《实验二——农夫过河问题的求解》

（一）问题描述

一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸。他要把这些东西全部运到北岸。他面前只有一条小船，船只能容下他和一件物品，另外只有农夫才能撑船。如果农夫在场，则狼不能吃羊，羊不能吃白菜，否则狼会吃羊，羊会吃白菜，所以农夫不能留下羊和白菜自己离开，也不能留下狼和羊自己离开，而狼不吃白菜。请求出农夫将所有的东西运过河的方案。

（二）解决思路

求解这个问题的简单方法是一步一步进行试探，每一步搜索所有可能的选择，对前一步合适的选择后再考虑下一步的各种方案。

要模拟农夫过河问题，首先需要对问题中的每个角色的位置进行描述。可用 4 位二进制数顺序分别表示农夫、狼、白菜和羊的位置。用 0 表示在南岸，1 表示在北岸。例如，整数 5 (0101)表示农夫和白菜在南岸，而狼和羊在北岸。

现在问题变成：从初始的状态二进制 0000(全部在河的南岸)出发，寻找一种全部由安全状态构成的状态序列，它以二进制 1111(全部到达河的北岸)为最终目标。总状态共 16 种(0000 到 1111 或者看成 16 个顶点的无向图)，可采用广度优先或深度优先的搜索策略——得到从 0000 到 1111 的安全路径。

选择以深度优先算法进行搜索，每访问一个顶点就将其入栈，然后访问栈顶元素的邻接顶点，若没有邻接顶点则将其出栈，不断重复这个过程直至找到目标顶点。

最终的过河方案应用汉字显示出每一步的两岸状态。

因此，问题的求解共分为三个模块：无向图的创建、寻找所有的安全路径、输出所有的安全路径。

（三）数据结构

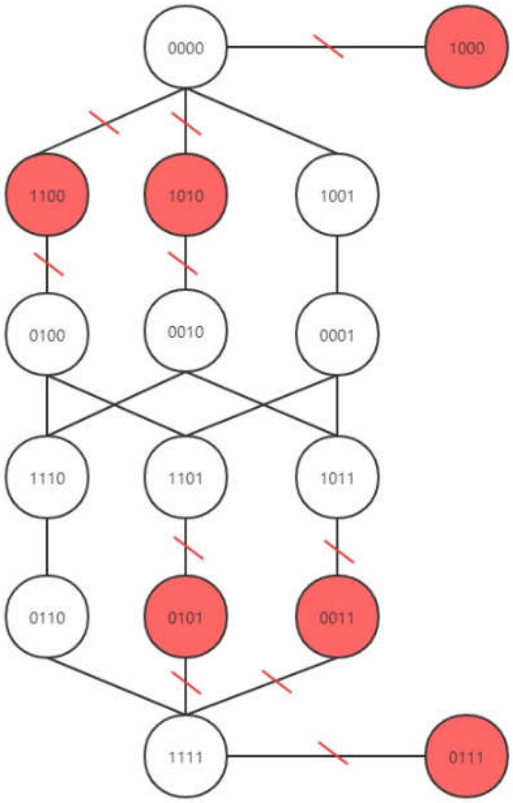
1、图：将四位二进制数字 0000 到 1111（即十进制中的 0 到 15）16 种状态作为顶点，若能从某一状态一步进入另一状态，则认为两顶点邻接。由于顶点个数固定，故采用邻接矩阵的方法存储图。

2、栈：在进行深度优先搜索路径时，需要使用到栈。由于栈中存储的元素较少，不容易产生溢出现象，故采用顺序表（即数组）存储栈。

（四）算法分析

① 无向图的创建：

（a）初始化邻接矩阵 matrix[16][16]，将对角线元素初始化为 0，其余元素初始化为 1；
（b）依次访问所有顶点，判断其是否处于安全状态。若不是，将标志位设置为 1，并将邻接矩阵中的相应元素设置为 0，使得该顶点无邻接顶点；
（c）依次访问所有标志位为 0 的顶点，判断任意两个顶点间是否直接相连。若没有，将邻接矩阵中的相应元素设置为 0；
（d）将标志位全部重置为 0。
建立好的无向图（红色顶点表示不安全状态）如下所示：



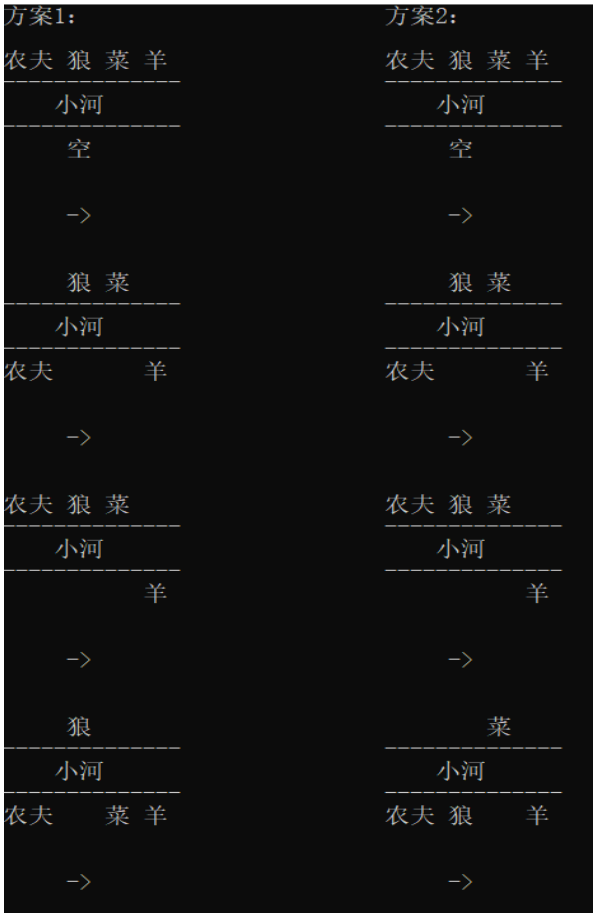
② 寻找所有的安全路径：

（a）访问某一顶点，入栈，并将标志位设置为 1；
（b）检查该顶点是否是目标顶点，若是，则输出路径；
（c）访问栈顶元素的未访问过（即标志位为 0）的邻接顶点（若存在）；
（d）若栈顶元素不存在满足（c）中条件的邻接顶点，则出栈，并将标志位重置为 0；
（e）重复执行上述操作。

③ 输出所有的安全路径：

（a）创建指针数组，用于指向 16 种状态所对应的情形（如 0100 对应于“农夫　菜 羊”）；
（b）从栈底元素开始直至栈顶，先输出二进制对应的字符串，然后输出“河流”，再将二进制按位取反，得到河对岸情形所对应的二进制表示，并将其对应字符串输出。

（五）运行结果


方案1：
农夫 狼 菜 羊
小河
空
->
狼 菜
小河
农夫 羊
->
农夫 狼 菜
小河
羊
->
狼
小河
农夫 菜 羊
->
方案2：
农夫 狼 菜 羊
小河
空
->
狼 菜
小河
农夫 羊
->
农夫 狼 菜
小河
羊
->
菜
小河
农夫 狼 羊
->
农夫 狼 羊
小河
菜
->
羊
小河
农夫 狼 菜
->
农夫 羊
小河
狼 菜
->
空
小河
农夫 狼 菜 羊
农夫 菜 羊
小河
狼
->
羊
小河
农夫 狼 菜
->
农夫 羊
小河
狼 菜
->
空
小河
农夫 狼 菜 羊
Process exited after 0.3482 seconds with return value 0
请按任意键继续. . .

（六）反思总结

程序中建立的邻接矩阵如下：

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

该矩阵的稀疏因子 $\Gamma = \frac{20}{16\times16} = 0.078125$，故该邻接矩阵近似于一个稀疏矩阵，矩阵中含有大量零元素，造成极大的空间浪费。此外，使用邻接矩阵存储图时，遍历全部顶点所需的时间为$O(n^2)$，效率较低。

因此，可以采用邻接表替代邻接矩阵来存储图，从而对程序进行改进和优化。

通过本次实验，我进一步掌握了图和栈这两种数据结构，独立实现了图的深度优先搜索算法以及出栈与入栈的操作。此外，我还了解了 C 语言的位操作，并学会了如何对一个实际问题进行数学建模以及将其转化为程序语言去解决问题的方法。

（七）源程序

```
//农夫过河问题的求解
//0表示在南岸，1表示在北岸；从左到右依次表示农夫、狼、菜、羊
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define SIZE 16

int visited[SIZE];              //标志位（用于标志结点是否被访问过）
int stack[SIZE];                //栈
int top = 0;                    //栈顶指针
int number = 0;                 //方案数目
int interface_bottom = 0;       //使程序退出时返回信息在界面底端

char* situation[SIZE] = { "农夫 狼 菜 羊", "农夫 狼 菜", "农夫 狼    羊", "农夫 狼",
                          "农夫    菜 羊", "农夫    菜", "农夫       羊", "农夫",
                          "    狼 菜 羊", "    狼 菜", "    狼    羊", "    狼",
                          "       菜 羊", "       菜", "          羊", "    空"};

//函数原型
void CreateGraph(int matrix[][SIZE]);

int farmer_location(int state);
int wolf_location(int state);
int cabbage_location(int state);
int sheep_location(int state);
int unsafe(int state);

int case1(int i, int j);
int case2(int i, int j);
int case3(int i, int j);
int unavailable(int i, int j);

void DFS(int matrix[][SIZE], int start, int end);

int StackIsEmpty(void);
int StackIsFull(void);
int push(int data);
int pop(void);

void PrintPath(void);

void SetCCPos(int x, int y);
```

```
int main(void) {
    int matrix[SIZE][SIZE];            //邻接矩阵存储图
    CreateGraph(matrix);               //创建图
    DFS(matrix, 0, 15);                //深度优先搜索

    return 0;
}

//创建图
void CreateGraph(int matrix[][SIZE]) {
    //初始化邻接矩阵
    //对角线元素为0，其余都为1
    for (int i = 0; i < SIZE; i++) {
        visited[i] = 0;       //将标志位初始为0
        for (int j = i; j < SIZE; j++) {
            if (i == j)
                matrix[i][j] = 0;
            else              //邻接矩阵的对称性
                matrix[i][j] = matrix[j][i] = 1;
        }
    }

    //判断不安全状态（狼和羊、羊和菜）
    //使得其它顶点与不安全状态的顶点间无路径
    for (int i = 0; i < SIZE; i++) {
        if (unsafe(i)) {
            visited[i] = 1;       //访问了不安全状态的顶点
            for (int j = 0; j < SIZE; j++)
                matrix[i][j] = matrix[j][i] = 0;
        }
    }

    //判断其它顶点是否相连
    //即从某一顶点出发能否直接到达另一顶点
    for (int i = 0; i < SIZE - 1; i++)
        for (int j = i + 1; j < SIZE; j++)
            if (visited[i] == 0 && visited[j] == 0 && unavailable(i, j))
                matrix[i][j] = matrix[j][i] = 0;

    //将标志位重置为0
    for (int i = 0; i < SIZE; i++)
        visited[i] = 0;
}
```

```
//判断农夫、狼、菜、羊的位置
int farmer_location(int state) {return ((state & 0x08) != 0);}

int wolf_location(int state) {return ((state & 0x04) != 0);}

int cabbage_location(int state) {return ((state & 0x02) != 0);}

int sheep_location(int state) {return ((state & 0x01) != 0);}

//判断不安全状态
int unsafe(int state) {
    int judge = 0;
    //狼和羊单独在一起
    if (wolf_location(state) == sheep_location(state) &&
        wolf_location(state) != farmer_location(state))
        judge = 1;
    //羊和菜单独在一起
    if (cabbage_location(state) == sheep_location(state) &&
        sheep_location(state) != farmer_location(state))
        judge = 1;
    return judge;
}

//狼到达对岸
int case1(int i, int j) {
    if (wolf_location(i) != wolf_location(j)) {
        if (sheep_location(i) == sheep_location(j) &&
            cabbage_location(i) == cabbage_location(j))
            return 1;
        else return 0;
    }else
        return 0;
}

//菜到达对岸
int case2(int i, int j) {
    if (cabbage_location(i) != cabbage_location(j)) {
        if (sheep_location(i) == sheep_location(j) &&
            wolf_location(i) == wolf_location(j))
            return 1;
        else return 0;
    }else
        return 0;
}
```

```
//羊到达对岸
int case3(int i, int j) {
    if (sheep_location(i) != sheep_location(j)) {
        if (wolf_location(i) == wolf_location(j) &&
            cabbage_location(i) == cabbage_location(j))
            return 1;
        else return 0;
    }else
        return 0;
}

//判断其它顶点是否相连
int unavailable(int i, int j) {
    int judge = 1;
    // <i,j>边存在
    if (farmer_location(i) != farmer_location(j)) {
        //农夫独自返回对岸
        if ((i & 0x07) == (j & 0x07))
            judge = 0;
        //农夫带东西返回对岸
        if (case1(i, j) || case2(i, j) || case3(i, j))
            judge = 0;
    }
    return judge;
}

//深度优先搜索
void DFS(int matrix[][SIZE], int start, int end) {
    visited[start] = 1;
    push(start);          //入栈
    if (start == end)
        PrintPath();
    for (int j = 0; j < SIZE; j++)
        if (!visited[j] && matrix[start][j] == 1)
            DFS(matrix, j, end);
    pop();                //出栈
    visited[start] = 0;
}
```

```
//输出路径
void PrintPath(void) {
    int i;
    int row = 10;
    int col = 30;
    SetCCPos(col * number, row * i);
    printf("方案%d：", number + 1);
    for (i = 0; i < top - 1; i++) {
        SetCCPos(col * number, row * i + 2);
        puts(situation[stack[i]]);
        SetCCPos(col * number, row * i + 3);
        puts("--------------");
        SetCCPos(col * number, row * i + 4);
        puts("    小河");
        SetCCPos(col * number, row * i + 5);
        puts("--------------");
        SetCCPos(col * number, row * i + 6);
        int opposite = (~(stack[i])) & 0x0F;
        puts(situation[opposite]);
        SetCCPos(col * number, row * i + 7);
        SetCCPos(col * number, row * i + 8);
        SetCCPos(col * number, row * i + 9);
        puts("     ->");
        SetCCPos(col * number, row * i + 10);
        SetCCPos(col * number, row * i + 11);
    }
    SetCCPos(col * number, row * i + 2);
    puts(situation[stack[i]]);
    SetCCPos(col * number, row * i + 3);
    puts("--------------");
    SetCCPos(col * number, row * i + 4);
    puts("    小河");
    SetCCPos(col * number, row * i + 5);
    puts("--------------");
    SetCCPos(col * number, row * i + 6);
    int opposite = (~(stack[i])) & 0x0F;
    puts(situation[opposite]);
    interface_bottom = ((row * (top - 1) + 7) > interface_bottom) ? (row * (top - 1) + 7) :
interface_bottom;
    SetCCPos(0, interface_bottom);
    number++;
}
```

```
//判断栈是否为空
int StackIsEmpty(void) {
    return (top == 0) ? 1 : 0;
}

//判断栈是否已满
int StackIsFull(void) {
    return (top == SIZE) ? 1 : 0;
}

//入栈
int push(int data) {
    if (StackIsFull())
        return 0;
    else
        stack[top++] = data;
    return 1;
}

//出栈
int pop(void) {
    if (StackIsEmpty()) {
        printf("False! Stack is Empty.\n");
        exit(0);
    }
    else
        return stack[--top];
}

//设置光标位置
void SetCCPos(int x, int y) {
    HANDLE hOut;
    hOut = GetStdHandle(STD_OUTPUT_HANDLE);     //获取标注输出句柄
    COORD pos;
    pos.X = x;
    pos.Y = y;
    SetConsoleCursorPosition(hOut, pos);        //偏移光标位置
}
```