Project 2

Rectangle Puzzle

Introduction

You may have seen the Triangle Puzzle, perhaps in a restaurant. It consists of a triangular board with a triangular array of holes, five holes on a side, and 14 pegs, leaving one hole unoccupied. The goal is remove all but one peg by jumping pegs as in checkers, always jumping one peg over another peg and into an unoccupied hole and then removing the peg that was jumped.

In this project we will create a similar puzzle, except that it will be played on a rectangular board. That will be simpler to program, as every row will contain the same number of holes.

Data Structures

The `Point` Class

We will use the same `Point` class that we used in Project 1.

The `Vectr` Template Class

We will use the `Vectr` template class, which is included in Laboratory 4.

The `Move` Class

A `Move` is a `Vectr` of three `Point`s. The first point represents the location of the peg that is jumping. The second point represents the location of the peg that is jumped over and removed. The third point represents the location of the unoccupied hole where the jumping peg lands. See the document `Move Class.pdf` for the details of the data members and the member functions.

Moves can be made in four possible directions: to the left, to the right, upwards, and downwards. Thus, a `Move` object with have one of the following forms, where `(x, y)` is the location of the jumping peg.

```
((x, y), (x, y - 1), (x, y - 2))    // Jump to the left
((x, y), (x, y + 1), (x, y + 2))    // Jump to the right
((x, y), (x + 1, y), (x + 2, y))    // Jump upwards
((x, y), (x - 1, y), (x - 2, y))    // Jump downwards
```

The `PuzzleBoard` Class

The `PuzzleBoard` is a rectangular array of boolean values, where `true` means that the square is occupied by a peg and `false` means that it is not occupied. The `PuzzleBoard` class will have only one data member: `m_board` which is a vector of vectors of `bool`s.

For example, a 3×4 puzzle board with the starting hole in position $(0, 0)$ would look like

```
((0, 1, 1, 1), (1, 1, 1, 1), (1, 1, 1, 1))
```

where `1` means `true` and `0` means `false`.

The number of rows in the array is the size of the "outer" vector, namely, `m_board.size()`. The number of columns in the array is the size of any one of the "inner" vectors, say, `m_board[0].size()`.

See the document `PuzzleBoard Class.pdf` for details of the member functions.

The Algorithm

The program should begin by prompting the user for the number of rows and the number of columns in the board. Then the user is prompted for the location of the initial unoccupied square. This location will be entered as a `Point` object. Then the rows, columns, and starting location are used to construct a `PuzzleBoard` object.

Before attempting to solve the puzzle, we must create a list of all possible moves. Each move is a `Move` object that represents two occupied squares followed by an unoccupied square, which allows a legal move to be made. The total number of moves to the right is the number of rows times 2 less than *cols*, where *cols* is the number of columns, because the jumping peg must be in one of the columns 0 through $cols - 2$. There are an equal number of moves to the left. Similarly, the number of moves upwards is the number of columns times $rows - 2$, where *rows* is the number of rows, and there are an equal number of downward moves. Thus, the total number of possible moves is

$$2(rows)(cols - 2) + 2(cols)(rows - 2) = 4(rows \cdot cols - rows - cols).$$

You should create a vector of that size and then fill it systematically with all the possible moves.

The algorithm will be recursive, much like the algorithm in Project 1. I suggest that you write a function `solvePuzzle()` that will receive the `PuzzleBoard` object and the vector of possible moves. The `solveProblem()` function should first check to see whether the problem has been solved by calling the `solved()` member function of the `PuzzleBoard` class. If the puzzle has been solved, then `solvePuzzle()` should return `true`.

Otherwise, begin trying the possible moves. Use a `for` loop to work sequentially through the list of possible moves. For each move in the list, first check whether it is legal. If it is, then make that move by calling the `move()` function of the `PuzzleBoard` class and then call `solvePuzzle()` (a recursive call), passing the board and the list of possible moves.

If `solveProblem()` returns `true`, then return `true`. If `solveProblem()` returns `false`, then undo that move by calling the `remove()` function of the `PuzzleBoard` class and try the next possible move. If all possible moves have been tried, but without success, then the `for` loop will be exited and you should return `false`.

When execution returns to the `main()` function, `solvePuzzle()` will return either `true` or `false`. If it returns `true`, then display the sequence of all moves that were made, by calling the `displayHistory()` function of the `PuzzleBoard` class. If it returns `false`, then print a message indicating that this instance of the Rectangle Puzzle is not solvable.

When you are finished, place the files `move.h`, `move.cpp`, `puzzleboard.h`, `puzzleboard.cpp`, and `RectanglePuzzle.cpp` in a folder named `Project_2` and drop it in the dropbox by 5:00 pm, Friday, February 16.