Python Coursework: A report investigating Latin squares

Arwyn Thandrayen:1805468, Ben Thomas:1823663, Jui-Hsuan Su:1888337, Meng Wang:1988603

MA2760: Mathematical Investigations with Python
School of Mathematics,
Cardiff University.

April 28, 2020

1 Introduction

In this report we will be discussing the optimisation of Latin squares. Latin squares are examples of puzzles first conceptualized in depth by mathematicians like Leonhard Euler in the 18th century. The world famous Sudoku conundrums are based on the fundamental ideas of Latin, and partial Latin, squares. The idea behind these Latin squares is to fill a n x n matrix containing n **different** symbols whereby each symbol only occurs once per row and once per column. In this report we develop a series of python programs in order to tackle four tasks investigating different scenarios focusing on these enigma squares.

2 Methodology

2.1 Task 1

This first task required our program to be able to read an n x n matrix in a specified problem file containing gaps and hence produce an 'initial solution' to form a partial Latin square. To do this, our code had to satisfy the initial constraint that all numbers 0,...,n-1 appeared exactly once in each row, but didn't require this same property to apply to the columns.

Due to columns being exempted from this property, this is why it is only classed as a partial Latin square as only the row constraint is focused upon. Only when both the rows and columns simultaneously adhere this property does it then become a Latin square.

2.1.1 Reading files and neat display

We began our program by allowing the reading of a desired file using the python's in built "with open" command. This allowed for a user input of any file saved on the computer and therefore by saving the p1, p2 and p3 files on learning central, the program was able to open these and begin to analyse. These files contained n x n matrices with various number of 'given' elements and then the remaining placeholders were 'blank' elements.

Next, we needed a method of displaying what was in the notepad++ documents in a more eligible form within python. As a consequence, using a function called "printSquare", this read the file line by line and transformed the problem into a matrix form within python. This new layout is far more aesthetic as well as practical. By making the problem documents easier to display it becomes easier to proceed further with the tasks because now we can vividly see how our functions alter this matrix.

2.1.2 Determining 'given' and 'blank' elements

The first line of these text files specifies the dimension of the matrix. Next, we had to ensure the program interpreted the matrix correctly by differentiating between 'given' elements, which had fixed predetermined values, and 'blank' elements which had unspecified values. The values that were 'given' were essential, these were unable to change when programming our random initial solution.

In order to separate the 'given' and 'blank' terms, the file labelled the placeholders with 'blank' elements to have values of -1. Therefore the program is able to detect this and hence provide a new value for that placeholder in the set $\{ x \mid x$ is in (0, n-1) and not already in that row $\}$. By creating a new function called specialPos, this remembers the 'given' placeholders and their values and ensures they do not get changed, moved or swapped in later functions.

2.1.3 Coding a random initial solution

As previously stated an initial solution for a specified problem is when the 'givens' are all fixed and in their corresponding placeholder and each **row** satisfies containing all numbers in the range (0, n-1) with no repetitions. There are various different solutions to these partial Latin squares with the number of potential solutions increasing exponentially as n gets larger following the sequence:

$$N(n,n) = n!(n-1)!L(n,n)$$

where n is the size of the matrix and L is the number of **normalised** Latin squares.

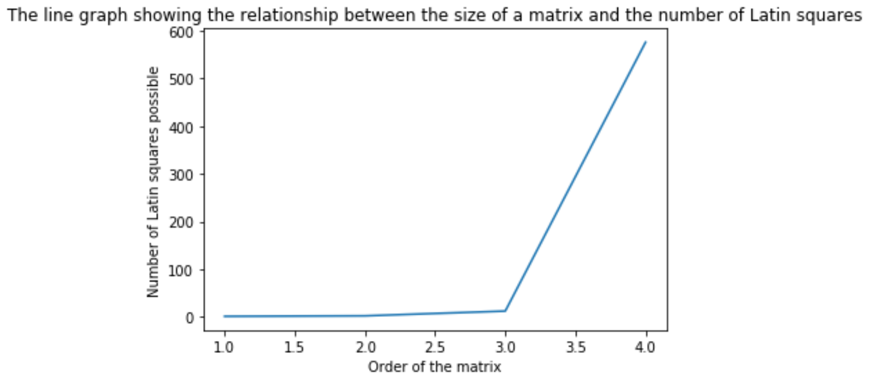

Figure 1:

To achieve our initial solution our program recorded whenever a -1 value occurred in the problem file and instead changed it to become a random number between (0, n-1) using the import random feature. In addition, in order for our solution to satisfy the property of no reoccurring numbers in the rows, we had to program the fact we cannot have any repetitions.

Running the file "p1.txt" in our program, the following result is just one example of a random partial Latin square initial solution.

```
n           = 5
num. givens = 7
3       4       1       2       0
1       0       2       3       4
1       4       0       2       3
2       3       0       1       4
2       1       4       0       3
```

Figure 2: Initial partial Latin square solution for p1.txt

This matrix is of size n = 5 and we can see the following solution has all rows contain 0, 1, 2, 3, 4 with no repetitions. Hence all constraints have been satisfied.

The exact same can be shown for the file "p2.txt" with its initial solution being:

```
n           = 9
num. givens = 17
2       5       6       8       0       3       7       1       4
4       3       7       6       2       1       0       5       8
0       2       6       3       4       5       8       7       1
3       1       0       2       5       8       6       7       4
1       5       8       6       0       7       3       4       2
8       4       1       3       0       7       5       6       2
3       8       1       4       6       7       2       5       0
3       5       6       1       2       4       8       7       0
1       3       4       8       0       7       2       6       5
```

Figure 3: Initial partial Latin square solution for p2.txt

Although very unlikely for larger values of n, it is possible for a randomly generated partial solution to also be a solution for a complete Latin square. However, this is dependent on n and the number of 'given' entries compared to 'blank' entries. One way this occurs is via achieving a root solution.

2.1.4 Identifying the "root solution"

For any value n, if there has been no 'given' elements from the matrix supplied in the problem file ,then our program is able to determine the 'root solution'. This solution essentially has values (0,...,n-1) on the first row, (1,2,...n-1,0) on the second row and this process is repeated after each row seeing a shift by 1 as you descend the rows.

In order to do this, a function called 'rootSolution' recognises if the matrix in the problem file consists entirely of -1's. If an nxn matrix contains only these 'blank' elements, then each element can be generated by the formula:

$$(i + j)\%n$$

where i and j represent the number of rows and columns and n is the 'size' of the matrix.

This function can be seen applied when inserting ”p3.txt” into our program which is a 10x10 matrix made up of -1’s (no ’given’ elements).

```
n            = 10
num. givens = 0
Root Solution
0      1      2      3      4      5      6       7       8       9
1      2      3      4      5      6      7       8       9       0
2      3      4      5      6      7      8       9       0       1
3      4      5      6      7      8      9       0       1       2
4      5      6      7      8      9      0       1       2       3
5      6      7      8      9      0      1       2       3       4
6      7      8      9      0      1      2       3       4       5
7      8      9      0      1      2      3       4       5       6
8      9      0      1      2      3      4       5       6       7
9      0      1      2      3      4      5       6       7       8
```

Figure 4: Showing how the program identifies and produces a root solution for file p3.txt

2.2 Task 2

The purpose of this second task is to create a method of assessing how close our initial solution is to achieving a completely optimized Latin square. In addition, we were also required to implement a local search technique whereby we can gradually increase this assessment value by applying a certain small change.

2.2.1 Programming the ’cost’ function

In order to achieve some sort of assessment regarding the quality of our random initial partial Latin square we established a ’cost’ function. This is able to identify how our initial solution compares to a complete Latin square solution for that matrix. It works by reviewing each column in our solution matrix and calculates the number of terms missing from the set [0,n-1] in that column. The function will run through each column separately and the sum of all the individual columns will be the initial solution’s cost value. Ideally, we want to have a total cost of 0 as this means we have achieved a complete Latin square.

If the starting matrix consists of only ’blank’ elements then by definition, our ’root solution’ is exactly a Latin square and hence its score value will be 0. However, its more probable to have a matrix which does have ’given’ elements and therefore a score will be achieved.

2.2.2 Local search technique

This next step involves an iteration technique in order to obtain a complete Latin square. It works by undergoing a small change for a number of iterations to see whether the score value changes after the change is applied. We implemented a local search technique in attempt to decrease this cost function without jeopardising the initial row constraint.

To do this, we created a ’swap’ function. This works by swapping two elements in our initial solution which were ’blank/non-given’ placeholders in order to see if the cost is affected. By selecting a row at random and two columns at random then interchanging the elements, as long as they’re not ’given’ elements, this is classed as our ’swap’.

Due to the ’swap’ function being implemented, this changes our matrix and hence the ’cost’ function is affected. As we want to achieve a cost of 0, we want our cost to be decreasing. Therefore, if a ’swap’ function outcomes a matrix with a lower cost function or the same as its predecessor, then the swapped matrix becomes the ”winner” of that iteration and this becomes our new initial solution. However, if the swap occurs and the cost doesn’t increase, then we have to return back to the original matrix before that swap. To do this, we created a ’changeBack’ function which does exactly this, by returning the matrix after the swap function back to the matrix before it.

The program has a user input for the number of times you wish to carry out the ’swap’ function, and the program will run until this iteration number is reached or the matrix achieved has a cost of 0.

2.3 Task 3

After running our program with 50 iterations, the final solution we obtain For ’p1.txt’ is:

```
3       4       2       0       1
1       0       3       4       2
0       1       4       2       3
2       3       0       1       4
4       1       1       0       3
The cost for this final puzzle is 2
```

Figure 5: Final solution obtained after 50 iterations for ’p1.txt’50

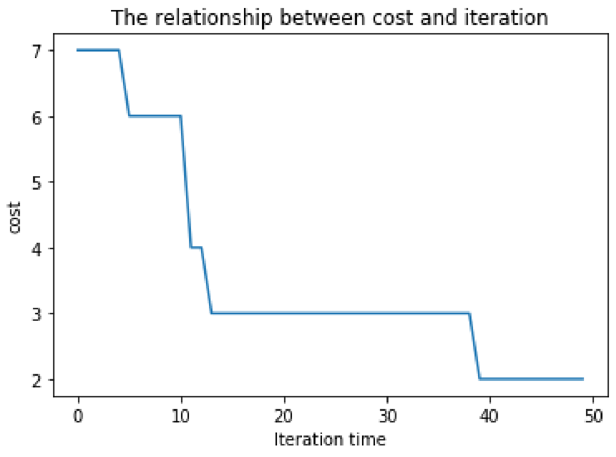

Figure 6: A line graph to show how the cost changes as number of iterations increases for p1.txt

The graph shown in figure 6 demonstrates how our cost has decreased after 50 iterations. This method is clearly effective the cost has reduced significantly in what is considered fairly few number of iterations. Due to the randomness of the function, it is likely the graph will remain at 2 for a long time until the cost can be will be zero (a perfect Latin square) as a cost of 1 is unable to occur.

Repeating the same process for the file 'p2.txt', after 50 iterations the final solution we obtain is:

```
2       5       3       0       8       6       7       1       4
4       3       5       6       2       1       0       8       7
0       2       1       3       4       5       8       7       6
3       1       8       7       5       0       6       2       4
0       5       8       6       1       7       3       4       2
7       4       1       8       0       3       5       6       2
3       8       6       4       7       1       2       5       0
2       5       6       1       3       8       4       7       0
1       3       4       8       6       7       2       0       5
The cost for the final puzzle is: 18
```

Figure 7: Final solution obtained after 50 iterations for 'p2.txt'50

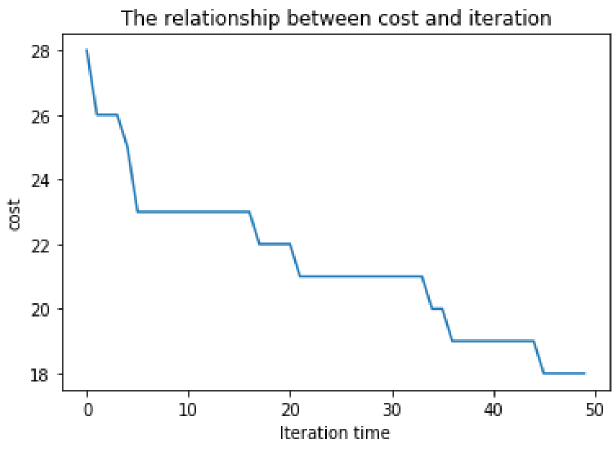

Figure 8: A line graph to show how the cost changes as number of iterations increases for p2.txt

From the graph shown in figure 8, we can see that this procedure is very effective with the cost declining a value of 10 in just 50 iterations.

2.4 Task 4

In this task we were encouraged to think creatively on how we can carry out this investigation further. Therefore we decided to focus upon the following bullet points.

- Making the local search code more efficient
- Altering ways the solution can be changed
- How to adapt our code so that it can be used to solve Sudoku puzzles

Using the copy module we were able to compare the task 4 experiments with the task 2 result as we were able to use the same starting puzzle.

2.4.1 Making the local search code more efficient

Here we realised that when undergoing the 'swap' function, the rows continually satisfy the property that (n-1)numbers appear only once. In other words, each row had a score of 0 no matter what. Therefore, we can change our program to only consider the score of the columns. If the overall score of the columns is zero, then we know we have achieved a Latin square.

Therefore, our new search method is to create a function called columnCost which calculates the total cost of two selected columns. Combining this with the 'swap2' function which randomly selects the 2 columns the columnCost function works on. Then, the 'swap2' function picks a random row and swaps the elements in this row corresponding to the selected columns. Then the transformed matrix undergoes the columnCost function again with another two random columns. The puzzle with the lowest columnCost will "win" that iteration and will become the next puzzle that the swap function performs on. This all occurred whilst still maintaining the integrity of the row condition as well as not moving any of the 'given' positions by including the specPos function and maintaining the better puzzle if necessary using the changeBack function.

2.4.2 Altering ways the solution can be changed

Our first idea to begin optimizing our matrix was to apply the improved local search method and instead of randomising the two columns that are selected to undergo the swap, we choose these columns to be the ones with the highest individual score values.

However, after producing the new 'Columcost2' and 'swap3' functions and creating a graph showing how the overall cost changes over time we got the following result:

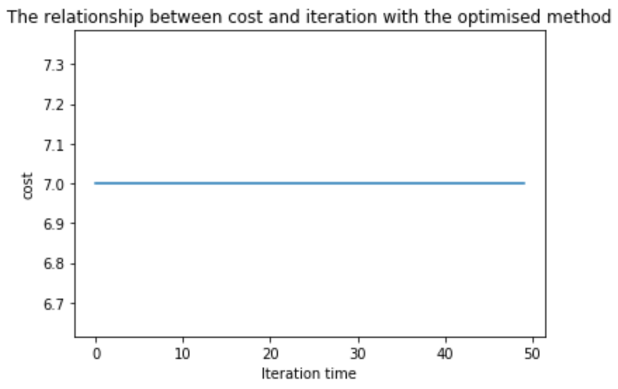

Figure 9: A line graph to show how the cost changes as number of iterations increases

Clearly from figure 9, this is not as effective as the method in Task 2 and it was here that we realised that the reason for this was that there was too many restrictions being applied. The constraints meant that it was actually difficult for the swap3 function to even be applied. As a consequence of the 'given' elements which were unable to move, then sometimes a swap can't occur. In addition with the first and second columns with the highest columCost2 value being mostly the same two due to a lack of swaps, hence, the function remains in a loop until a suitable swap can actually occur.

However, although this method doesn't produce a higher score than the method in task2, it does achieve an outcome faster (with the same number of iterations) which is another significant factor in the optimization process. These are two graphs which show the time taken to perform the 'cost' function in task 2, compared to the 'columncost2' method in task 4.

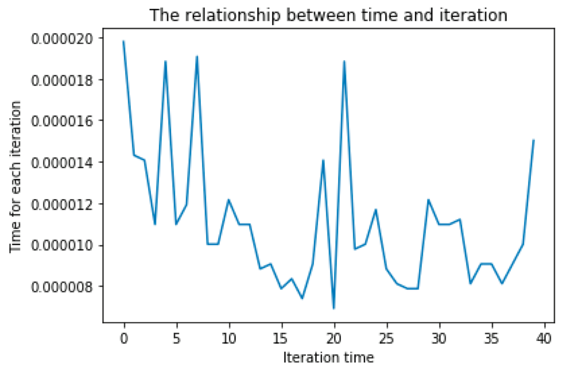

Figure 10: Task 2's duration of "cost" function

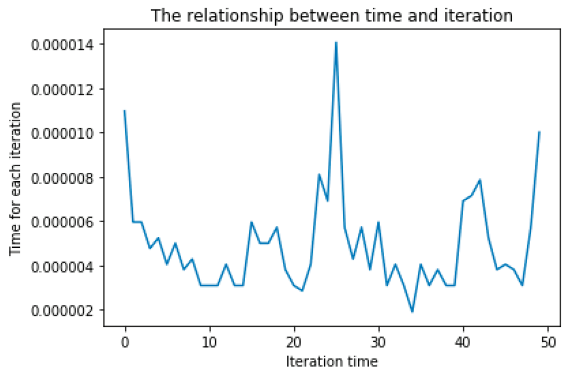

Figure 11: Task 4's duration of the "colsCost2" function

From figures 10 and 11 we can see that the time taken to conduct the task 4 technique is indeed shorter than the task 2 technique. The task 2 technique has 4 main peaks approximately taking the computer 0.000019 seconds to compute in 40 iterations. Whereas, the second task only has one major peak at 0.000014 and this was with 50 iterations.

2.4.3 How to adapt our code so that it can be used to solve Sudoku puzzles

Sudoku problems are extremely similar to what we have been doing already but need one further constraint to be classed as solved. They also require not only a cost of zero for both rows and columns to occur but the additional requirement that the 9x9 matrix, when split into nine 3x3 grids, there must be numbers 1-9 with no repeats in each sub grid.

In our program, we have enabled it so that you can insert the Sudoku problem as a grid whereby zero's play the same role as -1's in the problem sheet files and are the 'blank' elements.

3 Conclusion

To conclude, this project is extremely malleable in what you can do. From just the investigations that we have done, we are able to see that there is a lot of potential to improve our optimisation techniques. If we were to continue this experiment, we would investigate alternative optimisation methods such as simulated annealing and tabu search to see how they compare with the brute force approach not only in number of iterations required but also in time to compute and length of the code.