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';
```

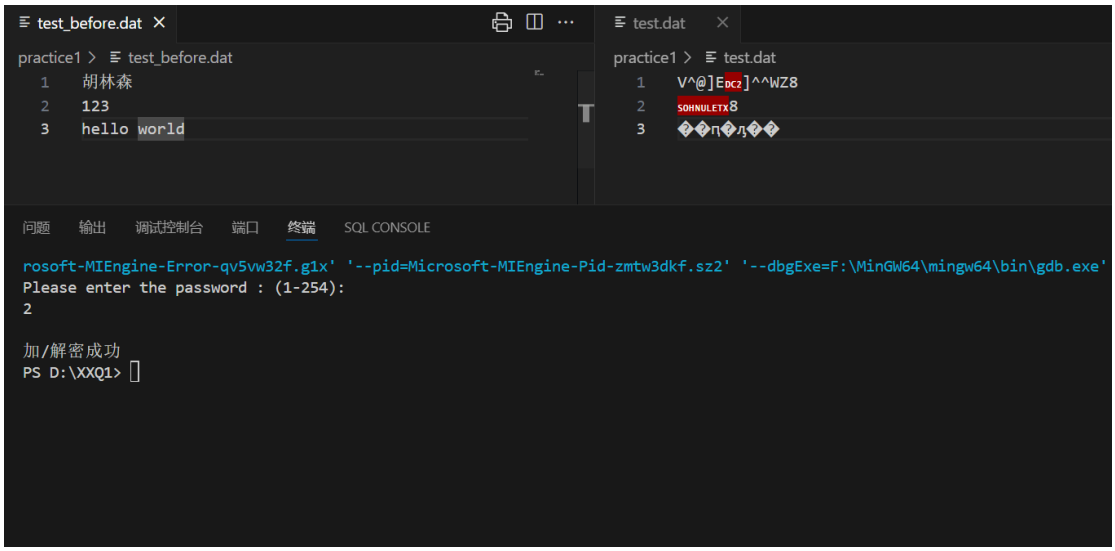
```
    FILE *fp2;

    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


test_before.dat
practice1 > test_before.dat
1 胡林森
2 123
3 hello world
test.dat
practice1 > test.dat
1 V^@]E^C2]^^WZ8
2 SOHNULETX8
3 ??п?д??
问题 输出 调试控制台 端口 终端 SQL CONSOLE
rosoft-MIEngine-Error-qv5vw32f.g1x' '--pid=Microsoft-MIEngine-Pid-zmtw3dkf.sz2' '--dbgExe=F:\MinGW64\mingw64\bin\gdb.exe'
Please enter the password : (1-254):
2
加/解密成功
PS D:\XXQ1>

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

```c
//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; // 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;
    }
```

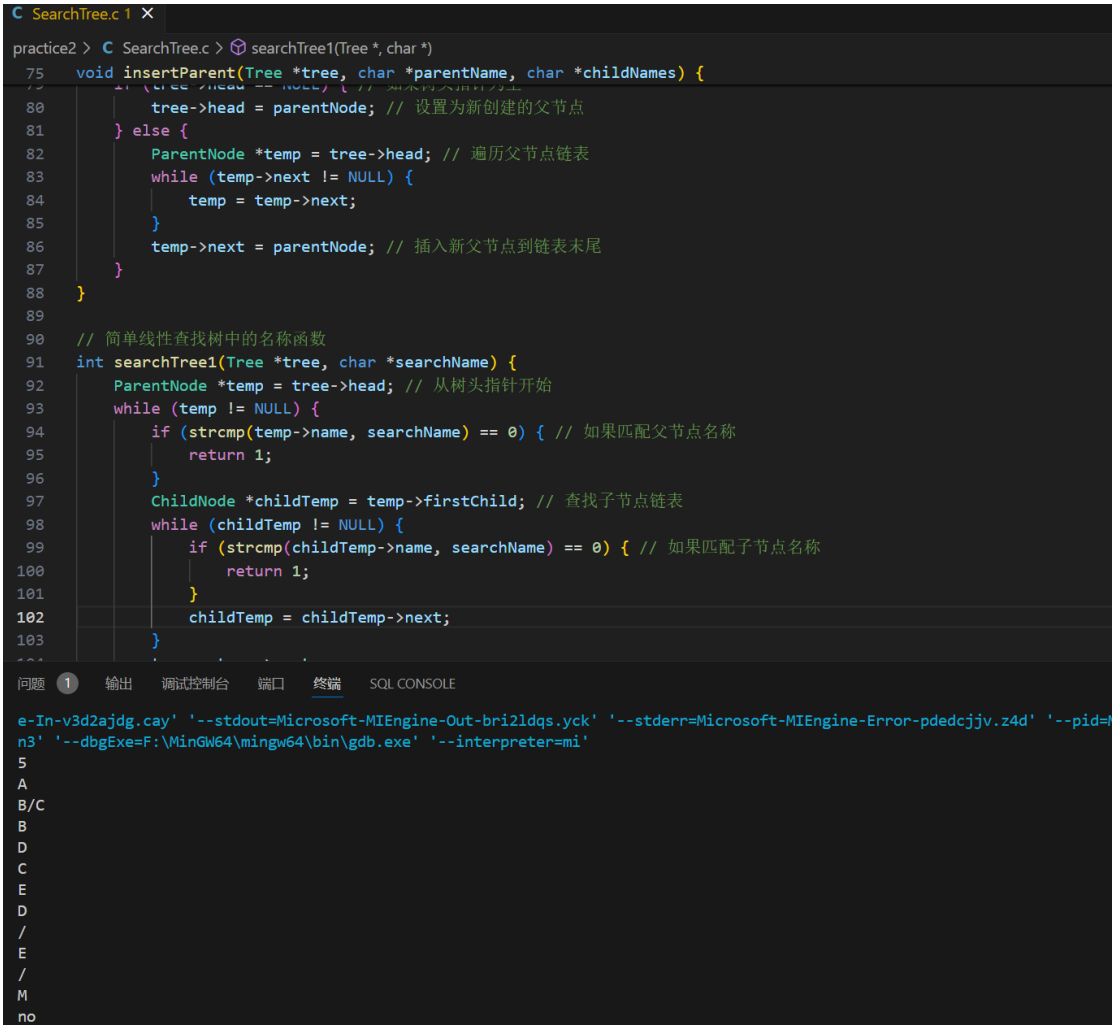
2.4 应用方法

```
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



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

```c
// 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

```c
// 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
```

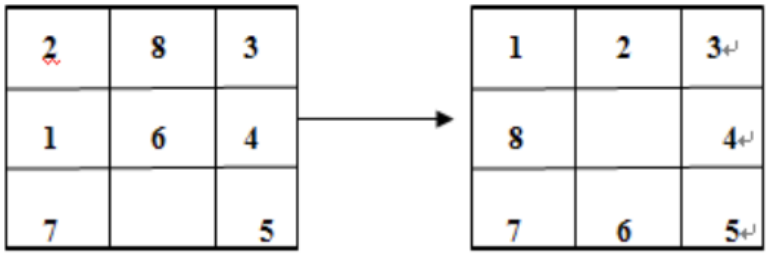
```
        Node* neighbors[4];
        int neighbor_count;
        get_neighbors(current, neighbors, &neighbor_count);  // 生成当前
节点的所有邻居节点

        // 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
则跳过
                    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-vscod
rosoft-MIEngine-In-n2skleib.wwr' '--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。

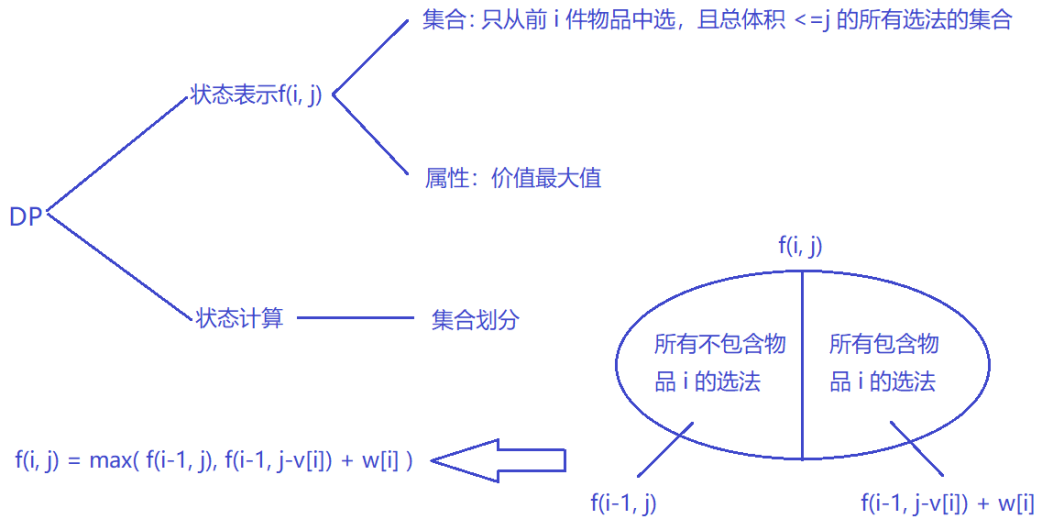
5.2.1 背包问题的解法——动态规划法

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。