Software Engineer Graduate Assessment

Samuel Teed

Software Engineer Graduate Assessment

Problem

This section will describe the problem which the remainder of this report will be answering.

Puzzle

A jigsaw like the one shown to the right sits in a fixed border with many different shaped pieces that interlock together. The puzzle is solved by placing all pieces within the border with no gaps; there may be multiple ways of placing the pieces within the boarder to solve the puzzle.

Task 1 (Algorithm)

Describe an efficient algorithm using pseudo code to solve any puzzle based on the conditions described in the above statement.

Task 2 (Database Schema)

Design a database schema that can be used to store the puzzle properties.

Algorithm

This section of the report will be answering **Task 1**. That is, it will be describing an efficient algorithm for solving the puzzle problem using pseudo code.

Notes:

Underlined text indicates a variables type

A position is the starting space for where a piece will be placed

The puzzle board and pieces are both made up of many spaces

Puzzle board spaces range from bottom-left(0,0) to top=right(length, width)

Code

```
function SolvePuzzle(number length, number width, number number of pieces of each shape):
    stack of pieces unplacedPieces = new stack of all pieces
    puzzle puzzle = new Puzzle(length, width)
    piece p = get piece from unplacedPieces
    PlacePiece(p, puzzle)
    return error "Unsolvable Puzzle"
```

Elaboration:

Imagine a puzzle with board size 4x4 and there are 3 possible shapes of pieces (a 2x2 square, a T shape, and a Line shape). 4 square pieces are given as inputs. The resulting function input would read: SolvePuzzle(4, 4, 4, 0, 0).

The stack used here would be a last on first off kind and when get is used on the stack the item on top of the stack is removed.

```
function PlacePiece(piece piece, puzzle puzzle):
    for each position on puzzle board
        if position does not already contain a piece
        for each orientation piece can take
            try:
                for each space piece occupies in current position and orientation
                    if space does not already contain a piece and
                            space is not outside puzzle board:
                        assign piece to space
                    else:
                        return error to try
            catch:
                skip – move to next orientation
            if unplacedPieces sum is 0 and
                    for each space there is no unassigned space:
                return solved puzzle
            else:
                piece nextPiece = get piece from unplacedPieces
                PlacePiece(nextPiece, puzzle)
    Add piece back to unplacedPieces
```

Elaboration:

The input PlacePiece(line piece, puzzle) is given to the program. The program would pick a position on the board, (0,0) for example. It checks if the space at this position is assigned to a piece. If not it will then pick an orientation for the current piece, let's say vertically for the current line piece. It will then take the bottom-left space of the current piece and assign it to the position. It will then continue to check the spaces above the current position and assign the piece to the current space if possible. If this is not possible the newly assigned spaces will lose their assignments and the program will pick a different orientation for the piece and try again. Once all space of a piece are assigned the program picks the next piece and recursively continues the cycle until there are no more pieces.

If the program fails to find a place for a piece it returns this piece back to the stack and then returns to the previous iteration of the recursive loop and begins finding a different orientation or position to place the piece. If at any point all piece are on the board the program will do a final check to see if there are any gaps on the board and if not it will then return the solved puzzle. Where as if he program manages to attempt to place the first piece in every position in every orientation without finding a solution it will return that the puzzle is impossible.

Class Puzzle(length, width):

A 2d array of length multiplied by width spaces represented by tuples containing positional data and which piece is assigned to the space.

Example:

```
[(0,0), NULL]
[(0,1), Line-piece]
```

Class Piece(shape):

Each piece shape would have a list of spaces that the piece would take up for each orientation the piece can take.

Example:

```
Piece(4-Space-Line):
    Orientation 1:
        [(0,0), (0,1), (0,2), (0,3)]
    Orientation 2:
        [(0,0), (1,0), (2,0), (3,0)]
```

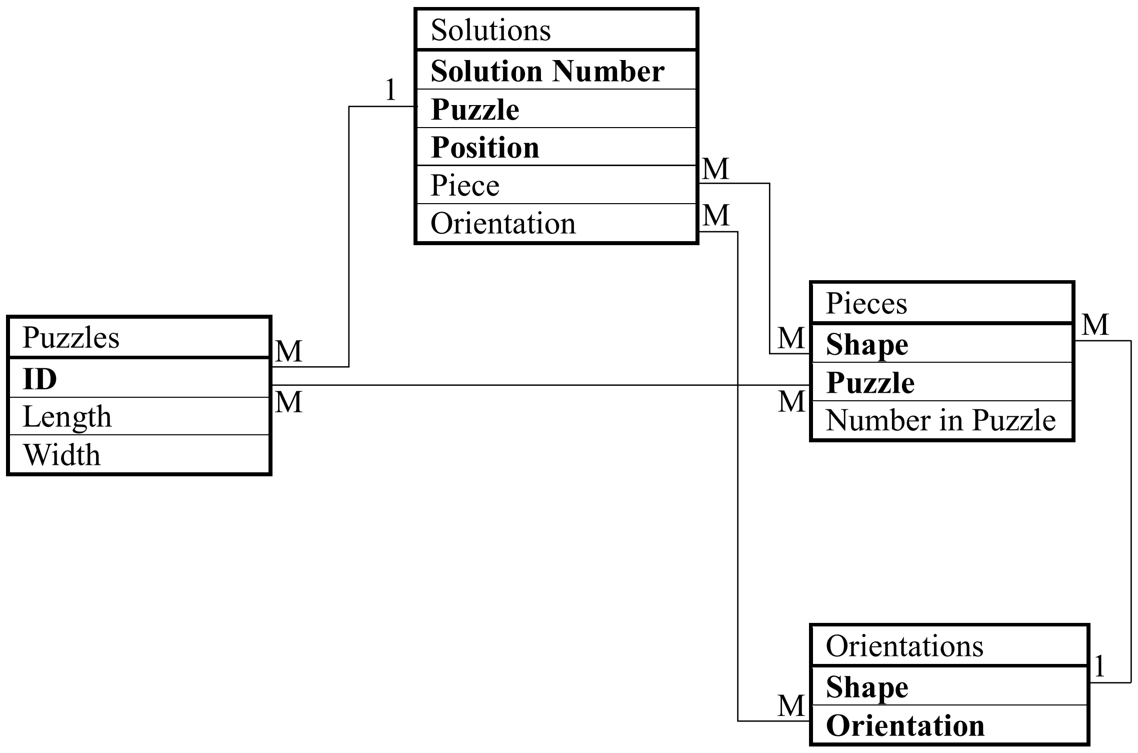
Analysis

With this algorithm the program will test if a piece can fit a space in every orientation, this is a constant time use. It will do this however for every space for every piece for every space. This means this algorithm would have a worst case time complexity of O((P.S)^S) where P is the number of pieces and S is the number of spaces on the puzzle board.

Database Schema

This section of the report will be answering **Task 2**. That is, it will be describing a database schema for storing a puzzles properties.

Schema



Research

https://codegolf.stackexchange.com/questions/39885/solve-a-tetris-puzzle-pack-predefined-shapes-into-optimal-form

https://github.com/adolfintel/tetrispuzzlesolver