CSE221: Data Structures
Programming Assignment 2
Due on Thursday 28 October, 23:59

Antoine Vigneron

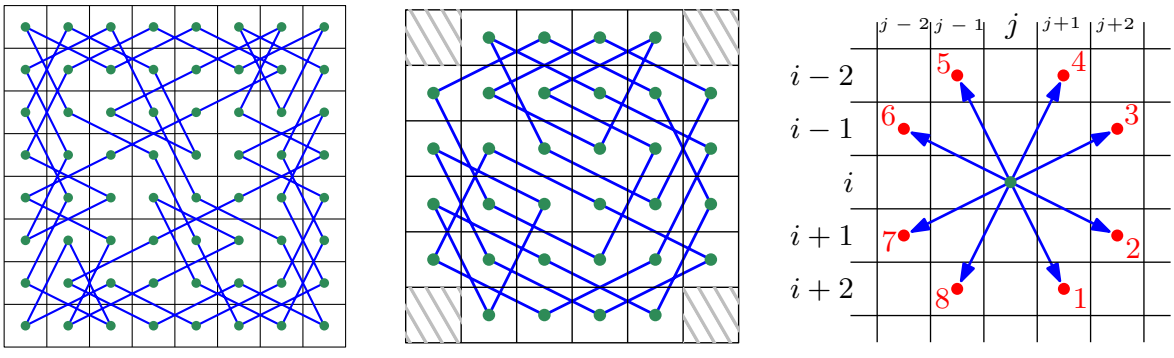
1 Introduction

The *knight's tour* problem is to find a sequence of moves for a knight on a chessboard, such that the knight visits each square exactly one time, and comes back to his starting position. Figure 1a shows a solution on the standard 8x8 chessboard. The board will not necessarily be square: Figure 1b shows the example of a 6x6 board where the 4 corners are obstacles, and the knight must traverse all squares that are not obstacles. In some instances, there is no knight's tour.

The goal of this assignment is to write a program that solves this problem. You will have to use *backtracking* (see below), which is a standard technique to solve some combinatorial problems. The solution is usually implemented as a *recursive* algorithm, but in this assignment, you will not be allowed to use recursion. So the main procedure will be *iterative*, and will use a stack to records the solution currently being explored. This is a general fact about recursion: You can always replace a recursive algorithm by an iterative algorithm that uses a stack. However, it can make the program longer and more complicated.

2 Problem Statement

A *square* is a pair $s = (i, j)$ of integers. So the squares of the board are the pairs (i, j) such that $0 \leqslant i < m$ and $0 \leqslant j < n$. A knight can move from a square (i, j) to any square (i', k') such that $|i - i'| = 1$ and $|j - j'| = 2$, or $|i - i'| = 2$ and $|j - j'| = 1$. The 8 possible moves are shown on Figure 1c.



(a) (b) (c)

Figure 1: (a) A tour of the standard 8x8 chessboard. (b) A tour of the 6x6 boardwith 4 corners removed. (c) The 8 knight's moves from square (i, j).

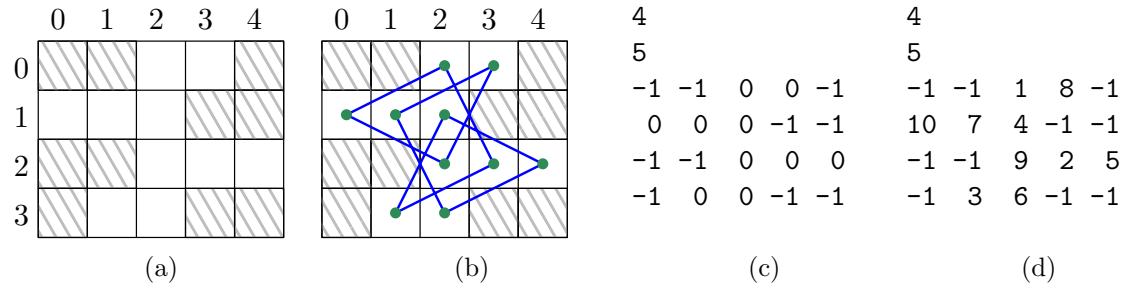


Figure 2: (a) Input $m \times n$ board, with $m = 4$ rows and $n = 5$ columns . (b) Knight's tour. (c) Input file. (d) Solution file recording the tour.

The board is represented by an $m \times n$ matrix M, where each square M_{ij} is either *empty*, and thus $M_{ij} = 0$, or is an *obstacle*, and $M_{ij} = -1$. A solution to the knight's tour problem is a sequence of knight moves, such that each empty square is visited once, and only once. In addition, it must be a closed path: The first square of the tour must be reachable from the last square of the tour in one knight's move.

The input files are given as "instance?.in" where "?" represents a number. The first line of the file represents m, the second line represents n, and then each line represents a row of the chessboard, from i=0 to m-1. (See Figure 2.) In the solution file, the "0"s are replaced with the move number in the knight's tour. So in addition to the "-1"s recording the obstacles, it should overwrite the "0"s from the input file with numbers 1, 2, 3 ....

3 Implementation

You are given a header file `knight.h`, which you should not modify. You should write the `knight.cpp` file that defines *all* the functions declared in `knight.h`. Here are some more detailed instructions that are not detailed in the comments.

3.1 Moves

The member functions of the Square class

```
Square move(Square s, int d);          //  dth move of square r
Square moveBack(Square s, int d);      //  dth move backwards
```

must be implemented using the same encoding as presented in Figure 1c.

`move(s,d)` returns the square reached after one move from square s in direction d. So for instance, if $s = (i, j)$, then `move`(s,3) should return $(i - 1, j + 2)$.

`moveBack(s,d)` is the inverse operation, so it moves in the direction opposite to d. So `moveBack`(s,3) should return $(i + 1, j - 2)$.

3.2 Range

The member function of the class Board

```
bool inRange(const Square &s) const;
```

returns `true` if the square s lies on the board (whether it is an obtacle or not). So if $s = (i, j)$, it just checks whether $0 \leqslant i < m$ and $0 \leqslant j < n$.

When calling one of the two functions below,

```
int get(Square s) const;          // gets M[s.x][s.y]
void set(Square s,int v);         // sets M[s.x][s.y]=v
```

you should check whether s is in range, and if not you should throw an exception of the type `runtime_error`.

3.3 Empty Squares

An empty square is a square (i, j) on the board that records 0, i.e. M[i][j]=0.

```
int nEmpty() const;                  // number of empty squares
Square emptySquare() const;     // returns an empty square of B
```

The function nEmpty() returns the number of empty squares, so it just counts the number of 0s in M.

The function emptySquare() returns one empty square in the board. There are several possible answers, it should just return one. The point of this function is that the tour must start from an empty square, so you will need to call it at the beginning of the algorithm.

3.4 Destructor

You should implement it in such a way that there is no memory leak.

3.5 Overloaded operators

The operator

```
bool operator==(const Square &s, const Square &r);
```

checks whether squares s and r are equal. So if $s = (i, j)$ and $r = (a, b)$, it checks whether $a = i$ and $b = j$.

The operator

```
ostream& operator<<(ostream & out, const Board & B);
```

prints a board. So given a board B, if you type `cout << B`, it should print the board B in the same format as explained in Section 2 and Figure 2; in particular it should first print m on the first line, then n on second line, and then the matrix.

3.6 Knight's tour

The main function computes the knight tour. So given the board shown in Figure 2c, it should update the values and obtain Figure 2d. The algorithm is explained in the next section.

Remember that your should *not* use recursion. You should use a stack instead. The code of the simple array-based stack is given in the file `arraystack.h`. You should use this stack class, not another one such as the STL stack.

4 Algorithm

An example of the execution of the backtracking approach is given in Figure 3. The idea is to extend the solution one step at a time. At some steps, there will be several possibilities, and we pick one. At some steps, there will be no possibility left (dead end), and we move back, i.e. we shorten the tour.

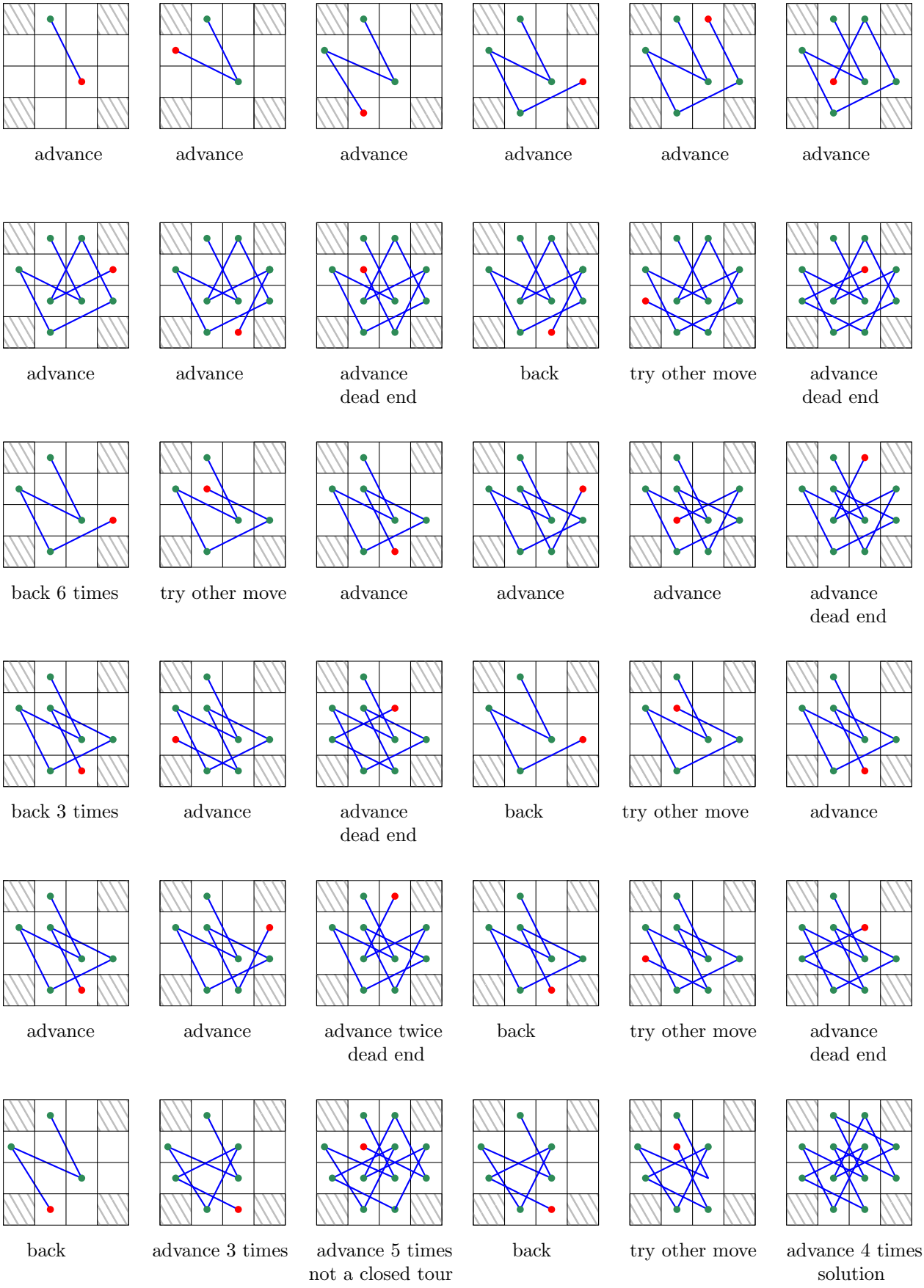


Figure 3: Solving a small instance by backtracking

When moving back, we move to the last square of the current tour where there is at least one untried possibility. Then we try this possibility. In order to implement this approach, we store in a stack the set of moves we made, that is, the direction $d \in \{1, 2, \ldots, 8\}$ of each move in the current solution.

More precisely, we keep the position `pos` of the the last square in the current tour (the red dot in Figure 3). We also keep the largest index d such that we tried to extend the path from `pos` to `move(pos,d)`. If we have not attempted such an extension before, we set $d = 0$.

Now at each step, if $d < 8$, then we increment d and try to move `pos` in direction d. If this move is possible, we extend the path by adding the square `move(pos,d)`, and we push 0 onto the stack. Then we check if the resulting tour is a full solution to the problem, and if so we return true.

On the other hand, if $d = 8$, we have no other possibility to try at this point, so we backtrack: we pop the stack, and we move backwards in the direction given by the top of the stack. The pseudocode is given below (Algorithm 1).

Algorithm 1 Pseudocode of the stack-based knight's tour algorithm.

```
 1: procedure KnightTour(board B)
 2:     S ← empty stack
 3:     pos ← an empty square                                   ▷ starting square
 4:     S.push(0)
 5:     write 1 in square pos
 6:     while true do
 7:         d=S.top()
 8:         if d < 8 then                                       ▷ try to advance
 9:             S.pop()
10:             d ← d + 1
11:             S.push(d)
12:             s ← the square obtained by moving in direction d from pos
13:             if square s is empty then
14:                 pos ← s                                     ▷ advance
15:                 S.push(0)
16:                 write S.size() in square s
17:                 if the current solution is a knight's tour then
18:                     return true
19:         else                                                ▷ moving back
20:             S.pop()
21:             if stack S is empty then
22:                 return false                 ▷ all possibilities were exhausted
23:             write 0 in square pos
24:             move pos in the direction opposite from S.top()
```

5 Testing your program

Checking programs for all the functions are provided in the "code.zip" file. Please read the README.txt file that is provided.

Instances of the knight tour problem are also given in the file "instances.zip", with their solutions.

input file	running time
Instance4.in	3.7s
Instance7.in	0.21s
Instance8.in	2m12s

Table 1: Running time of our program on the course server for 3 instances.

5.1 Submission

You should submit your code by simply uploading your `knight.cpp` file to your home directory on the server.

6 Grading

Your program will be graded automatically. Make sure that the provided test programs compile and run without error on your code. If not, it will make it considerably more difficult to grade your program, and there will be a 10% penalty.

Your program should be reasonably efficient to get full mark. You will get full grade if it is at most twice slower than our solution. (See Table 1.)

Late submissions up to one day after the deadline will be penalized 15%, and 30% two days after the deadline. After that, your assignment will not be graded (hence you get 0).

The grading program we will use in the end may differ from the ones that have been/will be provided.

7 Other rules

The general rules for the CSE department, given at the beginning of the semester (see course information folder), apply to this assignment. You may contact one of our two TAs for help:

- Seonghyeon Jue (`shjue@unist.ac.kr`)
- Hyeyun Yang (`gm1225@unist.ac.kr`)

Hyeyun Yang will be grading this assignment.

8 Typos and bugs

If you find typos in this handout, or bugs in the code that is being supplied, please send email to the TAs or to me.