PROGRAMMING ASSIGNMENT

Knight's Tour Problem

Zhihao Cao

Spring 2014

CSCI 36200

Index

Description

As the project requirement says, the overall objective is to get familiar with the use of several data structures. To solve "Knight's Tour" problem, arrays and stack are typically utilized. Besides, this project also helps to improve the ability to research an unknown problem and to understand difficult algorithm.

-Algorithm

The two algorithms to be used in this project are Warnsdorff's rule and Back-Tracking algorithm. Warnsdorff's rule is a heuristic for finding a knight's tour. We move the knight so that we always proceed to the square from which the knight will have the fewest onward moves. In this way, we can find the knight's tour without revisiting any square already visited. The Back-Tracking algorithm is actually a brute force method. It will go through every possible move to find the knight's tour. When there is no further move to go, it will turn back to the previous square and try another available move.

-Order of magnitude

Warnsdorff's rule:

According to the size of the chess board, the total number of square is N.

In each possible one of eight next moves, it will check the available move after it. Therefore, the running time is 8*8N. The order of magnitude is O (N).

Back-Tracking:

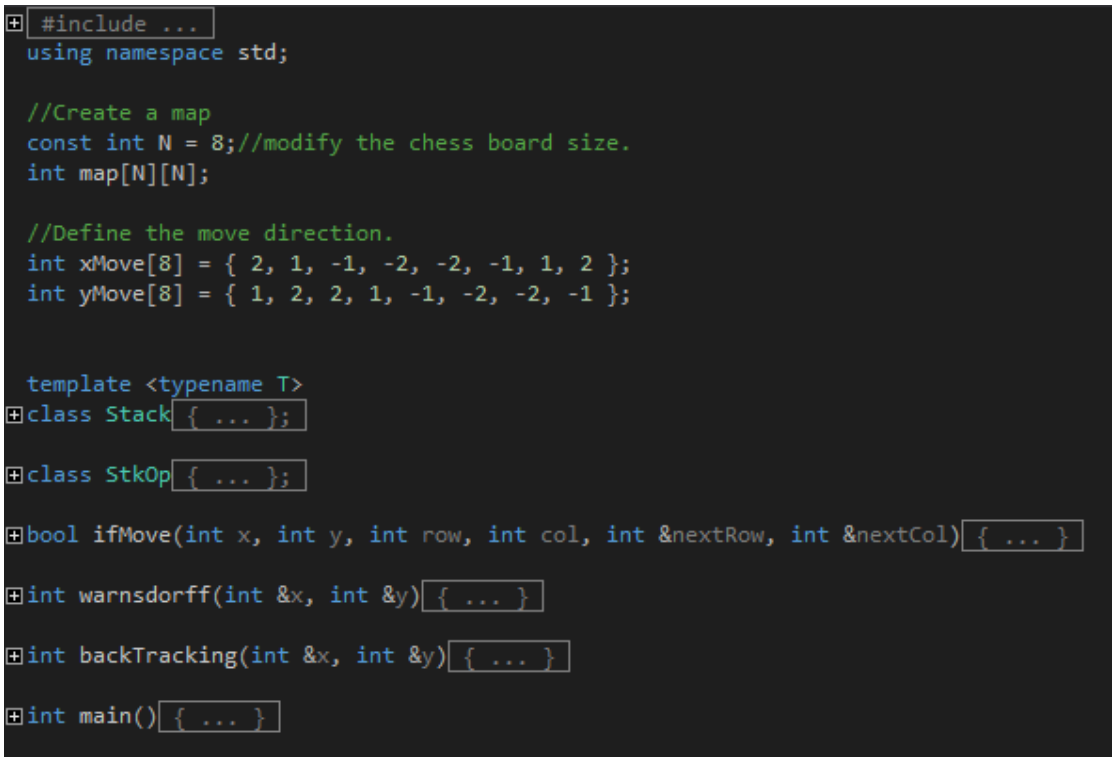
There are approximately 4×10^{51} possible move sequences. The running time varies between the direction and initial position. It just can be extremely large.

-Environment

This is a C++ program written through Visual Studio 2013.

Source Code

The overall structure:




```
//Author: Zhihao Cao
//Modify date: 2014/2/24
//Subject: Solving Knight's tour problem
//Course: CSCI-36200

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <ctime>
using namespace std;

//Create a map
const int N = 8;//modify the chess board size.
int map[N][N];

//Define the move direction.
int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };
```

```
template <typename T>
class Stack {
    T *sa;
    int sp;
public:
    Stack() {
        sp = -1;
        sa = new T [100];
    }


    void push(T i){
        if (isFull()){
            throw std::exception("Stack Overflow!");
            return;
        }
        sa[++sp] = i;
    }

    T pop() {
        if (isEmpty())
            throw std::exception("Stack Underflow!");
        return sa[sp--];
    }

    bool isFull(){
        if (sp < 99)
            return false;
        return true;
    }

    bool isEmpty(){
        if (sp == -1)
            return true;
        return false;
    }

    T getTop() {
        if (isEmpty())
            throw std::exception("Stack Underflow!");
        return sa[sp];
    }
```

```
};

class StkOp
{
    int row, col, dir;
public:
    StkOp()
    {
        set(0, 0, 0);
    }

    StkOp(int r, int c, int d)
    {
        set(r, c, d);
    }

    void set(int r, int c, int d)
    {
        row = r;
        col = c;
        dir = d;
    }

    void get(int& r, int & c, int & d)
    {
        r = row;
        c = col;
        d = dir;
    }

};

bool ifMove(int x, int y, int row, int col, int &nextRow, int &nextCol){
    //if (row + x >= 0 && row + x < 8 && col + y >= 0 && col + y < 8){
    if (row + y >= 0 && row + y < N && col + x >= 0 && col + x < N && map[row
+ y][col + x] == 0){
        //cout << "Current position: (" << row << ", " << col << "),  move
direction:(" << x  << ", " << y << ")" << endl;
        nextRow = row + y;
        nextCol = col + x;
        //cout << "   Next position: (" << nextRow << ", " << nextCol <<
")" << endl;
        return true;
    }
```

```
    //cout << "No further move, need trace back" << endl;
    return false;

}

int warnsdorff(int &x, int &y){
    //The position of next move.
    int nextX[8] = { 0 };
    int nextY[8] = { 0 };
    //Save the number of possible moves for the next move.
    int exist[8] = { 0 };
    int k, m, l;
    int tmpX, tmpY;
    int count, min, tmp;
    //First move of the night.
    map[x][y] = 1;

    //Start the main loop of Warnsdorff's rule.
    for (m = 2; m <= ((N*N) / 2); m++) {
    //for (m = 2; m <= (N*N); m++) {
        //Initialize the next available move counter && the array for the
number of possible moves for the next move.
        for (l = 0; l < 8; l++)
            exist[l] = 0;
        l = 0;

        //Check next move from current position.
        for (k = 0; k < 8; k++){
            tmpX = x + xMove[k];
            tmpY = y + yMove[k];
            //check if next move is valid.
            if (tmpX >= 0 && tmpX < N && tmpY >= 0 && tmpY < N && map[tmpX][tmpY]
== 0){
                //save the next available move position in the array.
                nextX[l] = tmpX;
                nextY[l] = tmpY;
                //Increase the number of next available move.
                l++;
            }
        }

        count = l; //create a counter for l(the next available move
counter);
        //if there is no next move.
```

```
        if (count == 0){
            return 0;
        }
        //if there is only one next move, we direclty go to this move.
        else if (count == 1){
            min = 0;
        }
        //if there are more than 1 available moves.
        else{
            //Check the next available moves of the next move.
            for (l = 0; l < count; l++){
                for (k = 0; k < 8; k++){
                    tmpX = nextX[l] + xMove[k];
                    tmpY = nextY[l] + yMove[k];
                    if (tmpX >= 0 && tmpX < N && tmpY >= 0 && tmpY < N &&
map[tmpX][tmpY] == 0)
                        exist[l]++;
                }//end loop k
            }//end loop l

            //look for the minimun for exist[].
            tmp = exist[0];
            min = 0;
            for (l = 1; l < count; l++){
                if (exist[l] < tmp){
                    tmp = exist[l];
                    min = l;
                }
            }
        }//end else

        //Move to the least possible moves of next move.
        x = nextX[min];
        y = nextY[min];
        map[x][y] = m;
    }
    return 1;
}

int backTracking(int &x, int &y){
    //Instantiate the stack and push the first stack
    int row;
    int col;
    int counter = (N*N) / 2;
```

```
    int direction = 0;
    Stack<StkOp> myStack;
    row = x;
    col = y;
    StkOp stkT(row, col, 0);
    myStack.push(stkT);

    //the main loop of the algorithm
    int nextRow = 0, nextCol = 0;

    //When there is any empty block.
    while (counter <= (N*N)-1 && !myStack.isEmpty()){
        stkT = myStack.getTop();
        stkT.get(row, col, direction);

        //check if the next move is valid. If not, changes direction.
        while (direction < 8 && !ifMove(xMove[direction], yMove[direction],
row, col, nextRow, nextCol)){
            direction++;
        }
        //Push next move into stack
        if (direction != 8){
            myStack.pop();
            stkT.set(row, col, direction + 1);// When the pointer turns back,
it poting to next direction.
            myStack.push(stkT);
            stkT.set(nextRow, nextCol, 0);//Push next move into stack, the
direction should begin from 0.
            myStack.push(stkT);

            map[nextRow][nextCol] = ++counter;
            //print counter and position
            cout << "Counter: " << counter << ", position: " << "(" << nextCol
+ 1 << ", " << nextRow + 1 << ")" << endl << endl;
        }
        //No further move to go, pop the stack.
        else{
            myStack.pop();
            map[row][col] = 0;
            counter--;
        }
    }
    return 0;
```

```
}

int main() {

    //Input the initial position.
    int initX, initY;
    cout << "Please enter a valid initial row number(start from 1):\n";
    cin >> initX;
    cout << "Then, enter a valid initial column number(start from 1):\n" ;
    cin >> initY;
    cout << "Your initial position is: (" << initX << "," << initY << ")"
<< endl << endl;
    initX--;
    initY--;

    //initialize the chess board.
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            map[x][y] = 0;

    //Start the Warnsdorff's rule.
    warnsdorff(initX, initY);
    //Start the back-tracking's rule.
    backTracking(initX, initY);

    //Print the chess board.
    ofstream myfile;
    myfile.open("a.out");


    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++){
            myfile << map[x][y] << "     ";
            cout << map[x][y] << "     ";
        }
        myfile << endl << endl;
        cout << endl << endl;
    }
    myfile.close();

    cin.ignore();
    cin.get();
    return 0;
}
```

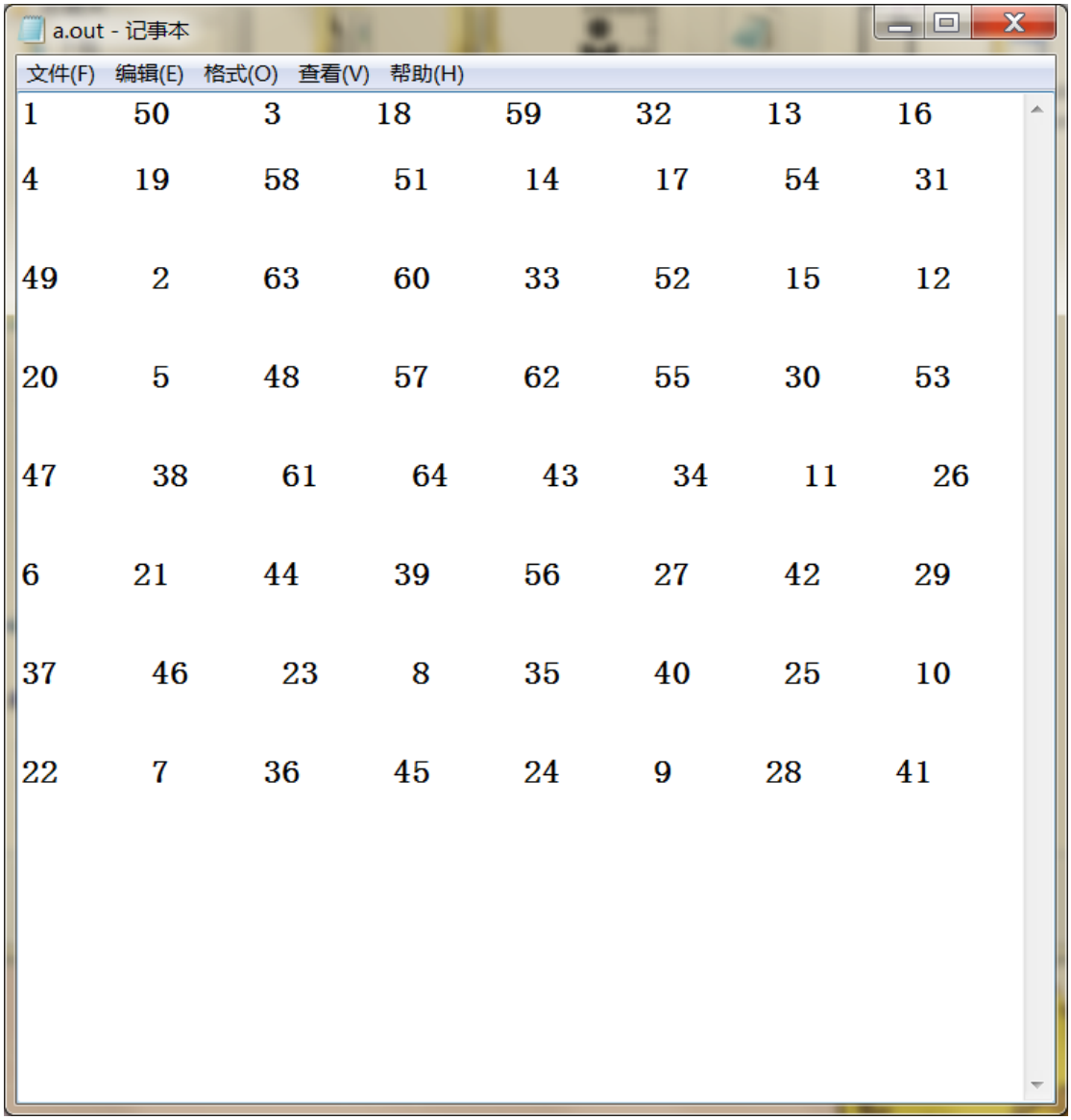
Output

User should input the initial position to start the program.
The first 32 steps using Warnsdorff's rule will not generate log. The log is generated by Back-tracking algorithm.

```
Please enter a valid initial row number(start from 1):
1
Then, enter a valid initial column number(start from 1):
1
Your initial position is: (1,1)

Counter: 33, position: (5, 3)

Counter: 34, position: (6, 5)

Counter: 35, position: (5, 7)

Counter: 36, position: (3, 8)

Counter: 37, position: (1, 7)

Counter: 38, position: (2, 5)

Counter: 39, position: (4, 6)

Counter: 40, position: (6, 7)

Counter: 41, position: (8, 8)

Counter: 42, position: (7, 6)

Counter: 43, position: (5, 5)

Counter: 44, position: (3, 6)

Counter: 45, position: (4, 8)

Counter: 46, position: (2, 7)

Counter: 47, position: (1, 5)

Counter: 48, position: (3, 4)

Counter: 49, position: (1, 3)

Counter: 50, position: (2, 1)

Counter: 51, position: (4, 2)
```

```
Counter: 52, position: (6, 3)

Counter: 53, position: (8, 4)

Counter: 54, position: (7, 2)

Counter: 55, position: (6, 4)

Counter: 56, position: (5, 6)

Counter: 57, position: (3, 5)

Counter: 58, position: (4, 3)

Counter: 59, position: (5, 1)

Counter: 60, position: (3, 2)

Counter: 61, position: (4, 4)

Counter: 58, position: (5, 4)

Counter: 59, position: (3, 3)

Counter: 60, position: (4, 5)

Counter: 57, position: (4, 4)

Counter: 58, position: (3, 2)

Counter: 59, position: (5, 1)

Counter: 60, position: (4, 3)

Counter: 61, position: (3, 5)

Counter: 62, position: (5, 4)

Counter: 63, position: (3, 3)

Counter: 64, position: (4, 5)

1     50     3     18     59     32     13     16

4     19     58     51     14     17     54     31

49     2     63     60     33     52     15     12

20     5     48     57     62     55     30     53

47     38     61     64     43     34     11     26

6     21     44     39     56     27     42     29

37     46     23     8     35     40     25     10

22     7     36     45     24     9     28     41
```

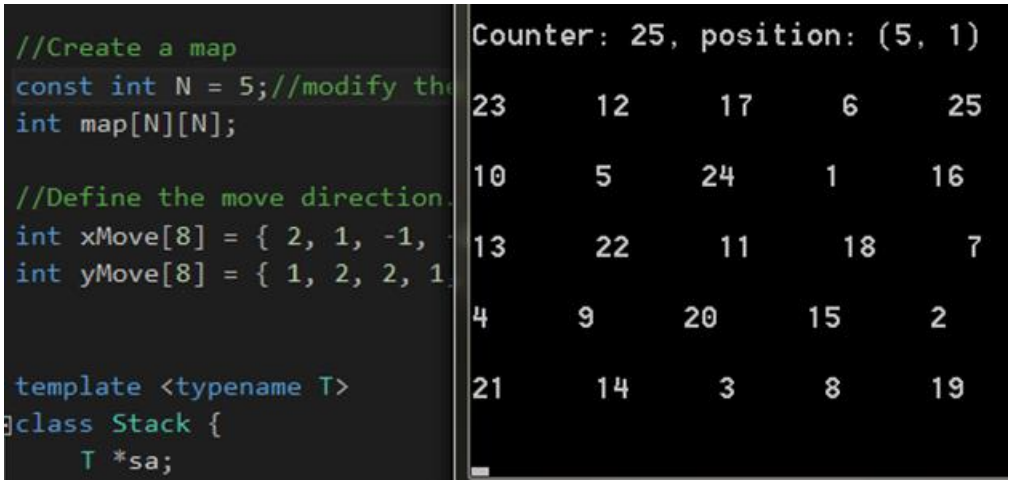
Output file:


a.out - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
1 50 3 18 59 32 13 16
4 19 58 51 14 17 54 31
49 2 63 60 33 52 15 12
20 5 48 57 62 55 30 53
47 38 61 64 43 34 11 26
6 21 44 39 56 27 42 29
37 46 23 8 35 40 25 10
22 7 36 45 24 9 28 41

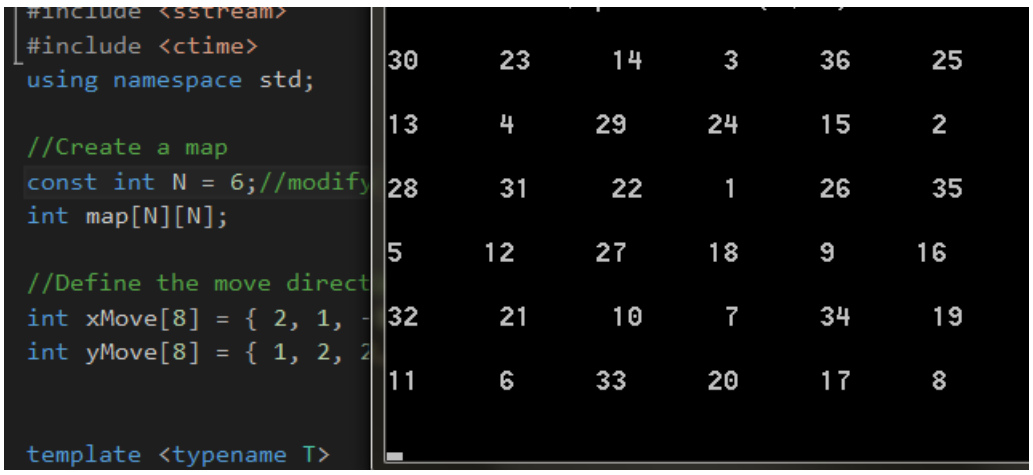
Problems & Conclusion

The Warnsdoff's rule works perfect. If the program only runs in this algorithm, the solution will be found just in an instant. However, when the Back-Tracking algorithm is added to solve the next half steps, the program will keep running for a long time until the final solution is found. It may take hours to finish or more. It looks like in an infinity loop, but actually it works no problem at all. This Back-Tracking algorithm has been checked to find a solution for smaller size chess board like 5*5.

Here is the screenshot of the result for 5*5 chess board using Warnsdoff's rule and Back-Tracking algorithm (Random initial position):

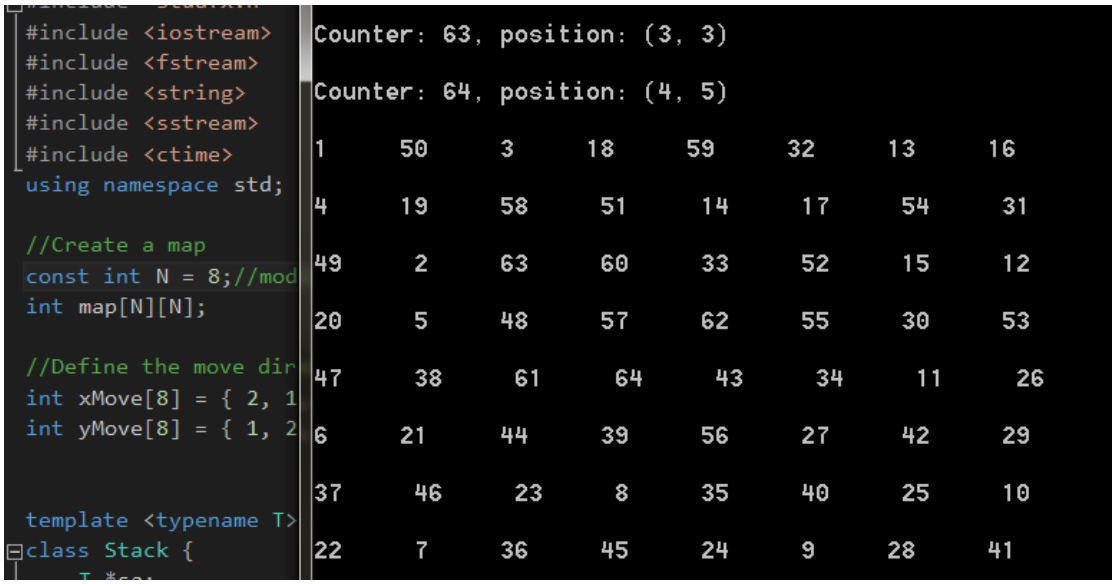


Even for the 6*6 chess board, it still can find the solution in seconds(Random initial position).

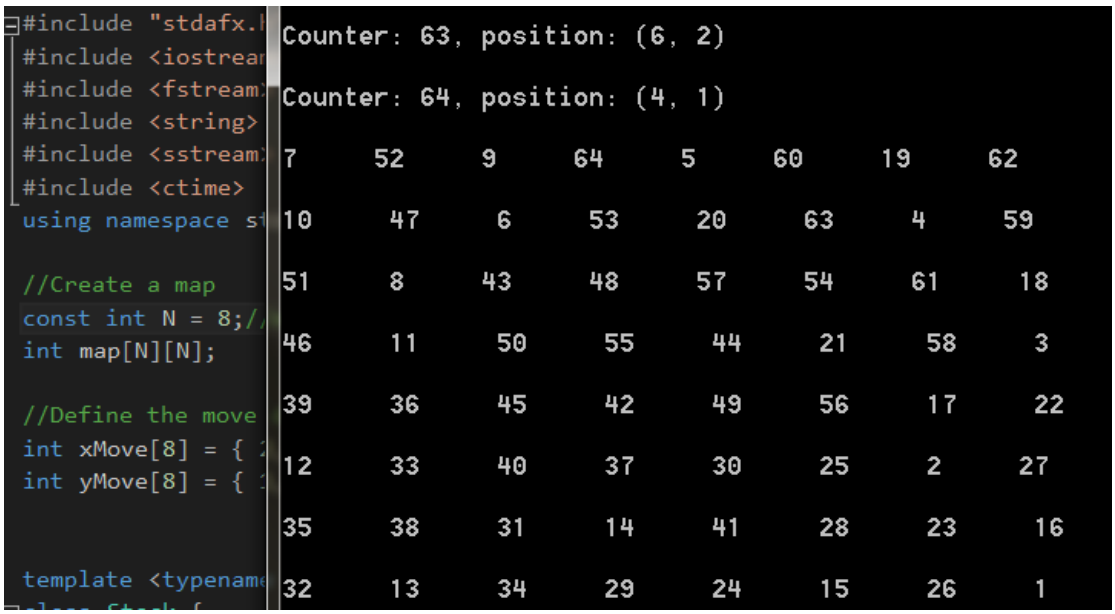


In 7*7 and 8*8 chess board, because the possible tours are too many, the running time goes to exponentially large. However, there are some particular points that can lead to the final solution quickly like (1, 1) and (8, 8).

It just takes an instant to find the final solution in 8*8 (initial position (1, 1)):



Initial position (8, 8):



Through the running experiment, we can see Warnsdoff's rule is far more efficient than Back-Tracking algorithm. It reflects how significant that an algorithm will affect the efficiency of a program.

The problem for this project is that when it is run in the SUN UNIX machines, it turns out errors says:

std::exception::exception(const std::exception&)

and

In member function ‘T Stack<T>::getTop() [with T = StkOp]’:

```
[caozh@pegasus A1_Cao_Zhihao]$ ls
cplusplus  java  output  README.txt  run.sh
[caozh@pegasus A1_Cao_Zhihao]$ sh run.sh
CSCI 36200 Assignment 1 :  Sample Run Script
Preferred Language is
CPlusPlus
Compiling CPlusPlus
knight_tour.cpp:1:20: error: stdafx.h: No such file or directory
knight_tour.cpp: In member function ‘void Stack<T>::push(T) [with T = StkOp]’:
knight_tour.cpp:190:   instantiated from here
knight_tour.cpp:31: error: no matching function for call to ‘std::exception::exc
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
knight_tour.cpp: In member function ‘T Stack<T>::getTop() [with T = StkOp]’:
knight_tour.cpp:197:   instantiated from here
knight_tour.cpp:57: error: no matching function for call to ‘std::exception::exc
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
knight_tour.cpp: In member function ‘T Stack<T>::pop() [with T = StkOp]’:
knight_tour.cpp:206:   instantiated from here
knight_tour.cpp:39: error: no matching function for call to ‘std::exception::exc
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../include/c++/4.4.7/exception:6
run.sh: line 27: ./a.out: No such file or directory
[caozh@pegasus A1_Cao_Zhihao]$
```

So far this problem did not be worked out yet.

The project will be updated as soon as this problem be solved.