Solving the Latin Square Problem with the MiniZinc System and Stochastic Hill Climbing (Local Search)

Terry Tian

The University of Tokyo / Monash University

Topics in Information Science II: Efficient Search Methods in Artificial Intelligence

$$\begin{bmatrix}1&2&3\\2&3&1\\3&1&2\end{bmatrix},\begin{bmatrix}1&2&3\\3&1&2\\2&3&1\end{bmatrix},\begin{bmatrix}1&3&2\\2&1&3\\3&2&1\end{bmatrix},\begin{bmatrix}1&3&2\\3&2&1\\2&1&3\end{bmatrix},$$

$$\begin{bmatrix}2&1&3\\1&3&2\\3&2&1\end{bmatrix},\begin{bmatrix}2&1&3\\3&2&1\\1&3&2\end{bmatrix},\begin{bmatrix}2&3&1\\1&2&3\\3&1&2\end{bmatrix},\begin{bmatrix}2&3&1\\3&1&2\\1&2&3\end{bmatrix},$$

$$\begin{bmatrix}3&2&1\\1&3&2\\2&1&3\end{bmatrix},\begin{bmatrix}3&2&1\\2&1&3\\1&3&2\end{bmatrix},\begin{bmatrix}3&1&2\\1&2&3\\2&3&1\end{bmatrix},\begin{bmatrix}3&1&2\\2&3&1\\1&2&3\end{bmatrix},$$

tytia@g.ecc.u-tokyo.ac.jp / ttia0006@student.monash.edu

Table of Contents

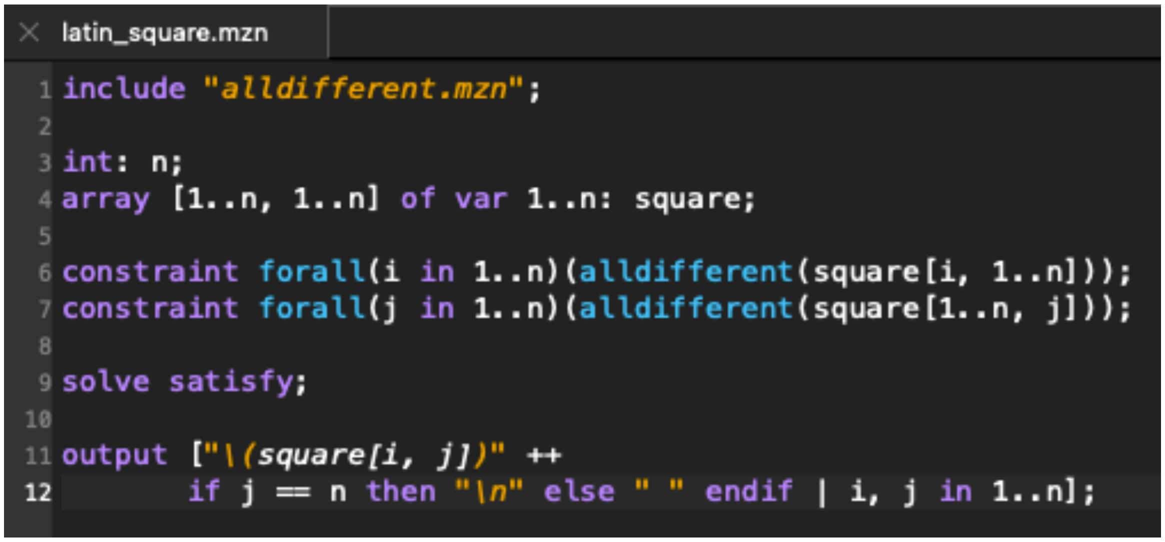
Abstract

The Latin square problem is a combinatorial puzzle that seeks to arrange symbols in a grid such that each symbol appears exactly once in each row and each column. This report will be dealing with the variation of the problem where a Latin square is an $n \times n$ array filled with n different symbols, with the numbers $\{1..n\}$ each occurring exactly once in each row and column. The problem was solved with two different methods – 1: with MiniZinc, a free and open-source constraint programming language, and 2: with a stochastic hill climbing (local search) algorithm in Python. The models and their implementations will be discussed along with experimental results and runtimes for various problem instances (e.g. N=3,5,10,15,20 or more). The performance and scalability for both methods will be investigated.

All performance tests were carried out with a MacBook Air (M1, 2020) with 16GB of RAM.

Solving with MiniZinc

This section will discuss and analyze the method of solving with a constraint programming language, in this case MiniZinc. The benefits of this approach include the overall efficiency, simplicity, abstraction and versatility provided, allowing the user to focus on defining constraints and letting the solver handle the specifics regarding the algorithm. This approach is likely to perform better than an inefficiently written custom algorithm, while taking significantly less effort to program.



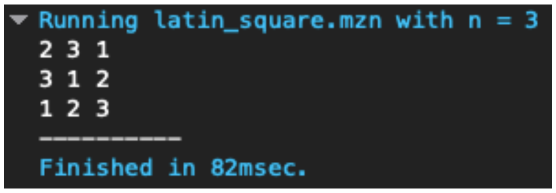
```
include "alldifferent.mzn";

int: n;
array [1..n, 1..n] of var 1..n: square;

constraint forall(i in 1..n)(alldifferent(square[i, 1..n]));
constraint forall(j in 1..n)(alldifferent(square[1..n, j]));

solve satisfy;

output ["\(square[i, j])" ++
        if j == n then "\n" else " " endif | i, j in 1..n];
```

Figure 1. MiniZinc program to generate a Latin square

As you can see, we only need a few lines of code in MiniZinc to successfully generate a valid Latin square. We first generate an initial array of n by n dimensions, with each integer from $\{1..n\}$ appearing once in each row. We then define the constraints of 'every element of each row and column must be unique'. Lastly, we solve.

All problem instances were solved with the default Chuffed 0.13.1 solver configuration.

With n = 3, we get the following result:

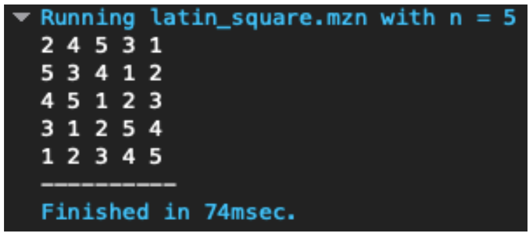


```
▾ Running latin_square.mzn with n = 3
  2 3 1
  3 1 2
  1 2 3
  ----------
  Finished in 82msec.
```

Figure 2. MiniZinc result with n = 3

---

n = 5:

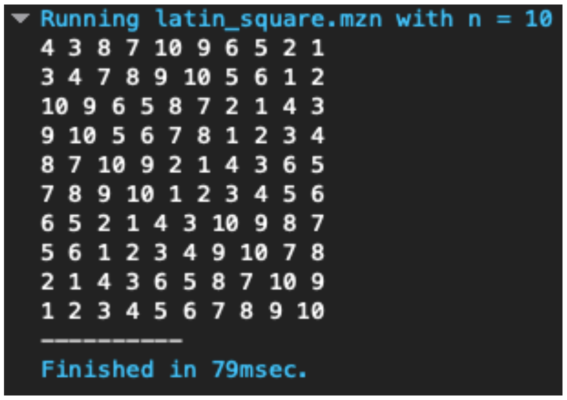


```
▾ Running latin_square.mzn with n = 5
  2 4 5 3 1
  5 3 4 1 2
  4 5 1 2 3
  3 1 2 5 4
  1 2 3 4 5
  ----------
  Finished in 74msec.
```

Figure 3. MiniZinc result with n = 5

---

n = 10:



```
▾ Running latin_square.mzn with n = 10
  4 3 8 7 10 9 6 5 2 1
  3 4 7 8 9 10 5 6 1 2
  10 9 6 5 8 7 2 1 4 3
  9 10 5 6 7 8 1 2 3 4
  8 7 10 9 2 1 4 3 6 5
  7 8 9 10 1 2 3 4 5 6
  6 5 2 1 4 3 10 9 8 7
  5 6 1 2 3 4 9 10 7 8
  2 1 4 3 6 5 8 7 10 9
  1 2 3 4 5 6 7 8 9 10
  ----------
  Finished in 79msec.
```

Figure 4. MiniZinc result with n = 10

Further results will be analyzed but not displayed.

Please note that while MiniZinc is deterministic by nature, it is possible to generate the entire set of solutions by modifying the solver configuration.

Tabulated solver runtimes in milliseconds

(averaged over 5 runs each)

***n* =**	**3**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
MiniZinc	58.6	62.8	64.3	63.2	69.2	81.8	94.8	112	140.8	170	311.2
Local Search	-	-	-	-	-	-	-	-	-	-	-

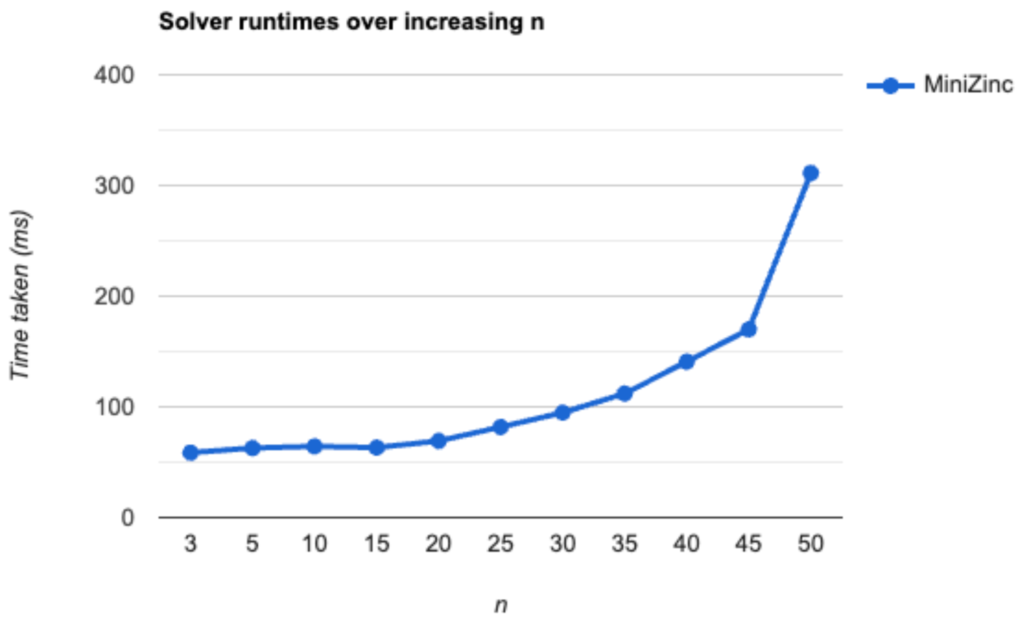


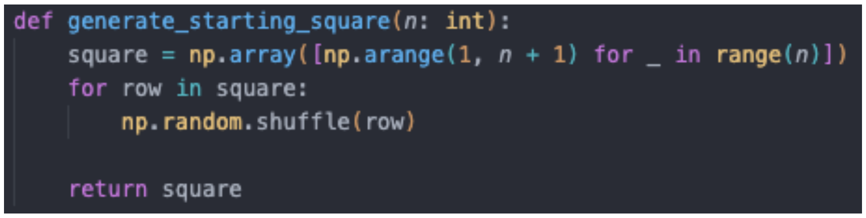
Figure 5. Graph of MiniZinc runtimes over increasing n

As we can see from the collected runtimes, although the Latin square is a combinatorial problem, the time complexity with the MiniZinc solver is far from exponential. This is likely due to the various optimizations MiniZinc performs under the hood and the relative simplicity of the problem.

Solving with Stochastic Hill Climbing (local search)

This section will discuss the method of solving with a custom-written stochastic hill climbing algorithm, a sub-specification of local search. Compared to using an existing constraint programming language, writing your own algorithm from scratch takes significantly more time to implement, and the resulting time complexity can vary drastically depending on the chosen heuristic function. However, with the right heuristic function, this approach has the potential to be very efficient.

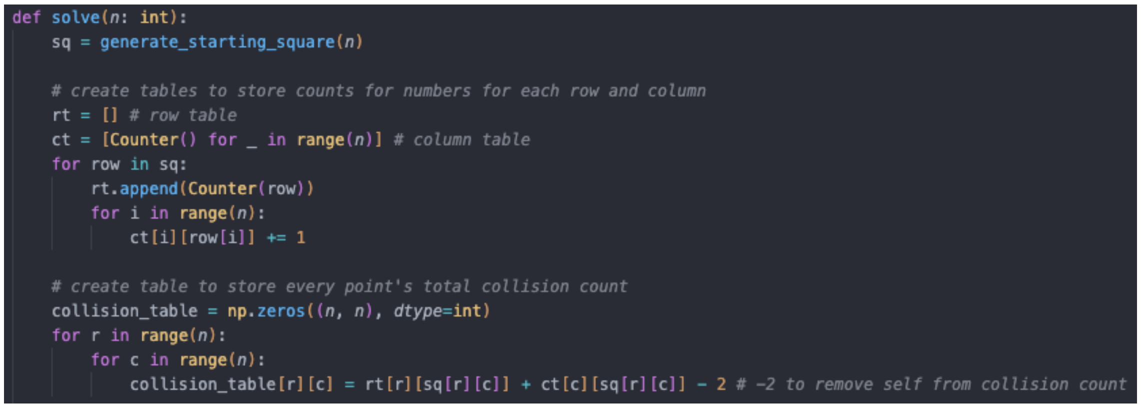
To start off, we first define a function to generate a starting configuration to solve into a Latin square. To generate a better configuration and make the algorithm non-deterministic, n rows of numbers $\{1..n\}$ were stacked then shuffled.



```python
def generate_starting_square(n: int):
    square = np.array([np.arange(1, n + 1) for _ in range(n)])
    for row in square:
        np.random.shuffle(row)

    return square
```

Figure 6. Function that generates a starting configuration to solve.

Next, we create three tables: one to store counters for each row, another of the same for columns, and lastly one to store every point’s total collision count for the current configuration. For the purpose of this report, a point’s ‘collision count’ will refer to the number of identical symbols in the same row **and** column, excluding the point itself. That is, a point is unique in its row and column if it has a collision count of 0.



```python
def solve(n: int):
    sq = generate_starting_square(n)

    # create tables to store counts for numbers for each row and column
    rt = [] # row table
    ct = [Counter() for _ in range(n)] # column table
    for row in sq:
        rt.append(Counter(row))
        for i in range(n):
            ct[i][row[i]] += 1

    # create table to store every point's total collision count
    collision_table = np.zeros((n, n), dtype=int)
    for r in range(n):
        for c in range(n):
            collision_table[r][c] = rt[r][sq[r][c]] + ct[c][sq[r][c]] - 2 # -2 to remove self from collision count
```

Figure 7. The start of the solve() function

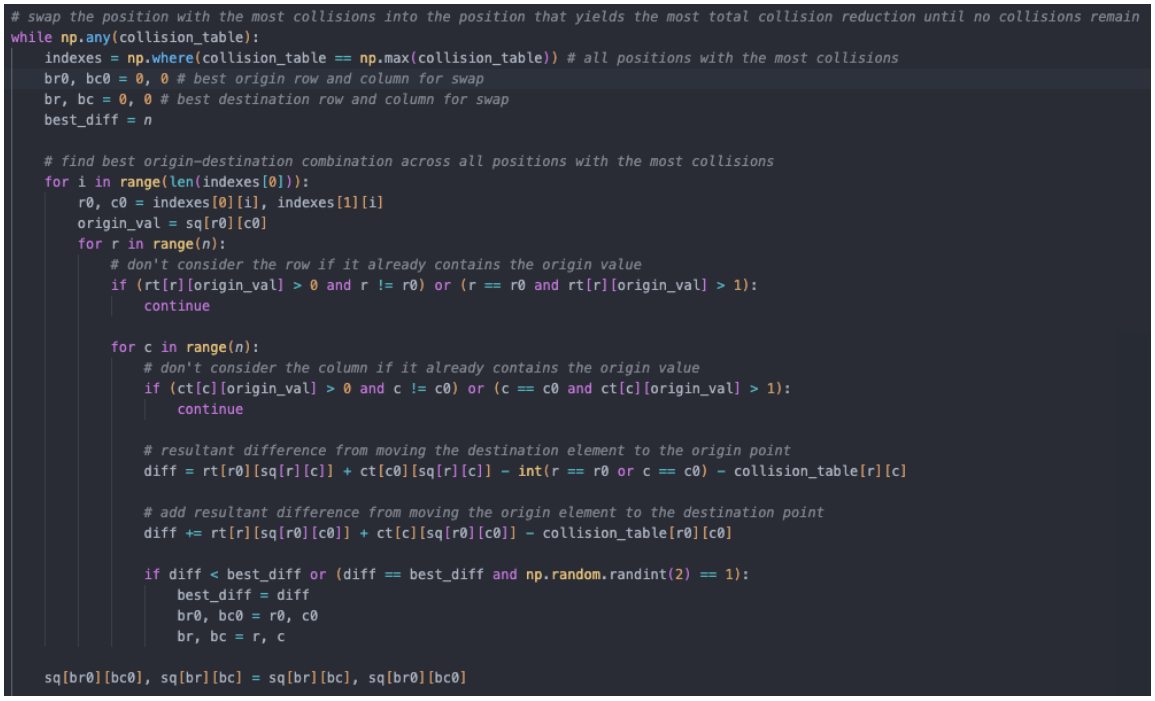
For the rest of the algorithm, two variations with very similar heuristics will be explored. The two heuristics can be described in the following manner:

1. Iterate through all points with the highest collision count, finding all potential swaps that would yield the highest global collision count reduction. Randomly choose one and swap. Repeat until solved.

2. Out of all points with the highest collision count, randomly select one. From the selected origin point, find and swap to the position that would yield the highest global collision count reduction. Repeat until solved.

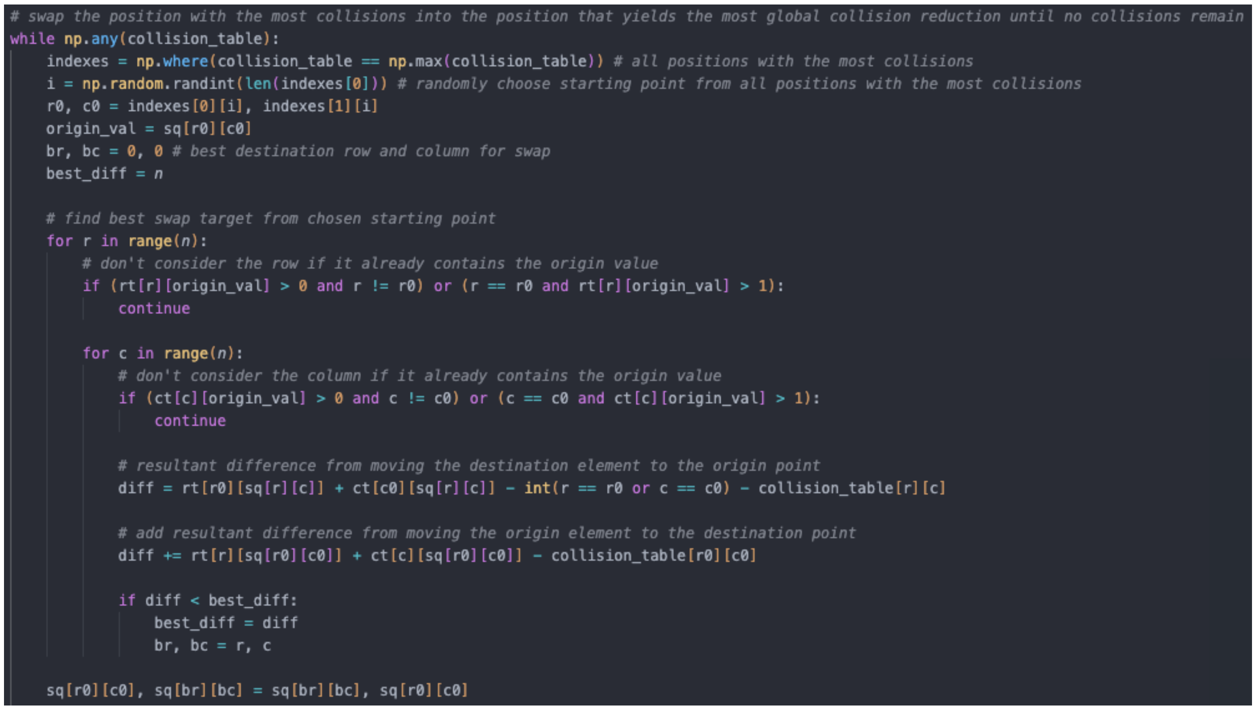
The Latin square problem has no local minima, with all solutions being equally correct/optimal, so normal hill climbing was considered at first. However, in situations where the best possible swaps resulted in no change of the global collision count, it was observed that the algorithm would get stuck in an infinite loop trying the same swaps, as it would always pick the first found ‘best swap’. Thus, random choice was introduced to break out of the infinite loops.

In the heart of the algorithm is the implementation of the previously stated heuristics. Because it was found that aiming to swap the origin into a position resulting in no collisions was always optimal **and** possible, rows and columns that already contain a copy of the origin symbol are skipped when searching for a swap target. Swaps will continue until the entire collision count table comprises of zeroes.



```python
# swap the position with the most collisions into the position that yields the most total collision reduction until no collisions remain
while np.any(collision_table):
    indexes = np.where(collision_table == np.max(collision_table)) # all positions with the most collisions
    br0, bc0 = 0, 0 # best origin row and column for swap
    br, bc = 0, 0 # best destination row and column for swap
    best_diff = n

    # find best origin-destination combination across all positions with the most collisions
    for i in range(len(indexes[0])):
        r0, c0 = indexes[0][i], indexes[1][i]
        origin_val = sq[r0][c0]
        for r in range(n):
            # don't consider the row if it already contains the origin value
            if (rt[r][origin_val] > 0 and r != r0) or (r == r0 and rt[r][origin_val] > 1):
                continue

            for c in range(n):
                # don't consider the column if it already contains the origin value
                if (ct[c][origin_val] > 0 and c != c0) or (c == c0 and ct[c][origin_val] > 1):
                    continue

                # resultant difference from moving the destination element to the origin point
                diff = rt[r0][sq[r][c]] + ct[c0][sq[r][c]] - int(r == r0 or c == c0) - collision_table[r][c]

                # add resultant difference from moving the origin element to the destination point
                diff += rt[r][sq[r0][c0]] + ct[c][sq[r0][c0]] - collision_table[r0][c0]

                if diff < best_diff or (diff == best_diff and np.random.randint(2) == 1):
                    best_diff = diff
                    br0, bc0 = r0, c0
                    br, bc = r, c

    sq[br0][bc0], sq[br][bc] = sq[br][bc], sq[br0][bc0]
```

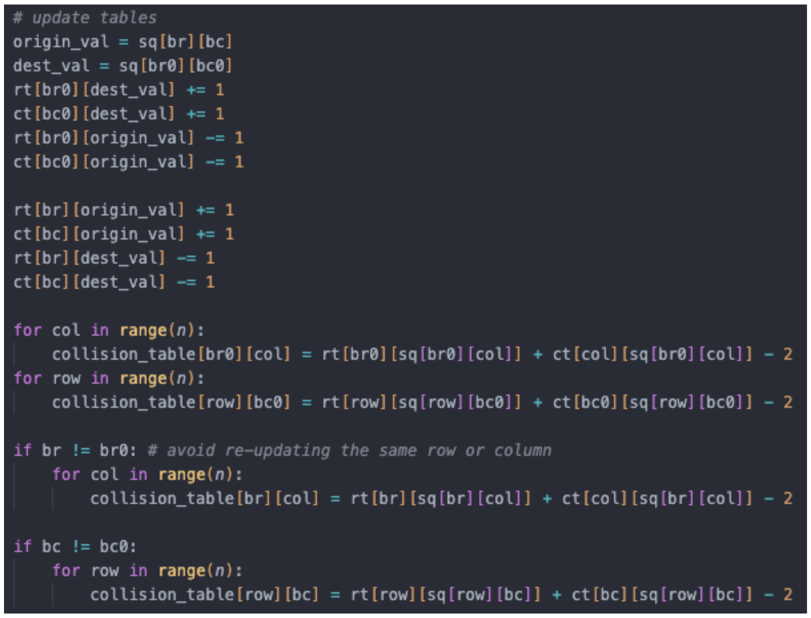
Figure 8. Core of local search algorithm (version 1)



```python
# swap the position with the most collisions into the position that yields the most global collision reduction until no collisions remain
while np.any(collision_table):
    indexes = np.where(collision_table == np.max(collision_table)) # all positions with the most collisions
    i = np.random.randint(len(indexes[0])) # randomly choose starting point from all positions with the most collisions
    r0, c0 = indexes[0][i], indexes[1][i]
    origin_val = sq[r0][c0]
    br, bc = 0, 0 # best destination row and column for swap
    best_diff = n

    # find best swap target from chosen starting point
    for r in range(n):
        # don't consider the row if it already contains the origin value
        if (rt[r][origin_val] > 0 and r != r0) or (r == r0 and rt[r][origin_val] > 1):
            continue

        for c in range(n):
            # don't consider the column if it already contains the origin value
            if (ct[c][origin_val] > 0 and c != c0) or (c == c0 and ct[c][origin_val] > 1):
                continue

            # resultant difference from moving the destination element to the origin point
            diff = rt[r0][sq[r][c]] + ct[c0][sq[r][c]] - int(r == r0 or c == c0) - collision_table[r][c]

            # add resultant difference from moving the origin element to the destination point
            diff += rt[r][sq[r0][c0]] + ct[c][sq[r0][c0]] - collision_table[r0][c0]

            if diff < best_diff:
                best_diff = diff
                br, bc = r, c

    sq[r0][c0], sq[br][bc] = sq[br][bc], sq[r0][c0]
```

Figure 9. Core of local search algorithm (version 2)

After the swap, we need to update the tables.

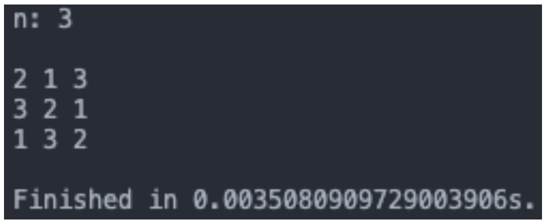


```python
# update tables
origin_val = sq[br][bc]
dest_val = sq[br0][bc0]
rt[br0][dest_val] += 1
ct[bc0][dest_val] += 1
rt[br0][origin_val] -= 1
ct[bc0][origin_val] -= 1

rt[br][origin_val] += 1
ct[bc][origin_val] += 1
rt[br][dest_val] -= 1
ct[bc][dest_val] -= 1

for col in range(n):
    collision_table[br0][col] = rt[br0][sq[br0][col]] + ct[col][sq[br0][col]] - 2
for row in range(n):
    collision_table[row][bc0] = rt[row][sq[row][bc0]] + ct[bc0][sq[row][bc0]] - 2

if br != br0: # avoid re-updating the same row or column
    for col in range(n):
        collision_table[br][col] = rt[br][sq[br][col]] + ct[col][sq[br][col]] - 2

if bc != bc0:
    for row in range(n):
        collision_table[row][bc] = rt[row][sq[row][bc]] + ct[bc][sq[row][bc]] - 2
```

Figure 10. Updating the tables after the swap

Finally, some results.

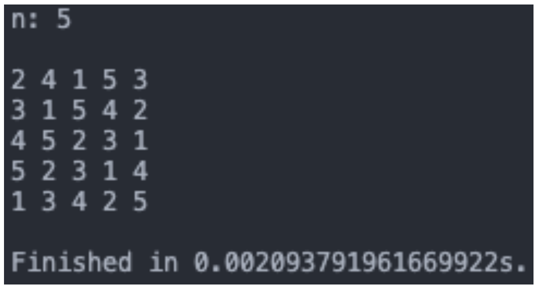
With n = 3, we get the following result:



```
n: 3

2 1 3
3 2 1
1 3 2

Finished in 0.0035080909729003906s.
```

Figure 11. Local search result with n = 3

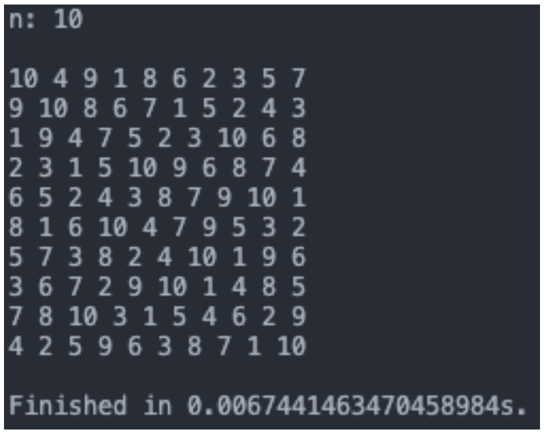
N = 5:



```
n: 5

2 4 1 5 3
3 1 5 4 2
4 5 2 3 1
5 2 3 1 4
1 3 4 2 5

Finished in 0.002093791961669922s.
```

Figure 12. Local search result with n = 5

N = 10:



```
n: 10

10 4 9 1 8 6 2 3 5 7
9 10 8 6 7 1 5 2 4 3
1 9 4 7 5 2 3 10 6 8
2 3 1 5 10 9 6 8 7 4
6 5 2 4 3 8 7 9 10 1
8 1 6 10 4 7 9 5 3 2
5 7 3 8 2 4 10 1 9 6
3 6 7 2 9 10 1 4 8 5
7 8 10 3 1 5 4 6 2 9
4 2 5 9 6 3 8 7 1 10

Finished in 0.0067441463470458984s.
```

Figure 13. Local search result with n = 10

Further results will be analyzed but not displayed.

The tests results shown here were produced by the version of the algorithm using the second heuristic.

Tabulated solver runtimes in milliseconds

(averaged over 5 runs each)

***n* =**	**3**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
MiniZinc	58.6	62.8	64.3	63.2	69.2	81.8	94.8	112	140.8	170	311.2
Local Search v1	1	1.1	7.3	25.3	75.8	196.2	434	932.3	1894.5	3349	5533.2
Local Search v2	0.8	0.9	4.4	13.2	18.9	35	42.4	76.3	91.7	129.9	198.9

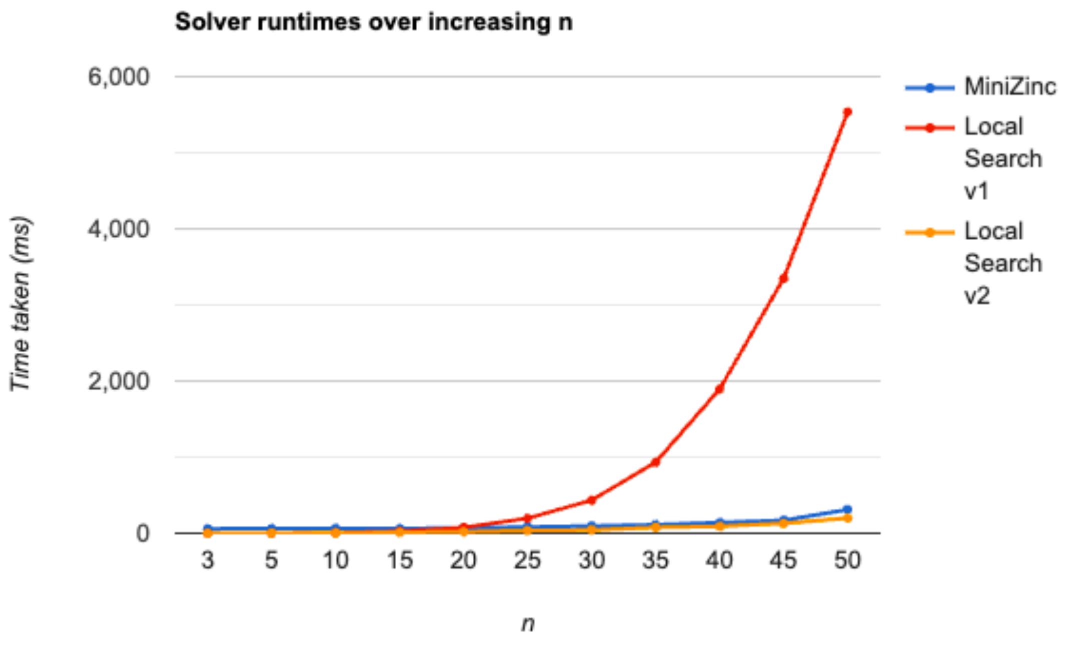


Figure 14. Graph comparing all three solver runtimes

By looking at the data, we can observe that while Local Search v1 performs well at lower values of n, it is not very scalable compared with MiniZinc and Local Search v2. This is due to trying to search for the best swap destination out of **all** origins with the highest collision count.

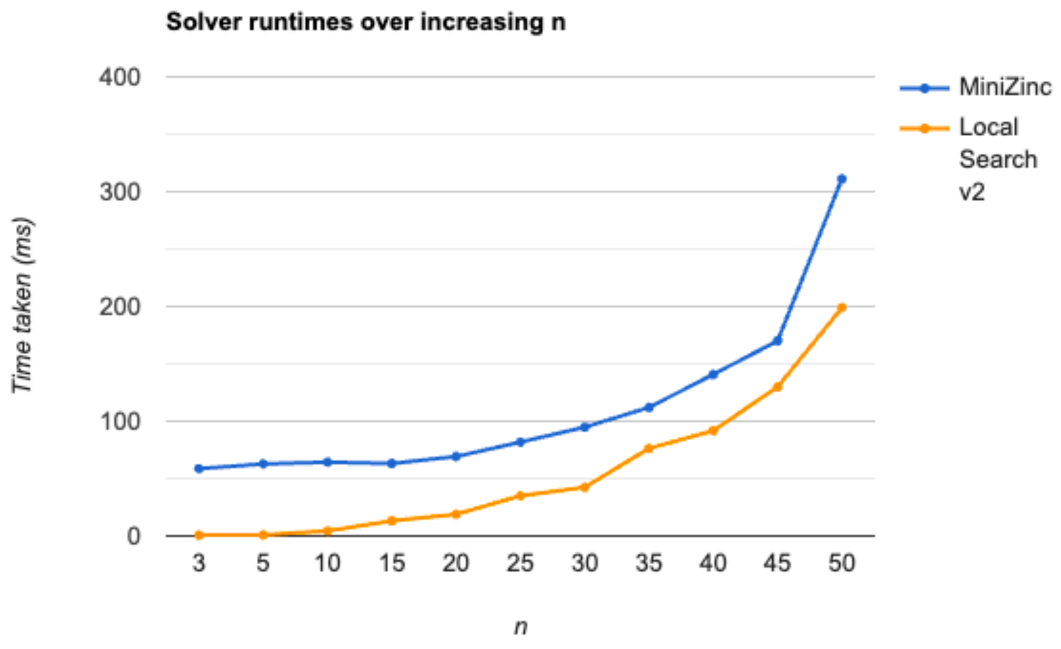


Figure 15. Graph comparing runtimes of MiniZinc to Local Search v2

On the other hand, we can observe that Local search v2 performs even better than MiniZinc at all values of n. This is because instead of trying to search from every highest collision count origin every iteration, we instead randomly choose one and only search for a swap target from the chosen origin. By moving the random component in this manner, we are still able to break out of infinite cycles through randomness while drastically improving the performance of our algorithm.

Solving the Latin square with MiniZinc vs Local Search in Python

Although using MiniZinc takes less programming time, effort, and is likely to result in an efficient solver, it is not the optimal method regarding time-complexity. While writing your own algorithm takes more time, effort, and is prone to mistakes, there is potential for it to solve faster than a constraint programming language such as MiniZinc.

Which method you choose will ultimately come down to how much time you have, how confident you are in your programming abilities, and whether you prioritize a shorter development time or a faster algorithm.