Final assignment: Latin Square Problem

Simon Andreas Glomb

Efficient Search Methods in Artificial Intelligence

Contents

1 Approach

The goal of this report is to describe my programming project for solving the Latin Square problem. It will cover the solving with Minizinc, a constrained modeling language as well as my development process for my own local search method. All approaches have been tested in experiments to test their scalability.

An approach was to run it on the IST cluster from the University of Tokyo [1]. The installation had quite many hurdles due to permissions. It was not possible to use sudo, that's why I could not use the recommended way of using bundled binary packages. I had to install by source code, i.e. using Make. I ran into errors on installation, some could be resolved by specifying a desired installation directory where I have enough permissions (i.e. using the following command: `cmake -DCMAKE_INSTALL_PREFIX=~/comb_opt/gecode/build ..`). In the end I could make it run on the IST cluster.

I implemented some test methods to validate functional correctness of some methods, the tests can be found in `tests.py`.

2 Minizinc

This section will describe the modeling and solving of the Latin Square problem using the constrained modeling language Minizinc.

2.1 Modeling

We first define our Constraint Satisfaction Problem (CSP) as a triple (V, D, C). All of our variables V are defined in the matrix "a". Each of the n^2 variables has the domain $1, ..., n$. All of this is defined in the first two lines of the minizinc model found in listing 5. `array[1..n,1..n]` defines the number of variables while `var 1..n` defines the domain.

Listing 1: Minizinc Model

```
int: n;
array[1..n,1..n] of var 1..n: a;
include "alldifferent.mzn";
constraint forall(i in 1..n) (
    alldifferent(j in 1..n) (a[i,j]) /\
    alldifferent(j in 1..n) (a[j,i])
);
solve satisfy;
```

I use the constraint `alldifferent()` which enforces that all elements in each row and column are different. The iteration with i does two things: `alldifferent(j in 1..n) (a[i,j])` iterates over the rows and makes sure that all elements in each row are distinct. `alldifferent(j in 1..n) (a[j,i])` iterates over the columns with the same i and makes sure that all elements in each column are distinct. The `/\` is a logical *and*. The line `solve satisfy;` specifies that one wants to find a solution regarding the constraints.

2.2 Solving

I chose the solver "Chuffed" as this was the solver I had the least problems to install on the IST cluster. All solutions have been generated with this solver. The disadvantage of this solver is that it does not provide parallel execution yet[2].

To get a deterministic solution, one can just run: `minizinc --solver Chuffed latin_square.mzn`

To get all the solutions, use: `minizinc --solver Chuffed -a latin_square.mzn`

In order to achieve non-determinism and get all solutions, I started to use python with the python package *pymzn*.

Listing 2: Python Code to evaluate Minizinc

```
import pymzn
import time
```

[1] https://www.i.u-tokyo.ac.jp/edu/facility/index_e.shtml
[2] https://github.com/chuffed/chuffed/issues/20

```
3
4  values = [5, 10, 15, 20]
5  with open('results.txt', 'w') as file:
6      for n in values:
7          data = {"n": n}
8          start_time = time.time()
9          solutions = pymzn.minizinc('latin_square.mzn', all_solutions=False,
               data=data, solver=pymzn.Chuffed())
10         end_time = time.time()
11         computation_time = end_time - start_time
12         print(computation_time)
13         file.write(f'n = {n}, time = {computation_time:.2f} seconds\n')
14         print(solutions)
15 print('Results saved to results.txt')
```

I print a random solution as output as it was a requirement that the solution is non-deterministic.

2.3 Performance

As a stress-test, I wanted to see how fast it is to generate all solutions.

As I ported the code to the cluster, it began to run into an error I could not fix. Therefore, I ported the python code to bash.

Listing 3: Bash script to evaluate Minizinc

```
1  #!/bin/bash
2
3  values=(3 5 10 15 20 25 30 35 40 45 50)
4  for n in "${values[@]}"; do
5      data="n=$n"
6      start_time=$(date +%s.%N)
7      minizinc --solver chuffed -a latin_square.mzn -D "$data" > /dev/null
           2>&1
8      end_time=$(date +%s.%N)
9      computation_time=$(echo "$end_time - $start_time" | bc)
10     echo "n = $n, time = $(printf '%0.2f' ${computation_time}) seconds"
11     #echo "$solutions" | shuf -n 1
12 done > results.txt
13
14 echo "Results saved to results.txt"
```

In the following we can see the runtimes for generating all solutions using the same solver but running it on the IST cluster:

The runtime results for generating all solutions from my machine can be found in the following:

Listing 4: Generating all solution on my Computer: Performance

```
1  n = 3, time = 0.77 seconds
2  n = 4, time = 0.84 seconds
3  n = 5, time = 64.83 seconds
```

Listing 5: Generating all solution on the IST cluster: Performance

```
1  n = 3, time = 0.09 seconds
2  n = 4, time = 0.15 seconds
3  n = 5, time = 27.35 seconds
```

We can see that the time is significantly lower, even without any parallelization. And using the cluster would have the option for parallelization, which is not possible on my machine. The foundation for parallelization is given, but beyond the scope of this report.

2.4 Experimental Results

The following experimental results have been generated on the cluster. The computation time is the time needed to find one solution.

```
n = 3, time = 0.09 seconds
n = 5, time = 0.08 seconds
n = 10, time = 0.09 seconds
n = 15, time = 0.15 seconds
n = 20, time = 0.25 seconds
n = 25, time = 0.86 seconds
n = 30, time = 31.00 seconds
n = 35, time = 27.62 seconds
n = 40, time = 683.60 seconds
n = 45, time = 1459.29 seconds
```

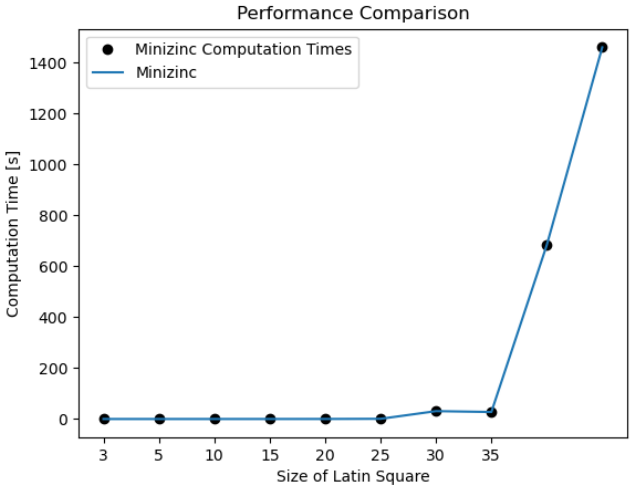

Figure 1: Scalability of Minizinc Approach using "Chuffed"

I could not resolve why for $n = 30$ and $n = 35$ the computation time was so close to each other. It was consistently that way over many runs. We see that with larger n the needed computation time goes up a lot. We will come back to the aspect of scalability of Minizinc in the comparison in section 4.

3 Constrained Optimization Model

This section will describe the modeling and solving of the Latin Square problem using a self-implemented local search method in python. The problem is modelled as a constrained optimization problem.

3.1 Choice of Local Search Method

Configuration: $(a_{11}, a_{12}, ..., a_{1j}, a_{2,1}, ..., a_{ij}, ..., a_{nn})$

Objective function: minimize the number of conflicts. Where we define the number of conflicts the following way: Conflicts occur when two or more numbers in a row or column are the same. More precisely, the number of conflicts is the number of elements for a column which are misplaced (e.g. having three times the number 1 in a column would have 2 conflicts, because we have to move 2 elements). This is a property that is very useful as can be seen in section 3.3.5.

But as I create the rows randomly which each element exactly once in a row, the only conflicts that can appear are in the columns. We will make use of that property that rows cannot have conflicts. One could also have initialized everything random, but then we would have less structure.

This structure had led to the idea of using fixpoint-free permutations (derangements) as a basis of most local search methods I will propose. The idea is to see two rows as a permutation. If the permutation is fixpoint-free (i.e. a derangement), then all rows and columns are distinct for these two rows. If this property holds pairwise for all elements, then we would have solved the Latin Square problem

Neighborhood: Looking at all possible combinations of rows is not feasible. Therefore, we reduce our neighborhood to the derangements of rows (fixpoint-free permutations). As we know that each row has every element exactly once, a derangement can be computed. Therefore we search for a fixpoint-free permutation to get another row without generating a conflict to the row used to generate the derangement.

Computing a derangement is not too hard, because, with larger n, the probability of a random generated sequence being a derangement approaches $\frac{1}{e}$ [3]. This means that, in expectation, we have to generate e samples to find such a derangement. This is exactly how I implemented the generation of a fixpoint-free permutation.

[3] https://en.wikipedia.org/wiki/Derangement

The function `fixpoint_free_permutation` can be found in *utils.py*. The code shuffles/permutes a copy of the row and then checks for that instance if it is fixpoint-free. The loop only stops if we found a derangement.

We can describe the whole neighborhood the following way: All derangements for each row which replace a row. More formally the neighborhood can be described as: $\{M|$ matrix $M, M[i] = Derangement(M[j]), i \in [n], j \in [n]\}$. Therefore a local search does the following: Compute a derangement for one or more rows and pick a row to replace.

3.2 Local Search Strategies

I have tried out multiple strategies as a local search. The strategies can differ in the following properties:

- Selecting a row to compute a derangement
- Selecting a row to replace with the computed derangement

The main theme of the strategies is that we have solved the problem if and only if all rows are pairwise fixpoint-free permutations to each other. Therefore, we try to find fixpoint-free permutations which reduce the number of conflicts.

It was clear that the methods can get stuck at a local optimum. See fig. 4 for an example of a method being stuck in a local optimum.

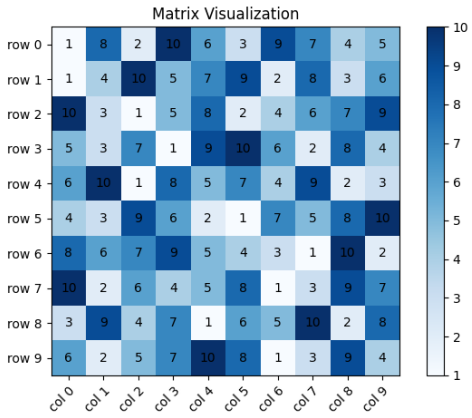

Figure 2: Local optimum with 20 conflicts. One can see that conflicts only occur withing the columns.

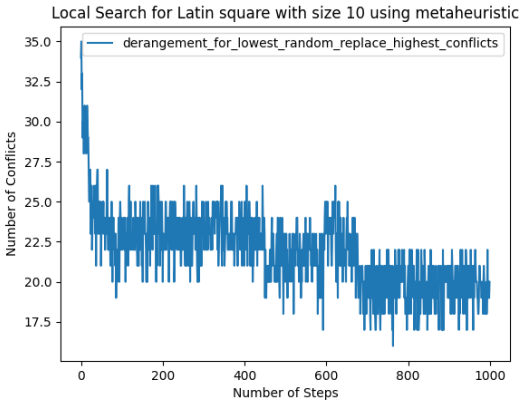

Figure 3: Performance: A local search method stuck in a local optimum. One can also see that the method takes choices which increase the number of conflicts.

Figure 4: Local search stuck in a local optimum

So almost all of the upcoming search strategies are a method to find a derangement. In my code I refer to them as `local_search_strategy` and they return an `index_to_replace` which tells us where to put a new row and a `new_row` which is the row that will be placed at the row with index `index_to_replace`.

It is implemented in such a way that one can combine every local search strategy with every metaheuristic. The metaheuristic uses the local search strategy as an argument. Everything is implemented in Python. As I did not know how to reach a global optimum with a local search in advance, I tested out many local search ideas. The implementation and evaluation of these ideas is a major part of this programming project.

3.2.1 Random Strategy

Function name: `random_strategy`

For comparison reasons, I implemented a random strategy which computes a derangement for a random row and replaces a random row. I use the random module from python to generate random indices.

3.2.2 Derangement for row with lowest conflict

For this idea I implemented multiple functions which all have a similar functionality. All the functions here do have a neighborhood size of 1.

`derangement_for_lowest_replace_highest_conflicts`

My first idea was to compute a derangement for the row with the lowest number of conflicts and replace the row with the highest number of conflicts by the derangement. We use `fixpoint_free_permutation` to compute such a derangement. The other methods are implemented similarly.

`derangement_for_lowest_random_replace_highest_conflicts`

Compute a derangement for a row with lowest number of conflicts and replace the row with the highest conflicts. Differs to previous method in the way that if multiple lowest number of conflict rows are present, picks one at random.

`derangement_for_lowest_replace_highest_conflicts_random`

Compute a derangement for the row with lowest number of conflicts and replace a row with the highest conflicts. Differs to previous method in the regard that when multiple highest number of conflict rows are present, picks one at random.

`derangement_for_lowest_random_replace_highest_conflicts_random`

Compute a derangement for a row with lowest number of conflicts and replace a row with the highest conflicts. Unlike previous methods, when multiple rows with the lowest number of conflicts are present, one row is picked at random. Additionally, when multiple highest number of conflict rows are present, one row is picked at random.

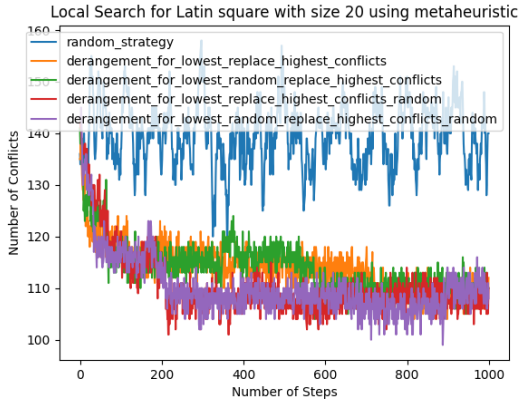

Figure 5: Performance: Comparison for described methods for $n = 20$ and $n\_iter = 1000$

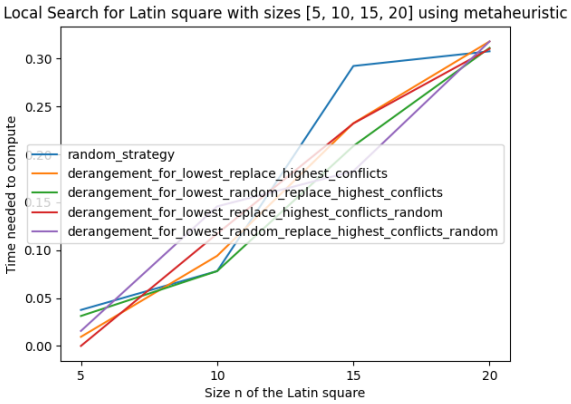

Figure 6: Scalability

Figure 7: Experiment 1: Derangement for row with lowest conflict

How to evaluate

One can create an evaluation through the use of following command
`python fixpoint_free.py --n 5 10 15 20 --n_iter 1000` Which will compute a Latin Square with the methods set in code and set the maximum iterations to 1000.

Evaluation

A resulting image can be seen in fig. 5. We can see that the methods do perform significantly better than random. We also see that the difference in performance between the different methods is not very large. Therefore, we will select the method `derangement_for_lowest_replace_highest_conflicts` for following evaluations as reference, because it is the simplest.

When looking at the computation times, they are all similar and all scale linearly to the size of the problem. Computing a derangement is also in $O(1)$. The strategies itself are also in $O(1)$ (e.g. selecting the row with the lowest conflicts), but the linearity comes from the fact that I compute the number of conflicts for each row. So as a whole we are in $O(n)$.

3.2.3 Derangement for row with lowest conflict + Neighborhood

These methods here are similar to before, but have a neighborhood size of $O(n)$ as they check the objective function for all rows.

`lowest_conflict_derangement_neighborhood`

Compute derangement for the worst row, and then check for all rows which replacement brings the most improvement regarding the objective function.

We can see in fig. 11 that the performance is improved by looking at a larger neighborhood.

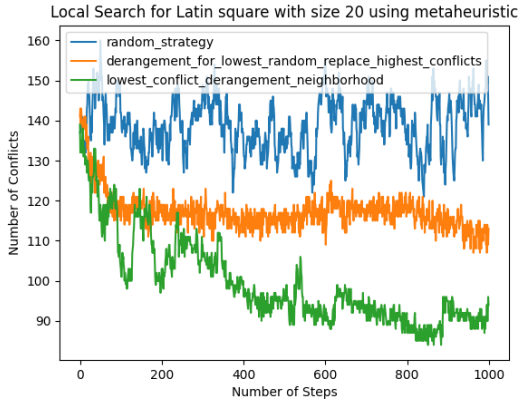

Figure 8: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

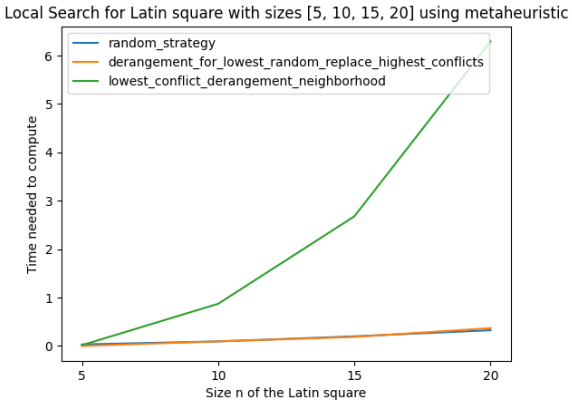

Figure 9: Scalability

Figure 10: Experiment 2: Derangement for row with lowest conflict + Neighborhood

But when looking at the time consumed to get to the we see a huge difference. The method from before took 0.33 seconds and the method which checks the neighborhood took 6.88 seconds for a $n = 20$. This means that the consumed time is over 20 times higher. Additionally, it is clear that the method is in $O(n^2)$ which makes a huge difference to the methods scaling linearly.

3.2.4 Derangement for row with highest conflict

`derangement_two_worst_solutions`

Compute a derangement for the worst one and replace the second worst one.

`derangement_two_worst_solutions_random`

Compute a derangement for the worst one and replace the second worst one. Picks them at random if multiple times the same value.

3.2.5 Derangement for row with highest conflict + Neighborhood

`most_conflict_derangement_neighborhood` We select the row with the highest number of conflicts and then compute a derangement for this row. Then we exchange the row such that our objective function is improved the most. So the neighborhood has n elements, and is the set of squares where we replace a row i with the computed derangement. In the code I iterate create a copy and iterate over all possible row changes.

Analysis of Method

This idea makes sense as this will fix most conflicts that the row with the highest conflicts has. The problem of this idea is that it can introduce many new problems as the derangement can still conflict with all other rows. Or more formally one can say that if the highest number of conflicts of a row is k, then it is an improvement if we do introduce less than k new conflicts. But the latter is not guaranteed.

As we know that the probability of a row being a derangement to another row is $\frac{1}{e}$. The probability of a row being a derangement to all n rows is $\frac{1}{e^{n-1}}$. Which means that we would have to generate e^{n-1} samples in expectation to find such a derangement. As we need exponentially many samples.

Given a row, the expected number of fixpoints (conflicts) when generating another row randomly is $\frac{1}{n}$ at each position. This means that looking at row 1 we have in expectation 1 fixpoint in each other row j ($j > 1$).

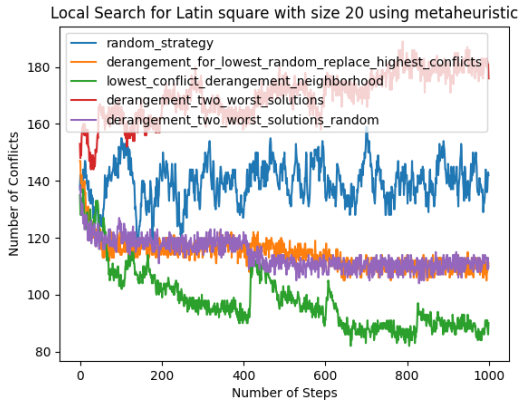

Figure 11: Performance: Comparison for described methods to previously selected method for $n = 20$ and $n\_iter = 1000$

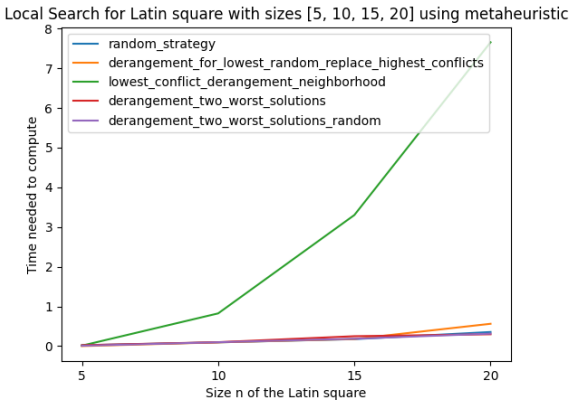

Figure 12: Scalability

Figure 13: Experiment 3: Derangement for row with highest conflict

This means that for row 1 we have in expectation $n-1$ fixpoints (conflicts). As we do not want to count each conflict twice, we now compare for row 2 each $j > 2$ and have in expectation $n-2$ conflicts for row 2. From here we can easily see that the number of fixpoints in the randomly generated matrix is described by $\sum_{k=1}^{n-1} k = -n + \sum_{k=1}^{n} k = -n + \frac{n^2+n}{2} = \frac{n^2-n}{2}$ using Gauß's sum.

And as we can mostly fix n conflicts by one operation but introduce in expectation $\frac{n^2-n}{2}$ we would not find a goal if we would choose a row randomly to replace. Therefore, we look at the whole neighborhood to choose the best row to replace.

Using Randomness

At this moment, I had preliminary results which indicated that this method has severe problems escaping local optima. Randomness is known to be used to escape local optima. I had several ideas to escape local optima using randomness.

`most_conflict_derangement_neighborhood_rand1` First idea for randomization was to randomize if we always use the row with the highest conflicts as a basis to compute a derangement. There is a 50% chance of using it, and a 50% chance of using a random row. When we look at the results, there is a small to no improvement to the normal method as seen in fig. 14.

`most_conflict_derangement_neighborhood_rand2` The second idea was to pick a random row to replace if no element in the neighborhood was strictly better than the current configuration.

`most_conflict_derangement_neighborhood_rand3` The third idea was the combination of the first and second idea.

As we can see in fig. 14, the second and third idea are only slightly better than a completely random approach. The other randomized approaches also do not exceed the deterministic method. So the observation is that the idea of using randomization did not help to escape local optima.

3.2.6 Large Neighborhood

These methods have the property of looking at a neighborhood of size $O(n^2)$. `all_possible_derangements` The idea is quite similar to `most_conflict_derangement_neighborhood`, but we don't always take the most conflicting row to compute a derangement. We check for each row its derangement and then check for each row how the value would change if we would replace it. The downside of this approach is that we have to check $O(n^2)$ neighbors.

`all_possible_rand` This method does not compute a derangement, but just generates a random row and checks which row is the best to replace. It is more of a reference method.

At this point I had the idea of computing derangements for the columns. But this idea is not usable. The problem is that if I get a column "[1, 3, 3]", we cannot find a derangement of this. So the essential property of

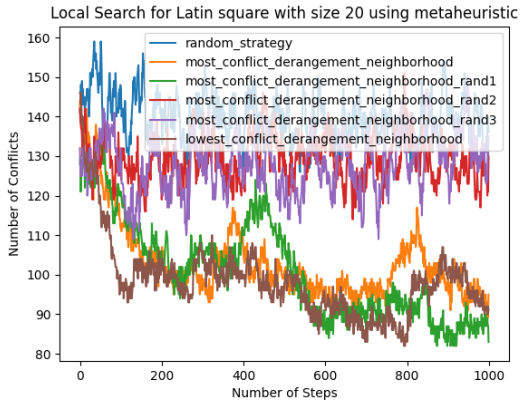

Figure 14: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

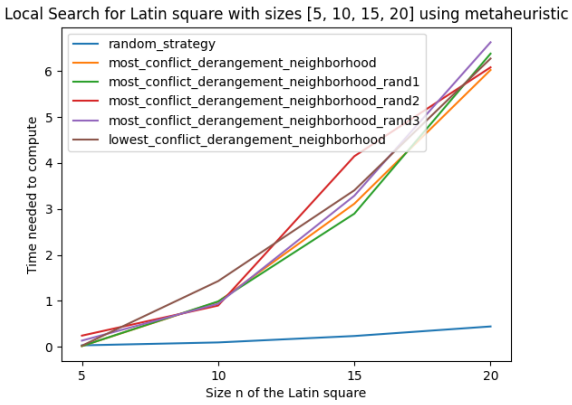

Figure 15: Scalability

Figure 16: Experiment 4: Derangement for row with highest conflict + Neighborhood

the rows is not given for the columns. A value can be multiple times in a column. Therefore, I had to develop a method to resolve multiple elements in each column without destroying the property that each row has unique elements. What I tried is the following: Assume you have a matrix and there occurs the following in a column [..., 6, ..., 6, ...], then just swap 2 elements in one row with such a 6 (only 2 elements to shuffle, to not create too many new conflicts). This is how this method was designed. This thought process led to the next idea I implemented.

`all_possible_derangements_swap` The idea is similar to `all_possible_derangements`, but with 50% chance, swap two elements which are both conflicts.

`all_possible_derangements_shuffle` Idea similar to `all_possible_derangements`, but shuffle a row that makes problems. Shuffle means her that we just give all the elements a new random position in the row.

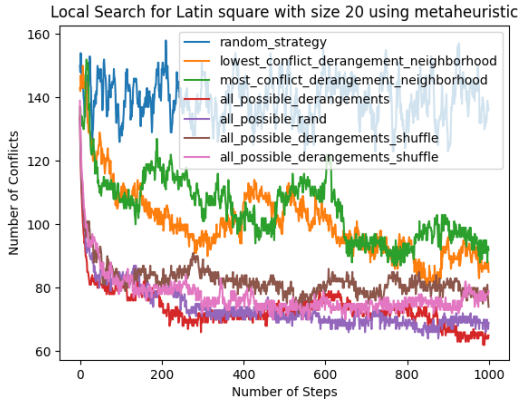

Figure 17: Performance: Comparison for described method to previously selected methods for $n = 20$ and $n\_iter = 1000$

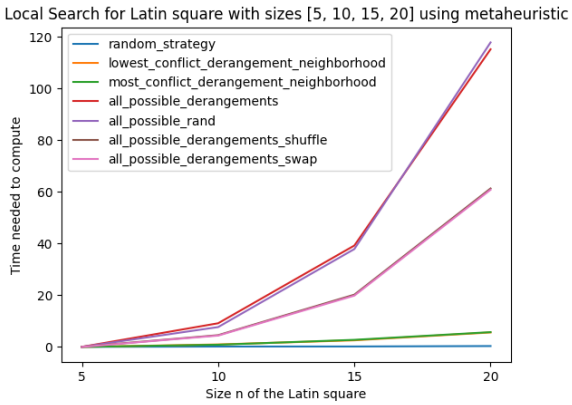

Figure 18: Scalability

Figure 19: Experiment 5: Large Neighborhood

3.2.7 What I thought until now

With all the experiments, I realized that the different local search ideas I had that they neither bring me the results I wanted nor did they help me find a global optimum. This was a major problem I encountered. Therefore, I had to develop metaheuristics which help me find a global optimum.

3.3 Metaheuristics: Escaping Local Optima

The functions I present in this section serve for applying the local search strategy. They are on the level of metaheuristics as they do steer how the `local_search_strategy` is used.

Each of the following metaheuristics count the number of iterations and stop when a global optimum is found or the maximum number of iterations is reached. When one looks into the code, one can also see that they serve me for computing our objective function in each iteration so that we can evaluate the methods.

For the following evaluations, I did choose several methods from before and now just change the metaheuristic used.

3.3.1 Base Case

`metaheuristic` This metaheuristic just applies the specified local search method in each iteration. This can be seen as the base case, it was used for all the experiments before.

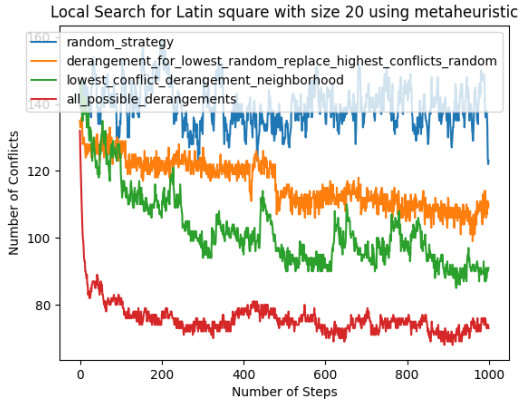

Figure 20: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

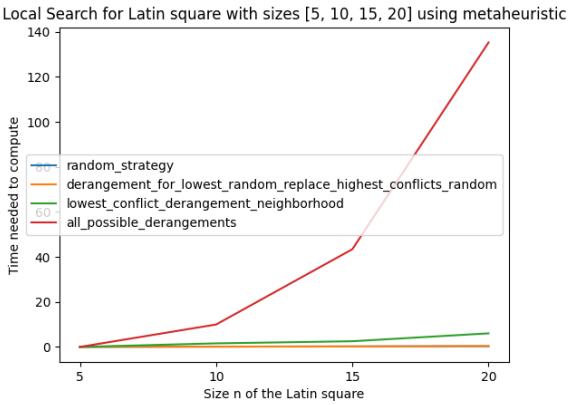

Figure 21: Scalability

Figure 22: Experiment Metaheuristics 1: Comparing Selected methods with our base case metaheuristic

3.3.2 Simulated Annealing

`metaheuristic_SA` This method implements a form of simulated annealing, to better find global optimum. It does not use a derangement strategy. The implementation goes as follows, if the objective function difference is positive, then we just take the row. Otherwise, we do have an acceptance probability with respect to the temperature and the objective function difference. This dictates how likely we are to take a step in a direction which does not improve our objective function directly. The temperature changes with the cooling rate. So this is quite a classical implementation of the idea of simulated annealing.

When comparing fig. 20 with fig. 23 we can see that this method is not better than the base case. One has to note that the runtime is way faster than before.

`metaheuristic_shuffle` This method just applies the `local_search_method` with 50% chance and with 50% chance shuffles two elements in a row when the element happens to be a duplicate for the column (so that we switch). The latter is inspired by the local search `all_possible_derangements_swap`. We pick a column with the maximum number of conflicts randomly (if there are multiple ones with the maximum number of conflicts).

Using shuffle on all rows with at least one conflict did not help with escaping local optima as can be seen in fig. 26. We can just see a significant increase in fluctuations compared to fig. 20

3.3.3 Random Restarts

`metaheuristic_random_restart`

This method has random restarts for the rows which have at least one conflict. The random restart is triggered every 100 iterations.

fig. 29 clearly shows that it does not help so escape local optima.

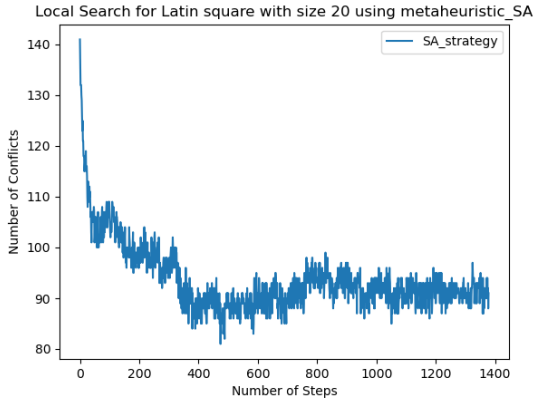

Figure 23: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

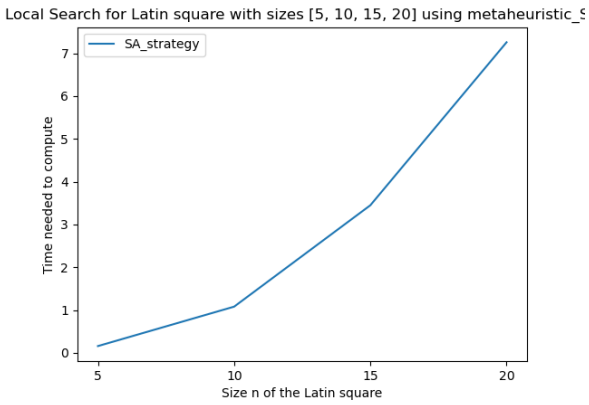

Figure 24: Scalability

Figure 25: Experiment Metaheuristics 2: Comparing Selected methods with simulated annealing metaheuristic A temperature of 1000 and a cooling rate of 0.05 was chosen for this evaluation

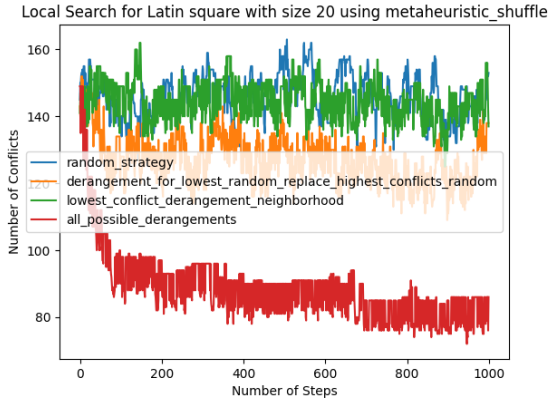

Figure 26: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

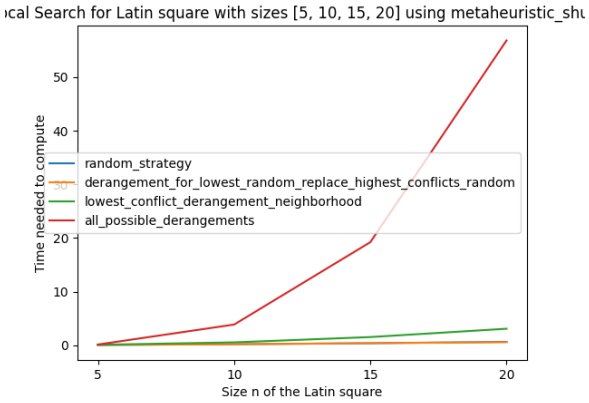

Figure 27: Scalability

Figure 28: Experiment Metaheuristics 3: Comparing Selected methods with metaheuristic shuffle

3.3.4 Some Remarks on Random Restarts

At this point I started doing a bit of an analysis on why random restarts did not help me escape local optima. The problem can be illustrated on the minimal example of having only one row wrong. When we would only use random restarts to fix it (so not computing a derangement), we would need e^{n-1} restarts. As this grows exponentially with growing n, the method is not scalable. E.g. for $n = 20$, in expectation we would already have to do 178482301 restarts to find the one configuration that fits. And in our example, if I do this every 100 iterations, it is quite clear that this does not help us escape the local optimum. This is also the explanation of why my former local search method did not really benefit from randomization.

Some remarks on using this metaheuristic with methods. Experiments showed the following: The method `most_conflict_derangement_neighborhood` only reduces the number of rows with conflicts. This means that `most_conflict_derangement_neighborhood` with random restarts will eventually find a solution.

Another property I wanted to point out is the following: The method `all_possible_derangements` can also increase the number of rows with conflicts. This shows that the choice for the local search method can play a huge role in interplay with the respective metaheuristic. One can see some numbers from an experiment in fig. 34. This means that the number of rows to restart is not strictly decreasing.

This is best illustrated on a small example. In my example, rows have a red background when they have at least

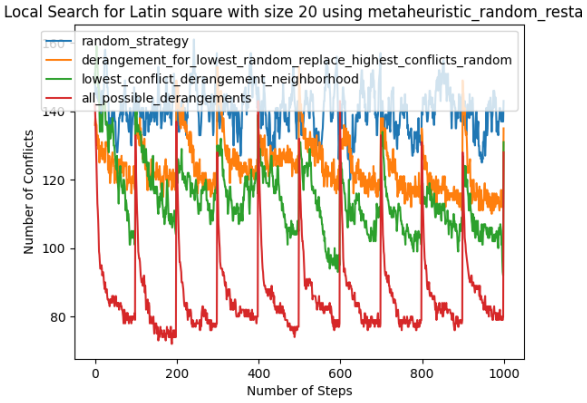

Figure 29: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

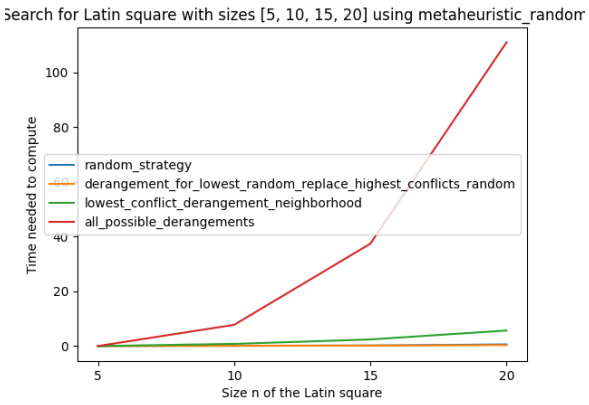

Figure 30: Scalability

Figure 31: Experiment Metaheuristics 4: Comparing Selected methods with random restarts every 100 iterations

```
Number of Conflicting Rows:16
Number of Conflicting Rows:15
Number of Conflicting Rows:15
Number of Conflicting Rows:14
Number of Conflicting Rows:13
Number of Conflicting Rows:13
Number of Conflicting Rows:13
Number of Conflicting Rows:13
Number of Conflicting Rows:13
Number of Conflicting Rows:13
```

Figure 32: most_conflict_derangement_neighborhood

```
Number of Conflicting Rows:18
Number of Conflicting Rows:16
Number of Conflicting Rows:18
Number of Conflicting Rows:17
Number of Conflicting Rows:18
Number of Conflicting Rows:17
Number of Conflicting Rows:17
Number of Conflicting Rows:18
Number of Conflicting Rows:17
Number of Conflicting Rows:18
```

Figure 33: all_possible_derangements

Figure 34: Experiment on Choice of Local Search for Randomness: These findings were also observed during longer runs

one conflict, while rows with a green background are conflict-free. The dark red entries indicate the conflicts. The light red cells are no conflict, but they are in a row which has conflicts.

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

random restart of second and fourth row →

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

3.3.5 Tabu Search

`tabu_search` I called this method `tabu_search` because that is where I got the inspiration from. One could also view at as a "restricted random restarts" method.

The idea is that every 100 iterations, we remember which rows are conflict-free. It is inspired by the non-successful application of random restarts to reduce the number of viable restarts.

We make use of the property that the rows without conflicts do not have to be changed anymore (this is also an advantage of my method of counting conflicts, when the number first occurs it is not seen as a conflict, only the row where the number occurs the second or subsequent times is seen as a conflict). We do not have to change them to find the optimal solution. And, the first row will always be without conflicts, by definition of the way we count conflicts.

The method works the following way: We can save for each column all the numbers which have been used in the rows which will not change anymore. And now the number of possible entries when doing a random restart is reduced a lot (so we have a way smaller neighborhood), because we cannot use the ones already used in the

column by the rows which will not change. The combinatorics still say that the number of possibilities is still exponential with respect to n, but it is greatly reduced.

This also has the advantage that compared to a naive tabu search, we do not have to save things for every entry in the square, but only for the columns.

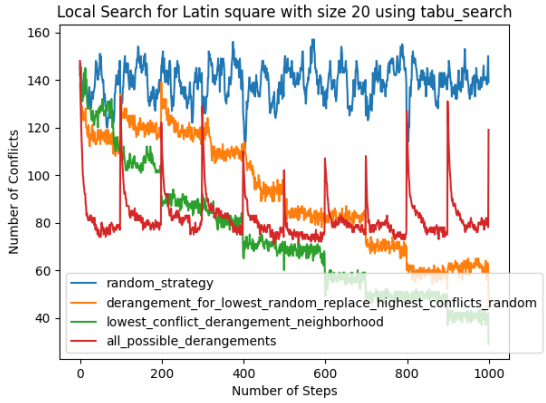

Figure 35: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

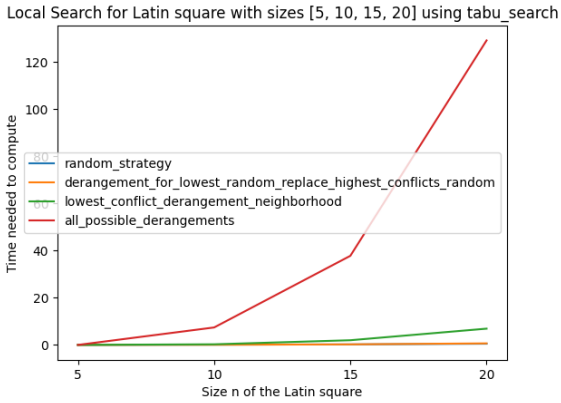

Figure 36: Scalability

Figure 37: Experiment Metaheuristics 5: Comparing Selected methods with tabu search

When looking at the performance results, we can clearly see that the proposed method does significantly help us with escaping local optima. In the diagrams before, the best methods could only get down to a number of conflicts of 80. Here we see that the best method achieves 40 conflicts. And with more iterations it would probably find a global optimum, which led to the next method.

`only_tabu_search` This method does not use a local search method from the previous section. It just applies the method described before in every iteration. The advantage is that with the method described that the neighborhood is quite small. I am aware that this is not a local search method regarding derangements in the sense I have explained at the beginning. But as a derangement was also just computed by generating random samples, so it still is the same approach.

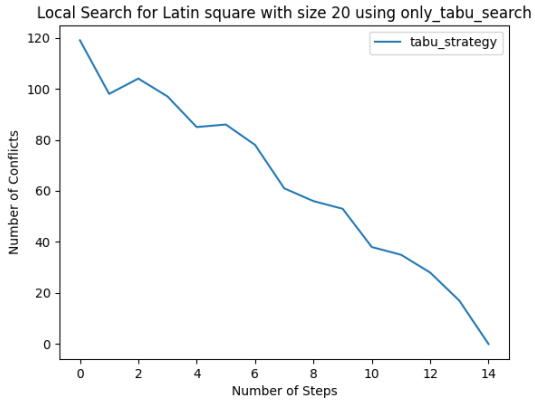

Figure 38: Performance: Comparison for described method to the previously selected method for $n = 20$ and $n\_iter = 1000$

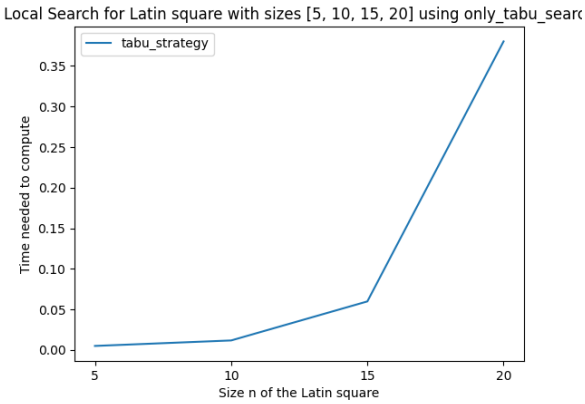

Figure 39: Scalability

Figure 40: Experiment Metaheuristics 6: Comparing Selected methods with only tabu search

When looking at fig. 38, we can see that we do find a global optimum in quite a few iterations. And looking at fig. 39 we see that the time needed to reach it is very low. This is by far the most promising method, as it is the only method reaching a global optimum in a reasonable amount of time. The method `tabu_search` would be the only other method which would find a global optimum consistently, but it is the same idea why it finds it.

3.3.6 Implementation of *tabu_search_only*

As this is the most important method in this report regarding finding a global optimum, I will describe its implementation more in detail.

The first thing one had to do was to save for each column which numbers are already occupied: This is done in `metaheuristics.py` in the `only_tabu_search` where we compute a `constraints_dict`. We first need a dict `curr_number_of_conflicts_row` which tells us how many conflicts each row has. Now we check all rows without conflicts, and then iterate over the columns of that row and appends the entries to a list of the `constraint_dict` for a certain column. The snippet which does this can be found here:

```
constraints_dict = {i: [] for i in range(0, n)}
for row, conflicts in enumerate(curr_number_of_conflicts_row):
    if conflicts == 0:
        for col in range(0, n):
            constraints_dict[col].append(square[row][col])
```

Furthermore, we iterate over all rows which have at least one conflict and assign a new row to our current solution. Computing the new row is the most interesting part, therefore I have to explain how to create a restart with the given constraints.

One can find the implementation in `utils.py`, both as recursion `create_restart_with_constraints_recursion` and as iterative approach `create_restart_with_constraints`.

At first, I did it with recursion but during later experiments I saw the following: On n=25 I encountered maximum recursion depth in python (as Python does not support proper tail-recursion optimization) so I had to also write the method without recursion. Both implementations can be found in `utils.py`. I will now explain the recursive approach as it is the more intuitive approach.

The code can be found here:

```
1  def create_restart_with_constraints_recursion(n, constraints_dict):
2  all_possible_numbers = range(1, n+1)
3  arr = np.random.permutation(n) + 1
4  already_used_in_this_row = set()
5  for col, constraints in constraints_dict.items():
6      available_numbers = set(all_possible_numbers) - set(constraints) -
           already_used_in_this_row
7      if available_numbers == set() and len(constraints)<n-1:
8          return create_restart_with_constraints_recursion(n,
               constraints_dict)
9      arr[col] = np.random.choice(np.array(list(available_numbers)))
10     if len(available_numbers) == 1:
11         arr[col] = list(available_numbers)[0]
12     already_used_in_this_row.add(arr[col])
13 return arr.tolist()
```

We do have our `constraints_dict`. We will save the numbers which have already been used in a row, so that we will never create duplicates in a row. And then we iterate over all columns and use its `constraints` (line 5). We can compute the numbers which are available for a given cell as all numbers which are not in the `constraints` and which are not `already_used_in_this_row`.

And if there are no numbers available, then we know that we made a wrong choice before in this row (because the Latin Square is at any stage solvable). Therefore, we just call the function again to reset the row.

In this is not the case, we know that we can pick a number for this cell and do this randomly from the available numbers. As `np.random.choice` gave me some errors if there is only one option, I had to include the code from line 10 - 11. After this, we can add the selected number to the list of numbers already in this row. After this the next iteration starts for same row and the next column (so the cell right to the cell we just looked at).

As explained, if a wrong choice is made, the method just starts again through the recursive call. This is one of the main ideas besides saving the constraints for each column.

If all choices that have been made are viable, we return our solution.

3.4 Overview

We saw that the biggest hurdle I encountered was that my local search methods did not find a global optimum for a larger n. This was successfully overcome through an approach similar to Tabu Search.

Regarding the metaheuristics part, we can see in fig. 21 that `all_possible_derangement` scales really badly because of the exploration of the large neighborhood. Also, we could observe that my simulated annealing approach did not escape local optima sufficiently well. The metaheuristic `metaheuristic_shuffle` also did not bring the quality that we wanted. The same holds for `metaheuristic_random_restart`. Using `tabu_search` did bring a significant improvement to the methods from before, but when looking at `only_tabu_search` it is not even close, neither in the number of conflicts nor the time needed to compute the Latin Square.

So the method `only_tabu_search` is outperforming everything I have implemented. This is more of an analytical method than a local search as we do not use the neighborhood or the objective function to compute the next neighbor. This approach works so well because it cannot get stuck in local optima.

4 Comparison: Minizinc and Local Search

This section compares Minizinc against `only_tabu_search`. I chose `only_tabu_search` because it is the only method reliably finding a global optimum in a reasonable amount of time.

4.1 Scalability

This section will compare the scalability of the Minizinc approach and my local search method. All the following code has been run on one CPU of the IST cluster.

I will now compare the computation for the generation of one random Latin Square.

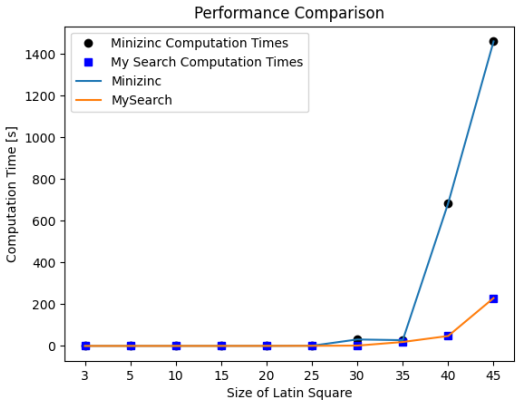

Figure 41: Scalability: Time needed to find a global optimum

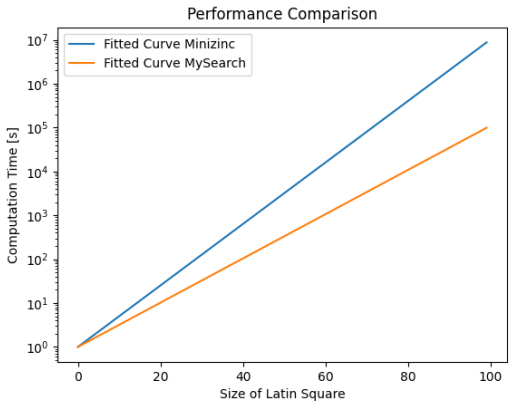

Figure 42: Scalability: Logarithmic scale of extrapolated function

Figure 43: Experiment 5

In fig. 41 we can see the performance plotted and we see that my method outperforms Minizinc.

In fig. 42 I have extrapolated the trend from the data and approximated an exponential function using `scipy.optimize.curve_fit` to approximate the time for even larger n. The scale is chosen to be logarithmic. Assuming that the extrapolation is reasonable, we can make a simple illustration of the difference it does make. For $n = 100$ my method takes roughly 10^5 seconds, i.e. around 27.78 hours. The minizinc would take roughly 10^7 seconds, i.e. 115.74 days to complete. This can make a difference between a method being useful in practice and a method not being useful in practice.

Therefore, I propose a competitive approach to creating a Latin Square. The results imply that it is more scalable than using a simple CSP-Solver like Minizinc.

4.2 Performance

This section is about the usability of the different methods.

When the goal is to find a random Latin Square as fast as possible, my method is superior. The downside of the method I propose is that it is not useful for an extensive search, e.g. finding all Latin Squares of size n. This is due to the fact that my method does not have a systematic approach to do so. In theory the method can generate all Latin Squares of size n, but it does not so systematically. For this problem one needs a more systematic method, but I think it is clear that a local search method is not designed with that goal in mind.

Where my local search method clearly is superior is the ability to give useful intermediate results. The rows are made conflict-free from top to bottom of the matrix, a visualization is found in the appendix in section 6. Coming back to derangements, the method can be seen as an online generation of derangements. This means if one needs pairwise fixpoint-free permutations, my method does generate them in an online fashion.

5 Conclusion

This report does explain my programming project for solving the Latin Square Problem. We first saw the modeling method of using a constraint-modeling language and used a respective solver. We saw the runtime for different n.

Then I used a constrained optimization model where I implemented a local search algorithm on my own. One saw how the definition of a conflict was used to design a good Tabu Search method.

For each of the methods proposed experimental results and runtimes for solving the Latin Square Problem for different problem sizes are given.

On the problems I encountered it was made clear that most of my approaches were not really scalable and most of them also did not find a global solution with an $n \geq 10$. Escaping the local optima was one of the biggest challenges I encountered. I could not escape the local optima by using randomness within my local search methods. If one only needs an approximation of a Latin Square, such methods could still be useful for a fast approximation.

I described many approaches as it did turn out to be quite hard to define a local search heuristic which finds global optima reasonably often. This was part of the development process of this programming project. All methods developed represent a big part of my thought process until developing the promising method `tabu_search_only`, where I used the idea of Tabu Search.

For the comparison between the constrained programming model and the constrained optimization model I did choose my best approach. It is clear that other approaches are not comparable as they do not produce global optima.

In the end, I could present a method which outperforms the classical solver with respect to being non-deterministic and generating any possible Latin Square. I also describe its different use cases as well as its advantages and disadvantages.

6 Appendix: Workings of tabu_search_only

Here is a visualization of how the `tabu_search_only` method works. The green rows are rows without conflicts.