Title of project: Keisuke

Group members:

Shelton Li (g2lishel)

Roles: Puzzle Generation, Report Writing, Testing, Sample Run Data Collecting

Imran Ariffin (c6ariffi)

Roles: Implement Keisuke CSP Version 2, Report Writing

Taewoo Kim(g5kimtae)

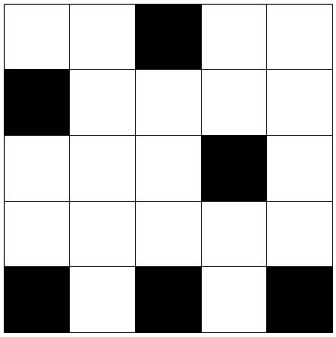
Roles: Implement Keisuke CSP Version 1, Hard Puzzle Generation, Report Writing

TA Consulted: Alberto

Project Motivation/Background

In our project, we are trying to solve the logic puzzle Keisuke. Keisuke is played on a n by n matrix in which some of the cells of the matrix are blacked out and the remaining cells are blank. Additionally, numeric values are given and they represent horizontal and vertical values in the matrix (every numeric value is not a single digit). The goal of the game is to fill the blank cells with single digits so that the numeric values show up in the specified order and orientation. There are no values placed in the blacked out cells. An example of a keisuke puzzle (left puzzle and numeric values) and its solution (right puzzle) can be seen below.

Numeric Values Provided:
ACROSS → 13, 23, 233, 3221, 21222, **DOWN** ↓ 12, 21, 22, 232, 3132, 33313



2	3	■	1	3
■	3	2	2	1
2	3	3	■	3
2	1	2	2	2
■	3	■	1	■

So for the across value of 13, the only possible locations it can go in the matrix is at positions [(0,0), (0,1)] or [(0,3), (0,4)] as they are the only horizontal sections of the matrix which are of size two. The same applies to other numeric value sizes and also to the vertical numeric values.

Our approach to this problem is through the form of a constraint satisfaction problem. We chose to frame Keisuke as a CSP since there are numerical constraints that have to be met in order for this puzzle to be solved. By properly framing the constraints of a Keisuke puzzle, we can easily apply the various CSP solving algorithms we learned in class to produce a solution.

Methods

Variables: Each individual cell in the matrix is treated as a variable.
Variable Domains: For variables that represent a blank cell, the possible domain is [1, 2, 3, 4, 5, 6, 7, 8, 9]. For variables that represent a blacked out cell, the domain would be [-1]. Furthermore, a variable's domain can be simplified by using the provided numeric values to narrow down the possible values. The only values that should be in the domain can be found in the given numeric values. For example, if we use the above sample puzzle, we can simplify the domain for each

variable to be [1, 2, 3]. This is because only the values 1, 2, and 3 are found in the given numeric values.

Constraints: There are two types of constraints that we employ. The first is the size and orientation constraint. We define a subsection of the matrix as any number of cells on the same row or column not separated by a blacked out cell (ie. [(0,0), (0,1)] or [(1,0), (1,1), (1,2), (1,3), (1,4)]). So for each subsection of the matrix, we create a constraint in which the scope would be the cells in the matrix that the subsection covers. The satisfying values for these constraints would be the numeric values that match the size and orientation of the subsection. An example of this constraint would be for the subsection [(0,0), (0,1)] in which the scope would include the variables for cells (0,0) and (0,1). The only possible values to this constraint would be 13 or 23.

The second type of constraint is similar to the binary alldifferent constraint. These will be constraints between all combinations of subsections that have the same size and orientation to ensure that every numeric value is used for only one subsection. An example of this constraint would be for subsections [(0,2), (0,3)], [(3,0), (3,1)] and [(3,3), (3,4)]. These three subsections are of size two and have the same orientation which is vertical. There would be three constraints made here; between the first and the second subsections, between the first and the third one, and between the second and the third one. So if [(0,2), (0,3)] is given the value of 22, then [(3,0), (3,1)] must be different and could have the value of 12, while [(3,3), (3,4)] could have 21.

Overall, we believe our formalization is correct since we developed our variables, domains, and constraints by following the puzzle's rules. Each cell is well suited to be a variable because the CSP algorithm will assign a number to a single cell at a time just how like we normally pick a single digit for each cell at a time. The domain of each variable can only be the numbers that are given in the numeric values. Only horizontal numeric values can go in subsections of the matrix that are also horizontal and have the same size. This applies for vertical numeric values as well. Each numeric value should be used to fill exactly one subsection.

Evaluation and Results

To test that our CSP is correct, we used our random puzzle generator function to create puzzles of different sizes. The random puzzle generator works by creating a solution puzzle by inserting numbers in range 1 to 9 and also -1 which represents a blacked out cell. It then creates the numeric values by splitting rows and columns on -1 values. We then solved the puzzles with our CSP. If all the cells are filled with valid numeric values, the solution is correct. Here is an example of a puzzle, its numeric values and the solution:

```
Puzzle            ACROSS          DOWN            Solution
[0, 0, 0, 0, 0]   [3, 4, 5, 2, 4] [3, 9]          [3, 4, 5, 2, 4]
[0, 0, 0, 0, 0]   [9, 3, 3, 6, 4] [4, 3]          [9, 3, 3, 6, 4]
[-1, -1, 0, 0, 0] [9, 6, 3]       [5, 3, 9, 8, 1] [-1, -1, 9, 6, 3]
[0, -1, 0, 0, -1] [8, 3]          [2, 6, 6, 3, 1] [1, -1, 8, 3, -1]
[-1, 0, 0, 0, 0]  [3, 1, 1, 5]    [4, 4, 3]       [-1, 3, 1, 1, 5]
```

After testing for correctness, we tested how it performs with different constraint propagations. We ran our CSP with 4 by 4 randomly generated puzzles using GAC and FC. We infer that GAC has better performance than FC because while GAC can check constraint inconsistency every time a variable is assigned, FC can't check it until there remains one unassigned variable in the scope of a constraint. The following charts show the results of five runs.

FC

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Time	0.185103	9.811287	0.253497	0.157246	5.75373
Variable Assignments	6282	356633	8747	5338	176613
Variables Pruned	37708	2155076	52900	32393	1437103

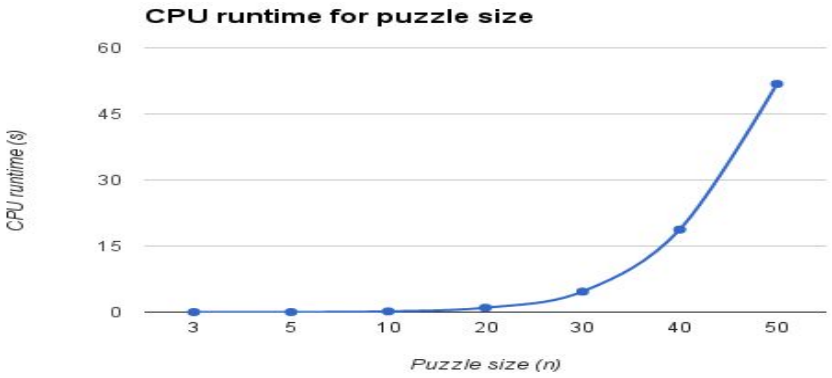
GAC

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Time	0.013748	0.012016	0.007835	0.004323	0.012067
Variable Assignments	16	16	16	16	16
Variables Pruned	112	112	105	98	96

The result shows that FC is significantly slower than GAC. We observed that the time is related to the number of variable assignments. Surprisingly GAC assigned values to variables 16 times which is the total number of cells in the puzzle. This means that GAC made a correct choice every time and never backtracked. On the other hand, FC made a lot of variable assignments and pruning. This is because FC made wrong choices and couldn't detect it early. The performance of FC fluctuated a lot in the five trials. This is also because FC can't detect its wrong choice early so its performance highly depends on how it assigns value to a variable. Moreover, the performance of FC will drop extremely as the puzzle size increases since FC has to wait more time or steps in order to check for constraint inconsistency. So we decided to further test robustness of our CSP with GAC.

Next, we want to explore if the runtime gets slower as we increase the size of the puzzle. We completed our tests by using our random puzzle generator function to create puzzles of different sizes. The size of the puzzles we tested are 3, 5, 10, 20, 30, 40, and 50. Since the puzzle

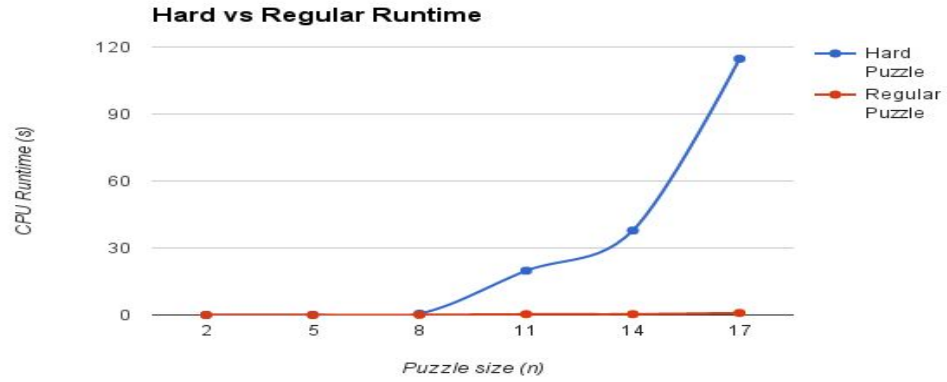
generator creates a random puzzle, for each size we create five different random puzzles and then we use GAC to solve the puzzle, and take the average run time of those five runs. The result can be seen in the chart below.

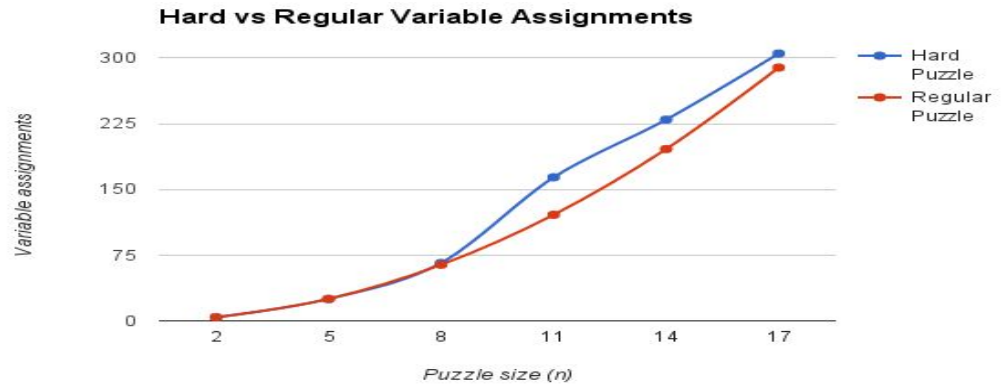


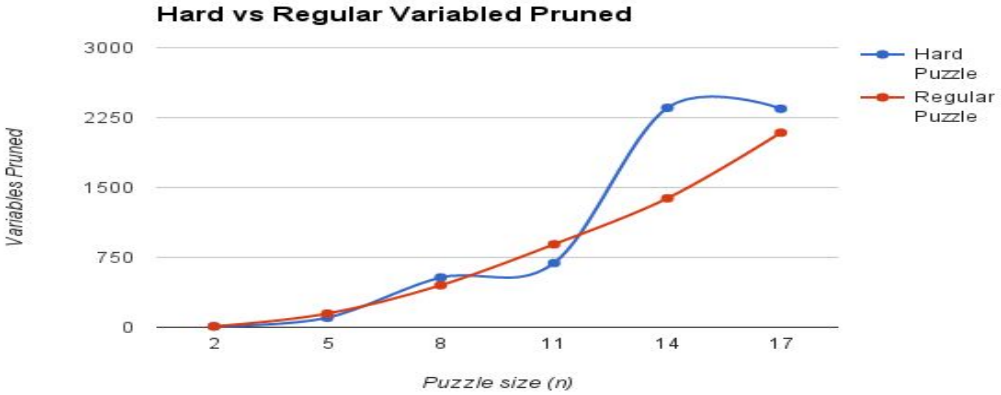
Using the data from our tests, it appears that the runtime increases dramatically as puzzle size increases. This is because the amount of variables and constraints increases as the size increases which will cause GAC to run slower. A 50 by 50 puzzle is a quite large puzzle and our CSP still solved in an acceptable time. We concluded that our CSP works well with randomly generated puzzles. We wanted to know whether there would be some configuration of the puzzle such that our CSP performs worse.

We defined a hard puzzle as one that has more subsections of the same size. We came up with an idea that if there are more supports for the constraints, then pruning will happen much later which causes our CSP to detect wrong choices made much later. We tested our CSP with puzzles that only have subsections of length two. On the right is an example of more difficult 5 by 5 puzzle board. We again made a puzzle generator function for harder puzzles and tested 5 times for each puzzle size and then took the average run time, number of variable assignments, and the number of variables pruned.

```
[0, 0, -1, 0, 0]
[0, 0, -1, 0, 0]
[-1, -1, -1, -1, -1]
[0, 0, -1, 0, 0]
[0, 0, -1, 0, 0]
```



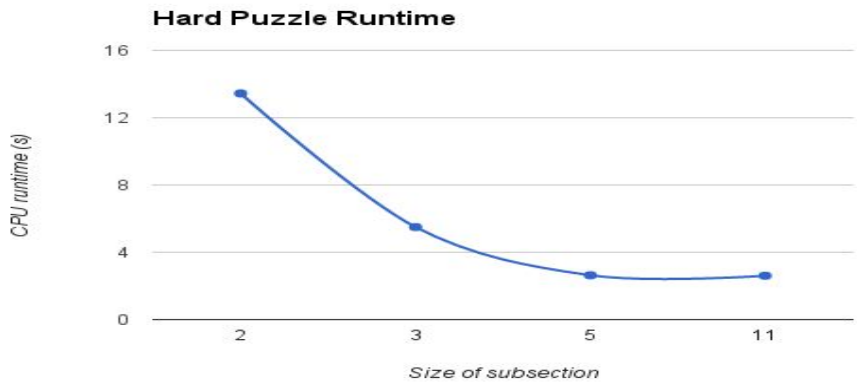




As we expected, the hard version of the puzzles took significantly more time than the regular puzzles (17 by 17 hard puzzles took around 2 minutes!). However, there is no big difference in the number of variable assignments. Rather, our CSP made correct assignments most of the time for hard puzzles. For the pruning, the hard puzzles of size 14 pruned a lot more than the regular puzzles but they are similar in the rest of the puzzles sizes. For size 11, even the hard puzzles have less pruning. So we can't conclude that significant performance drop is simply due to pruning and late detection of wrong variable assignments. We later noticed that the algorithm took lots of time on checking arc consistency. Since every subsection have the same size, many alldifferent constraints are related to each variable. Thus, in every variable assignment, the algorithm takes significant amount of time to check the consistency of all those constraints although it doesn't always do much more pruning than the case of random puzzles.

To further explore our insight of a hard puzzle, we tested our CSP with different size of subsections. We guess that as the size of subsection increases, the number of subsections decreases; this would result in less number of alldifferent constraints and will reduce time for checking arc consistency. We fixed the puzzle size to 11 by 11 and tested with subsections of

size 2, 3, 5, and 11. Again, we tested 5 times for each of the subsection size and took the average runtime. We observed that the time decreases dramatically and confirmed our predictions.



Limitations/Obstacles

In addition to the model described above, we also tried another way of representing the puzzle, as seen in file keisuke_csp2.py, which we believe could solve the puzzle more efficiently. Instead of the cells being treated as variables, the variables are subsections of the puzzle and the domain of each variable is the list of tuples from the numeric values, instead of single-digit values. The constraints exploited are binary alldifferent constraints for subsections of the same size and orientation, and binary intersection constraints. The last constraint is simply that two intersecting subsections have to have the same number at said intersection. Therefore, we had hoped to prune much more early on. However, it failed to find the solution and gave None to all assignments. We have not figured out what is the exact problems, but we would suspect that intersection constraints didn't work well. Unlike the first version, intersection constraints are necessary in the second version since the variables are now subsections. It is possible that the way we implement the constraints in the second version are erroneous and/or incomplete.

Conclusions

We observed that for our CSP, using GAC works well with regular puzzles. The algorithm made correct variable assignments most of the time. Large puzzles such as the 50 by 50 puzzle can be solved within a minute. In order to see if any alteration to the puzzle could make solving the CSP more difficult, we came up with hard puzzles where every subsection in a puzzle has the same size. As we expected, hard puzzles took a lot more time than regular puzzles. However, it was surprising that the reason it took a longer time was not due to late detection of wrong variable assignments since the number of variable assignments was not significantly higher than the regular one. We concluded that it is due to the number of binary alldifferent constraints for subsections of the same size and orientation. The amount of those constraints increases

exponentially as the puzzle size increases. Our CSP spends so much time on checking arc consistency. We hoped that our second version of CSP could solve the hard puzzle faster but the second version didn't work well. To extend our algorithm to solve the hard puzzles faster, the algorithm should not check all constraints since it didn't prune significantly more. We can skip some constraints from checking arc consistency instead of checking all relevant constraints.