. record\_nearby.push\_back(new\_label);
215. }
216. }
218. if(count)
219. {
220. std::random\_device rd;
221. std::mt19937 gen(rd()); *// 使用 Mersenne Twister 生成器*
222. std::uniform\_int\_distribution<int> dist(0,record\_nearby.size()-1); *// 设定范围*
223. int random\_label = record\_nearby[dist(gen)];
224. vexs[random\_label].visited = true;
225. arcs[locate.label][random\_label] = true;
226. arcs[random\_label][locate.label] = true;
227. S.Push(vexs[random\_label]);
229. }
230. }
231. Random\_Generate\_Disturbs();
232. Visual\_Maze();
233. Save\_Maze();
234. std::cout<<"Yeah";
235. }
237. inline void Display\_Arcs()
238. {
239. for(int i = 1;i <= vexnum; i++)
240. {
241. for(int j = 1;j <= i; j++)
242. {
243. if(arcs[i][j] == 1 && i!=j)
244. std::cout<<i<<" "<<j<<std::endl;
245. }
246. }
247. }
248. inline bool Locate\_Position(int x,int y,int &new\_label)
249. {
250. for(int k = 1;k <= vexnum; k++){
251. if(vexs[k].Position.first == x  && vexs[k].Position.second == y)
252. {
253. new\_label = k;
254. return true;
255. }
256. }
257. return false;
258. }
259. protected:
260. inline void Display\_Vex\_Position()
261. {
262. for(int i=1;i<=vexnum;i++)
263. {
264. std::cout<<vexs[i].Position.first<<" "<<
265. vexs[i].Position.second;
266. std::cout<<vexs[i].label<<std::endl;
267. }
268. }
270. bool Check\_Is\_All\_Visited()
271. {
272. for(int i = 1;i <= vexnum ;i++)
273. {
274. if(!vexs[i].visited) return false;
275. }
276. return true;
277. }
279. void Random\_Generate\_Disturbs()
280. {
281. int upper = 2\*sqrt(col\*row),count = 0;
282. while(count<upper)
283. {
284. std::mt19937 generator(static\_cast<unsigned int>(std::time(nullptr)));
285. std::uniform\_int\_distribution<int> distribution(1,vexnum-1);
286. int i = distribution(generator);
287. int j = i+1;
288. if((!arcs[i][j]) && (i%col!=0) && (j%col!=0))
289. {
290. arcs[i][j] = true;
291. count++;
292. }
293. }
294. }
295. inline void Display\_Maze()
296. {
297. for(int i = 0;i < 3\*row;i++)
298. {
299. for(int j = 0;j < 3\*row;j++)
300. {
301. std::cout<<Maze[i][j];
302. }
303. std::cout<<std::endl;
304. }
305. }
306. void Visual\_Maze()
307. {
308. for(int i = 0;i < 3\*row; i++)
309. {
310. for(int j = 0;j < 3\*col;j++)
311. {
312. int mod1 = i%3,mod2 = j%3;
313. switch (mod1)
314. {
315. case 0:
316. if(mod2 == 0) Maze[i][j] ='/';
317. else if(mod2 == 1) Maze[i][j] = '-';
318. else Maze[i][j] = '\\';
319. break;
320. case 1:
321. Maze[i][j] = '|';
322. break;
323. default:
324. if(mod2 == 0) Maze[i][j] ='\\';
325. else if(mod2 == 1) Maze[i][j] = '-';
326. else Maze[i][j] = '/';
327. break;
328. }
329. }
330. }
331. for(int k = 1;k <= vexnum; k++)
332. {
333. int x = vexs[k].Position.first;
334. int y = vexs[k].Position.second;
335. Maze[x][y] = '\*';
336. }
337. std::cout<<std::endl;
338. for(int i = 1;i <= vexnum; i++)
339. {
340. for(int j = 1;j <= vexnum; j++)
341. {
342. if(i!=j&&arcs[i][j]==1)
343. {
344. int distance = abs(vexs[i].label-vexs[j].label);
346. int x\_i = vexs[i].Position.first;
347. int y\_i = vexs[i].Position.second;
348. int x\_j = vexs[j].Position.first;
349. int y\_j = vexs[j].Position.second;
350. int min\_x = x\_i < x\_j ? x\_i : x\_j;
351. int min\_y = y\_i < y\_j ? y\_i : y\_j;
352. int max\_y = y\_i > y\_j ? y\_i : y\_j;
354. if(distance == 1)
355. {
356. Maze[min\_x][min\_y+1] = ' ';
357. Maze[min\_x][min\_y+2] = ' ';
358. }
359. else if(distance == col)
360. {
361. Maze[min\_x+1][min\_y] = ' ';
362. Maze[min\_x+2][min\_y] = ' ';
363. }
364. else if(distance == col+1)
365. {
366. Maze[min\_x+1][min\_y+1] = ' ';
367. Maze[min\_x+2][min\_y+2] = ' ';
368. }
369. else if(distance == col-1)
370. {
371. Maze[min\_x+1][max\_y-1] = ' ';
372. Maze[min\_x+2][max\_y-2] = ' ';
373. }
374. else
375. {
376. throw std::runtime\_error("The geneation of the maze failed");
377. }
378. }
380. }
381. }
382. Display\_Maze();
383. std::cout<<std::endl;
384. }
385. void Save\_Maze()
386. {
387. std::ofstream outFile("Maze.txt");
388. if (!outFile)
389. {
390. std::cerr << "Fail to open the txt" << std::endl;
391. return;
392. }
393. for (int i = 0; i < 3 \* row; i++)
394. {
395. for (int j = 0; j < 3 \* col; j++)
396. {
397. outFile << Maze[i][j];
398. }
399. if(i!=3\*row-1) outFile << std::endl;
400. }
401. outFile.close();
402. std::cout << "Succeed!" << std::endl;
403. }
404. };
405. Maze\_Operator.h
406. #include "Generate\_Maze.h"
407. #include <iostream>
408. #include <stack>
409. #include <queue>
410. #include <string>
411. #include <algorithm>
412. class Maze\_Operator:public Graze\_Maze
413. {
414. public:
415. std::vector<std::vector<char>> Received\_Maze;
416. Maze\_Operator(int \_row, int \_col)
417. : Graze\_Maze(\_row, \_col),Received\_Maze(3\*\_row, std::vector<char>(3\*\_col, '0')){
419. }
421. ~Maze\_Operator(){};
422. void Load\_Maze()
423. {
424. std::ifstream inFile("Maze.txt");
425. if (!inFile.is\_open()) throw std::runtime\_error("Failed to open txt");
426. for (int i = 0; i < 3 \* row; i++)
427. {
428. for (int j = 0; j < 3 \* col; j++)
429. {
430. Received\_Maze[i][j] = inFile.get();
431. }
432. inFile.get();
433. }
434. inFile.close();
435. std::cout << "Maze loaded successfully!" << std::endl;
436. for(int i=0;i<3\*row;i++)
437. {
438. for(int j=0;j<3\*col;j++)
439. {
440. std::cout<<Received\_Maze[i][j];
441. }
442. std::cout<<std::endl;
443. }
444. }
445. void Stack\_DFS\_Only\_Path()
446. {
447. int count\_path = 0;
448. std::stack<int> S;
449. *//M\_Stack<int> S;*
450. vexs[1].visited = true;
451. S.push(1);
452. while(!S.empty()&&!count\_path)
453. {
454. int locate = S.top();
455. S.pop();
457. for( std::vector<direction>::iterator it = Direction.begin();it!=Direction.end();it++)
458. {
459. bool jump\_flag = false;
460. int Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2,neighbor;
461. Detect\_Direction(locate,it,Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2);
462. Locate\_Position(Ver\_x, Ver\_y, neighbor);
463. if(Received\_Maze[x\_1][y\_1] == ' ' && Received\_Maze[x\_2][y\_2] == ' '
464. &&(!vexs[neighbor].visited))
465. {
466. S.push(locate);
467. S.push(neighbor);
468. vexs[neighbor].visited = true;
469. jump\_flag = true;
470. }
471. if(vexs[vexnum].visited)
472. {
473. Save\_Stack\_Only(S,"Stack\_Only\_Path.txt");
474. count\_path++;
475. std::cout<<"Stack\_Path:"<<count\_path<<std::endl;
476. Stack\_Display(S);
477. std::cout<<std::endl;
478. }
479. if(jump\_flag) break;
480. }
481. }
482. }
484. void Recursion\_DFS\_All\_Path()
485. {
486. *//M\_Stack<int> record;*
487. *//std::vector<M\_Stack<int>> All\_Paths;*
488. int time = 0;
489. std::stack<int> record;
490. std::vector<std::stack<int>> All\_Paths;
491. DFS(1,time,record,All\_Paths);
492. std::cout<<"The total paths are "<<time<<std::endl;
493. Save\_DFS\_All\_Paths(All\_Paths);
494. }
495. void BFS\_Shortest\_Distance()
496. {
497. std::stack<int>Path;
498. std::queue<int> Distance;
499. std::vector<int>father;
500. father.resize(100,0);
502. int locate = 1;
503. father[1] = 0;
504. vexs[locate].visited = true;
505. Distance.push(locate);
506. while(!vexs[vexnum].visited && !Direction.empty())
507. {
508. locate = Distance.front();
509. Distance.pop();
510. for (std::vector<direction>::iterator it = Direction.begin(); it != Direction.end(); it++)
511. {
512. int Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2,neighbor;
513. Detect\_Direction(locate,it,Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2);
514. Locate\_Position(Ver\_x, Ver\_y, neighbor);
515. if (Received\_Maze[x\_1][y\_1] == ' ' && Received\_Maze[x\_2][y\_2] == ' ' && !vexs[neighbor].visited)
516. {
517. Distance.push(neighbor);
518. *//std::cout<<neighbor<<" " ;*
519. vexs[neighbor].visited = true;
520. father[neighbor] = locate;
521. }
522. }
523. }
524. if(vexs[vexnum].visited)
525. {
526. int son = vexnum,dad = father[son];
527. while(dad)
528. {
529. dad = father[son];
530. Path.push(son);
531. son = dad;
532. }
533. Save\_Shortest\_Path(Path,"Shortest\_Path.txt");
534. std::cout<<"The shortest path is:";
535. while(!Path.empty())
536. {
537. std::cout<<Path.top()<<" ";
538. Path.pop();
539. }
540. std::cout<<std::endl;
541. }
542. else std::cout<<"No path"<<std::endl;
543. }
544. protected:
545. void Detect\_Direction(int locate,std::vector<direction>::iterator it,
546. int &Ver\_x,int &Ver\_y,int &x\_1,int &y\_1,int &x\_2,int &y\_2)
547. {
548. Ver\_x = vexs[locate].Position.first + it->Position.first;
549. Ver\_y = vexs[locate].Position.second + it->Position.second;
550. x\_1 = vexs[locate].Position.first + it->Position.first / 3;
551. y\_1 = vexs[locate].Position.second + it->Position.second / 3;
552. x\_2 = vexs[locate].Position.first + 2 \* it->Position.first / 3;
553. y\_2 = vexs[locate].Position.second + 2 \* it->Position.second / 3;
554. }
555. void DFS(int locate, int &time,std::stack<int>& record, std::vector<std::stack<int>>& All\_Paths)
556. {
557. vexs[locate].visited = true;
558. record.push(locate);
559. *//std::cout<<"Entering node"<<<locate<<std::endl;*
560. if (locate == vexnum)
561. {
562. time++;
563. std::cout<<"The path"<<time<<":";
564. Stack\_Display(record);
565. All\_Paths.push\_back(record);
566. record.pop();
567. vexs[locate].visited = false;
568. *//std::cout<<"Exiting1"<<locate<<"Found"<< std::endl;*
569. return;
570. }
571. for (std::vector<direction>::iterator it = Direction.begin(); it != Direction.end(); it++)
572. {
573. int Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2,neighbor;
574. Detect\_Direction(locate,it,Ver\_x,Ver\_y,x\_1,y\_1,x\_2,y\_2);
575. Locate\_Position(Ver\_x, Ver\_y, neighbor);
576. if (Received\_Maze[x\_1][y\_1] == ' ' && Received\_Maze[x\_2][y\_2] == ' ' && !vexs[neighbor].visited)
577. {
578. DFS(neighbor,time,record,All\_Paths);
579. *//if(neighbor!=vexnum) record.pop();这个pop和下面那个任选都行()....*
580. }
581. }
583. record.pop();
584. vexs[locate].visited = false;
585. *//std::cout<<"Exiting2<<locate<<std::endl;*
586. }
587. void Stack\_Display(std::stack<int> &record)
588. {
589. std::stack<int> copy = record,temp;
590. std::vector<int> out\_put;
591. while(!copy.empty())
592. {
593. temp.push(copy.top());
594. copy.pop();
595. }
596. while(!temp.empty())
597. {
598. out\_put.push\_back(temp.top());
599. temp.pop();
600. }
601. for(std::vector<int>::iterator it = out\_put.begin();it!= out\_put.end();it++)
602. {
603. std::cout<<\*it<<" ";
604. }
605. std::cout<<std::endl;
606. }
607. void Save\_Stack\_Only(std::stack<int> S,std::string location)
608. {
610. std::ofstream outFile(location);
611. if (!outFile)
612. {
613. std::cerr << "Fail to open the txt" << std::endl;
614. return;
615. }
616. std::vector<int> temp;
617. while (!S.empty()) {
618. temp.push\_back(S.top());
619. S.pop();
620. }
621. std::reverse(temp.begin(), temp.end());
622. for (std::vector<int>::iterator it=temp.begin();it!=temp.end();it++)
623. {
624. outFile << \*it << " ";
625. }
626. outFile.close();
627. }
629. void Save\_DFS\_All\_Paths(std::vector<std::stack<int>> All\_Paths)
630. {
632. std::ofstream outFile("All\_Path.txt");
633. if (!outFile)
634. {
635. std::cerr << "Fail to open the txt" << std::endl;
636. return;
637. }
638. int time = 0;
640. for (std::vector<std::stack<int>>::iterator it = All\_Paths.begin(); it != All\_Paths.end(); it++)
641. {
642. time++;
643. outFile << "Path " << time << ": ";
644. std::vector<int> temp;
645. while (!(\*it).empty())
646. {
647. temp.push\_back((\*it).top());
648. (\*it).pop();
649. }
650. std::reverse(temp.begin(), temp.end());
651. for (auto it\_ = temp.begin();it\_!=temp.end();it\_++)*//类型不匹配，用auto*
652. {
653. outFile<<\*it\_;
654. if (it\_!=temp.end()-1) outFile << " -> ";
655. }
656. outFile << std::endl;
657. }
658. outFile.close();
659. std::cout << "Saved Succeed!" << std::endl;
660. }
661. void Save\_Shortest\_Path(std::stack<int> S,std::string location)
662. {
663. std::ofstream outFile(location);
664. if (!outFile)
665. {
666. std::cerr << "Fail to open the txt" << std::endl;
667. return;
668. }
669. std::vector<int> temp;
670. while (!S.empty()) {
671. temp.push\_back(S.top());
672. S.pop();
673. }
674. for (std::vector<int>::iterator it=temp.begin();it!=temp.end();it++)
675. {
676. outFile << \*it << " ";
677. }
678. outFile.close();
679. }
680. };
681. main.cpp
682. #include "Maze\_Operator.h"
683. #include <iostream>
684. int main(void)
685. {
686. std::cout<<"You can generate the maze Randomly,or directly use the maze\_1.txt"<<std::endl;
687. std::cout<<"1.generate the maze Randomly  2.use the maze\_1.txt"<<std::endl;
688. int ch,row,col;
689. std::cin>>ch;
690. if(ch==1)
691. {
692. std::cout<<"Input row and col,both<=6"<<std::endl;
693. std::cin>>row>>col;
694. Graze\_Maze maze(row,col);
695. std::cout<<"---Slow------Slow---Slow-----Slow--------"<<std::endl;
696. maze.Random\_Generate\_Maze();
697. }
698. else
699. {
700. row = 6;
701. col = 6;
702. }
703. Maze\_Operator M1(row,col);
704. M1.Load\_Maze();
705. std::cout<<"----------------------------------------"<<std::endl;
706. std::cout<<"1.Only one path ? 2.Show all paths and the shortest one"<<std::endl;
708. std::cin>>ch;
709. if(ch==1)
710. {
711. std::cout<<"----------------------------------------"<<std::endl;
712. M1.Stack\_DFS\_Only\_Path();
713. std::cout<<"----------------------------------------"<<std::endl;
714. }
715. else
716. {
717. std::cout<<"----------------------------------------"<<std::endl;
718. M1.Recursion\_DFS\_All\_Path();
719. std::cout<<"----------------------------------------"<<std::endl;
720. M1.BFS\_Shortest\_Distance();
721. std::cout<<"----------------------------------------"<<std::endl;
722. }
724. return 0;
725. }