LATIN SQUARE COMPLETION

CMPT317 - KIRK MCCULLOCH – 11146754
MARCH 20, 2015

PROBLEM DESCRIPTION

A Latin Square is an NxN matrix of the positive integers 1 to N, such that no row or column contains the same value more than once. That is, each row and column contains a unique permutation of the numbers 1 to N. Latin Squares are similar to the well known Sudoku puzzles, except that they do not have the 3x3 block constraints and they can be of any order N, not just nine. Additionally, Latin Squares are related to quasigroups in that they can be used to represent the multiplication table of an order N quasigroup.

The Latin Square Completion Problem deals with the problem of filling in a partial Latin Square. Given an order N Latin Square with some number of cells left blank (or zero), we seek to find a way to fill in the blank cells that results in a valid Latin Square. The example below should give a clearer picture.

The following partial Latin Square

```
. 2 . . 4
3 . 1 . .
. 4 . . .
2 . . . 3
. . 2 1 .
```

Can be solved like so

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

Some partial Latin Squares have one or many possible solutions and some cannot be solved at all. In my experiments, I used a generating program that works backwards from a complete square to a partial square, to ensure that all of the Latin Squares in testing have at least one possible solution.

SOLUTION DESCRIPTION

The Latin Squares Problem can be represented as a Constraint Satisfaction Problem. Constraint Satisfaction Problems (CSP) are search problems, which take advantage of a special structure that allows them to be solved with powerful general-purpose algorithms. There are three main components to the structure of CSPs: X, a set of variables, D, a set of domains

specifying allowable values for the variables, and C, a set of constraints to specify allowable combinations of values.

In a Constraint Satisfaction Search, each search state is defined by a partial or complete assignment of values to the variables in X. A complete assignment assigns a value to every x_i in X and a partial assignment assigns values to only a subset of X. A constraint can be defined with a scope and relation, where the scope is a tuple of variables from X and the relation defines the values allowable within that scope. An assignment is legal or consistent if it does not violate any of the constraints in C. Thus, a solution to a CSP is a legal and complete assignment.

In the case of the Latin Squares Problem, we can define its structure as follows. The set of variables is every cell in the NxN matrix. Each variable will have a corresponding domain containing the integers from 1 to N. The constraints will be set such that, for every cell in each row of the square, the values must all be different, and for every cell in each column of the square, the values must all be different. Additionally, because we are solving partial Latin Squares some of the cells will have predefined values that cannot be changed. Thus, these cells will have unary constraints that restrict their domains down to one and only one option.

Using this structure, state-space searching can be combined with other techniques to simplify the search. With techniques such as inference through constraint propagation, variable and domain value ordering, forward checking, and backtracking search, a CSP can be solved without the need for problem-specific heuristics or pure brute force. The implementation I used on the Latin Squares problem uses the constraint set above along with backtracking search and some very simple variable and value ordering.

EXPERIMENTAL METHOD

To perform my experiments on the Latin Squares problem I made use of several open source tools and scripted the testing process in C#. To generate the sample problems, Professor Michael Horsch of the University of Saskatchewan provided a java program for students to use. In order to link this directly into my C# scripts, I compiled a jar file and used IKVM.NET to create a .NET dll, which I could call directly from my C# application. For the main CSP solver I made use of Google or-tools and a quasigroup completion model example by Hakan Kjellerstrand.

For my experimentation, my goal was to investigate the difficulty of solving partial Latin Squares. I wanted to see how the dimension and number of holes affect the time it takes to find a solution. For various dimensions, ranging from 10 to 100, I used the same five random seed values to generate test problems that would be as similar as possible with varying numbers of holes. This way the same five base Latin Squares could be generated repeatedly and only the cells removed would change. Across all tests, I removed cells by percentage of the total cell

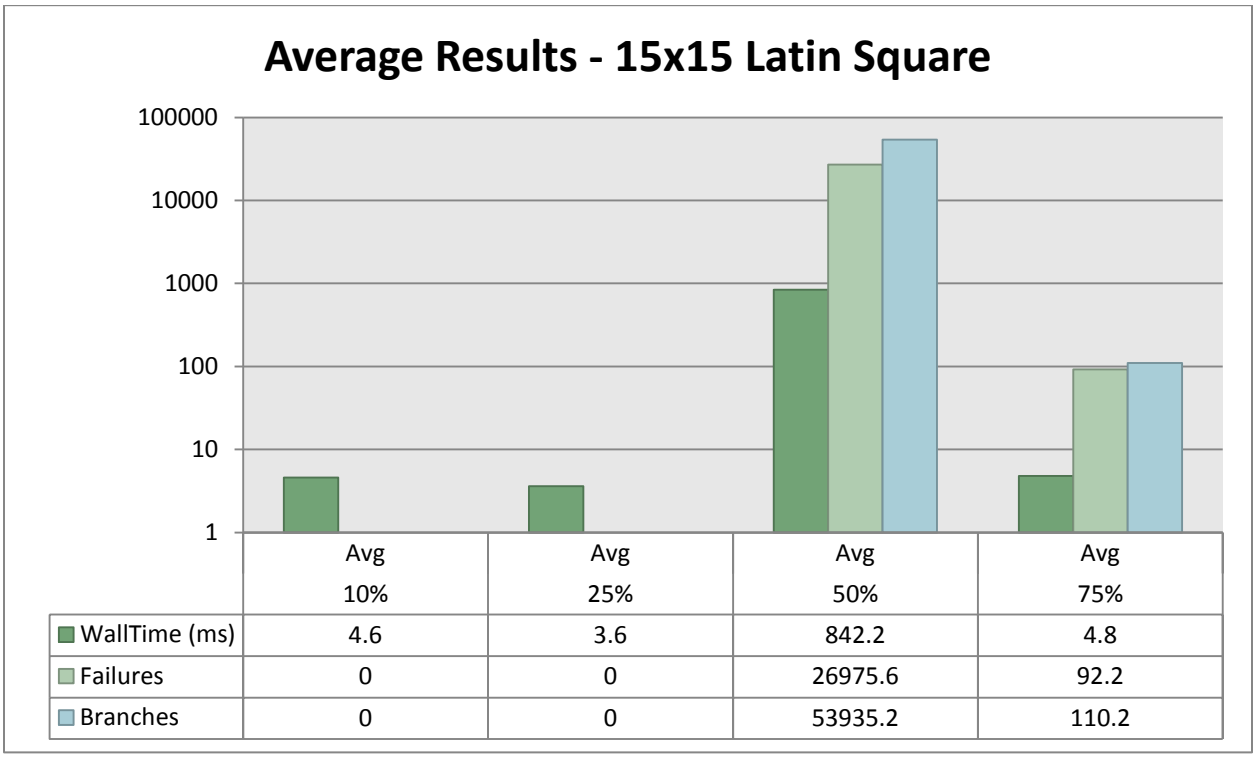
count. I used steps of 10%, 25%, 50%, and 75% to create holes proportional to the total size. Using these standard parameters to generate test problems, I ran the CSP solver against each problem and recorded the time taken, number of failed attempts, and branches explored to find the first available solution. Any further possible solutions could cause the running time to increase significantly for problems with numerous solutions compared to problems with only one, so I chose to focus on finding only the first solution.

In general, I found that running time increases by a small degree with size, but increases more significantly as the number of holes increases. However, passed 50% the time seems to drop as quickly as it grows.

An unexpected anomaly that emerged from the above tests was that for dimensions above 15, the 50% emptiness test case would frequently run indefinitely for certain random seed values. This prompted me to try some different tests to discern the cause. The first new test I created was to perform a more granular set of emptiness tests. For a given dimension, seed value, and five levels of emptiness, I generated 5 test problems to search on for up to 30 seconds. My intension was to figure out if it was a problem specific to the 50% mark, some range near 50%, or due to the seed value. The second new test I created was to compare a range of dimensions for the same seed and emptiness values. My intension was to see if this was simply a product of the increasing size of the square, because nothing under dimension 15 seemed to be affected.

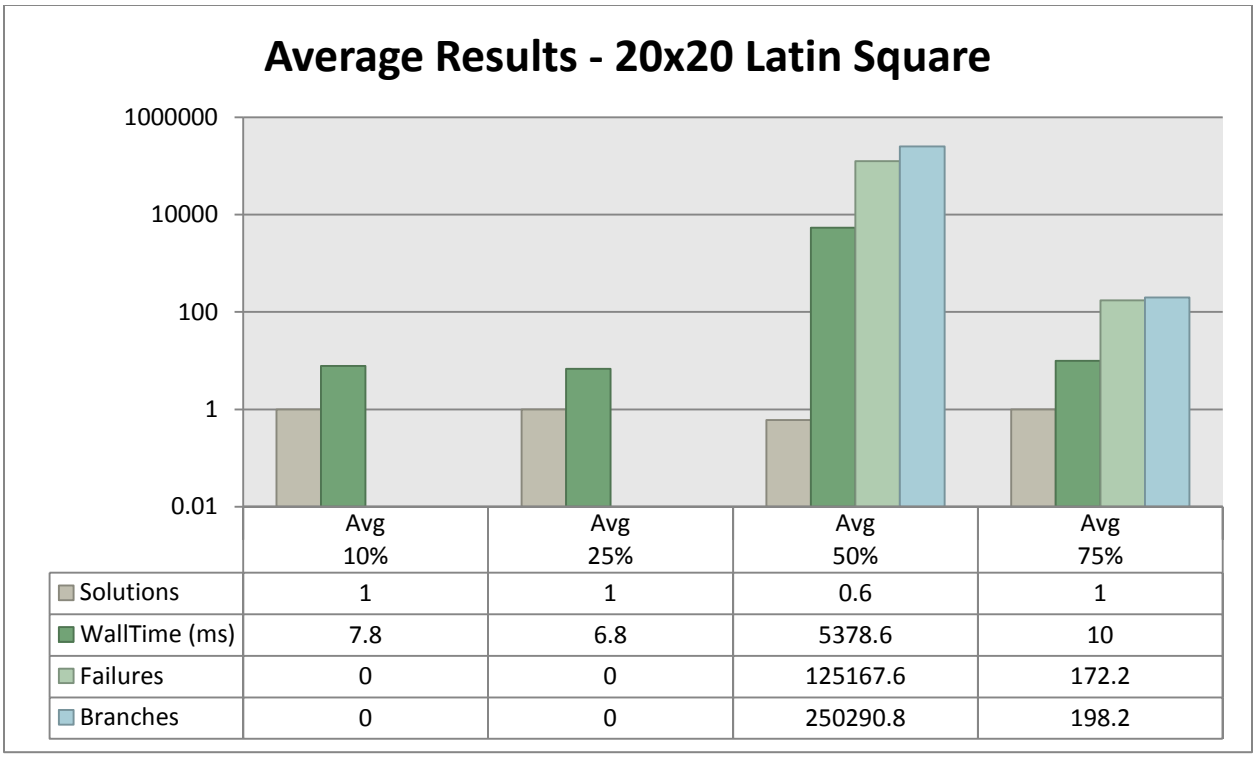
RESULTS

For my initial set of tests, I tested dimensions from 10 to 100. I started with a dimension of 10, but the search timer built into Google or-tools is not accurate enough to report any useful data. At dimension 10, almost all test squares were completed in under 1ms. The first dimension with interesting data was dimension 15.



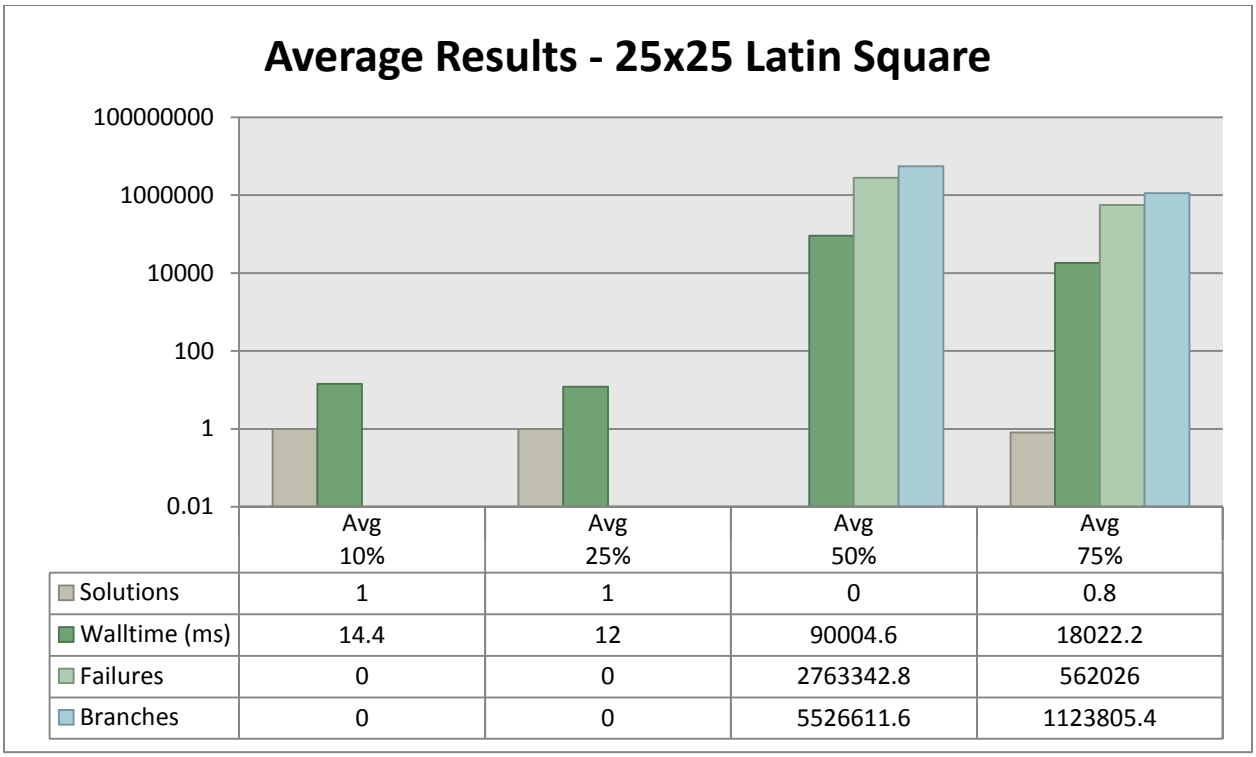
	Avg 10%	Avg 25%	Avg 50%	Avg 75%
WallTime (ms)	4.6	3.6	842.2	4.8
Failures	0	0	26975.6	92.2
Branches	0	0	53935.2	110.2

At dimension 15 a solution was found for every square and at the 50% emptiness mark the search time shoots up drastically. Moving beyond 50% the search time drops down again. The graph above shows results averaged across all five seed values. The data in the graph is a little bit misleading, because seeds 1 to 3 took less than 35ms each at 50% emptiness. Seeds 4 and 5 took 2932ms and 1208ms respectively. While emptiness at 50% is clearly a factor in the difficulty of solving the Latin Square, the degree of impact varies greatly from one square to the next. All five seed values generate squares that are functionally the same, and yet there is potential for significant gaps in search time.



	Avg 10%	Avg 25%	Avg 50%	Avg 75%
Solutions	1	1	0.6	1
WallTime (ms)	7.8	6.8	5378.6	10
Failures	0	0	125167.6	172.2
Branches	0	0	250290.8	198.2

The next interesting test set was dimension 20. This was the first time I saw it fail to find a solution at all. In this test, I was using a time limit of 10 seconds, so after 10 seconds it would give up and move to the next Latin Square. The 10% and 25% marks remain trivial, never running over 14ms. Again, the data at 50% deserves more detail than the average shows. Seeds 2 and 5 both ran beyond the 10-second limit and failed and seed 1 took 6684ms. Seeds 3 and 4 took only 153ms and 52ms respectively. This is surprising because previously the most difficult seed was number 4, by a wide margin, and here it was the easiest, by a wide margin. I can draw a similar conclusion to before. The emptiness is clearly a factor, but it is not the dominant aspect. Some uncontrolled factor in the randomly generated test squares is causing dramatic variations in difficulty, with no discernible pattern.



	Avg 10%	Avg 25%	Avg 50%	Avg 75%
Solutions	1	1	0	0.8
Walltime (ms)	14.4	12	90004.6	18022.2
Failures	0	0	2763342.8	562026
Branches	0	0	5526611.6	1123805.4

Next, I ran a dimension 25 test with an increased search limit of 90 seconds, 10 million fails, and 10 million branches. However, this had little effect on success at 50% emptiness. Again, 10% and 25% are trivial. The 50% mark did not have a single success, even with the dramatic limit increase. Since none of the 50% squares were solved, it is unclear whether this was due to the dimension or the unknown factor observed previously. However, the 75% mark is interesting here, because seed 3 did not complete, but all others took less than 45ms to complete.

I would also like to note that in all tests so far, the first two emptiness brackets are completing with 0 failures and 0 branches. This tells me that the solutions are being found linearly in the very first search branch, which is something I had not expected.

I also ran tests on a dimension 50 set and a dimension 100 set. In the dimension 50 set, at 25% emptiness, only 2 of the 5 squares were solved in less than 90 seconds and every test beyond that failed. In the dimension 100 set, I increased the time limit to 3 minutes. Despite the time limit, only the 10% bracket was solved and averaged 189ms. At this high dimension, the number of holes to fill is extremely high even at only 25% emptiness. It is understandable that there would be a point where the square would simply be too big to process quickly.

After discovering the odd behavior at dimension 20 with 50% emptiness, I devised some other tests to look into the behavior further. As described in the method section, I tested a more granular range of emptiness on a single seed value. I used a dimension 20 square and a

range of 48% to 52% empty on seeds 1, 3, and 4. I got some surprising results, considering each test only differed by 1%. Seed 1 solved 48% to 50% and failed after that. Seed 3 was interesting, because it was solved at every level, except for 49% and it seems strange that a difference of 1% would cause the running time to go from over 30 seconds, to just 157ms. Seed 4 was equally strange in that it only solved 49% and 50%, and jumped from 51ms at 50% to over 30 seconds at 51%. While the emptiness of the square does have an effect on the difficulty, from the drastic variations in these results, it is clear that the sheer *number* of empty cells is not the source of the difficulty.

The other extra test I ran was to compare a range of dimensions at the same level of emptiness for a given seed value. At 25% emptiness, I tested the following dimensions, 15, 20, 50, 75, and 100. I ran this on all five seed values and did not get any interesting results. In all cases, 15 and 20 were solved trivially and only for seed 3 was it able to solve the dimension 50. All others failed. I also ran this test on a more granular range of 16 to 20, with 50% emptiness, which better reflects the point where the test squares became unsolvable. This set returned more strange results as seen previously. For seed 2 it was only able to solve dimensions 16 and 19. For seed 4 it solved all but dimension 18. The difficulty really does not seem to increase directly with these small size changes and the results still do not follow a discernible pattern.

Further test results that I have not discussed directly here are included with the source. See the `../bin/Debug/` directory.

CONCLUSIONS

Ultimately, given the lack of a regular pattern in much of my results, I can only conclude that the difficulty of partial Latin Square problems has less to do with the number of holes and significantly more to do with *where* the holes appear. Since my test problems were all generated randomly, I could control how large they were and how many holes they had and still be certain that they were always solvable, but I could not control where the holes got made. Additionally, I know it was not due to the order of the values in filled squares, because every problem from a given dimension and seed value would always start from the same complete Latin square. Making a similar number of holes in that square (i.e. 48% to 52%) could lead to wildly different results. Thus, it must be the *placement*, and not the number, of holes that is the deciding factor in the complexity.

Size and hole count do still affect the solvability of a partial Latin Square, but not to the extent that I had expected. Extremely large problems will always be harder to solve than small ones with the same density of holes. Similarly, problems with only a few holes are easier than problems with holes up to about half the area of the square. I had expected the search time to be roughly parabolic given that with few holes the search space is smaller and beyond 50% the

opportunity for flexibility (i.e. more holes means more freedom to place numbers) should increase the odds of there being an obvious solution. To some degree, it roughly is. At dimension 15 the running time goes from 4ms to 800ms to 4ms again. In hindsight, I should have included a 90% empty test case to make the whole set properly symmetrical. The thing is the randomness in the generator can lead to completely different problems from the same base Latin Square. If only *new* holes were added, instead of re-shuffling the whole square, the results might be more predictable. If I had more time, I would explore this theory next.

REFERENCES

Frijters, Jeroen. 2011. IKVM.NET Home Page. *IKVM.NET.* [Online] 2011. <http://www.ikvm.net/>.

Google. 2014. Google Optimization Tools. *Google Developers.* [Online] 2014. <https://developers.google.com/optimization/>. <http://www.apache.org/licenses/LICENSE-2.0>.

Kjellerstrand, Hakan. 2012. My Google or-tools / CP Solver page. [Online] 2012. <http://www.hakank.org/google_or_tools/quasigroup_completion.cs>.

For implementation source code, see the included archive.