



中国石油大学(北京)克拉玛依校区  
CHINA UNIVERSITY OF PETROLEUM - BEIJING AT KARAMAY

中国石油大学（北京）克拉玛依校区  
2023—2024 学年第三学期

## 《数据结构与程序综合实践》

# 实 践 报 告

班级： 计算机 23-3 班

姓名： 胡林森

学号： 2023015509

时间： 2024 年 7 月 2 日

## 摘 要

**摘要：**本文介绍了笔者在《数据结构与程序综合实践》课程中完成的项目，分别涉及文件加密解密、树搜索和图搜索算法的实现和应用。实践 1 使用异或算法和栈数据结构实现了文件的加密和解密功能，实践 2 树搜索比较了三种树搜索算法（简单线性搜索、BFS、DFS）的优缺点。实践 3 介绍了四种图搜索算法（DFS、BFS、迪杰斯特拉算法、A\*算法），并以八数码九宫格问题为例，展示了 A\*算法在求解最优路径问题上的优势。

**关键词：**数据结构，文件加密，树搜索，图搜索，算法实现，八数码问题，A\*算法

---

目 录

|                              |    |
|------------------------------|----|
| 摘 要 .....                    | 2  |
| 实践 1: 文件加密/解密 .....          | 5  |
| 1.1 实现方法.....                | 5  |
| 1.2 数据结构.....                | 5  |
| 1.3 核心算法.....                | 5  |
| 1.4 应用方法.....                | 6  |
| 1.5 程序运行截图.....              | 8  |
| 实践 2: 树搜索 .....              | 9  |
| 2.1 实现方法.....                | 9  |
| 2.2 数据结构.....                | 10 |
| 2.3 核心算法.....                | 12 |
| 2.4 应用方法.....                | 13 |
| 2.5 程序运行截图.....              | 14 |
| 实践 3: 图搜索 .....              | 15 |
| 3.1 实现方法.....                | 15 |
| 3.1.1 深度优先搜索 (DFS) .....     | 15 |
| 3.1.2 广度优先搜索 (BFS) .....     | 15 |
| 3.1.3 A*搜索算法.....            | 15 |
| 3.1.4 小结 .....               | 16 |
| 3.2 数据结构.....                | 16 |
| 3.3 核心算法.....                | 16 |
| 3.3.1 DFS.....               | 16 |
| 3.3.2 BFS.....               | 17 |
| 3.3.3 A*算法.....              | 18 |
| 3.4 应用方法.....                | 19 |
| 3.5 程序运行截图.....              | 23 |
| 实践 4: 图搜索问题的应用拓展 .....       | 24 |
| 4.1 迷宫问题介绍.....              | 24 |
| 4.1.1 迷宫问题的解法——DFS .....     | 24 |
| 4.1.2 迷宫问题的解法——最优路径 DFS..... | 25 |

|       |                            |    |
|-------|----------------------------|----|
| 4.1.3 | 迷宫问题的解法——A*算法 .....        | 25 |
| 4.2   | 再谈九宫格.....                 | 25 |
| 实践 5: | 组合优化问题的拓展 .....            | 26 |
| 5.1   | TSP 问题——以管道建设规划为例 .....    | 26 |
| 5.2   | 背包问题.....                  | 26 |
| 5.2.1 | 背包问题的解法——动态规划法 .....       | 26 |
| 5.2.2 | 背包问题的解法——递归回溯法 (DFS) ..... | 27 |

## 实践1：文件加密/解密

### 1.1 实现方法

加密利用数学方法将明文转化为密文，从而达到保护数据的目的。生产生活中对加密的需求很多，例如软件公司的源代码程序文件，工程设计公司的图纸文件，制造企业的配方，军工企业涉及的军事秘密等，对于企业来说这些文件都非常重要，直接关系到企业的经济利益和国家机密，不希望流失到企业外部、竞争对手或敌对势力手里。但是，怎样才能使这些文件不被流失呢？加密是通常的解决办法。

本次实践对文件进行简单加密，以期提高对文件读取、简单加密算法的掌握。

实现方法：

- 1：通过文件操作语句打开数据文件，并执行文件读写操作。
- 2：采用异或算法对文件内容执行初步的加密处理。
- 3：利用堆栈数据结构实现字符串的逆序排列，以执行进一步的加密步骤。
- 4：再次调用加密函数，对文件执行逆向加密过程以实现解密。

### 1.2 数据结构

```
/* 顺序栈结构体定义 */
typedef struct
{
    ElemType *base; /* 栈底指针，在初始化之前和销毁之后，base 值为 NULL */
    ElemType *top; /* 栈顶指针，指向栈顶元素的下一个位置 */
    int stacksize; /* 当前栈已分配的内存空间大小（以元素个数为单位） */
} SqStack;
```

### 1.3 核心算法

```
/**
 * @brief 密钥输入函数，用户输入密钥。
 * @return 返回用户输入的密钥字符
 */
char InputKey() {
    char key;
    printf("Please enter the password : (1-254):\n");
    scanf("%c", &key);
    if((int)key > 255 || (int)key < 0) {
        printf("输入的密钥不符合要求，请检查后重新输入.\n");
        scanf("%c", &key);
    }
    return key;
}
```

```
/**
 * @brief 对文件中的内容进行加密或解密，并逆序存储。
 * @param fp 已打开的文件指针
 * @param key 密钥字符，用于异或加密/解密
 */
void decrypt(FILE * fp, char key) {
    SqStack S; // 栈的声明和初始化
    InitStack_Sq(&S);
    char ch;
    rewind(fp); // 将文件指针移到开头
    ch = fgetc(fp); // 读取文件内容并进行异或处理
    while(ch != EOF) {
        ch = ch ^ key;
        fseek(fp, -1L, SEEK_CUR); // 将文件指针向前移动一个位置
        fputc(ch, fp); // 写回处理后的字符
        fseek(fp, 0, SEEK_CUR);
        ch = fgetc(fp);
    }
    // 逆序操作
    fseek(fp, 0, SEEK_SET); // 将文件指针移到开头，实现正向压栈操作
    ch = fgetc(fp);
    while(ch != EOF) {
        Push_Sq(&S, ch);
        ch = fgetc(fp);
    }
    rewind(fp); // 将文件指针重新移到开头，进行反向出栈操作
    while(StackLength_Sq(S) != 0) {
        Pop_Sq(&S, &ch);
        fputc(ch, fp);
    }
    printf("\n 加/解密成功\n");
}
```

#### 1.4 应用方法

```
/**一个文件保留加/解密前文件，另一个文件进行加解密
 * @brief 主函数，负责文件操作和调用加密解密。
 * @param argc 参数个数
 * @param argv 参数列表
 * @return 返回执行状态
 */
int main(int argc, char* argv[]) {
    FILE *fp1;
    char ch;
    char key = '\0';
```

```
FILE *fp2;

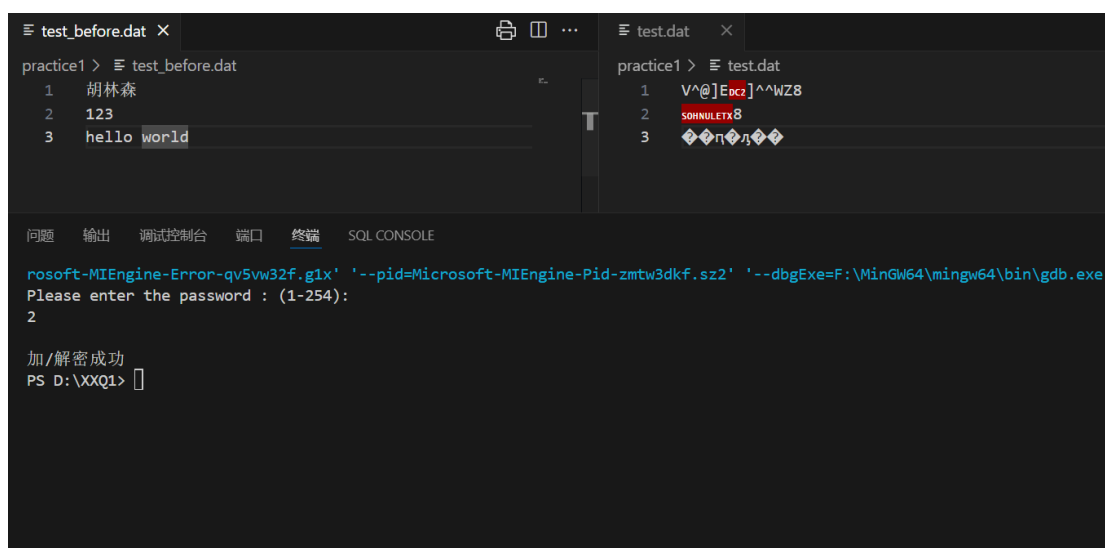
fp1 = fopen("D:\\XXQ1\\practice1\\test.dat", "rt+"); // 打开文件
if(fp1 == NULL) {
    printf("无法打开文件, 请检查后重试。\\n");
    exit(1);
}

fp2 = fopen("D:\\XXQ1\\practice1\\test_before.dat", "wt+"); // 打开
// 存储原始内容的文件
if(fp2 == NULL) {
    printf("无法创建文件, 请检查后重试。\\n");
    fclose(fp1);
    exit(1);
}

// 保存原始文件内容到另一个文件中
while ((ch = fgetc(fp1)) != EOF) {
    fputc(ch, fp2);
}

key = InputKey(); // 调用 InputKey 函数输入密钥
decrypt(fp1, key); // 调用 decrypt 函数对文件进行加密解密
// 由于加密方式为逆序+异或, 因此第一次调用函数为加密, 第二次调用函数为解密
fclose(fp1);
fclose(fp2);
return 0;
}/*
//同一文件中读写
int main(int argc, char* argv[]) {
    FILE *fp1;
    char key = '\\0';
    fp1 = fopen("D:\\XXQ1\\practice1\\test.dat", "rt+"); // 打开文件
    if(fp1 == NULL) {
        printf("无法打开文件, 请检查后重试。\\n");
        exit(1);
    }
    key = InputKey(); // 调用 InputKey 函数输入密钥
    decrypt(fp1, key); // 调用 decrypt 函数对文件进行加密解密
    // 由于加密方式为逆序+异或, 因此第一次调用函数为加密, 第二次调用函数为解密
    fclose(fp1);
    return 0;
}
*/
```

## 1.5 程序运行截图





## 实践2：树搜索

### 2.1 实现方法

使用有三个不同的搜索算法用于在树结构中查找名称：一个简单的线性搜索和广度优先搜索 (BFS)、深度优先搜索 (DFS)。

#### 1) 简单线性搜索 (`searchTree`)

- a) **工作原理**：从树的头节点开始，逐个遍历父节点及其子节点，如果找到匹配的名称就返回 1，否则返回 0。
- b) **实现方法**：首先检查父节点，如果发现匹配的名称则返回 1。然后遍历当前父节点的子节点链表，继续检查匹配。如果当前父节点、子节点及其链表中都没有找到匹配则继续检查下一个父节点。
- c) **优点**：简单直观，代码实现较为直接。
- d) **缺点**：效率较低，尤其在树的结构较为复杂或节点较多时，需要遍历更多节点。

#### 2) 广度优先搜索 (`BFS`, `bfsSearchTree`)

- a) **工作原理**：使用队列，逐层遍历树节点。先加入根节点，然后遍历其所有子节点，再将子节点的子节点加入队列，直到找到匹配的名称。
- b) **实现方法**：首先将树头节点加入队列，然后不断从队列中取出当前节点并检查其名称。对于当前节点的子节点，若存在下一个子节点则将其加入队列。
- c) **优点**：对于宽树（层次较多但每层节点较少）的情况下效率较高，因为可以逐层逐节点地查找。
- d) **缺点**：空间复杂度较高，因为需要存储每层的所有节点。

#### 3) 深度优先搜索 (`DFS`, `dfsSearchTree`)

- a) **工作原理**：使用递归方法，沿着每个节点的子节点链表进行深度优先遍历，一直到找到匹配的名称或者遍历完树。
- b) **实现方法**：先检查父节点名称，然后递归检查子节点链表，若当前节点的子节点链表没有匹配结果，则递归检查下一个父节点。
- c) **优点**：对于深树（每层子节点较少但层次很多）的情况下效率较高，因为可以沿着一条路径深入查找。
- d) **缺点**：容易造成较深的递归调用，可能会导致栈溢出。每层子节点数量多时，递归调用次数也会增加。

小结:

`searchTree` 适合简单结构的树，直接遍历但效率一般。

`bfsSearchTree` 适合宽度较大的树，依赖队列进行层次遍历，适合逐层查找。

`dfsSearchTree` 适合深度较大的树，使用递归进行深度优先遍历，可快速深入查找。

对于不同类型和规模的树结构，选择合适的搜索算法非常关键。

## 2.2 数据结构

```
//构建树:
// 定义最大长度常量
#define MAX_LEN 100

// 定义子节点结构体
typedef struct ChildNode {
    char name[MAX_LEN]; // 子节点的名称
    struct ChildNode *next; // 指向下一个子节点的指针
} ChildNode;

// 定义父节点结构体
typedef struct ParentNode {
    char name[MAX_LEN]; // 父节点的名称
    ChildNode *firstChild; // 指向第一个子节点的指针
    struct ParentNode *next; // 指向下一个父节点的指针
} ParentNode;

// 定义树结构体
typedef struct {
    ParentNode *head; // 指向树中第一个父节点的指针
} Tree;

// 创建树函数，初始化树
Tree *createTree() {
    Tree *tree = (Tree *)malloc(sizeof(Tree)); // 动态分配内存
    tree->head = NULL; // 初始化头指针为 NULL
    return tree;
}

// 创建父节点函数
ParentNode *createParentNode(char *name) {
    ParentNode *parentNode = (ParentNode *)malloc(sizeof(ParentNode));
    // 动态分配内存
    strcpy(parentNode->name, name); // 设置父节点的名称
    parentNode->firstChild = NULL; // 初始化第一个子节点指针为 NULL
```

```
    parentNode->next = NULL; // 初始化下一个父节点指针为 NULL
    return parentNode;
}

// 创建子节点函数
ChildNode *createChildNode(char *name) {
    ChildNode *childNode = (ChildNode *)malloc(sizeof(ChildNode)); // 动态分配内存
    strcpy(childNode->name, name); // 设置子节点的名称
    childNode->next = NULL; // 初始化下一个子节点指针为 NULL
    return childNode;
}

// 插入子节点到父节点函数
void insertChild(ParentNode *parentNode, char *childNames) {
    char *token = strtok(childNames, "\\"); // 使用“\”分割字符串
    while (token != NULL) {
        ChildNode *childNode = createChildNode(token); // 创建新的子节点
        if (parentNode->firstChild == NULL) { // 如果父节点第一个子节点为空
            parentNode->firstChild = childNode; // 设置为新创建的子节点
        } else {
            ChildNode *temp = parentNode->firstChild; // 遍历子节点链表
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = childNode; // 插入新子节点到链表末尾
        }
        token = strtok(NULL, "\\"); // 继续分割下一个子节点名称
    }
}

// 插入父节点到树函数
void insertParent(Tree *tree, char *parentName, char *childNames) {
    ParentNode *parentNode = createParentNode(parentName); // 创建新的父节点
    insertChild(parentNode, childNames); // 插入子节点到新的父节点

    if (tree->head == NULL) { // 如果树头指针为空
        tree->head = parentNode; // 设置为新创建的父节点
    } else {
        ParentNode *temp = tree->head; // 遍历父节点链表
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = parentNode;
    }
}
```

```
    }  
    temp->next = parentNode; // 插入新父节点到链表末尾  
}  
}
```

### 2.3 核心算法

```
int searchTree1(Tree *tree, char *searchName) {  
    ParentNode *temp = tree->head; // 从树头指针开始  
    while (temp != NULL) {  
        if (strcmp(temp->name, searchName) == 0) { // 如果匹配父节点名称  
            return 1;  
        }  
        ChildNode *childTemp = temp->firstChild; // 查找子节点链表  
        while (childTemp != NULL) {  
            if (strcmp(childTemp->name, searchName) == 0) { // 如果匹配子  
节点名称  
                return 1;  
            }  
            childTemp = childTemp->next;  
        }  
        temp = temp->next;  
    }  
    return 0; // 如果没找到, 返回 0  
}  
  
int bfsSearchTree(Tree *tree, char *searchName) {  
    if (tree->head == NULL) {  
        return 0;  
    }  
    ParentNode *queue[1000]; // 定义队列  
    int front = 0, rear = 0; // 定义队头和队尾指针  
  
    queue[rear++] = tree->head; // 入队树头指针  
  
    while (front != rear) {  
        ParentNode *temp = queue[front++]; // 出队父节点指针  
        if (strcmp(temp->name, searchName) == 0) { // 如果匹配父节点名称  
            return 1;  
        }  
        ChildNode *childTemp = temp->firstChild; // 查找子节点链表  
        while (childTemp != NULL) {  
            if (strcmp(childTemp->name, searchName) == 0) { // 如果匹配子  
节点名称
```

```

        return 1;
    }
    if (childTemp->next != NULL) { // 如果有下一个子节点
        queue[rear++] = childTemp->next; // 入队下一个子节点
    }
    childTemp = childTemp->next;
}
}
return 0; // 如果没找到, 返回 0
}

// 深度优先搜索树函数
int dfsSearchTree(ParentNode *parentNode, char *searchName) {
    // 检查当前父节点
    if (strcmp(parentNode->name, searchName) == 0) {
        return 1;
    }
    // 检查子节点链表
    ChildNode *childTemp = parentNode->firstChild;
    while (childTemp != NULL) {
        if (strcmp(childTemp->name, searchName) == 0) {
            return 1;
        }
        childTemp = childTemp->next;
    }
    // 递归检查下一个父节点
    if (parentNode->next != NULL) {
        return dfsSearchTree(parentNode->next, searchName);
    }

    return 0;
}

```

## 2.4 应用方法

```

int searchTree2(Tree *tree, char *searchName) {
    if (tree->head == NULL) {
        return 0;
    }
    return dfsSearchTree(tree->head, searchName);
}

int searchTree3(Tree *tree, char *searchName) {

```

```
    if (tree->head == NULL) {  
        return 0;  
    }  
    return bfsSearchTree(tree, searchName);  
}  
//search1 直接调用即可
```

## 2.5 程序运行截图

```
C SearchTree.c 1 X  
practice2 > C SearchTree.c > searchTree1(Tree *, char *)  
75 void insertParent(Tree *tree, char *parentName, char *childNames) {  
76     if (tree->head == NULL) { // 如果树头指针为空  
80         tree->head = parentNode; // 设置为新创建的父节点  
81     } else {  
82         ParentNode *temp = tree->head; // 遍历父节点链表  
83         while (temp->next != NULL) {  
84             temp = temp->next;  
85         }  
86         temp->next = parentNode; // 插入新父节点到链表末尾  
87     }  
88 }  
89  
90 // 简单线性查找树中的名称函数  
91 int searchTree1(Tree *tree, char *searchName) {  
92     ParentNode *temp = tree->head; // 从树头指针开始  
93     while (temp != NULL) {  
94         if (strcmp(temp->name, searchName) == 0) { // 如果匹配父节点名称  
95             return 1;  
96         }  
97         ChildNode *childTemp = temp->firstChild; // 查找子节点链表  
98         while (childTemp != NULL) {  
99             if (strcmp(childTemp->name, searchName) == 0) { // 如果匹配子节点名称  
100                 return 1;  
101             }  
102             childTemp = childTemp->next;  
103         }  
104     }  
105     return 0;  
106 }  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2198  
2199  
2200
```

## 实践3： 图搜索

### 3.1 实现方法

在计算机科学中，图是表示对象及其相互关系的一种数学结构。图搜索算法是一系列用于遍历或搜索图中节点的算法，它们对于路径查找、最短路径问题、拓扑排序等问题至关重要。本文旨在详细介绍几种常用的图搜索算法，包括深度优先搜索（DFS）、广度优先搜索（BFS）、迪杰斯特拉（Dijkstra）算法和 A\* 搜索算法。

#### 3.1.1 深度优先搜索（DFS）

深度优先搜索是一种用于遍历或搜索树或图结构的算法。这个算法会尽可能深地搜索图的分支。当节点  $v$  的所在边都已被探寻过，搜索将回溯到发现节点  $v$  的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。DFS 主要用在需要遍历所有节点并且需要尽可能深地搜索图中节点的场景，例如解决迷宫问题、拓扑排序以及找出图中所有的连通分量。

#### 3.1.2 广度优先搜索（BFS）

与 DFS 不同，广度优先搜索是一种从根节点开始，沿着图的宽度遍历节点的算法。它对每一个节点先访问它的所有邻接节点，然后再逐层向外扩展。BFS 通常使用队列来实现，并且适用于需要在不需要考虑路径成本的情况下找到两个节点之间的最短路径的问题。

迪杰斯特拉（Dijkstra）算法迪杰斯特拉算法是一种用于在加权图中找到某个节点到其他所有节点最短路径的算法。它采用的是贪心策略，声明一个数组  $dis$  来保存源点到  $i$  点的最短距离和一个  $path$  来保存已经找到了最短路径的顶点，其余顶点为还没有找到最短路径的顶点。它逐步更新这些数组，直到找到所有从源出发的最短路径。

#### 3.1.3 A\* 搜索算法

A\* 搜索算法是一种启发式搜索算法，通过维护一个代价函数评估每个节点的代价，该代价由两部分组成：一部分是起点到当前节点的实际代价，另一部分是从当前节点到目标点的估计代价。启发函数的设计对 A 算法的性能有很大影响。A\* 算法广泛应用于路径规划和游戏 AI 中的角色移动计算。

A\* 算法是一种在图中寻找从初始节点到目标节点最短路径的启发式搜索算法。它结合了 Dijkstra 算法的确保性（保证找到一条最短路径）和贪心算法的高效性（快速找到目标）。A\* 算法通过评估函数  $f(n) = g(n) + h(n)$  来工作，其中  $g(n)$  是从起始点到任何顶点  $n$  的实际成

本，而  $h(n)$  是从顶点  $n$  到目标的估计最低成本，通常用启发式函数来计算，这个函数需要事先设计来反映实际的地形或环境特征。理想情况下， $h(n)$  应该不会高估实际的成本，这种情况下，A\*算法保证找到一条最低成本路径。算法的性能和准确性高度依赖于启发式函数的选择。在实际应用中，A\*算法广泛应用于各类路径规划问题，如机器人导航、地图定位服务和游戏中的 AI 路径寻找等场景。通过适当选择和调整启发式函数，A\*算法能够在复杂的环境中有效地寻找最短路径，同时保持计算上的可行性和效率。

#### 3.1.4 小结

DFS 和 BFS 是最基本的图遍历方法，其中 DFS 适合目标明确但求解空间大的情况，而 BFS 适合找到最短路径或近距离的目标。Dijkstra 算法适用于不带负权边的图，且能找到单源最短路径，而 A\*算法则在 Dijkstra 的基础上增加了启发信息，使得搜索更加高效。在选择合适的图搜索算法时，需要根据具体问题的需要以及图的性质来决定。

### 3.2 数据结构

```
// 图的结构定义
typedef struct Graph {
    int n; // 顶点数量
    char** nodes; // 存储顶点字符串数组
    int** adjMatrix; // 邻接矩阵
} Graph;

❏ createGraph(int)
❏ getNodeIndex(Graph *, const char *)
❏ addEdge(Graph *, const char *, const char *)
❏ dfs(Graph *, int, int, int *)
❏ isPathExist(Graph *, const char *, const char *)
❏ freeGraph(Graph *)
```

### 3.3 核心算法

#### 3.3.1 DFS

```
// 深度优先搜索路径存在性检查
int dfs(Graph* graph, int start, int end, int* visited) {
    if (start == end) return 1;
    visited[start] = 1;

    for (int i = 0; i < graph->n; ++i) {
        if (graph->adjMatrix[start][i] && !visited[i]) {
            if (dfs(graph, i, end, visited)) {
```



```
        return 1;
    }
}
return 0;
}
```

### 3.3.2 BFS

如果使用 BFS 算法，需要使用一个队列来替换递归调用。

```
int bfs(Graph* graph, int start, int end, int* visited) {
    int *queue = (int*)malloc(graph->n * sizeof(int));
    int front = 0, rear = 0;

    queue[rear++] = start;
    visited[start] = 1;

    while (front < rear) {
        int current = queue[front++];

        if (current == end) {
            free(queue);
            return 1;
        }

        for (int i = 0; i < graph->n; ++i) {
            if (graph->adjMatrix[current][i] && !visited[i]) {
                queue[rear++] = i;
                visited[i] = 1;
            }
        }
    }

    free(queue);
    return 0;
}
```

}

### 3.3.3 A\*算法

A\*算法结合了广度优先搜索（BFS）的完整性和最佳优先搜索（Best-First Search）的效率。它通过评估每个节点的总代价函数  $f(n)$  来确定搜索路径，其中  $f(n) = g(n) + h(n)$ 。

$g(n)$ ：从起点到当前节点  $n$  的实际代价。

$h(n)$ ：从当前节点  $n$  到目标节点的估计代价（启发式函数）。

```
void astar_search(int initial[N][N]) {
    Node* open_list[MAX_LIST_SIZE];
    int open_count = 0;

    Node* closed_list[MAX_LIST_SIZE];
    int closed_count = 0;

    Node* initial_node = create_node(initial, 0, manhattan(initial),
    NULL); // 生成初始节点
    if (initial_node == NULL) {
        printf("Failed to create initial node.\n");
        return;
    }
    open_list[open_count++] = initial_node; // 初始节点加入开启列表

    while (open_count > 0) {
        qsort(open_list, open_count, sizeof(Node*), compare_nodes); //
        根据 f 值对开启列表排序

        Node* current = open_list[0]; // 取出 f 值最小的节点
        if (is_same_state(current->state, target)) { // 检查是否达到目标
        状态
            print_path(current); // 打印路径
            count = current->g; // 记录从初始状态到目标状态所经过的步数
            free_path(current); // 释放路径
            return;
        }

        // 从开启列表中移除当前节点
        for (int i = 0; i < open_count - 1; ++i) {
            open_list[i] = open_list[i + 1];
        }
        open_count--;

        closed_list[closed_count++] = current; // 将当前节点加入关闭列表
    }
}
```

```

Node* neighbors[4];
int neighbor_count;
get_neighbors(current, neighbors, &neighbor_count); // 生成当前
节点的所有邻居节点

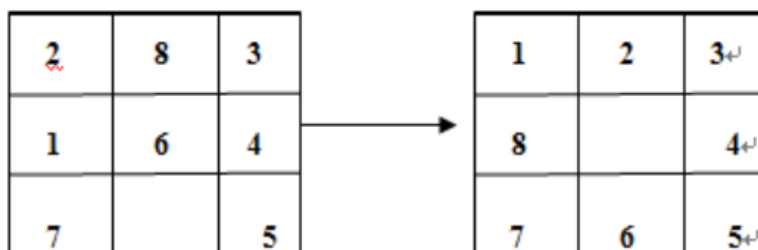
// 遍历所有邻居节点
for (int i = 0; i < neighbor_count; ++i) {
    Node* neighbor = neighbors[i];
    int is_in_closed_list = 0;
    for (int j = 0; j < closed_count; ++j) {
        if (is_same_state(neighbor->state,
closed_list[j]->state)) {
            is_in_closed_list = 1; // 如果邻居节点已经在关闭列表中
            break;
        }
    }
    if (!is_in_closed_list) {
        open_list[open_count++] = neighbor; // 将合法的邻居节点加
入开启列表
    } else {
        free(neighbor); // 若邻居节点在关闭列表中则释放节点
    }
}

printf("No solution found\n"); // 没有找到解决方案
}

```

### 3.4 应用方法

以八数码九宫格问题为例，它要求在一个 3x3 的网格中，通过上下左右移动八个数字和一格空位，从初始状态转变为目标状态。在移动过程中，每次只能将一个数字滑动到空格位置，直到整个九宫格的数字排列与目标排列一致。



图表 1 左方为初始状态，右方为目标状态

为了节约步骤，我们使用启发式搜索——A\*算法，但会使用较多的时间、空间，换取最优路径。

```
// 节点结构体(构建优先队列以便后续算法使用)
typedef struct Node {
    int state[N][N]; // 当前状态
    int g; // 实际代价
    int h; // 预估代价（曼哈顿距离）
    struct Node* parent; // 父节点指针
} Node;

// 曼哈顿距离计算
int manhattan(int state[N][N]) {
    int distance = 0;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] != 0) { // 忽略空格位置
                int target_x = (state[i][j] - 1) / N;
                int target_y = (state[i][j] - 1) % N;
                distance += abs(target_x - i) + abs(target_y - j);
            }
        }
    }
    return distance;
}

// 生成一个新节点
Node* create_node(int state[N][N], int g, int h, Node* parent) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node != NULL) { // 检查内存分配是否成功
        memcpy(new_node->state, state, N * N * sizeof(int)); // 复制状态
        new_node->g = g; // 设置实际代价
        new_node->h = h; // 设置预估代价
        new_node->parent = parent; // 设置父节点
    }
    return new_node;
}

// 节点比较函数（用于优先队列）
int compare_nodes(const void* a, const void* b) {
    Node* nodeA = *(Node**)a;
    Node* nodeB = *(Node**)b;
    return (nodeA->g + nodeA->h) - (nodeB->g + nodeB->h); // 按 f = g + h 排序
}
```

```

}

// 获取空格位置
void get_blank_pos(int state[N][N], int* blank_x, int* blank_y) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (state[i][j] == 0) {
                *blank_x = i;
                *blank_y = j; // 找到空格位置
                return;
            }
        }
    }
}

// 生成相邻节点
void get_neighbors(Node* current, Node** neighbors, int* count) {
    int blank_x, blank_y;
    get_blank_pos(current->state, &blank_x, &blank_y); // 获取空格的位置
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // 上下左右四个方向
    *count = 0;

    // 遍历四个方向，生成合法的相邻节点
    for (int i = 0; i < 4; ++i) {
        int new_x = blank_x + directions[i][0];
        int new_y = blank_y + directions[i][1];
        if (new_x >= 0 && new_x < N && new_y >= 0 && new_y < N) { // 检查新坐标是否在边界内
            int new_state[N][N];
            memcpy(new_state, current->state, N * N * sizeof(int)); // 复制当前状态
            new_state[blank_x][blank_y] = new_state[new_x][new_y]; // 交换空格与目标位置
            new_state[new_x][new_y] = 0;
            Node* neighbor = create_node(new_state, current->g + 1, manhattan(new_state), current); // 生成新节点
            if (neighbor != NULL) { // 检查节点生成是否成功
                neighbors[*count] = neighbor;
                (*count)++;
            }
        }
    }
}

```

```
int main() {
    int initial[N][N] = {
        {2, 8, 3},
        {1, 6, 4},
        {7, 0, 5}
    };

    astar_search(initial); // 调用 A*算法进行搜索
    printf("Total steps from initial to target: %d\n", count -1); // 打印步数
    return 0;
}
```

小结：

（1）在八数码问题中，启发式搜索算法表现出显著的性能优势。通过利用问题特有信息，启发式算法指导搜索过程，有效降低搜索空间和时间复杂度，加速寻找最优解或近似解的进程。

（2）启发式函数作为算法的核心，其设计要求精确评估当前状态与目标状态的差距。在复杂问题中，设计高效启发式函数是一项挑战，不当的设计可能导致算法效率低下或求解失败。

（3）启发式函数的指导能力有限，在某些场景下可能无法提供足够信息，导致算法收敛于局部最优而非全局最优。

（4）我们的算法采用可回溯的单向搜索策略，利用堆栈存储中间节点，评估所有潜在路径，实现接近最优解的搜索路径。该方法适用于目标明确的最优路径寻找问题。

（5）九宫格问题解决涉及算法设计、数据结构选择、评估函数设计等多个方面。通过程序优化，可提升问题解决效率与准确度，增强算法的可扩展性。未来研究可致力于改进九宫格问题解决方法，以适应更广泛的应用场景和复杂度。

## 3.5 程序运行截图

```
PS E:\CUPK-lib\XXQ1> & 'c:\Users\15638\.vscode\extensions\ms-vscode  
rosoft-MIEngine-In-n2skleib.wvr' '--stdout=Microsoft-MIEngine-Out-m  
ine-Pid-4mpr41gq.1yj' '--dbgExe=F:\MinGW64\mingw64\bin\gdb.exe' '--  
2 8 3  
1 6 4  
7 0 5  
  
2 8 3  
1 0 4  
7 6 5  
  
2 0 3  
1 8 4  
7 6 5  
  
0 2 3  
1 8 4  
7 6 5  
  
1 2 3  
0 8 4  
7 6 5  
  
1 2 3  
8 0 4  
7 6 5  
  
Total steps from initial to target: 4
```

## 实践4：图搜索问题的应用拓展

图搜索算法是通过图中的边对图中所有顶点进行系统访问的过程。

迷宫、九宫格

### 4.1 迷宫问题介绍

迷宫问题是指在一个二维的网格中，找到从起点到终点的一条路径。网格中的某些格子是可通行的，而某些则是不可通行的障碍物。迷宫的目标是在满足约束的情况下找到一条从起点到终点的路径。

#### 4.1.1 迷宫问题的解法——DFS

1. 定义迷宫和相关参数：

- a) 使用一个二维数组 `maze` 表示迷宫，其中 0 表示可以通行的路径，1 表示不可通行的障碍。
- b) 定义一个与迷宫同等大小的二维数组 `visited` 用来标记已访问的节点。
- c) 定义起点 (`startX`, `startY`) 和终点 (`endX`, `endY`) 的坐标。
- d) 定义四个移动方向：上 (`{-1, 0}`)，下 (`{1, 0}`)，左 (`{0, -1}`)，右 (`{0, 1}`)。

2. 输入与初始化：

- a) 用户输入迷宫数据，即每个格子是 0 或 1。
- b) 用户输入起点和终点的坐标。
- c) 初始化访问数组 `visited`，将所有节点标记为未访问状态。

3. DFS 算法：(3.1.1 中已做介绍，此处不做过多阐述)

- a) 从起点出发，标记起点为已访问。
- b) 尝试移动到四个方向的邻接节点：
- c) 检查新位置是否越界、是否是路径且未被访问（通过 `isSafe` 函数）。
- d) 如果可以移动，递归调用 DFS 函数处理新位置。
- e) 如果到达终点，则输出路径并结束递归。
- f) 如果无法到达终点，则回溯，继续尝试其他方向。

4. 输出结果：

- a) 如果 DFS 找到路径，则依次打印路径上的坐标。
- b) 如果 DFS 未找到路径，则输出提示“没有路径可以到达目的地”。



### 4.1.2 迷宫问题的解法——最优路径 DFS

普通 DFS 会沿着树的深度进行搜索，直到找到叶子节点或没有路径为止，然后再回溯到上一个节点，继续搜索其他未访问的路径。因此，DFS 的特点是沿着路径尽可能深入搜索。

而最优路径 DFS 是在普通 DFS 的基础上，改进其能找到最短路径或满足特定条件的路径。例如，在求解迷宫问题时，不仅仅是找到一条路径，更希望找到最短路径。这种情况下，可以记录每次找到的路径长度，若找到更短的路径则更新最优路径。

优化普通 DFS 以达到查找最优路径只需要调整 3 个关键步骤：

1. 使用一个变量来记录最佳路径长度和路径。
2. 每当找到终点时，比较当前路径的长度与记录的最优路径长度。
3. 如果当前路径更短，则更新最优路径。

### 4.1.3 迷宫问题的解法——A\*算法

在迷宫问题中，A\*算法可以如下实现：

1. 使用优先级队列（通常是最小堆）保存节点，按照  $f(n)=g(n)+h(n)$  的值进行排序。
2. 从起点开始，将其加入优先级队列中，并初始化  $g(n)$  值。其中  $g(n)$ ：从起点到节点  $n$  的实际路径长度； $h(n)$ ：节点  $n$  到目标节点的估计路径长度
3. 从优先级队列中取出  $f(n)$  最小的节点进行扩展，检查其是否为终点。
4. 对每个邻居节点，计算其新的  $g(n)$  和  $f(n)$  值，并更新优先级队列。
5. 重复上述过程直到找到终点或者优先级队列为空。

## 4.2 再谈九宫格

九宫格问题也称八数码问题，是人工智能中状态搜索中的经典问题

## 实践5：组合优化问题的拓展

组合优化问题有哪些？是什么？局部最优、全局最优、遗传算法……

### 5.1 TSP 问题——以管道建设规划为例

对于管道建设，一般会考虑站点间距离、建设代价（建设难度、费用）。在计算机科学中，旅行商问题（Traveling Salesman Problem，简称 TSP）是一个经典的组合优化问题。给定一个有  $n$  个城市的集合和每对城市之间的距离，目标是找到一条旅行路线，使得一个旅行商能够访问每个城市恰好一次并返回出发城市，使得所走的总距离最短。

### 5.2 背包问题

背包问题是一类经典的组合优化问题，通常表现为如何在有限的容量或资源限制下，优化某个目标值（如价值的最大化）。背包问题有很多变种，背包问题大概有九种典型的背包问题：

- 1) 01 背包问题：每个物品只能选择一次，且重量和价值各不相同。
- 2) 完全背包问题：每个物品可以选择无限次。
- 3) 多重背包问题：每个物品有固定的数量限制。
- 4) 分组背包问题：物品被分成若干组，每组中只能选择一个物品。
- 5) 二维费用背包问题：每个物品有两个费用限制，如重量和体积。
- 6) 多维费用背包问题：每个物品有多个费用限制。
- 7) 依赖背包问题：物品之间有依赖关系，如选择了物品 A 就必须选择物品 B。
- 8) 混合背包问题：包含多种背包问题的元素。
- 9) 其他复杂的背包变种：如带有不确定因素的背包问题。

#### 5.2.1 背包问题的解法——动态规划法

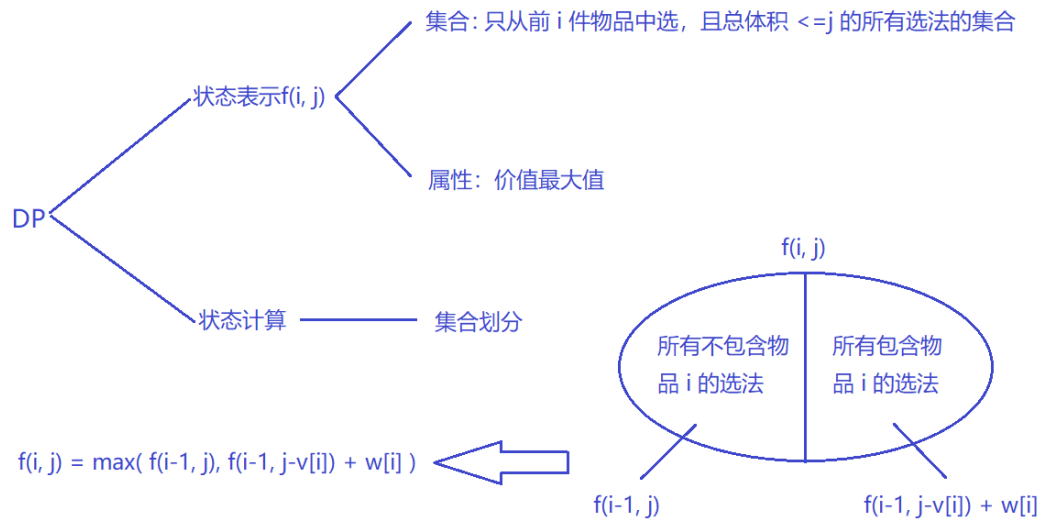
01 背包是其他背包问题的基础，其大致问题如下：

有  $N$  件物品和一个容量为  $V$  的背包，每件物品有各自的价值且只能被选择一次，要求在有限的背包容量下，装入的物品总价值最大。

01 背包是较为简单的动态规划问题，也是其余背包问题的基础。

动态规划是不断决策求最优解的过程，「0-1 背包」即是不断对第  $i$  个物品的做出决策，

01 正好代表不选与选两种决定。



图表 2 背包问题流程图

动态规划的核心就是状态转移方程, 01 背包的状态转移方程为

```
for (int i = 1; i <= n; i++)
{
    for (int j = V; j >= 0; j--)
    {
        if (j >= w[i]) // 如果背包装得下当前的物体
        {
            f[i][j] = max(f[i - 1][j], f[i - 1][j - w[i]] + v[i]); // 状态转移方程
        }
        else // 如果背包装不下当前物体
        {
            f[i][j] = f[i - 1][j];
        }
    }
}
```

$i$  代表对  $i$  件物体做决策, 有两种方式—放入背包和不放入背包。 $j$  表示当前背包剩余的容量。

### 5.2.2 背包问题的解法——递归回溯法 (DFS)

回溯法在确定了解空间的结构后, 从根结点出发, 以 DFS 的方式搜索整个解空间, 此时根结点成为一个活结点, 并且成为当前的扩展结点。每次都从扩展结点向纵向搜索新的结点, 当算法搜索到了解空间的任一结点, 先判断该结点是否肯定不包含问题的解 (是否还能或者还有必要继续往下搜索), 如果确定不包含问题的解, 就逐层回溯; 否则, 进入子树, 继续按照深度优先的策略进行搜索。当回溯到根结点时, 说明搜索结束了, 此时已经得到了一系

列的解，根据需要选择其中的一个或者多个解即可。

回溯法解决 01 背包步骤：

1. 初始化：
  - a) 创建一个变量 `maxValue`，用于跟踪最大价值，初始为 0。
2. 递归函数设计：
  - a) 定义递归函数 `dfs(index, currentWeight, currentValue)`。
  - b) `index` 表示当前考虑的物品索引。
  - c) `currentWeight` 表示当前选择物品后的总重量。
  - d) `currentValue` 表示当前选择物品后的总价值。
  - e) 更新 `maxValue` 为 `currentValue` 与当前最大 `maxValue` 中的较大者。
  - f) 递归结束条件：所有物品都考虑结束或者当前重量超过背包容量。
3. 选择与不选择：
  - a) 在每一步，可以选择当前物品或者不选择：
  - b) 选择当前物品：递归调用时更新 `currentWeight` 和 `currentValue`。
  - c) 不选择当前物品：递归调用时 `currentWeight` 和 `currentValue` 保持不变。
4. 剪枝：
  - a) 如果选择当前物品导致重量超过 `W`，则剪枝（不继续递归）。
  - b) 利用当前最大 `maxValue` 剪枝：如果当前价值达不到当前最大值，不继续递归。

通过回溯法，可以探索所有可能的选择，即穷举所有组合，从而找到能够使背包中物品总价值最大的一组物品。虽然回溯法比较直接，但在大规模问题上可能效率较低，实际应用中常使用动态规划方法。