数据结构实习报告

题目：编制一个马踏棋盘问题的求解程序。

组员：朱天瑶 梅晓东 杨雪莹

一、 需求分析

(1) 将马放在国际象棋 8×8 棋盘的某个方格中，马按走棋规则进行移动。要求每个方格只进入一次，走遍棋盘上全部 64 个方格。

(2) 编写非递归程序，由用户输入一个马的初始位置，求出马的行走路线，并按此路线将数字 1，2，…，64 依次填入答案数组 ans[8][8]，并输出。

(3) 采取某些策略选择每次试探的下一步可走位置，使得回溯次数尽可能少。

(4) 求出从某一起点出发的给定条数路线。

二、 概要设计

1. 问题分析：

马踏棋盘问题是一个经典的搜索问题，其本质是在一张特殊的图上求一条或若干条哈密顿路径。

2. 算法设计：

采用深度优先搜索为基础算法对问题进行求解。若要缩短得到第一个或前几个解的时间，可以尝试减小搜索树的规模。一个常见的可行策略为每次选择使得下一次搜索分支数最小的结点进行搜索，即基于贪心策略的启发式搜索。

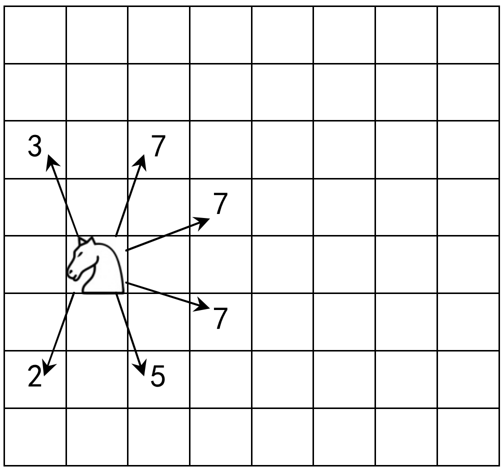

图 1. 所有下一步可走结点的搜索分支数

值得注意的是，这种启发式算法在马踏棋盘这种特殊的求哈密顿路径问题中有极好的效果，并被称作 Warnsdorff 法则。虽然求哈密顿路径是一个 NP 问

题，但是对于较小规模的棋盘，Warnsdorff 法则能够在大多数情况下于线性时间内求出一个解。

在 Warnsdorff 法则的实现中有一个明显的问题，即如何在使得下一次搜索分支数并列最小的若干个结点中选择。受到 Pohl 等的启发[1][2]，我们对马每步可走的 8 个方向进行编号（图 2），在每次选择结点时若遇到并列最小的结点，则按照某个固定的编号顺序选择并列最小结点中顺序最靠前的一个。经试验，48172635 的优先顺序可使得 Warnsdorff 法则在从 8×8 棋盘的全部 64 个位置起始的情况下均能不回溯即求出一个解，同时在求多个解的情况下也有较好的优化效果。

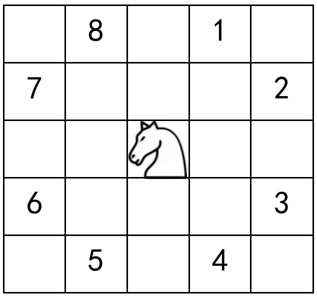

图 2. 8 个可走方向的编号

在求一条路径的算法中，我们采取线性选择方法选取马下一步所走的位置。

在求多条路径的算法中，我们对每个搜索结点的所有搜索分支进行插入排序，优先选择顺序靠前的分支进行搜索。

3. 数据结构设计：

马踏棋盘问题等价于一个精确覆盖问题[3]，为快速实现其回溯及启发式搜索，我们采用 Dancing Links 数据结构来实现 X 算法。

Dancing Links 的基础数据结构是一个双向循环十字链表，其类型定义如下：

```
ADT DLX{
  数据对象：D={e|e∈struct node*}
  struct node{
      int pos;
      struct node *up, *down, *left, *right;
  };
```

数据关系：R1={<e, e->up>, <e, e->down>, <e, e->left>, <e, e->right>}

R2={<e_i, e_{i+1}>, i=1, 2,…, n}

基本操作：

add(i, j)

初始条件：表头、表尾数组已存在。

操作结果：在(i, j)位置新增一个结点。

初始条件：双向循环十字链表已存在。

cover_r(struct node *p)

操作结果：从纵向删去结点。

cover_r(struct node *p)

操作结果：从横向删去结点。

recover_r(struct node *p)

操作结果：从纵向恢复结点。

recover_r(struct node *p)

操作结果：从横向恢复结点。

}

三、 详细设计

1. 求一条路径的程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// 8 directions
const int da[8] = {2, -2, -2, -1, -1, 1, 1, 2};
const int db[8] = {1, -1, 1, -2, 2, -2, 2, -1};

int w[8][8] = {{2, 3, 4, 4, 4, 4, 3, 2},
               {3, 4, 6, 6, 6, 6, 4, 3},
               {4, 6, 8, 8, 8, 8, 6, 4},
               {4, 6, 8, 8, 8, 8, 6, 4},
               {4, 6, 8, 8, 8, 8, 6, 4},
               {4, 6, 8, 8, 8, 8, 6, 4},
               {3, 4, 6, 6, 6, 6, 4, 3},
               {2, 3, 4, 4, 4, 4, 3, 2}};   // Numbers of branches
int ans[8][8];  // The Hamiltonian path
```

```
int legal(int a)
{
    if (a >= 0 && a < 8)
        return 1;
    return 0;
}   // Judge if the position is in the legal range

int main()
{
    int a0, b0;
    scanf("%d %d", &a0, &b0);   // Input starting position
    memset(ans, 0, sizeof(ans));
    int i, j, min, a, b, a_min, b_min;
    for (i = 1; i <= 64; i++)
    {
        ans[a0][b0] = i;
        min = 8;
        for (j = 0; j < 8; j++)
        {
            a = a0 + da[j], b = b0 + db[j];
            if (legal(a) && legal(b) && !ans[a][b] && --w[a][b] < min)
            {
                min = w[a][b];
                a_min = a, b_min = b;
            }
        }
        a0 = a_min, b0 = b_min; // Find the place with least branches
    }
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
            printf("%3d", ans[i][j]);
        printf("\n");
    }   // Output the path
    return 0;
}
```

2. 求多条路径的程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// 8 directions
```

```c
const int da[8] = {2, -2, -2, -1, -1, 1, 1, 2};
const int db[8] = {1, -1, 1, -2, 2, -2, 2, -1};

struct node{
    int pos;
    struct node *up, *down, *left, *right;
};  // Datatype of nodes in DLX

struct vertex{
    int pos, w;
};  // Datatype of vertexs for sorting

struct node *c[64], *r[64], *ct[64], *rt[64];   // DLX head and tail array
int w[64];  // Numbers of branches
int ans[8][8];  //The Hamiltonian path
int a0, b0, n, t;

int legal(int a)
{
    if (a >= 0 && a < 8)
        return 1;
    return 0;
}   // Judge if the position is in the legal range

void sort(struct vertex *list, int n)
{
    int i, j;
    for (i = 1; i < n; i++)
    {
        struct vertex key = list[i];
        j = i - 1;
        while (j >= 0 && list[j].w > key.w)
        {
            list[j + 1] = list[j];
            j--;
        }
        list[j + 1] = key;
    }
}   // Insert sorting

void add(int pos0, int pos)
{
    struct node *p = (struct node*)malloc(sizeof(struct node));
    p->pos = pos;
```

```
    ct[pos]->down = p, p->up = ct[pos], p->down = NULL, ct[pos] = p;
    rt[pos0]->right = p, p->left = rt[pos0], p->right = NULL, rt[pos0] = p;
}   // Add new nodes to DLX

// Dancing moves
void cover_r(struct node *p)
{
    p->up->down = p->down;
    p->down->up = p->up;
}

void cover_c(struct node *p)
{
    p->left->right = p->right;
    p->right->left = p->left;
}

void recover_r(struct node *p)
{
    p->up->down = p;
    p->down->up = p;
}

void recover_c(struct node *p)
{
    p->left->right = p;
    p->right->left = p;
}

void print()
{
    printf("Path %d:\n", t);
    int i, j;
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
            printf("%3d", ans[i][j]);
        printf("\n");
    }
}   // Output the path

void find(int pos0, int step)
{
    step++;
```

```
    int a = pos0 / 8, b = pos0 % 8;
    ans[a][b] = step;
    if (!w[pos0])   // No more position to go
    {
        if (step < 64)  // A wrong path
            return;
        t++;    // A right path
        print();
        return;
    }
    struct node *p;
    for (p = c[pos0]; p->down != c[pos0]; p = p->down)
        cover_c(p->down);
    struct vertex list[8];
    int i = 0, j, k;
    for (p = r[pos0]; p->right != r[pos0]; p = p->right)
    {
        cover_r(p->right);
        list[i].pos = p->right->pos, list[i].w = --w[p->right->pos];
        i++;
    }   // Cover
    sort(list, i);
    for (j = 0; j < i; j++)
        if (t < n)
            find(list[j].pos, step);    // Search the next step
    for (p = c[pos0]; p->down != c[pos0]; p = p->down)
        recover_c(p->down);
    for (p = r[pos0]; p->right != r[pos0]; p = p->right)
    {
        recover_r(p->right);
        w[p->right->pos]++;
    }   // Recover
}   // DFS

int main()
{
    scanf("%d %d", &a0, &b0);   //Input starting position
    scanf("%d", &n);    // Input the number of paths wanted
    memset(ans, 0, sizeof(ans));
    memset(w, 0, sizeof(w));
    t = 0;
    int i, j;
    for (i = 0; i < 64; i++)
    {
```

```
        c[i] = (struct node*)malloc(sizeof(struct node));
        c[i]->up = c[i]->down = c[i]->left = c[i]->right = NULL;
        ct[i] = c[i];
        r[i] = (struct node*)malloc(sizeof(struct node));
        r[i]->up = r[i]->down = r[i]->left = r[i]->right = NULL;
        rt[i] = r[i];
    }  // Initialize DLX
    for (i = 0; i < 64; i++)
        for (j = 0; j < 8; j++)
        {
            int a = i / 8 + da[j], b = i % 8 + db[j];
            if (legal(a) && legal(b))
            {
                add(i, a * 8 + b);
                w[i]++;
            }
        }
    for (i = 0; i < 64; i++)
    {
        ct[i]->down = c[i], c[i]->up = ct[i];
        rt[i]->right = r[i], r[i]->left = rt[i];
    }  // Build DLX
    find(a0 * 8 + b0, 0);   // DFS
    return 0;
}
```

四、 调试分析

1. 求一条路径的复杂度

在 8×8 棋盘中，设总步数为n，则算法的时间复杂度为$O(n)$。无论从哪一个位置出发，程序均将在 64 次选择后输出一条路径。

2. 求多条路径的复杂度

在 8×8 棋盘中，设所求路径数目为n。我们对n=1, 10, 100, ⋯, 1000000 的情形，分别以全部 64 个位置为出发点进行测试，以 find 函数总调用次数的平均值$f(n)$作为衡量，并对n和$f(n)$取对数后作图，结果如下：

n	1	10	100	1000	10000	100000	1000000
$f(n)$	64	194	3593	76233	1947984	41985040	833903040
$lg\ f(n)$	1.8062	2.2867	3.5554	4.8821	6.2896	7.6231	8.9211

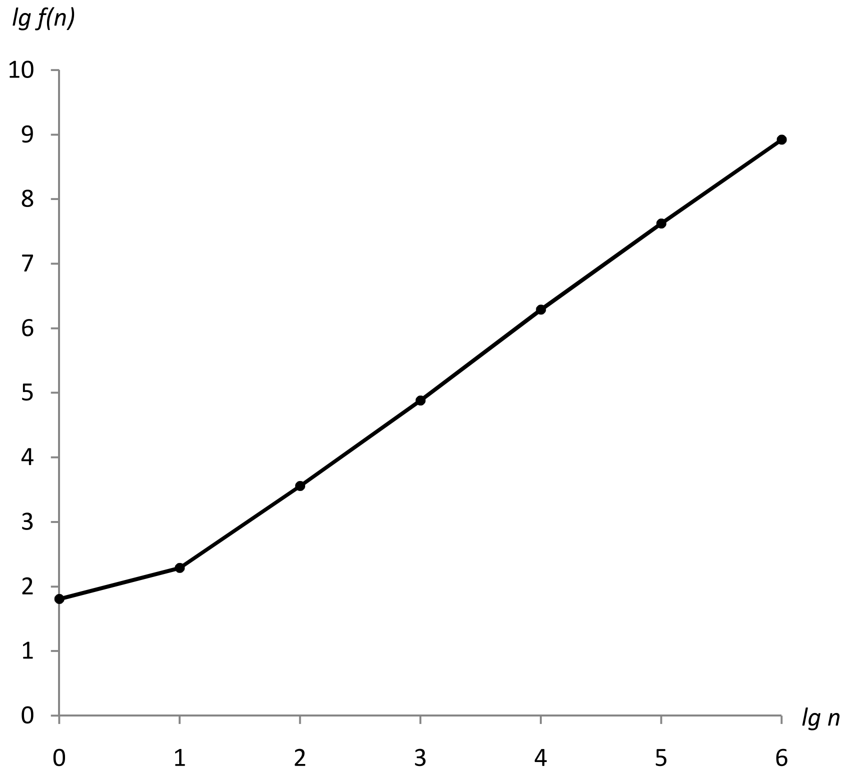

图 3.

从图 3 中可以看出，在 n≤1000000 的范围内，$f(n)$与n大致呈线性关系，其时间复杂度可能是$O(n)$的。

五、 用户手册

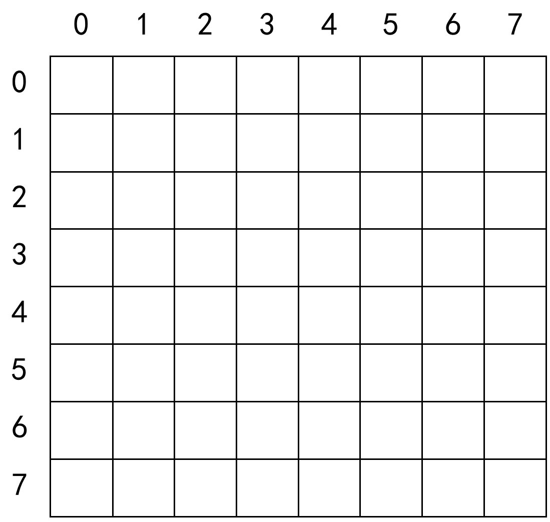

图 4.

1. 求一条路径:

输入两个整数，分别代表用户指定的初始位置的行和列的坐标（图 4)。

2. 求多条路径:

输入三个整数，前两个分别代表用户指定的初始位置的行和列的坐标（图 4)，后一个为用户指定要求出的路径条数。

六、 测试结果

1. 求一条路径:

样例输入：

```
0 0
```

样例输出：

```
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 60 53 36 43 30 41
57 48 37 44 61 52 11 26
 6 21 56 59 54 27 64 29
49 58 23  8 51 62 25 10
22  7 50 55 24  9 28 63
```

2. 求多条路径:

样例输入：

```
0 0
10
```

样例输出：

```
Path 1:
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 60 53 36 43 30 41
57 48 37 44 61 52 11 26
 6 21 56 59 54 27 64 29
49 58 23  8 51 62 25 10
22  7 50 55 24  9 28 63
Path 2:
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
```

```
 47  2 35 38 45 40 15 12
 20  5 60 53 36 43 30 41
 57 48 37 44 61 52 11 26
  6 21 56 59 54 27 62 29
 49 58 23  8 51 64 25 10
 22  7 50 55 24  9 28 63
Path 3:
  1 34  3 18 39 32 13 16
  4 19 46 33 14 17 42 31
 47  2 35 38 45 40 15 12
 20  5 64 53 36 43 30 41
 57 48 37 44 63 52 11 26
  6 21 56 59 54 27 62 29
 49 58 23  8 51 60 25 10
 22  7 50 55 24  9 28 61
Path 4:
  1 34  3 18 39 32 13 16
  4 19 46 33 14 17 42 31
 47  2 35 38 45 40 15 12
 20  5 58 53 36 43 30 41
 57 48 37 44 59 52 11 26
  6 21 56 63 54 27 60 29
 49 64 23  8 51 62 25 10
 22  7 50 55 24  9 28 61
Path 5:
  1 34  3 18 39 32 13 16
  4 19 46 33 14 17 42 31
 47  2 35 38 45 40 15 12
 20  5 62 53 36 43 30 41
 63 48 37 44 57 52 11 26
  6 21 56 61 54 27 58 29
 49 64 23  8 51 60 25 10
 22  7 50 55 24  9 28 59
Path 6:
  1 34  3 18 39 32 13 16
  4 19 46 33 14 17 42 31
 47  2 35 38 45 40 15 12
 20  5 64 53 36 43 30 41
 63 48 37 44 57 52 11 26
  6 21 56 61 54 27 58 29
 49 62 23  8 51 60 25 10
 22  7 50 55 24  9 28 59
Path 7:
  1 34  3 18 39 32 13 16
```

```
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 58 53 36 43 30 41
59 48 37 44 57 52 11 26
 6 21 56 61 54 27 64 29
49 60 23  8 51 62 25 10
22  7 50 55 24  9 28 63
Path 8:
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 60 53 36 43 30 41
57 48 37 44 59 52 11 26
 6 21 58 61 54 27 64 29
49 56 23  8 51 62 25 10
22  7 50 55 24  9 28 63
Path 9:
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 64 53 36 43 30 41
57 48 37 44 59 52 11 26
 6 21 58 63 54 27 60 29
49 56 23  8 51 62 25 10
22  7 50 55 24  9 28 61
Path 10:
 1 34  3 18 39 32 13 16
 4 19 46 33 14 17 42 31
47  2 35 38 45 40 15 12
20  5 58 53 36 43 30 41
57 48 37 44 63 52 11 26
 6 21 64 59 54 27 62 29
49 56 23  8 51 60 25 10
22  7 50 55 24  9 28 61
```

七、 附录

参考文献：

[1] Pohl, Ira (July 1967). "A method for finding Hamilton paths and Knight’s tours". *Communications of the ACM* 10 (7): 446–449.

[2] Squirrel, Douglas; Cull, P. (1996). "A Warnsdorff-Rule Algorithm for Knight's Tours on Square Boards".

[3] Knuth, Donald (2000). "Dancing links". *Millennial Perspectives in Computer Science*. P159–187.