**武汉大学计算机学院**

**本科生实验报告**

**算法设计环节实验**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：高级语言程序设计实验

指 导 教 师 一：常军 讲师

指 导 教 师 二：谭成予 副教授

学 生 学 号 ：2016301500373

学 生 姓 名 ：徐一恒

二○一八年四月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 徐一恒 日期： 2018/04/08

摘 要

算法设计环节实验的实验目的：练习较为复杂的算法，使用高级程序语言完成给定题目，通过完成这些题目来提高编程技巧，从原来的会用程序语言写程序，编程真正会设计程序、会用算法去优化程序，成为一个真正合格的程序员。

实验设计主要遵循：面向对象程序设计思想和软件工程原理。

实验内容主要包括：1. 编写程序建立一个节点数边数适中的图，实现此图的深度优先遍历。

2. 编写程序实现象棋中马的遍历问题，此问题是非常经典的递归回溯问题，找到一条不重复的遍历路线。在此基础上加入蹩马腿的棋子，再输出一条遍历路线。

实验结论为：1. 深度优先遍历可以很容易地成功实现

2. 马的遍历问题中，需要考虑时间复杂度的问题，需要用贪心算法来进行优化。而当实现高级要求，也就是加入蹩马腿的棋子后，遍历变的更加复杂，而且可能出现根本无法遍历全部的情况，此时一般的贪心算法优化力度也不够了，需要用其他算法来优化。除此之外，马的遍历问题还是可以用回溯和贪心算法很好地实现的。

**关键词：图论；深度优先；遍历；回溯；贪心算法**

**目 录**

**1** **题目一：图的深度优先遍历**

1.1 实验题目及要求说明…………………………………………………………1

1.2 开发环境及开发日期…………………………………………………………1

1.3 代码结构………………………………………………………………………1

1.4 数据结构………………………………………………………………………2

1.5 主要函数的功能描述…………………………………………………………2

1.6 主要算法描述…………………………………………………………………3

1.7 运行效果………………………………………………………………………3

1.8 问题分析与总结………………………………………………………………4

**2**  **题目二：象棋中马的遍历**

2.1 实验题目及要求说明…………………………………………………………5

2.2 开发环境及开发日期…………………………………………………………5

2.3 代码结构………………………………………………………………………5

2.4 数据结构………………………………………………………………………6

2.5 主要函数的功能描述…………………………………………………………7

2.6 主要算法描述…………………………………………………………………8

2.7 运行效果………………………………………………………………………10

2.8 问题分析与总结………………………………………………………………12

**结论** …………………………………………………………………………………13

**参考文献** ……………………………………………………………………………13

**附录** …………………………………………………………………………………14

**1 题目一：图的深度优先遍历**

**1.1** 实验题目及要求说明

给定某个图的结构数据和起点位置，创建出图后，再采用深度优先遍历算法，输出所有的遍历节点序列。

**基本要求**

1. 自己定义一组图的结构数据，节点数大于7个，边数大于15个，用于创建此图
2. 对图中各节点进行深度优先遍历
3. 遍历过程中的节点序列要求输出显示

**1.2** 开发环境及开发日期

环境：IntelliJ

日期：2018/4/9

**1.3** 代码结构

采用面向对象程序设计，使用Java作为程序语言。整个程序由一个Java文件构成，定义了两个类：Graph和Vertex。

Graph类是程序的主要部分，将整个图封装成一个类，包含了图的主要成分：节点和边。所有的函数包括深度优先遍历算法的函数都是写在这个类里的。

Vertex类是定义节点的类，储存节点的内容和是否被访问过。

**1.4** 数据结构

节点用定义的Vertex类来表示，其中存储该节点的相关信息，包括节点序号、内容、是否访问过等等，在图中所有节点用一个数组来储存。

边的信息通过邻接矩阵存储，邻接矩阵用一个二维数组表示，每个单元表示这两个节点间是否有边连接，若没有则为0，若有，则该数值表示这条边的权值。

由此，Vertex数组表示的节点和邻接矩阵表示的边构成了这个图，在添加和删除节点或边的时候，只需要在数组和邻接矩阵中做相应操作即可。

**1.5** 主要函数的功能描述

写在前面：关于函数，我在代码中严格按照Java注释的格式写了详细的注释，包括函数的作用，参数的意义，返回值的意义。因此，在此只取其中重要的比较复杂的函数做介绍。

* + 1. **void DFS(int current)**

从一个给定的节点开始，对这个图进行深度优先遍历。首先检测这个节点是否已经被访问过了，如果访问过，则直接返回。若没有访问过，则访问该节点，输出其标号，并将其标记为已访问。之后对该节点的临节点进行递归遍历。直到所有能访问的节点都遍历到了，遍历结束，函数返回。

参数current：进行遍历的开始节点。

* + 1. **void DFS()**

重载函数，表示对整个图进行深度优先遍历。考虑到某些图可能是非连通的，或者一些有向图但从一个节点开始遍历无法遍历整个图，因此编写这个函数保证整个图的节点都可以被遍历。函数内部就是依次对每个节点调用上一个DFS函数。

* + 1. **int nextNeighbor(int i)**

获取i号节点的下一个邻节点。通过搜索邻接矩阵的第i行，来找到下一个将要被访问的绩点

参数i：当前节点序号

返回值：其下一个邻节点的序号

**1.6** 主要算法描述

深度优先遍历算法非常简单直接，是非常基础的数据结构算法：从一个点开始，首先访问它自己并标为已访问，然后依次对其没有访问过的邻节点递归调用深度优先遍历函数，直到所有节点都被访问过。

**1.7** 运行效果

整个图为一个有10个节点，16条边的一个无向连通图，图如下所示:

0

2

5

3

6

8

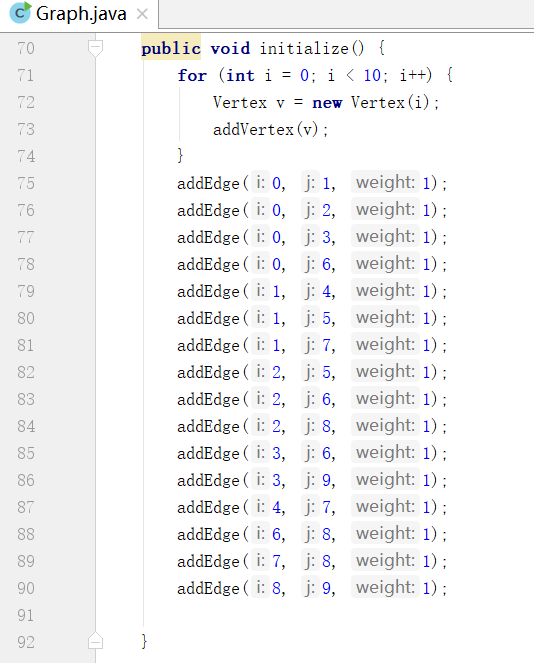
9

1

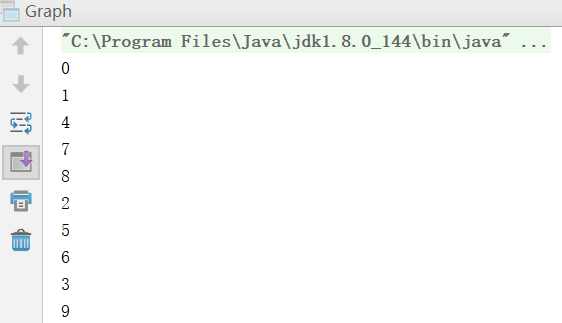
7

4

初始化如下所示：



深度遍历结果：



**1.8** 问题总结与分析

相对而言是一个简单的题目，考察对图这种数据结构的基础理解，需要用合适的方法来构造一个图，深度遍历只需用到简单的递归就可以实现。

**2 题目二：象棋中马的遍历**

**2.1** 实验题目及要求说明

在N\*N棋盘上，任意一个位置放置一个棋子马，要能选择一套合适的移动路线，按象棋中“马走日”的移动规则不重复地遍历棋盘上每一个位置点。扩展任务，在棋盘上设置一个或多个憋马腿的棋子，找到一套合适的不重复遍历路线。

**基本要求：**

1. 起始位置坐标由用户输入任意指定，然后依次输出所遍历的每个位置坐标。
2. 开始先不必考虑象棋中“蹩马腿”的规则，实现基本遍历功能。
3. 棋谱的大小N（N>5）自行设定，注意事先分析算法的时间复杂度。
4. 有兴趣的同学可在棋盘上设置一个或多个憋马腿的棋子，尝试遍历过程如何完善。

**2.2** 开发环境及开发日期

环境：IntelliJ

日期：2018/4/20

**2.3** 代码结构

采用面向对象程序设计，使用Java作为程序语言。整个程序由两个Java文件构成，定义了两个类：Move和ChessHorse。

Move表示马的一次移动，由于马的移动方式较为特殊，一次可以有8种方向，而且考虑到后来要加入蹩马腿的棋子，蹩马腿的棋位也需要计算，因此我将马的一次移动抽象成一个类，属性有起点和终点坐标、蹩马腿的那个点的坐标。其中包括8种情况，根据每个情况生成其起止坐标，也生成其被蹩马腿的棋位坐标。这样，在寻找遍历时会方便很多。同时，也在这个类中编写了toString函数来输出一个move。

ChessHorse类是程序的主要部分，用来表示象棋棋盘和其上马的移动，通过回溯和贪心算法来找到一条不重复的马的遍历路径。包含一个二维数组board属性，用来表示象棋棋盘；一个堆栈visited属性，用来保存所有已经走过的move，为回溯和最后输出路径做准备。其中编写了若干函数，共同实现马的遍历

**2.4** 数据结构

象棋棋盘用二维数组表示，每个点表示一个象棋格子，走过为1，没走过为0。马的当前位置用一组整数表示，每走一步更新一次。当前路径通过一个堆栈来表示，其中按时间顺序储存当前所有走过的move。

移动用Move这个类来表示，每次移动的相关信息储存在这个类中。属性有起点和终点坐标、蹩马腿的那个点的坐标。其中包括8种情况，根据每个情况生成其起止坐标，也生成其被蹩马腿的棋位坐标。

整个象棋棋盘的遍历用ChessHorse这个类来表示，

**2.5** 主要函数的功能描述

写在前面：关于函数，我在代码中严格按照Java注释的格式写了详细的注释，包括函数的作用，参数的意义，返回值的意义。因此，在此只取其中重要的比较复杂的函数做介绍。

蹩马腿的情况和普通情况我是统一考虑的，因此一下函数均同时适用于普通要求和扩展要求。

**1.5.1 boolean isLegalMove(Move move)**

判断一个move在当前棋盘状态下是否是合法的。输入的move是按照顺序生成的当前位置开始的8种move，首先判断这个move是否超出棋盘边界，再判断其落子点是否已经走过或被蹩马腿的棋子占据，最后判断这个move本身有没有被蹩马腿，都没问题则返回true。

参数move：即将要被执行的move

返回值：是否合法

**1.5.2 Stack<Move> getAllLegalMoves()**

获得从当前位置开始所有合法的move，最多是8个。在8种可能的move循环判断，将所有合法的move存到堆栈中返回出去。

返回值：包含当前情况所有可能的move的堆栈

**1.5.3 void undo()**

为了实现遍历，采用递归回溯算法，因此需要一个回溯用的函数。此函数用于回溯时撤销上一步棋，也就是上一个移动，将棋盘还原成之前的棋盘，马回到原位，把上一步棋从走过的棋的stack中拿走，搜索深度减一。

makeMove()和这个很像，一个是走一步棋，一个是撤销，操作就是反着的，不再赘述。

**1.5.4 int getMoveNum()**

为了优化遍历算法，减少运行时间，采用贪心算法，此函数为贪心算法做准备。此函数用于求当前棋盘状态下可以走的移动个数，也就是出口数，将这个出口数传递给给下面的getBestMove函数。具体实现方法是先执行这一个move，然后获取出口数，再调用上面的undo函数undo回去。

返回值：当前可执行的合法move数，也就是出口数。

**1.5.5 Move getBestMove(Stack<Move> moves)**

为了优化遍历算法，减少运行时间，采用贪心算法，此函数是贪心算法中的核心函数，旨在找出所有剩余合法移动中的“最佳”的那个move，最佳的move根据贪心算法，应当是执行这个move之后，出口最少的那个move，也就是跳完之后再一次跳动的move数最少的那个。对堆栈中的所有move依次计算其移动后的出口数，最终选取出口最少的那个返回出去。

参数moves：所有剩余的合法move

返回值：这些move中出口数最少的那个move。

**1.5.6 void traversal()**

执行不重复的遍历的寻找，输出一条可以不重复遍历整个象棋棋盘的路径，包括带蹩马腿棋子的情况。从指定起点开始执行，先获取当前情况所有的合法的move，放到一个stack中，只要这个stack非空，就循环继续执行，从这些move中选出出口最少的move，执行这个move，搜索深度加一，改变棋盘状态，将move存到走过的moves里面去。然后对之后的棋盘递归调用traversal()。如果发现此路不通，则回溯，调用undo()，回到执行这步棋之前，并将这一个move从之前的所有合法move的stack中移除，再挑出剩下move中最好的那个，执行traversal。直到有一条路走通为止。

**2.6** 主要算法描述

蹩马腿的情况和普通情况我是统一考虑的，因此以下算法均同时适用于普通要求和扩展要求。主要包括两个大算法点：

**1.6.1 完成遍历的回溯算法：**

此问题是非常典型的回溯问题，基本思想是通过深度优先遍历来尝试是否可以不重复地遍历整个棋盘，当无路可走且未完全遍历时，则回溯到上一层搜索，执行另一个可能的走法，直到找到一条符合要求的路线。具体算法跟写在traversal()函数中的算法是一样的，在此不再重复书写。

**1.6.2 对遍历进行优化的贪心算法：**

有了基本的回溯算法之后，肯定是可以找到一条符合要求的遍历路线，但由于时间复杂度的问题，当棋盘变大后，运行时间大大变长，尤其是加入蹩马腿的棋子后，时间长到无法接受。因此，需要用到一些算法来进行优化。这里的贪心算法就是对时间的优化。具体思想为每次移动时都找移动后出口最少的那个move来移动，因为这样做可以使出口少的节点越来越少，就越不容易出现死路，成功的几率会大很多，有效地提高了运行效率。

贪心算法的基本实现就是在每次移动前，计算所有可能移动之后的出口数，找出出口数最少的那个move来执行，调用getMoveNum和getBestMove来实现这个算法，具体的算法内容在函数部分的以上两个函数中有详细的说明，不再重复。

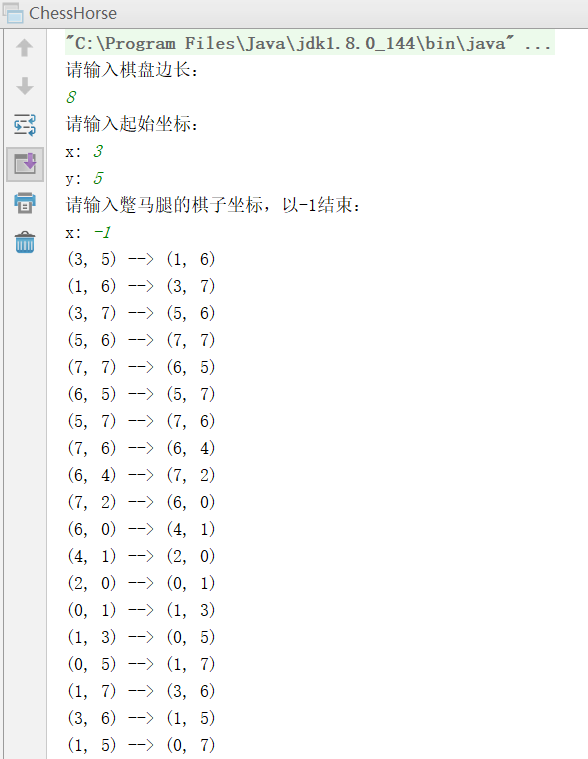
可以发现，当使用了贪心算法之后，效率显著提高，正常的8\*8棋盘可以极快地求出遍历路径。带有蹩马腿棋子的情况的效率也有了提高，不过仍然存在一定的时间效率问题。

**2.7** 运行效果

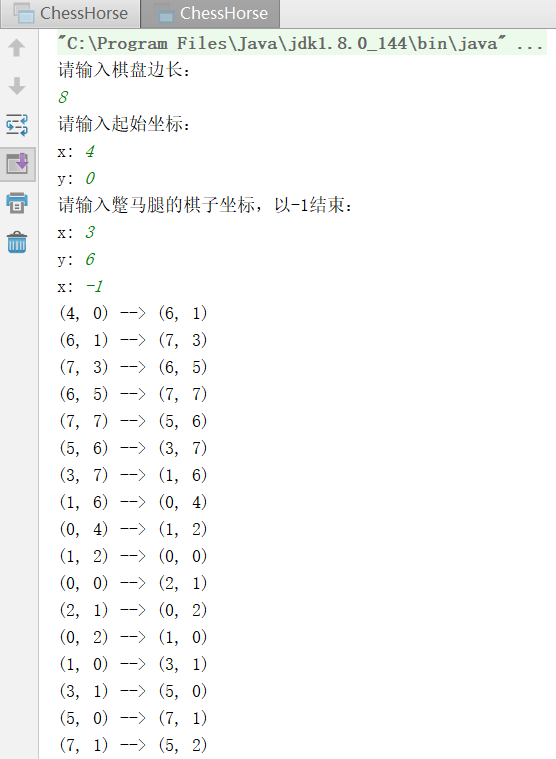
首先输出提示语让用户输入相关信息：棋盘大小、起始位置、所有蹩马腿的棋子的坐标，以-1结束。

然后程序根据用户的输入信息，输出一条可以完全遍历的路径。

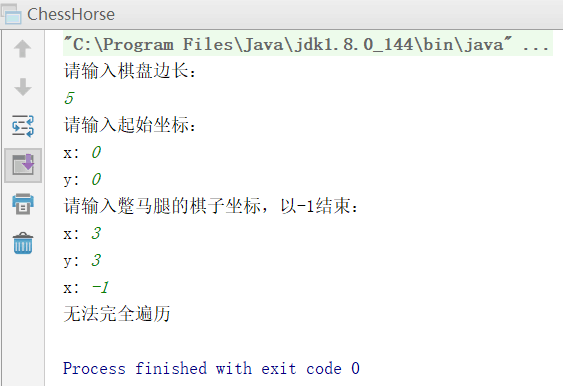
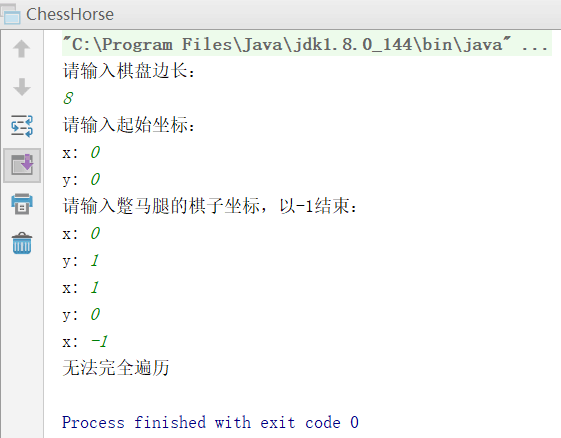
8\*8不带蹩马腿的情况，任何起点开始都可以极快得到结果：



带蹩马腿的情况，有的快有的慢，取决于棋子的位置：



同时，带蹩马腿的某些情况是会出现无法完全遍历的情况的，读者可以结合下面的例子，自己思考一下为什么会有这种情况。此时，输出无法完全遍历：



**2.8** 问题总结与分析

基本的马的遍历问题可以用回溯算法实现，这并不难，最大的难点在于如何对算法进行优化，使得运行效率提高到人可以接受的速度。因此引入了贪心算法，成功地做出了优化，提高了效率。

同时，我也实现了加入蹩马腿的棋子的扩展要求，其算法其实不需要做多大变化，只需要在判断是否合法时多考虑一点即可。其带来的最大的问题就是运行时间大大边长，这也是引入贪心算法的重要原因。

总的来说同时成功实现了题目的基本要求和扩展要求。

结 论

达到了算法设计实验环节的要求和目的，提高了自己的编程技巧，巩固了数据结构中的算法。学会了如何使用更好的算法来优化复杂的程序。成功完成了此环节的所有任务。

参考文献

[1] Kathy Sierra. Head First Java，2007

附 录

题目一核心代码：

*/\*\* Initialize the graph.\*/***public void** initialize() {  
 **for** (**int** i = 0; i < 10; i++) {  
 Vertex v = **new** Vertex(i);  
 addVertex(v);  
 }  
 addEdge(0, 1, 1);  
 addEdge(0, 2, 1);  
 addEdge(0, 3, 1);  
 addEdge(0, 6, 1);  
 addEdge(1, 4, 1);  
 addEdge(1, 5, 1);  
 addEdge(1, 7, 1);  
 addEdge(2, 5, 1);  
 addEdge(2, 6, 1);  
 addEdge(2, 8, 1);  
 addEdge(3, 6, 1);  
 addEdge(3, 9, 1);  
 addEdge(4, 7, 1);  
 addEdge(6, 8, 1);  
 addEdge(7, 8, 1);  
 addEdge(8, 9, 1);  
  
}  
  
*/\*\* Get the next neighbour vertex that is not visited.  
 \** ***@param i*** *the current vertex.  
 \** ***@return*** *the next neighbour vertex that is not visited.\*/***public int** nextNeighbour(**int** i) {  
 **for** (**int** j = 0; j < **vertexNum**; j++) {  
 **if** (**adjacencyMatrix**[i][j] != 0 && !**vertexList**.get(j).**isVisited**) {  
 **return** j;  
 }  
 }  
 **return** -1;  
}  
  
*/\*\* Set the given vertex visited and print its label.  
 \** ***@param i*** *the vertex to be visit. \*/***public void** visit(**int** i) {  
 Vertex v = **vertexList**.get(i);  
 v.**isVisited** = **true**;  
 System.***out***.println(v.**label**);  
}  
  
*/\*\* Execute the depth first searching. \*/***public void** DFS() {  
 **for** (**int** i = 0; i < **vertexNum**; i++) {  
 DFS(i);  
 }  
}  
  
*/\*\* Perform DFS from the current vertex.  
 \** ***@param current*** *the current vertex.\*/***public void** DFS(**int** current) {  
 **if** (**vertexList**.get(current).**isVisited**) {  
 **return**;  
 }  
 visit(current);  
 **int** next = nextNeighbour(current);  
 **while** (next != -1) {  
 DFS(next);  
 next = nextNeighbour(current);  
 }  
}  
  
*/\*\* Vertex. \*/***class** Vertex {  
 */\*\* Indicate if the vertex has been visited. \*/* **boolean isVisited**;  
 */\*\* Content of the vertex. \*/* **int label**;  
  
 */\*\* Constructor of Vertex Class.  
 \** ***@param label*** *label of the vertex. \*/* **public** Vertex(**int** label) {  
 **isVisited** = **false**;  
 **this**.**label** = label;  
 }  
}

题目二核心代码：

*/\*\* Check if the given move is a legal move on current board.  
 \** ***@param move*** *the move to be checked. \*/***public boolean** isLegalMove(Move move) {  
 **if** (move.**x1** < 0 || move.**x1** > **\_size** - 1 || move.**y1** < 0 || move.**y1** > **\_size** - 1) {  
 **return false**;  
 }  
 **if** (**\_board**[move.**x1**][move.**y1**] == 1 || **\_board**[move.**x1**][move.**y1**] == 2) {  
 **return false**;  
 }  
 **if** (**\_board**[move.**blockX**][move.**blockY**] == 2) {  
 **return false**;  
 }  
 **return true**;  
}

*/\*\* Undo a move. Reset changes made by this move,  
 \* including current position, visited points and so on. \*/*

**public void** undo() {  
 **\_board**[**\_currentX**][**\_currentY**] = 0;  
 Move move = **\_visited**.pop();  
 **\_currentX** = move.**x0**;  
 **\_currentY** = move.**y0**;  
 **\_num** -= 1;  
}  
  
*/\*\* Actually make a move. Set changes to be made by it,  
 \* including current position, visited points and so on.  
 \** ***@param move*** *the move to be made.\*/***public void** makeMove(Move move) {  
 **\_currentX** = move.**x1**;  
 **\_currentY** = move.**y1**;  
 **\_board**[**\_currentX**][**\_currentY**] = 1;  
 **\_num** += 1;  
 **\_visited**.push(move);  
}  
  
*/\*\* Get the number of all legal moves on current status.  
 \** ***@return*** *the number of all legal moves on current status. \*/***public int** getMoveNum() {  
 **int** num = 0;  
 **for** (**int** i = 0; i < 8; i++) {  
 Move move = getMove(i);  
 **if** (isLegalMove(move)) {  
 num += 1;  
 }  
 }  
 **return** num;  
}  
  
*/\*\* Prepare to use the Greedy Algorithm to optimize the program.  
 \* Get the move that has the lest legal moves after making it.  
 \** ***@param moves*** *The set of all current legal moves.  
 \** ***@return*** *the move having the lest legal moves after making it.\*/***public** Move getBestMove(Stack<Move> moves) {  
 Move result = **null**;  
 **int** num = 9;  
 **for** (Move move : moves) {  
 makeMove(move);  
 **int** temp = getMoveNum();  
 undo();  
 **if** (temp < num) {  
 num = temp;  
 result = move;  
 }  
 }  
 **return** result;  
}  
  
*/\*\* Get all legal moves of the current status.  
 \** ***@return*** *The set of all current legal moves. \*/***public** Stack<Move> getAllLegalMoves() {  
 Stack<Move> result = **new** Stack<Move>();  
 **for** (**int** i = 0; i < 8; i++) {  
 Move move = getMove(i);  
 **if** (isLegalMove(move)) {  
 result.push(move);  
 }  
 }  
 **return** result;  
}  
  
*/\*\* Execute the traversal. Get the sequence made by the horse from given beginning  
 \* point to cover all squares on the board without repeating steps. \*/***public void** traversal() {  
 **if** (isOver()) {  
 **return**;  
 }  
 Stack<Move> moves = getAllLegalMoves();  
 **while** (!moves.isEmpty()) {  
 Move move = getBestMove(moves);  
 makeMove(move);  
 traversal();  
 **if** (isOver()) {  
 **return**;  
 }  
 undo();  
 moves.remove(move);  
 }  
  
 **return**;  
}

教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）