学　　号：	0121610870910

武漢理工大學

《算法设计与分析 B》课程报告

题　　目	基于广度优先搜索方法的骑士游历问题求解
学　　院	计算机科学与技术
专　　业	软件工程
班　　级	软件 1604 班
姓　　名	冯钢果
指导教师	李晓红

2018 年 5 月 28 日

目录

1 注册资料

题目来源：Peking University Online Judge 和 Zhejiang University Online Judge

用户名：Law_less 和 WillAlso

题目 ID：poj2243（或 zoj1091）

扩展题目：poj1915、zoj3467

2 选题描述

英文题目描述：

A friend of you is doing research on the Traveling Knight Problem (TKP) where you are to find the shortest closed tour of knight moves that visits each square of a given set of n squares on a chessboard exactly once. He thinks that the most difficult part of the problem is determining the smallest number of knight moves between two given squares and that, once you have accomplished this, finding the tour would be easy.

Of course you know that it is vice versa. So you offer him to write a program that solves the "difficult" part.

Your job is to write a program that takes two squares a and b as input and then determines the number of knight moves on a shortest route from a to b.

Input Specification

The input file will contain one or more test cases. Each test case consists of one line containing two squares separated by one space. A square is a string consisting of a letter (a-h) representing the column and a digit (1-8) representing the row on the chessboard.

Output Specification

For each test case, print one line saying "To get from xx to yy takes n knight moves.".

中文题目描述：

你的一位朋友正在研究旅行骑士问题（TKP），在那里你可以找到骑士移动的最短闭环行程，该行程只需一次访问棋盘上给定 n 个方格的每个方格。他认为，问题中最困难的部分是确定两个给定方格之间的最小骑士移动数量，并且一旦你完成了这一步，找到巡回赛将很容易。

当然你知道它反之亦然。所以你让他写一个解决“困难”部分的程序。

你的工作是编写一个程序，它需要两个方格 a 和 b 作为输入，然后确定在从 a 到 b 的最短路线上的骑士移动数量。

输入规格

输入文件将包含一个或多个测试用例。每个测试用例由一行包含两个用一个空格分隔的正方形组成。正方形是由表示列的字母（a-h）和表示棋盘上的行的数字（1-8）组成的字符串。

输出规格

对于每个测试用例，打印一行“从 xx 到 yy 需要 n 次骑士移动”。

3 算法设计

3.1 问题分析

输入样例	输出样例	时间限制	内存限制	问题类型
e2 e4 a1 b2 b2 c3 a1 h8 a1 h7 h8 a1 b1 c3 f6 f6	To get from e2 to e4 takes 2 knight moves. To get from a1 to b2 takes 4 knight moves. To get from b2 to c3 takes 2 knight moves. To get from a1 to h8 takes 6 knight moves. To get from a1 to h7 takes 5 knight moves. To get from h8 to a1 takes 6 knight moves. To get from b1 to c3 takes 1 knight moves. To get from f6 to f6 takes 0 knight moves.	2 Seconds	65536 KB	求两点最短路径问题

分析：问题大意是国际象棋中骑士如何从起点用最短的步数走到指定终点。

相关背景知识：国际象棋中骑士行走规则是“日”字形状，类似于中国象棋中的马的行走规则：走 1X2 或 2X1 的矩形对角线；另外是国际象棋棋盘为 8X8 的网格状。

3.2 算法设计方法比较选择

骑士游历问题是一个经典图的搜索问题，我们能想到的方法是广度优先搜索和深度优先搜索，本题目是求图的最短路径问题，而在解决最短路径问题，首先想到的方法是广度优先

搜索方法（Breadth-First-Search）、Floyd 算法、Dijkstra 算法。本题目解决的问题规模比较小，也可以使用深度优先搜索（Depth-First-Search）、利用数学推到的方程进行枚举、递归算法。当然，本文主要围绕广度优先算法进行解决并给出优化方法，其他方法只简单介绍思想，并对主要代码进行解释。以下是深度优先搜索、广度优先搜索、Floyd 算法、Dijkstra 算法特点：

1) BFS 是用来搜索最短径路的解是比较合适的，比如求最少步数的解，最少交换次数的解，因为 BFS 搜索过程中遇到的解一定是离根最近的，所以遇到一个解，一定就是最优解，此时搜索算法可以终止。这个时候不适宜使用 DFS，因为 DFS 搜索到的解不一定是离根最近的，只有全局搜索完毕，才能从所有解中找出离根的最近的解。
2) DFS 适合搜索全部的解，因为要搜索全部的解，那么 BFS 搜索过程中，遇到离根最近的解，并没有什么用，也必须遍历完整棵搜索树，DFS 搜索也会搜索全部，但是相比 DFS 不用记录过多信息，所以搜素全部解的问题，DFS 显然更加合适。
3) 空间优劣上，DFS 是有优势的，DFS 不需要保存搜索过程中的状态，而 BFS 在搜索过程中需要保存搜索过的状态，而且一般情况需要一个队列来记录。
4) Dijkstra 是单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止；是贪心算法在图类问题的典型应用。
5) Floyd 是多源最短路径，是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包，是一个动态规划在图中的应用。

接下来主要介绍深度优先算法即分支界限法，并给出相关证明和优化分析。（当然也会给出其他方法简单的介绍：递归法、深度优先搜索、枚举法、Floyd 算法，优化时主要介绍 A*算法、双向广度优先搜索）

3.3 算法设计方法

1. 分支界限法及类似算法设计：广度优先算法、深度优先算法；
2. 其他思路方法设计：枚举法、递归法；（同样 Floyd 算法、Dijkstra 算法也能解决，在此不介绍）
3. 优化算法设计：A*算法、双向广度优先算法设计。

骑士行走示意图、框图设计如下：

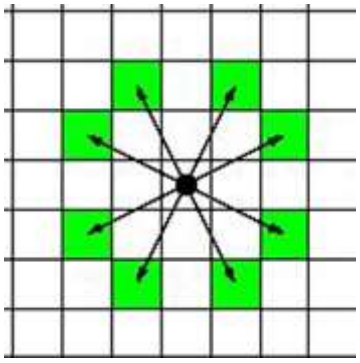

图 1.骑士行走示意图

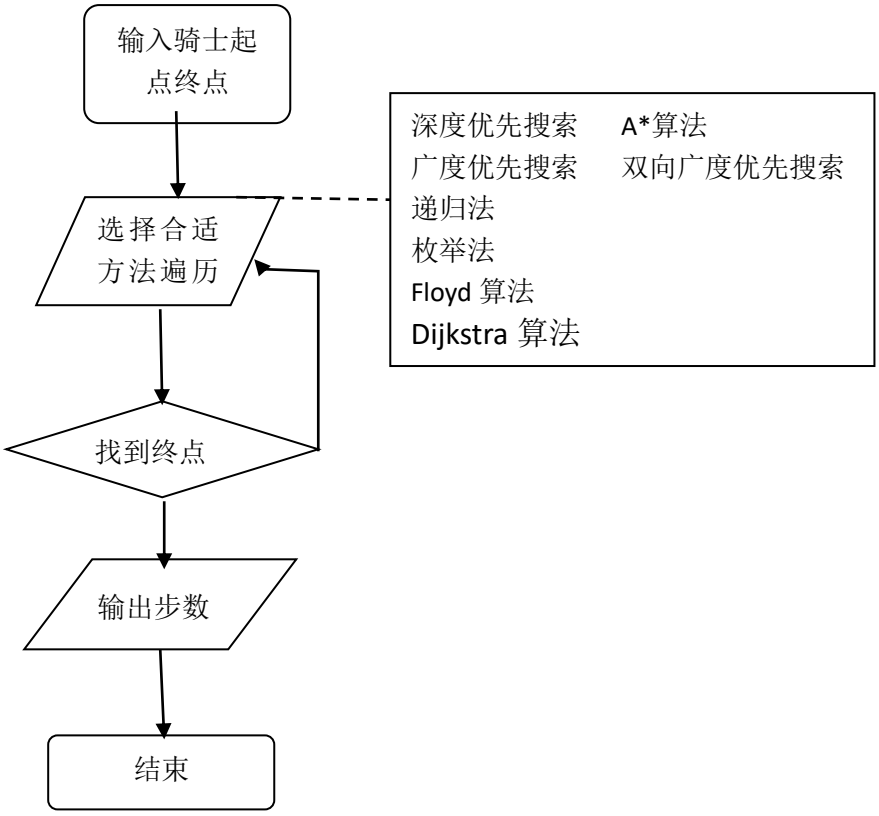

图 2.流程图

具体设计如下：

1.1 广度优先算法

设计思路：本问题遍历骑士游历问题时，主要从起点向八个方向搜索，但不是采用递归方式，而是建立一个队列，将待搜索的节点放到队列里面，已经搜索的节点从队列里面删除，由于没有重复搜索，所以效率要比接下来的深度优先算法高很多。

关键伪码：

```
BFS(poiny src,point dist){                          //src 为起点，dist 为终点
    queue_list.push(src)                            //起点先进入队列
    while(true)
        from = q.front()
        q.pop()
        if (from.x == to.x && from.y == to.y)       //判断是否到达终点
            break
        for i in (0,8)                              //将八个方向的节点加入队列
            temp.x = from.x + dx[i]
```

```
            temp.y = from.y + dy[i]
            queue_list.push(temp)
    }
```

证明：（在这里只给出 BFS 算法正确性证明）

//这里 L(s,v)表示 s 到 v 的最短路径，d 为现节点到源点距离。

先假定某些节点获取的 d 值并不等于其最短路径距离。

设 v 为这样的一个节点，其最短距离为 L(s,v)，而其获得的 d 值不等于该数值，显然 v 不等于 s，所以 v.d >= L(s,v)，因此 v.d > L(s,v)。另外，节点 v 必定是从 s 可以到达的，如果不是这样，则将出现 L(s,v) = ∞ >= v.d。设 u 为从源节点 s 到节点 v 的最短路径上的直接前驱结点，则 L(s,v) = L(s,u) + 1。因为 L(s,u) < L(s,v)，并且因为我们对节点 v 的选择，所以 u.d = L(s,u)。因此：

v.d > L(s,v) = L(s,u) + 1 = u.d + 1

节点从 queue 中取出前，节点状态现在并不确定，而此时如果假设 v 是未访问节点则有 v.d = u.d + 1 与上个式子矛盾；如果假设 v 是访问过的，则有 v.d <= u.d，也与上个式子矛盾，因此对于所有节点 v∈V，v.d = L(s,v)，得证。

1.2 深度优先算法

设计思路：本问题深度优先搜索要便利所有情况，从起点开始向八个方向递归，计算该起点到棋盘所有位置的最短路径。本算法致命弱点是速度慢，因为要遍历所有状况然后比较，而深度优先搜索只需要按照队列先进先出顺序，就能保证最先得到的结果是最短路径，不需要进行便利所有情况。但由于本问题规模较小，所以 DFS 写出的程序也能在 poj 上提交通过，但问题从 8*8 的棋盘改为 n*n 的棋盘就要另外一说了。

关键伪码：

```
DFS(int si,int sj,int moves){
    if arrived at destination              //如果到达终点结束程序
        return 0;
    knight[si][sj] = moves                 //步数记录
    for i in (0,8)                         //遍历相邻的八个方向
        DFS(si + x[i],sj + y[i],moves + 1)
}
```

2.1 枚举法

设计思路：在本问题中枚举法就是列出骑士可能到达的所有位置，然后进行比较；但 DFS 也可以理解为一个图的枚举，这里的枚举与 DFS 不同的是主要应用数学方法进行枚举推导：在本问题中，我们可以先假设起点终点位置差值分别为 X、Y，而骑士向八个方向扩展，可以表示为方向矢量(2,1)、(2,-1)、(-2,1)、(-2,-1)、(1,2)、(1,-2)、(-1,2)、(-1,-2)，可以发现八个方向有四个是另外四个的相反向量，若要求最短路径，不可能同时使用相反向量，因此，可以设方向向量为(1,2),(2,1),(2,-1),(1,-2)的分别使用 a、b、c、d 次，可以列出方程：

$$\begin{cases} a+2b+2c+d=x \\ 2a+b-c-2d=y \end{cases} \xrightarrow{\text{推导得}} \begin{cases} c=(-4a-5b+2x+y)/3 \\ d=(5a+4b-x-2y)/3 \end{cases}$$

根据同余定理的(-4a-5b)与(2x+y)模 3 同余，即(a+2b)与(2x+y)模 3 同余，由于 2x+y 已知，对于 b 进行枚举，由于-n/2<=b<=n/2，进行枚举，对每个知道 a 模 3 是多少，进而再对可能的 a 进行枚举，从而解出 c,d，进而求出总步数，即求|a|+|b|+|c|+|d|的最小值。

关键伪码：

```
FUN(point src,point dist){
    X = dist.x  -  src.x
    Y = dist.y  -  dist.y                          //求起点和终点坐标插值
    for i in (-4,4)
        m=y+2*x-2*b+30                             //a 模 3 的余数
        m %= 3
        for(a=m-6;a<=m+6;a+=3)                     //对 a 枚举
                c=(2*x+y-4*a-5*b)/3;               //求出 c 和 d
                d=(5*a+4*b-x-2*y)/3;
                if (s>abs(a)+abs(b)+abs(c)+abs(d))
                    s=abs(a)+abs(b)+abs(c)+abs(d);  //判断是否是最小的
}
```

2.2 递归法

设计思路：深度优先搜索使用的是递归法，而在这里使用的递归和 DFS 中的递归不同的是没有用去存储图的信息，而是直接使用骑士向八个方向进行递归，直到到达符合要求的节点。如使用 void visit (int row, int col, int moves)去访问列 row 和 col 的节点，在函数内递归调用实现向八个方向递归，没有用结构体解决储存，与 DFS 实现方法类似，要便利

所有的情况才能得出答案，并且时间复杂度和 DFS 相同，不同的是空间使用，并且这种方法易于实现，不用为储存空间分配懊恼，但同样，效率比 BFS 低，不适用于解决大规模求最短路径问题。

关键伪码：

```
Visit(int row, int col, int moves){
    if arrived at destination              //判断是否到达终点
        return 0;
    record move path                       //记录路径步数
    visit(row-2,col-1,moves+1);            //遍历该点相邻的八个点
    visit(row-2,col+1,moves+1);
    visit(row+2,col-1,moves+1);
    visit(row+2,col+1,moves+1);
    visit(row-1,col-2,moves+1);
    visit(row-1,col+2,moves+1);
    visit(row+1,col-2,moves+1);
    visit(row+1,col+2,moves+1);
}
```

3.1 使用 A*算法优化

设计思路：前面提到 BFS 主要解决求最短路径方法，而在求最短路径时进行同宽度搜索时，如果是计算机的 BFS 算法并不会辨别终点位置，而是让骑士直接向八个方向扩展，而对于人来讲，我们肯定会越来越往终点位置去靠，而不是去向八个方向去走，因此，我们可以提出优化方案，通过一定的方法实现剪枝，使算法便利情况向好的方向发展，因此我们采用 A*算法：基于评估函数，在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径。这个算法可以理解为 BFS 和 Dijkstra 的结合，从而使算法效率大大提升。公式如下：

$$f(x) = g(x) + h(x)$$

g(n)表示从起点到任意顶点 n 的实际距离

h(x)表示任意顶点 n 到目标顶点的估算距离

关键伪码：

```
Astar(){
    while(!que.empty())
        t = que.top                                  //队列头节点出队列
        que.pop
        visited[t.x][t.y] = false                    //标记访问
        if arrived at destination                    //判断到达终点
            ant = t.step                             //记录步数
            break
        for i in (0,8)                               //遍历八个方向，同 BFS 方法
            s.x = t.x + dir[i][0]
            s.y = t.y + dir[i][1]
            if (in(s) && !visited[s.x][s.y])         //判断节点是否访问
                s.g = t.g + 23;                      //进行 g(x)函数计算
                s.h = Heuristic(s);                  //曼哈顿估价函数
                s.f = s.g + s.h;                     //计算估值函数
                s.step = t.step + 1;
                que.push(s);
}
```

3.2 使用双向广度优先算法优化

设计思路：本问题主要使用 BFS 求解最短路径，我们除了使用 A*算法进行剪枝，还能想办法缩小 BFS 的搜索树深度，树的深度越深，搜索时间越长，而这个时间增加速度是指数级的增加，因此，我们可以使用从终点和起点同时扩展，在增加同一层搜索树的节点的同时减少树的深度，从而达到时间优化，这种方法我们叫做“双向广度优先算法”。在这里，我们可以使用骑士的起点和终点进行扩展，直到两个扩展方向出现交点，那么可以认为找到了一条路径。本方法特别适用于给出了起始节点和目的节点，要求它们之间的最短路径问题。

关键伪码：

```
DBFS(point src,point dist){
    queue_list1.push(src)                            //起点先进入队列
    queue_list2.push(dist)                           //终点进入队列
```

```
    while(true)
        traverse queue_list1                          //遍历队列 1 的元素
        traverse queue_list2                          //遍历队列 2 的元素
        if there is an intersection between queue_list1 and queue_list2
                                                      //如果队列 1 和队列 2 存在交点
            print path                                //说明到达终点，输出信息
}
```

4 算法实现

数据结构设计及表示

使用结构体表示节点信息：

```
struct point{
    int x,y;          //节点纵横坐标
    int c;            //节点其他信息（步数）
 };
```

关键算法代码说明（BFS 和 DFS 的关键代码）

1. BFS 关键代码：

```
    while(true){
                from = q.front();
                q.pop();
                if(from.x == to.x && from.y == to.y){
                    break;}
                for(int i = 0;i < 8;i++){
                    temp.x = from.x + dx[i];
                    temp.y = from.y + dy[i];
                    temp.c = from.c + 1;
                    if(temp.x < 0|| temp.x > 7 || temp.y < 0 || temp.y > 7){
                        continue; }
                    q.push(temp);
```

```
        }   }
```

2. DFS 关键代码：

```
void DFS(int si,int sj,int moves){
    if(si < 0 || sj < 0 || si >= 8 || sj >= 8 || moves >= knight[si][sj])
    {
        return;
    }
    knight[si][sj] = moves;
    int i;
    for(i = 0;i < 8;i++){
        DFS(si + x[i],sj + y[i],moves + 1);
    }
}
```

代码测试

经过调试编译代码通过，然后提交到 POJ 上，但需要注意的是 POJ 是多组数据测试，须将之前的测试一组数据改为 while 循环来实现多组数据输入，最后 POJ 提交通过。

代码优化（使用双向广度优先算法、A*算法优化，关键代码如下）

1. A*算法

```
for(int i=0;i<8;i++){
            s.x = t.x + dir[i][0];
            s.y = t.y + dir[i][1];
            if(in(s) && !visited[s.x][s.y])
             {
                s.g = t.g + 23;
                s.h = Heuristic(s);              //曼哈顿估价函数
                s.f = s.g + s.h;
                s.step = t.step + 1;
                que.push(s);
            }
        }
```

2. 双向广度优先算法

```
for(i=0;i<8;i++)  {
        next.x=now.x+fx1[i];
        next.y=now.y+fx2[i];
        if(inside(next.x,next.y)&&vis1[next.x][next.y]!=-1){
          printf("%d\n",vis2[now.x][now.y]+1+vis1[next.x][next.y]);
          return;  }
        if(inside(next.x,next.y)&&vis2[next.x][next.y]==-1){
          vis2[next.x][next.y]=vis2[now.x][now.y]+1;
          w.push(next);
        }  }
```

5 算法效率分析

对于 BFS 方法，因为要遍历所有的点和边，所以时间复杂度为 O(V+E)使用空间为 $O(n^2)$；

DFS 方法主要使用递归遍历搜索的顶点 n*n，所以时间复杂度为 $O(n^2)$,空间为 $O(n^2)$；

枚举法要对 n*n 的大部分情况进行列举，所以时间复杂度为(n^2)，而空间只使用了几个变量，所以算法空间复杂度为 O(1)；

递归法与 DFS 相似，时间复杂度为 $O(n^2)$，只不过空间是使用一个四维的数组，因此空间复杂度为 $O(n^4)$；

A*算法在这里并不能简单的计算出时间复杂度，这里给出的是情况最差的时间复杂度 O(V+E)，使用的空间与 DFS 相同，空间复杂度为 $O(n^2)$；

双向 BFS 算法时间复杂度与 BFS 相似，只不过时间搜索的树的深度不同，但只是系数不同，所以时间复杂度为 O(V+E)，空间使用是 BFS 的 2 倍，空间复杂度为 $O(n^2)$。

分析：可以看出 A*算法的巨大效率，因此优化方案可行；但我们也可以看出 DFS 在最短路径求解小规模问题的效率低的缺点已经暴露无遗，因此，对于不同特点的问题要学会挑选合适的方法很重要。

方法	理论时间复杂度	理论空间复杂度	实际消耗时间	实际消耗空间
BFS 法	O(V+E)	$O(n^2)$	220 ms	420 kb
DFS 法	$O(n^2)$	$O(n^2)$	330 ms	288 kb
枚举法	$O(n^2)$	O(1)	140 ms	288 kb
递归法	$O(n^2)$	$O(n^4)$	140 ms	304 kb
A*算法	O(V+E)(最差)	$O(n^2)$	20 ms	288 kb
双向 BFS 法	O(V+E)	$O(n^2)$	94 ms	1536 kb

6 问题扩展

本问题是 8*8 的棋盘，如果将 8*8 的棋盘条件改为 n*n 的条件（如问题 poj1915）又如何求解，或者把二维空间改为三维的空间（如问题 zoj3467）有如何求解？

这里求解方法同 8*8 的解决方案大致是相同的，只不过对于这类大规模问题我们不能使用 DFS 和递归方法，因为这两种方法要遍历所有情况，存在超时危险，因此，可以采用双向 BFS 方法，或者 A*算法，在附录中给出的是 n*n 的双向 BFS 解决方法源码，而对于三维的情况我们也可以使用双向 BFS，但同样不能使用 DFS 法和递归方法，最后附录未给出源码，但可以通过二维的双向 BFS 改动即可实现。

7 运行结果

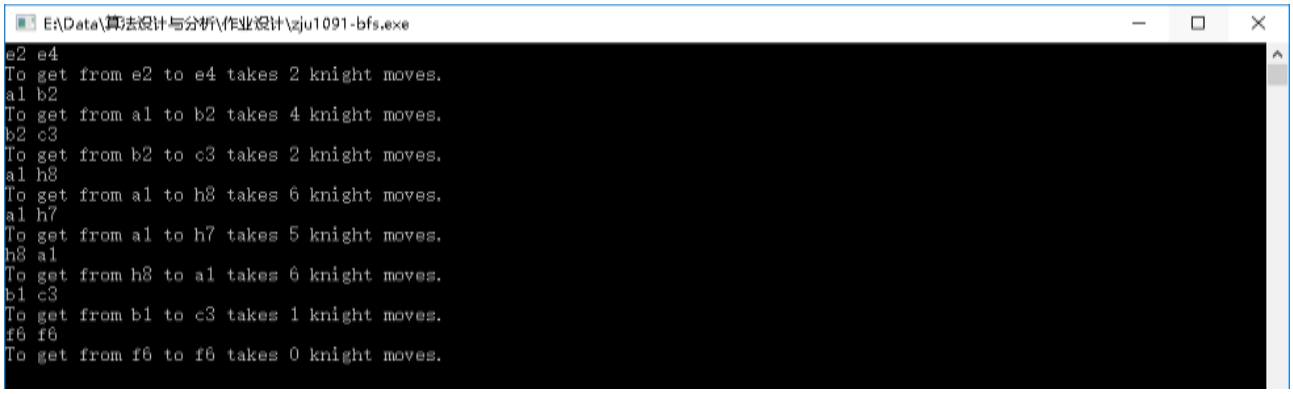

图 1 程序运行截图

Run ID	Submit Time	Judge Status	Problem ID	Language	Run Time(ms)	Run Memory(KB)	User Name
4253092	2018-05-22 22:50:37	Accepted	1091	C++	20	288	WillAlso
4253090	2018-05-22 22:49:56	Accepted	1091	C++	130	288	WillAlso
4253087	2018-05-22 22:49:13	Accepted	1091	C++	330	288	WillAlso
4253085	2018-05-22 22:48:34	Accepted	1091	C++	220	420	WillAlso
4253083	2018-05-22 22:48:01	Accepted	1091	C++	200	420	WillAlso
4253080	2018-05-22 22:47:20	Accepted	1091	C++	140	304	WillAlso
4253078	2018-05-22 22:46:37	Accepted	1091	C++	140	288	WillAlso
4253048	2018-05-22 21:41:43	Accepted	1091	C++	10	288	WillAlso
4253004	2018-05-22 20:38:38	Accepted	1091	C++	160	288	WillAlso

图 2 提交结果截图

说明：这里是在 Zhejiang University Online Judge 提交截图，题目编号为 1091；

总共提交了八种方法：（除了本文介绍的方法，还包括）Dijkstra 算法和 Floyd 算法，状态均为 Acceped，从图中可以看到一个 Run Time 为 20ms 的提交记录，这里即为 A*算法，从这里也体现了 A*算法的高效性，因此，前面的说明 A*算法的结论是正确的。

7 总结

7.1 大作业总结

本次作业主题是对于图的搜索方法的探讨：深度优先搜索方法、广度优先搜索方法、Dijkstra 算法和 Floyd 算法，对于深度优先搜索方法，主要用于穷举所有路径，而对于深度优先搜索方法主要由于求最短路径，这两种方法在本问题中均可以使用，但使用 BFS 要比 DFS 效果更好，但如果换做大规模问题（如针对本问题把 8*8 的棋盘改为 n*n 的规模）如果考虑超时问题，就不能使用 DFS 进行求解，因为要遍历所有情况才能得出结果——最短路径。

同时在考虑优化的时候，我们可以先考虑减小搜索树的深度、考虑情况剪枝，如使用 A*算法的时候，可以不用往不必要的方向扩展，效率大大提升，在使用双向广度优先搜索的时候，减小了搜索树的深度，使指数增长的时间大大减小。

但对于 A*算法使用时，由于刚开始时不能对估值函数进行灵活运用，导致不能得到正确结果，因此经查阅资料得到结论：一般网格情况使用哈曼顿估值，对于坐标系问题使用欧式估值，估值函数的选取是 A*的核心，因此，通过这次实验我学会了如何应用估值函数，但同时也猜想并尝试自己写出一个估值函数，A*算法的趣味和它的效率一样令人着迷。我深深体会到了算法的巧妙性和趣味性。

对于本题目的实际应用：我们可以把它应用到博弈树(棋类)建立、游戏 NPC 寻路等问题，解决了这样的问题，在加上约束条件和游戏规则，可以尝试设计一个象棋程序；同样对于游戏开发 A*算法的效率是毋庸置疑的，因此可以设计一个简单的贪吃蛇自动寻路程序。

7.2 对本门课程的体会及建议

体会：在本次课程中，主要学习了一些基础和常用的算法，如贪心算法、动态规划、穷举法等等，通过学习后认识到了自己的不足，意识到自己的动手能力还是太差，比如解决一个简单的八皇后问题，这个经典的回溯法问题由于没有动手写过代码，导致不看别人的样例代码不能写出合格的程序。但还是有其他收获的：如了解到如何求解最短路径问题，什么时间应用贪心法，什么时间应用穷举法，我认为这是我在这门课程上最大的收获——学会使用算法。我认为这门课程可以和之前修的《数据结构》相辅相成，数据结构注重理论思想讲解，而算法设计与分析注重方法实际应用，两者都是培养一个合格程序员必不可少的。

同时我也体会到自己在算法方面仍有不足，还需要课外多补充其他解决方案，如本问题中使用的 A*算法和双向广度优先搜索都是课本上没有提到或者没有详细讲解的，因此在今后的学习中，我要不仅加强自己的理论知识学习，还要注重全面发展，不能“目光短浅”，毕竟“人外有人，天外有天”，抱着终身学习的态度去做好自己的事。

建议：可以根据实际情况对算法的正确性给出简单证明，或者在讲解算法前引导学生思路以便加深印象。

8 参考文献

[1] 王红梅. 算法设计与分析（第二版）. 北京：清华大学出版社，2013.

[2] 严蔚敏.数据结构（C 语言版）.北京：清华大学出版社，2011.

[3] 俞勇.ACM 国际大学生程序设计竞赛题目与解读.北京：清华大学出版社，2012.

[4] 何克右.从实例中学习 C/C++程序设计.北京：清华大学出版社，2014.

[5] 赵端阳.ACM 国际大学生程序设计竞赛题解.北京：电子工业出版社，2010.

[6] Thomas H.Cormen,Charle E.Leiserson,Ronald L.Rivest Clifford Stein.算法导论.北京：机械工业出版社，2017.

附录（代码）

1.//BFS 解决 8*8 骑士游历问题

```
#include <iostream>
#include <queue>
using namespace std;
 struct point{
    int x,y;
    int c;
 }from,to;

 int main()
 {
    queue<point> q;
    char src[3],dist[3];
    int dx[] = {1,1,2,2,-1,-1,-2,-2};
    int dy[] = {2,-2,1,-1,2,-2,1,-1};
    while(cin >> src >> dist)
    {
        cout << "To get from " << src << " to " << dist;
        while(!q.empty())
        {
            q.pop();
        }
        from.x = src[0] - 'a';
        from.y = src[1] - '1';
        from.c = 0;
        to.x = dist[0] - 'a';
        to.y = dist[1] - '1';
        q.push(from);
        point temp;
```

```
        while(true)
        {
            from = q.front();
            q.pop();
            if(from.x == to.x && from.y == to.y)
            {
                break;
            }
            for(int i = 0;i < 8;i++)
            {
                temp.x = from.x + dx[i];
                temp.y = from.y + dy[i];
                temp.c = from.c + 1;
                if(temp.x < 0|| temp.x > 7 || temp.y < 0 || temp.y > 7)
                {
                    continue;
                }
                q.push(temp);
            }
        }
        cout <<" takes " << from.c << " knight moves." << endl;
    }
    return 0;
 }
```

2.//DFS 解决 8*8 问题

```
#include <iostream>
#include <memory.h>
using namespace std;

int knight[8][8];
int x[] = {1,1,2,2,-1,-1,-2,-2};
```

```
int y[] = {2,-2,1,-1,2,-2,1,-1};

void DFS(int si,int sj,int moves)
{
    if(si < 0 || sj < 0 || si >= 8 || sj >= 8 || moves >= knight[si][sj])
    {
        return;
    }
    knight[si][sj] = moves;
    int i;
    for(i = 0;i < 8;i++)
    {
        DFS(si + x[i],sj + y[i],moves + 1);
    }
}

int main()
{
    char a[10],b[10];
    while(cin >> a >> b)
    {
        memset(knight,10,sizeof(knight));
        DFS(a[0] - 'a',a[1] - '1',0);
        cout << "To get from " << a << " to " << b << " takes " << knight[b[0] -'a'][b[1]
- '1'] << \
        " knight moves." << endl;
    }
    return 0;
}
```

3.//枚举法解决 8*8 问题

```
#include <iostream>
```

```
#include <string>
#include <cmath>
using namespace std;
int main()
{
    string s1,s2;
    int a,b,c,d,x,y,s,m;
    while (cin >> s1 >> s2)
    {
        if((s1=="a1" && s2=="b2") || (s1=="b2" && s2=="a1") || (s1=="g2" && s2=="h1")
|| (s1=="h1" && s2=="g2"))
        {
            cout << "To get from " << s1 << " to " << s2 << " takes 4 knight moves."
<< endl;
            continue;
        }
        if((s1=="a8" && s2=="b7") || (s1=="b7" && s2=="a8") || (s1=="g7" && s2=="h8")
|| (s1=="h8" && s2=="g7"))
        {
            cout << "To get from " << s1 << " to " << s2 << " takes 4 knight moves."
<< endl;
            continue;
        }
        x=s2[0]-s1[0];
        s=9999;
        y=s2[1]-s1[1];
        for(b=-4;b<=4;b++)
        {
            m=y+2*x-2*b+30;
            m%=3;
            for(a=m-6;a<=m+6;a+=3)
            {
```

```
                c=(2*x+y-4*a-5*b)/3;
                d=(5*a+4*b-x-2*y)/3;
                if (s>abs(a)+abs(b)+abs(c)+abs(d))
                    s=abs(a)+abs(b)+abs(c)+abs(d);
            }
        }
        cout << "To get from " << s1 << " to " << s2 << " takes " << s << " knight moves." << endl;
    }
    return 0;
}
```

4.//递归法解决 8*8 问题

```
#include <iostream>
using namespace std;
#define oo 1000000000
int d[8][8][8][8];      /* distance matrix */
int startrow,startcol;  /* starting square */
int destrow,destcol;    /* destination square */
void visit (int row, int col, int moves)
{
  if (row<0 || row>7 || col<0 || col>7 ||
      moves>=d[startrow][startcol][row][col]) return;
  d[startrow][startcol][row][col] = moves;
  visit(row-2,col-1,moves+1);
  visit(row-2,col+1,moves+1);
  visit(row+2,col-1,moves+1);
  visit(row+2,col+1,moves+1);
  visit(row-1,col-2,moves+1);
  visit(row-1,col+2,moves+1);
  visit(row+1,col-2,moves+1);
  visit(row+1,col+2,moves+1);
```

```
}
int main()
{
  char a[3],b[3];
  int i,j,k,l;
  for (i=0; i<8; i++)
    for (j=0; j<8; j++)
      for (k=0; k<8; k++)
   for (l=0; l<8; l++)
     d[i][j][k][l] = oo;
  while (cin >> a >> b)
    {
      startcol = a[0]-'a';
      startrow = a[1]-'1';
      destcol = b[0]-'a';
      destrow = b[1]-'1';
      visit(startrow,startcol,0);
      cout << "To get from " << a << " to " << b << " takes " << d[startrow][startcol][destrow][destcol] << " knight moves." << endl;
    }
  return 0;
}
```

5.//A*算法解决 8*8 问题

```
#include<iostream>
#include<queue>
#include<cstdio>
#include<cmath>
#include<cstdlib>
#include<cstring>
using namespace std;
struct knight
```

```
{
    int x,y,step;
    int g,h,f;
    bool operator < (const knight &k) const
    {
        return f > k.f;
    }
}k;
bool visited[8][8];
int x2,y2,ans;
int dir[8][2] = {{-2,-1},{-2,1},{2,-1},{2,1},{-1,-2},{-1,2},{1,-2},{1,2}};
priority_queue<knight> que;

bool in(const knight &a)
{
    if(a.x < 0|| a.y <0 || a.x >= 8 || a.y >= 8)
    {
        return false;
    }
    return true;
}
int Heuristic(const knight &a)
{
    return (abs(a.x - x2) + abs(a.y - y2)) * 10;
}
void Astar()
{
    knight t,s;
    while(!que.empty()) //如果开启列表不为空
    {
        t = que.top();
```

```
        que.pop();
        visited[t.x][t.y] = true;
        if(t.x == x2 && t.y == y2) //如果到达目的地
        {
            ans = t.step;
            break;
        }
        for(int i=0;i<8;i++)
        {
            s.x = t.x + dir[i][0];
            s.y = t.y + dir[i][1];
            if(in(s) && !visited[s.x][s.y])
            {
                s.g = t.g + 23;
                s.h = Heuristic(s); //曼哈顿估价函数
                s.f = s.g + s.h;
                s.step = t.step + 1;
                que.push(s);
            }
        }
    }
}
int main()
{
    char line[5];
    int x1,y1;
    //freopen("111","r",stdin);
    while(gets(line))
    {
        x1 = line[0] - 'a'; //起点
        y1 = line[1] - '1';
```

```
        x2 = line[3] - 'a';//终点
        y2 = line[4] - '1';
        memset(visited,false,sizeof(visited));
        k.x = x1;
        k.y = y1;
        k.g = k.step = 0;
        k.h = Heuristic(k);
        k.f = k.g + k.h;
        while(!que.empty()) que.pop();
        que.push(k);
        Astar();
        printf("To get from %c%c to %c%c takes %d knight moves.\n",line[0],line[1],line[3],line[4],ans);
    }
    return 0;
}
```

6.//双向 BFS 算法解决 n*n 骑士游历问题

```
#include<iostream>
#include<cstdio>
#include<queue>
#include<cstring>
#define qq 330
using namespace std;
int vis1[qq][qq];      //既标记路径 也统计步数
int vis2[qq][qq];
int fx1[8]={2,2,-2,-2,1,1,-1,-1};
int fx2[8]={1,-1,1,-1,2,-2,2,-2};
struct node {
 int x,y;
}start,end;                //双向 BFS 的两端起点
int sx,sy,ex,ey;
```

```
int m;
bool inside(int xx,int yy)           //判断越界
{
    if(xx>=0&&yy>=0&&xx<m&&yy<m)
        return true;
    else
        return false;
}
void dbfs()  {
    int i,tq,tw;
    queue<node>q,w;              //两个队列
    start.x=sx;start.y=sy;
    end.x=ex;end.y=ey;
    q.push(start);
    w.push(end);
    vis1[sx][sy]=0;          //后面的步数是从0开始加的
    vis2[ex][ey]=0;
    while(!q.empty()&&!w.empty())
    {
        node now,next;
        tq=q.size();
        while(tq--)
        {
          now=q.front();
          q.pop();
          if(inside(now.x,now.y)&&vis2[now.x][now.y]!=-1)
              printf("%d\n",vis1[now.x][now.y]+vis2[now.x][now.y]);
              return;
          }
          for(i=0;i<8;i++)
          {
```

```
        next.x=now.x+fx1[i];
        next.y=now.y+fx2[i];
        if(inside(next.x,next.y)&&vis2[next.x][next.y]!=-1)
        {
            printf("%d\n",vis1[now.x][now.y]+1+vis2[next.x][next.y]);
            return;
        }
        if(inside(next.x,next.y)&&vis1[next.x][next.y]==-1)
        {
            vis1[next.x][next.y]=vis1[now.x][now.y]+1;
            q.push(next);
        }
    }
}
tw=w.size();
while(tw--)          //同上
{
    now=w.front();
    w.pop();
    if(inside(now.x,now.y)&&vis1[now.x][now.y]!=-1)
    {
        printf("%d\n",vis1[now.x][now.y]+vis2[now.x][now.y]);
        return;
    }
    for(i=0;i<8;i++)
    {
        next.x=now.x+fx1[i];
        next.y=now.y+fx2[i];
        if(inside(next.x,next.y)&&vis1[next.x][next.y]!=-1)
        {
            printf("%d\n",vis2[now.x][now.y]+1+vis1[next.x][next.y]);
```

```
                return;
            }
            if(inside(next.x,next.y)&&vis2[next.x][next.y]==-1)
            {
                vis2[next.x][next.y]=vis2[now.x][now.y]+1;
                w.push(next);
            }
        }
    }
}
int main()
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d",&m);
        scanf("%d%d%d%d",&sx,&sy,&ex,&ey);
        memset(vis1,-1,sizeof(vis1));       //标记为未走过
        memset(vis2,-1,sizeof(vis2));
        dbfs();
    }
        return 0;
}
```

《算法设计与分析 B》课程报告成绩评定表

班级：软件 1604 班　　姓名：冯钢果　　学号：0121610870910

序号	评分项目	满分	实得分
1	学习态度、遵守纪律	10	
2	算法设计与分析合理性	30	
3	设计难度、结果正确性	20	
4	代码实现风格、规范性、效率	15	
5	设计报告的规范性	10	
6	设计验收及答辩情况	15	
		总得分	

指导教师签名：

2018 年 6 月 1 日