COSC 4P03 A2

Christopher Delo, 6418024

March 16, 2023

1 Usage

Although the program can be run very easily, please not that the text output files will appear in either

`\LatinSquare\bin\Release\net6.0`

or

`\LatinSquare\bin\Debug\net6.0`

2 Basis

In this definition of equivalent, two squares are called equivalent if there is a set of row and column permutations that can transform the first into the second. The approach I took to this was finding a way to reduce all squares to a set of equivalent possibilities. I found that all Latin squares can be sorted into a set of equivalent ones through the following method.

Reduce original matrix

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 2 & x & x & x \\ 3 & x & x & x \\ 4 & x & x & x \end{matrix}$$

Create a new set of matrices by sorting remaining columns or rows into natural order

$$\begin{matrix} x & 1 & x & x \\ x & 2 & x & x \\ x & 3 & x & x \\ x & 4 & x & x \end{matrix} \quad \begin{matrix} x & x & 1 & x \\ x & x & 2 & x \\ x & x & 3 & x \\ x & x & 4 & x \end{matrix} \quad \begin{matrix} x & x & x & 1 \\ x & x & x & 2 \\ x & x & x & 3 \\ x & x & x & 4 \end{matrix}$$

After this, apply to each a transformation such that the topmost row is of increasing order, like the original reduced matrix. This can also be applied the other way, using rows then columns. However, this means that for each valid square there are roughly n permutations that must be verified as inequivalent to existing solutions. Therefore, the CPU bottleneck of this project will inevitably be in the verification stage. With that in mind:

3 Explanations

3.1 ValidationSort(int[][] square, int row)

This method does what mentioned in the general theory portion, it permutes the columns the square such that the selected row is in natural order. Then following this is permutes the rows such that the leftmost column is in natural order. This has a side effect of placing the previously sorted row at the top, thereby recreating the reduced form.

3.2 Backtrack(int[][] square, int x, int y)

This method is the main method for this project. Though not too different from other backtrack algorithms, it's worth mentioning that by it's construction it is assumed that a square will be valid after each decision is made or it will recurse. Therefore there is no piece of code validating an entire square, just individual placement options.

3.3 SquareExists(int[][] square)

This method is responsible for making sure that a square is not already present in the list of in-equivalent squares. It does this by implementing the processes mentioned in ValidationSort and section 1. It gathers a list of all possible reduced form permutations of the square, then checks them against each existing in-equivalent squares.

3.4 IsEquivalent(int[][] squareOne, int[][] squareTwo

This method is the most important one to optimize, since in testing with n=7 for part one the program reached several trillion ms of checking equivalences alone. This makes sense, since there ¿ 14 million reduced form 7x7 Latin squares, and each has to test each of its permutations against the previously found in-equivalent squares. By its nature as a reduced Latin square, the leftmost column and topmost row are identical by definition, so the iterators for x and y coordinates can be started at index 1 instead of index 0. In addition, if a square missing a single row is identical to another square, then they must be identical seeing as those cells are automatically known, like a negative exposure. With this in mind, an additional row and column can be skipped in running comparisons. I chose to skip the final one for symmetry. This means that the time cost of comparing two matrices is reduced from n^2 to $(n-2)^2$. Though it remains in the order of $O(n^2)$, it is still a slight time save.

3.5 EmptySquare(int size)

The final method that requires elaboration is the generation method. Since all the squares will end up in reduced form regardless, I chose to generate them all following the template provided by their definition, with the topmost row and leftmost column in natural order. This reduces the depth of a backtracking tree by $2n-1$, which is a significant reduction considering their compounding nature.

4 Images

The following images are screenshots of my CPU ms while running the project for n= [4,6].

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
LatinSquare (PID: 31584)	1292 (100.00%)	749 (57.97%)	LatinSquare
LatinSquare.Backtrack(int32[][], int, int)	53 (4.10%)	37 (2.86%)	latinsquare
system.private.corelib.dll!0x00007ffb93e9b1af	27 (2.09%)	27 (2.09%)	system.private.cor...
LatinSquare.ctor()	159 (12.31%)	22 (1.70%)	latinsquare
system.private.corelib.dll!0x00007ffb94144fd8	21 (1.63%)	21 (1.63%)	system.private.cor...
system.private.corelib.dll!0x00007ffb94147deb	21 (1.63%)	21 (1.63%)	system.private.cor...
LatinSquare.Part_1(int)	109 (8.44%)	17 (1.32%)	latinsquare

Figure 1: n=4

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
LatinSquare (PID: 20124)	1273 (100.00%)	729 (57.27%)
LatinSquare.Backtrack(int32[][], int, int)	56 (4.40%)	39 (3.06%)
system.private.corelib.dll!0x00007ffb93e9b1af	27 (2.12%)	27 (2.12%)
system.private.corelib.dll!0x00007ffb94144fd8	24 (1.89%)	24 (1.89%)
LatinSquare.ctor()	169 (13.28%)	23 (1.81%)
LatinSquare.Part_1(int)	116 (9.11%)	21 (1.65%)

Figure 2: n=5

LatinSquare (PID: 33580)	6401 (100.00%)	789 (12.33%)	LatinSquare
LatinSquare.IsEquivalent(int32[][], int32[][])	3962 (61.90%)	3962 (61.90%)	latinsquare
LatinSquare.SquareExists(int32[][])	4868 (76.05%)	676 (10.56%)	latinsquare
LatinSquare.EmptySquare(int)	154 (2.41%)	111 (1.73%)	latinsquare
LatinSquare.PositionIsValid(int32[][], int, int, int)	144 (2.25%)	98 (1.53%)	latinsquare
LatinSquare.Backtrack(int32[][], int, int)	5082 (79.39%)	68 (1.06%)	latinsquare
System.Runtime.CompilerServices.CastHelpers.St...	52 (0.81%)	52 (0.81%)	System.Private.Co...
LatinSquare.ValidationSort(int32[][], int)	191 (2.98%)	29 (0.45%)	latinsquare
system.private.corelib.dll!0x00007ffb93e9b1af	29 (0.45%)	28 (0.44%)	system.private.cor...
system.private.corelib.dll!0x00007ffb94144fd8	27 (0.42%)	27 (0.42%)	system.private.cor...
LatinSquare.ctor()	5203 (81.28%)	27 (0.42%)	latinsquare
LatinSquare.Part_1(int)	5148 (80.42%)	24 (0.37%)	latinsquare

Figure 3: n=6

Function Name	Total CPU [unit, ...	Self CPU [unit, %]
LatinSquare (PID: 33620)	6352 (100.00%)	890 (14.01%)
LatinSquare.Main(System.String[])	5079 (79.96%)	4 (0.06%)
LatinSquare.ctor()	5075 (79.90%)	22 (0.35%)
LatinSquare.Part_1(int)	5023 (79.08%)	18 (0.28%)
LatinSquare.Backtrack(int32[][], int, int)	4963 (78.13%)	66 (1.04%)
LatinSquare.SquareExists(int32[][])	4772 (75.13%)	706 (11.11%)
LatinSquare.IsEquivalent(int32[][], int32[][])	3835 (60.37%)	3835 (60.37%)

Figure 4: alt n=6

4.1 Regrets

There is another time save I considered but ultimately didn't feel the need to implement as it was trivial. When a square is almost done being created, the final row/column is trivial. It can be completed automatically, as mentioned above in the IsEquivalent description. It would also be a negligible time save, maybe saving $n(n-1)$ operations per square in the end. In the same way, when a row is almost done generating the final value could be substituted in, but the same idea of only saving a handful of operations. I also had the idea that the final value is equal to the sum of the first n-1 values subtracted from the algebraic series sum, but again found it too small of a time save to be worth implementing. As it turns out, the main issue with runs of $n < 6$ is indeed the backtracking algorithm, so it's possible I would have saved a few ms here and there. Alas.