CSC384: Numbrix to Hidato CSP

Due on April 5, 2017

Angus Fung, Alvin Lee, Yoonsun You
fungangu, leealvi3, youyoons

April 7, 2017

Tasks

Angus Fung: Altered the Numbrix CSP to be able to handle obstacles (transition to Hidato CSP). Made hypotheses regarding the different inference techniques and performances (BT, FC, GAC).
Alvin Lee: Planned the CSP variable and constraint representation. Programmed the Numbrix CSP and debugged the Hidato CSP.
Yoonsun You: Programmed the transition from Numbrix CSP to Hidato CSP and devised the test cases. Ran BT, FC, and GAC on the test cases.

Background

We decided to explore the Constraint Satisfaction Problem (CSP) of Numbrix and Hidato. Numbrix was explored first before deciding to move onto Hidato. Numbrix and Hidato are popular grid games in which the entries of the grid must be such that a trace from the lowest number to the largest number can be performed without raising the pencil. The detailed rules are below:

In Numbrix, the lowest number (1) and highest number are always provided (not considered to increase complexity of problem). The goal of the puzzle is to fill the cells in such a way that there is a series of consecutive numbers adjacent to each other in the *horizontal, or vertical direction.* In addition, a Numbrix puzzle must be a *square shape*, similar to Sudoku (see Figure 1 in Appendix). Hidato is similar to Numbrix, except that for each cell, adjacency includes diagonally adjacent cells, and the shape of the grid can be irregular (see Figure 2 in Appendix). One aspect of Numbrix that makes it a great logic puzzle for this project is that it has a unique solution, given that a sufficient number of initial entries are provided.

This problem was framed as a CSP because (1) the nature of the game is built on constraints (2) it has a uniform and simple state representation (3) the user has complete control of the environment (e.g no adversary), among many other reasons.

Method

The approach to the project was to begin by solving a subset of the original problem, either a smaller grid size or relaxing some of the rules and constraints. All the while, keeping the code as modular as possible. Then, layers of complexity was slowly introduced, and as will be seen later, more constraints were added on top of the original problem as time allowed. This was to explore the maximum potential of the AI.

As mentioned earlier, the first phase of the project was to solve Numbrix for a 3x3 square, with vertical and horizontal constraints (elaborated later), and start and end numbers defined (Grid a0). Then, this was extended to larger grids (6x6), including rectangular shaped grids (ie. a2: 4x3). At this point, our initial goal for the Numbrix CSP was completed. However, the original problem was made more challenging by adding in diagonal constraints, as well as being able to solve *irregular shaped* grids, with no beginning or end numbers defined. This was the start of transitioning into the Hidato CSP. Finally, the algorithm was adjusted to solve problems with boundaries [1] within a given shape. Custom test puzzles were needed to be created for these additional constraints, as they go beyond the scope of the original Numbrix puzzle. By the end of the transition to Hidato, the AI was able to solve up to 8x8 Hidato with obstacles with around half of the entries defined in under 3 minutes (Grid hid10). Other simple puzzles were set up and solved at a much quicker rate (generally between 0.0 seconds to 1 minute).

Numbrix was formulated as the following:
Variables: The number of variables is domain-dependent and problem-specific, and is defined as the number of cells in which a number can be placed. For example, a 3x3 square grid would have 9 variables (no need to consider boundaries/obstacles as this is Numbrix and not Hidato).

Domain of Values: For cells that are already filled in (i.e. initially on the grid), the domain is the singleton set containing the predefined value. For cells that are not already filled in, the domain is the set of all possible numbers that could be on the grid, and lie between 1 and the total number of valid [2] cells.

[1]Boundaries: regions in which no numbers can be placed, which are also referred to as **obstacles** in our code and report. Cells that are on the boundary are called a boundary cells.

[2]Valid Cells: All cells that are not a boundary cell although this is not an issue for Numbrix.

State: Any completed board given by specifying the value in each cell. Note that not all states are valid. A **partial state** is any incomplete filling of the board.

Solution: A value for each cell satisfying the following constraints:

1. The number be between 1 and the total number of valid cells in the grid.
2. The number must be unique (no two numbers in the grid are the same).
3. For numbers that are not 1 or the maximum number on the grid, the number that is one greater and one less must be in an adjacent cell [3].
4. For the number 1, a 2 must be in an adjacent cell, and for the maximum number on the grid, the number which is one less than the maximum must be in an adjacent cell.

The methods of enforcing these constraints will be explored, discussed, and compared.

Hidato was formulated as the following (a more complicated application of Numbrix):
Variables: The number of variables is domain-dependent and problem-specific, and is defined as the number of cells in which a number can be placed. For example, a 4x4 square grid could have 16 variables. However, if there are two obstacles within the grid, there would be 14 variables.

The **Domain of Values** and **States** are identical to Numbrix, and are not included for sake of brevity. Similarly, **Solution** 1. and 2. are identical to Numbrix, and slight modifications to 3. and 4. are shown below:

3*. For numbers that are not 1 or the maximum number on the grid, the number that is one greater and one less must be in an adjacent cell (for Hidato, adjacent includes diagonally adjacent cells).

4*. For the number 1, a 2 must be in an adjacent cell, and for the maximum number on the grid, the number which is one less than the maximum must be in an adjacent cell.

Numbrix Variable and Constraint Setup
The variables and constraints were set up in a similar manner to Assignment 2. Each valid cell on the grid was assigned as a variable, and given a unique variable identifier (i.e Var i,j where i,j is the matrix indices of the variable). Each variable that did not have a pre-defined value was given a domain ranging from 1 to the number of valid cells on the grid. A variable that had an initially assigned number only had that single number assigned to its domain.

Afterwards, depending on whether a variable was in a corner, on an edge, or not touching any edges, the variables for each constraint were specified (as each constraint was built around a selected variable). A constraint for a variable on a corner would have 3 variables, itself and its two cardinal adjacent variables (the other two cardinal directions are blocked by the boundary edges and hence, have no variable). A constraint for a variable on an edge would have 4 variables, itself and its three valid cardinal adjacent variables. Finally, a constraint for a variable not touching any edges would have 5 variables with a similar reasoning as above. Each of satisfying tuples for the listed constraints involved finding tuples of numbers such that each number in the tuple was unique, and that the tuple included a number which was one greater and one less than the number assigned to the variable of interest (i.e. the variable that the constraint was built around). There is an exception for when the variable of interest was assigned the number 1 or the maximum number on the grid, in which case the satisfying tuple only required a number which was one greater or one less,

[3]Adjacent Cell: A cell that is horizontally or vertically adjacent for Numbrix.

respectively (all numbers being unique in the tuple was still required). Binary not equal constraints between all variables were also chosen over n-ary all different constraints (as n would be equal to the total number of variables, this was not a computationally feasible choice) in order to solve the problem and print the results.

Transition to Hidato
After being able to solve all Numbrix puzzles within reasonable sizes, a more challenging CSP, Hidato, was explored. The major changes from Numbrix to Hidato were that consecutive adjacent numbers were allowed across diagonals instead of only horizontal and vertical directions. In addition, irregular shapes within a grid format are allowed. That is, a grid of say, 8x8, can have some of its entries replaced with a -2 to indicate that it is an obstacle/boundary. No number can be placed in these locations, and the specific entry is not a variable. The specific variable constraints in Hidato were no longer 3,4,5-ary like in Numbrix, but ranged from 2-ary to 9-ary depending on the locations of boundaries and obstacles. The binary not equal constraints worked in the same way as in Numbrix.

Evaluation and Results

Evaluation
The goals of the **Numbrix** AI was to be able to solve all **Numbrix** puzzles of different sizes, ranging from 3x3 to 6x6 (grids a0 to a8; See Appendix) in a reasonable amount of time (maximum of 1 minute for setting up CSP and a maximum of a 30 seconds for solving the grid). Puzzles ranging from 6x6 to 10x10 (a9 to a11) were desired to be setup within 2 minutes and solved within 1 minute (min(FC time,GAC time)). It was found that depending on the number of pre-defined entries, the set-up and solve time varied drastically. The initial occupancy of the grid was generally set to above 25% (to ensure a unique solution). These puzzles would be classified as "difficult". Puzzles between 3x3 and 6x6 were given difficult cases but puzzles between 6x6 and 10x10 were given up to "intermediate" (50% initial occupancy) cases . It is crucial to re-emphasize that our most important metric was correctness, followed by the time constraint. The number of variable assignments and prunings were not considered as an important metric.

After the adjustments were made to make the AI to solve Hidato puzzles, similar evaluation metrics were set. If too many numbers were removed, the solutions were no longer unique and hence, it was made sure that there would be enough undefined numbers to make some of the puzzles challenging but there would be enough to ensure a unique solution. The Hidato test cases were divided into three sets (c-class, hidsq-class, and hid-class). The c-class was to evaluate the ability of the Hidato AI to solve 3x3 to 6x6 grids. The hidsq-class were official 6x6 Hidato puzzles (http://www.mathinenglish.com/puzzleshidato.php?pstid=802) ranging from easy (hidsq1,hidsq2), intermediate (hidsq3,hidsq4) to difficult (hidsq5,hidsq6). These two classes did not have any obstacles. The hid-class was used to see if our AI would be able to solve puzzles for sizes up to 8x8 with obstacles (set-up time of 2 minutes and solving time of 1 minute for min(FC,GAC)). The same time constraints applied to the c-class and hidsq-class.

The different search and inference methods covered in class were explored. This included the plain back-track (BT) search, forward checking (FC) inference, and GAC inference. These were compared in order to determine which method was most effective for solving the CSP.

Results
All the Numbrix test cases ranging from 3x3 to 6x6 (and rectagular variants such as 3x4) were solved under 30 seconds with most of them solved under 5 seconds (please refer to Table 1). This can be attributed to the reasonable sizes and pre-defined values for the given test cases. It was desired to test the limits of the Numbrix AI and so an 8x8 puzzle was tested on. It turned out that assuming that if approximately 1/2 of the Numbrix was pre-defined, the AI could set up and solve the puzzle in under two minutes. This is a

limitation of our model that will be discussed in the limitations section. Other larger puzzles such as 10x10 could also be solved but they required much more time to complete.

The Hidato results for the first batch of test cases, the c-class, (to test the time performance for smaller grid sizes) was successful. However, similar to Numbrix, as the number of variables increased, the runtime increased drastically. Overall, it appeared that the set-up and solve time of Hidato puzzles took longer than Numbrix puzzles of a similar difficulty. This can generally be attributed to the increased n in the n-ary constraints due to the addition of diagonal constraints (please refer at Table 2). The hidsq-class and hid-class cases were able to meet the time constraints with the exception of hidsq6 in Table 3. The reason is explained later on in this section.

Along the way, there were problems found within the AI implementation, but these problems were fixed as they were discovered. Initially, it was discovered that the Numbrix AI could not solve puzzles more complicated than 4x4 if the first and last numbers were not pre-defined in the grid. This was fixed by including a conditional if statement, where the extra permutations including the first and last numbers were added to the set of satisfying tuples if they were not predefined (the original code did not include this feature because it was assumed that the first and last numbers would always be predefined). For the Hidato AI, it was found that it could not solve grids that were shaped as a single row or a single column. This was due to the nested list (2-D list) implementation that was used to represent the problem. Hence, an additional check was made in the code to deal with these special cases (i.e. an if statement was used to check the size of the grid, and the CSP would be formulated accordingly depending on if the grid was part of these special cases). However, it is to be noted that Hidato puzzles will essentially never occur in the shapes of (1 x n) or (n x 1), and these cases were included in the implementation for completeness of the AI. There were other errors such as the solution returning a list of `[None]`. These were due to errors in the indexing when setting up the constraints and were fixed. **The results for setup, BT, FC, and GAC times for the test cases are in the Appendix.**

One key observation was that FC consistently outperformed GAC across almost all test cases. This can be explained in terms of the idea of of exploration vs. exploitation [4], the way the problem was encoded, and the nature of the algorithms themselves including the complexity thereof. One significant advantage of GAC over FC is that is does more exploitation and inference, whereas FC does more exploration and less inference. Intuitively, it does seem that exploration is a more idealized strategy in Numbrix and Hidato. For example, solving a 4x4 or even a 5x5 solution, there are only so few paths from start to end; because of this, the advantages of GAC are wasted. It is *much* faster to just guess at the solution than expensively enforcing arc consistency on all variables. Furthermore, the constraints of the problem are binary-not-equal for uniqueness and n-ary constraints for neighbouring cells. Since binary constraints were predominately used, then in fact, FC is actually arc-consistent as well for those constraints. In terms of complexity, FC is $\mathcal{O}(c)$ and GAC is $\mathcal{O}(cd^3)$ in the worst case, where c the number of constraints and d is the size of the largest domain. FC in the worse case has every constraints containing every variable, so every instantiation of a variable will force a check over every variable. Similarly, the worst case of GAC is the same as FC, where the d^2 comes from checking and each node is check at most d times since every constraint is *always* placed on the queue. Thus, the run-time of GAC scales *rapidly* with increased domain size.

However, there were some cases where GAC *does* outperform FC. For Numbrix, there are two cases for which this is true on the more harder problems (a7 and a8). One hypothesis for why this is true is because in a7, unlike the previous test cases, it has a lot more unassigned cells *in one area* (note: not necessarily more total unassigned cells). This means that the puzzle has much more satisfying tuples and has less clues on which moves are *wrong*. In these cases, an algorithm with stronger inference would be better than one that makes more random guesses at unassigned variables. This could be explain why FC performs much

[4]By exploitation, I am referring to inference.

better than GAC on a10 – there aren't too many blocks of unassigned variables in one area. In Hidato, the differences between GAC and FC are much more pronounced. This can be explained in terms of the number of constraints. Since Hidato takes into consideration diagonal adjacency, GAC has *much* more constraints to check for arc consistency. Comparing the run-time complexities of both algorithms, adjacency constraints introduce a total of 9 constraints for each cell. This means that in the worse case, GAC could be $9^3 \cdot 2 \cdot C = 1458C$ times longer than FC, where 2 takes into consideration pre-processing and C is a constant that comes from the definition of big $\mathcal{O}$. Finally, this explains the anomaly seen in the the case `hidssq6` for Hidato. Compared to all other test cases in the same category, `hidssq6` takes *60 seconds* to complete for FC, compared to 0-1 seconds. Looking at the construction of `hidssq6`, it has large groups of unassigned variables in various locations on the grid, which is not present in the other puzzles of the same category. From the exploration vs. exploitation discussion earlier, it is apparent that too more exploration will not be a good idea – this is why FC performs much slower for this particular test case. However, that trade-off is not significant enough to perform worse than GAC. To take this idea further, if we did even *more* exploration and even less inference (FC does do some inference which is why it did better than GAC), then an algorithm like back-tracking (BT) would likely perform longer than GAC.

To conclude the discussion on inference, an analysis of BT reveals that it performs worse than all other inference techniques used. For very small problems of Numbrix however, BT performs just as well as FC (and better than GAC), because of the exploration vs. exploitation trade-off mentioned earlier. For very small problems, exploration is much better. However, this trade-off is quickly compromised as the run-time of BT explodes (e.g a6 to a7).

Limitations

The major limitation for both the Numbrix and Hidato AI was the time taken to solve larger, more complicated grids. Although these puzzles could be solved as well, the time and memory that is required to solve them is excessive, and this is an undesirable property of the AIs.

Conclusion

It can be seen from our limitations that the run time for more complex puzzles (ie. sizes larger than 10x10) can be undesirable. Although our Numbrix and Hidato models are capable of solving most reasonable puzzles in a short time, a different approach may be required to be solve the most complicated puzzles in Hidato. One possible improvement includes having a more efficient algorithm to find satisfying tuples when building the constraints, as this contributes to the total time it takes the AI to solve a problem. Also, removing redundant constraints (i.e. possibly the binary not equal constraints) could also improve efficiency, although this was not done because we could not exhaustively confirm (or prove) that these constraints are not required. Future improvements to the inference techniques used could include the addition of degree heuristics along with MRV (which was already being used), as using degree heuristics would decide tie-breakers in MRV by choosing the variable the affects the most other variables. Using the least constrained variable (LCV) heursitic as opposed to MRV could also have been an option, and would be a heuristic that should be explored in the future to investigate possible improvements. The performances of the various inference techniques reveals that Numbrix and Hidato are games for which a delicate balance of exploration and exploitation (i.e inference) is required (subjected to the aforementioned trade-offs). Thus, it was discovered that FC is the perfect candidate for this problem, in contrast to BT (all exploration, no exploitation) and GAC (little exploration, heavy exploitation). The success of modelling Hidato and Numbrix as a CSP demonstrates its breadth and ubiquity in AI, and it is our hope that this program will find use for the typical consumer irated by a challenging problem.

Appendix

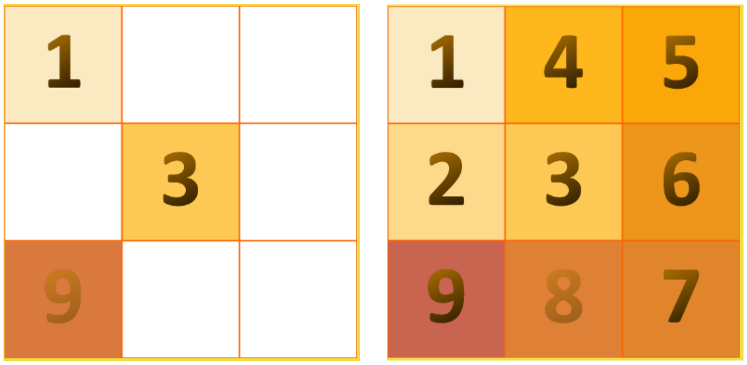

Figure 1: Numbrix Sample Problem (left) and solution (right)

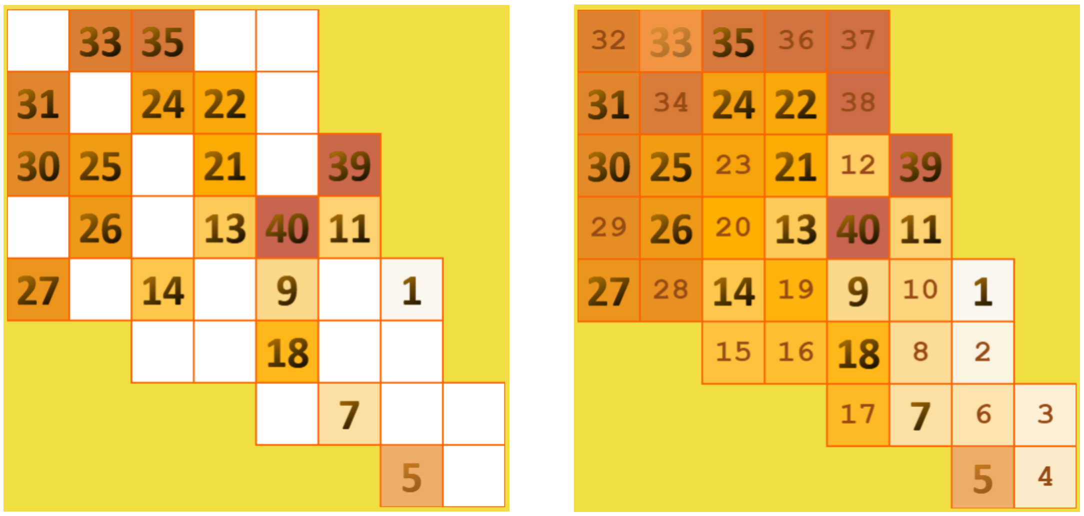

Figure 2: Hidato Sample Problem (left) and solution (right)

Table 1: Test Cases for Numbrix

Grid	a0	a1	a2	a3	a4	a5
Setup	0.0	0.015625	0.015625	0.0	0.046875	0.015625
BT	0.0	0.015625	0.0	0.0	0.0	0.0
FC	0.0	0.0	0.015625	0.015625	0.03125	0.03125
GAC	0.03125	0.015625	0.015625	0.03125	0.046875	0.03125

Grid	a6	a7	a8	a9	a10	a11
Setup	0.203125	1.765625	12.40625	14.140625	92.640625	106.515625
BT	16.671875	2650.575	N/A	N/A	N/A	N/A
FC	0.03125	2.375	2.34375	0.109375	0.421875	0.796875
GAC	0.171875	0.578125	1.15625	2.515625	7.125	38.890625

Table 2: Test Cases Hidato (No Obstacles)

Method	c0	c1	c2	c3	c4	c5	c6	c7
Setup	0.0	0.0	0.25	3.265625	1.125	1.21875	60.859375	7.875
FC	0.0	0.0	0.015625	0.03125	0.015625	0.03125	2.9875	0.421875
GAC	0.046875	0.0	0.609375	5.48375	2.765625	3.453125	58.93125	16.46375

Table 3: Test Cases for Official Hidato (No Obstacles)

Method	hidsq1	hidsq2	hidsq3	hidsq4	hidsq5	hidssq6
Setup	0.796875	2.015625	2.015625	15.265625	48.5	82.390625
FC	0.046875	0.0625	0.0140625	0.03125	1.796875	60.078125
GAC	6.140625	33.140625	13.59375	51.875	175.734375	90.59375

Table 4: Test Cases for Hidato with Obstacles

Method	hid1	hid2	hid3	hid4	hid5
Setup	0.0	0.0	8.75	0.046875	0.015625
FC	0.0	0.0	0.03125	0.0	0.015625
GAC	0.015625	0.046875	4.453125	0.015625	0.03125

Method	hid6	hid7	hid8	hid9	hid10	hid11
Setup	0.0	0.0	12.515625	7.765625	23.59375	33.78125
FC	0.0	0.0	0.015625	0.0625	0.109375	0.265625
GAC	0.015625	0.015625	11.859375	6.125	176.78125	25.03125