

实验六

1. 问题描述与实验目的:

给定 8*8 方格棋盘, 求棋盘上一只马从一个位置到达另一位置的最短路径长。注意马是走“日”形的。

2. 要点分析和算法描述:

该题的主要思想是使用 dfs 或 bfs 从棋盘指定点开始遍历, 直到遍历到棋盘边缘停止。用 st[9][9]二维数组记录棋盘从某点到所有点的最少步数, 最后输出 st[i][j]指定点即可。

为了测试 dfs 和 bfs 时间复杂度差异, 由于棋盘只有 8*8 大小, 从点 a 到点 b 一共有 64*64 种, 可以先生成所有的测试数据, 放入 in.in 文件种作为输入文件, 将结果输出到 out_dfs.out/out_bfs.out 文件中。从读取字符串开始使用时钟信号, 结束时输出时间差, 单位 ms, 即可比较 dfs 和 bfs 性能差异。

3. 代码实现:

(1) 解决方法主代码:

```
#include <cstring>
#include <fstream>
#include <iostream>
#include <queue>

#define PII pair<int, int>
#define x first
#define y second

#define rep(a, b) for (int i = (a); i < (b); i++)

using namespace std;

// 8 个方向, 逆时针, 横x, 纵y
int dx[] = {2, 1, -1, -2, -2, -1, 1, 2};
int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};
int st[9][9]; // 表示棋盘

// 广度优先遍历
void bfs(int &sx, int &sy)
{
    queue<PII> q;
    q.push({sx, sy}); // 起点入队
    while (!q.empty())
    {
        int ux = q.front().x, uy = q.front().y;
        q.pop();
        // 枚举8个方向
        rep(0, 8)
```

```

{
    int vx = ux + dx[i], vy = uy + dy[i];
    if (vx <= 0 || vy <= 0 || vx > 8 || vy > 8)
        continue;
    if (st[vx][vy] > st[ux][uy] + 1)
    {
        st[vx][vy] = st[ux][uy] + 1;
        q.push({vx, vy});
    }
}
}

// 深度优先遍历
void dfs(int ux, int uy)
{
    rep(0, 8)
    {
        int vx = ux + dx[i], vy = uy + dy[i];
        if (vx <= 0 || vy <= 0 || vx > 8 || vy > 8)
            continue;
        if (st[vx][vy] > st[ux][uy] + 1)
        {
            st[vx][vy] = st[ux][uy] + 1;
            dfs(vx, vy);
        }
    }
}

// 主函数
int main()
{
    ifstream r;
    r.open("../in.in", ios::in);
    ofstream w;
    w.open("../out_bfs.out", ios::out);
    clock_t start;
    start = clock();
    string s, t;
    while (r >> s >> t)
    {
        // 坐标转化, 如 "a1", 将 a 转为 1, 字符 '1' 转为数字 1
        int sx = s[0] - 'a' + 1, sy = s[1] - '0'; // s 表起点的 x, y 坐标
        int tx = t[0] - 'a' + 1, ty = t[1] - '0'; // t 表终点的 x, y 坐标

```

```

        memset(st, 0x3f, sizeof(st));
        st[sx][sy] = 0;
        bfs(sx, sy);
        w << s << "=>" << t << ":" " << st[tx][ty] << "moves" << endl;
    }

    // 输出运行时间
    w << "运行时间为: " << double(clock() - start) << "ms" << endl;
    r.close();
    w.close();
    return 0;
}

```

(2) 生成棋盘所有测试数据的代码:

```

// 构造跳马测试数据
#include <iostream>
#include <fstream>
using namespace std;
string s(int x)
{
    int k = (x - 1) / 8;
    char a = k + 'a';
    char b = (x - 1) % 8 + 1 + '0'; // 这种取余方法避免了0的情况
    string res;
    res += a;
    res += b;
    return res;
}

int main()
{
    ofstream w;
    w.open("../in.in", ios::out);
    // 8 行8 列的棋盘
    for (int i = 1; i <= 64; i++)
        for (int j = 1; j <= 64; j++)
            w << s(i) << " " << s(j) << endl;
    w.close();
    return 0;
}

```

4. 运行结果:

(1) 所有测试数据生成结果:

in.in		
4079	h8	t7
4080	h8	f8
4081	h8	g1
4082	h8	g2
4083	h8	g3
4084	h8	g4
4085	h8	g5
4086	h8	g6
4087	h8	g7
4088	h8	g8
4089	h8	h1
4090	h8	h2
4091	h8	h3
4092	h8	h4
4093	h8	h5
4094	h8	h6
4095	h8	h7
4096	h8	h8
4097		

(2) bfs 测试结果:

out_bfs.out	
4084	h8==>g4: 3moves
4085	h8==>g5: 2moves
4086	h8==>g6: 1moves
4087	h8==>g7: 4moves
4088	h8==>g8: 3moves
4089	h8==>h1: 5moves
4090	h8==>h2: 4moves
4091	h8==>h3: 3moves
4092	h8==>h4: 2moves
4093	h8==>h5: 3moves
4094	h8==>h6: 2moves
4095	h8==>h7: 3moves
4096	h8==>h8: 0moves
4097	运行时间为: 92ms
4098	

(3) dfs 测试结果:

```
out_dfs.out
4087 h8==>g7: 4moves
4088 h8==>g8: 3moves
4089 h8==>h1: 5moves
4090 h8==>h2: 4moves
4091 h8==>h3: 3moves
4092 h8==>h4: 2moves
4093 h8==>h5: 3moves
4094 h8==>h6: 2moves
4095 h8==>h7: 3moves
4096 h8==>h8: 0moves
4097 运行时间为: 165ms
4098
```

5. 实验体会:

可以看到，使用 bfs 方法所用的时间比 dfs 方法缩短很多，大约只占到 dfs 方法的 55%时间。这是因为 bfs 优先搜索逐层扩展，先访问距离起点最近的节点，然后向外逐层扩展，保证最先到达的目标点必然是通过最短路径到达的。而 dfs 是沿着一个方向走到尽头，然后回溯，再尝试其他方向，这意味着 dfs 会在找到最短路前，搜索了很多较长的路径，耗费了很多无用的时间。