**MP3: CSP - Nonogram**

Bum Jun Kim (*bkim102*) Samuel Hung (*shung5*)

**Introduction**

Most of this assignment seemed to revolve around the design aspect of the problem, rather than the actual implementation. Breaking the nonogram solver down into smaller subproblems, we realized that we had to first find a way to get all the different permutations for the given row or column constraint. We saw the nonogram row permutation problem as the most important step of the problem — reducing the number of permutations could provide a significant reduction in runtime.

**Constructing Nonogram as CSP**

***Variables*** - pixels on the map at each given (row, col)

***Domain*** - ‘1’ for black, ‘0’ for white. If multiple colors exist in the given constraints, there are *n* additional numbers for *n* additional colors.

***Constraints*** - each pixel has to follow the given constraint for the row/column. So if the constraint does not allow any 1’s in the column, the variable cannot have the value ‘1’.

**Breaking Down the Permutation Problem**

We realized that this row permutation problem is conceptually the same as the “Stars and Bars” combinatorics problem. In “Stars and Bars”, there are *n* objects (or stars) that need to be split into *k* bins (*k*-1 bars). So instead of treating a block as a series of contiguous elements, we treat it as a singular object. The spaces between the blocks and the remaining open spaces become objects as well. We take this number of objects to be the number of “stars” in this analogy. They have to be split by *k*-1 bars, which is the number of remaining open spaces.

In our code, we have a function that generates all of the possible permutations for this “Stars and Bars” problem. After that, we convert the block elements back into their multi-element counterparts so that we end up with a row or column with the correct dimensions, as well as having the proper representation of a row permutation.

**Constraint Propagation**

However, we are not finished here. Given the row and column permutations, we can determine what remains fixed within each row and column just by extracting what the permutations have in common. Instead of starting with a blank solution map, we can put down some elements that we know for sure and reduce the total number of permutations to take into account. In fact, this process is repeated until the starting solution map stops changing — basically constraint propagation. We chose to use -1 as an indicator that “this block cannot be 1 at all times”.

Below is the process in more depth:

1. *Process Row Constraints*
   * Taking all the possible row permutations, we find the elements that are consistent across all permutations and formulate a row map, partially filled in with values that we know are constant. In our code, we call this the “fixed row map”
   * ex) say the permutations for a given rowindex is :

permutations[rowindex] =[ [1,1,-1], [-1,1,1] ] then,

Fixed\_map[rowindex] = [0, 1, 0]. First and last element can be 1 or -1, but the middle is always 1.

1. *Get Column Data*
   * The idea is to get column data out of our fixed row map. This step essentially transposes the row map to get a column map.
2. *Process Column Constraints*
   * This is the same as step 1, except change “rows to column” and vice versa.
3. *Update Column Permutations*
   * Having extrapolated some row data as column data, we can use this information to narrow down the amount of column permutations we want to process. This step removes the permutations that do not align with the given information thus far, namely the transposed row data in this case.
4. *Get Row Data*
   * This is the same process as step 3, but change “columns” to “rows” and vice versa.
5. *Update Row Permutations*
   * This is the same process as step 4, except change rows to columns and vice versa.
6. *Update Starting Map*
   * Since we now have a “fixed map” for both the rows and columns, we want to combine the two to form a starting map that contains their union. This provides us with an optimized starting map.
7. *Repeat* 
   * Store the optimized starting map, we repeat entire process until the new optimized map matches the previous one, meaning that there can be no further optimization

To our surprise, constraint propagation alone solves most of the provided nonograms! The trend here is that the given nonograms need to be smaller or more given more constraints — the ones that are more sparse will require more work to solve, which leads to the next step.

**Post Constraint Propagation: DFS**

Thanks to the constraint propagation done on predetermined states, we are left with a significantly reduced number of permutations to try. Although DFS is not the fastest searching mechanism, DFS would be, in our case, sufficient due to the reduction of possible states.

1. Out of the row permutation list, we chose the rows with the least number of permutations, yet still higher than 1, since the rows with only one permutation option does not expand the number of options.
   1. Create N number of elements, each with a single permutation for that row.

If the found row has *n* permutations, push into the stack

Row permutations, but for the found row, permutation is first of the *n* permutations

Then push into the stack row permutations, but for the found row, permutation is the second of the *n* permutations . . .

Thus, effectively trying one permutation of that row every time we pop from the stack.

b. The stack consists of row permutations, column permutations, and

row/col\_modified(variable to keep track of whether row or col was modified)

1. Pop an element from the stack.
2. From the shortened row permutation list, we created a fixed map, the same way we did on the constraint propagation. If all the permutation for a row has 1 for a given index, set it as 1, if it always have -1, set it as -1, if it could be either 1 or -1, leave it as 0
3. Transpose the fixed map, and compare with the column permutation list. If any of the permutation list does not agree with the fixed map generated from row permutations, delete the element. If any of the column permutations end up with 0 element, continue back to top of loop, and pop another configuration to try. **This is the forward checking process, decreasing the domain of each row/col as we move on.**
   1. The column permutation list elements are set up as 1D list, so rather than comparing it directly, we transposed the fixed map for better debugging purposes.
4. Repeat step 1, instead of using rows, use column permutations.
5. Pop from the stack, if the element we popped has row/col\_modified indicates row was modified last iteration, we do operations 3-5. If row/col\_modified indicates column was modified last iteration, do operations 3-5 but we would substitute rows for columns and columns for rows.
6. Repeat until solution is found.

**Extra Credit (Heuristic)**

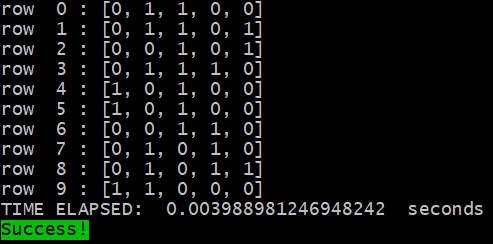
On the 23\_27 extra credit nonogram, the runtime was the same for both Constraint Propagation + DFS and just pure DFS (~12.6 seconds). On 34\_34, there was a 17% reduction in runtime, when using simple Constraint Propagation with DFS, compared to using only DFS

On the 35\_25 bigger input nonogram, surprisingly DFS only took 2125 seconds, compared to with initial Constraint Propagation, it took mere 97% of the time. The runtime reduction factor correlates with how much fixed or always 1/0 state the given constraint has, and how much it reduces the possible permutations of the columns/rows.

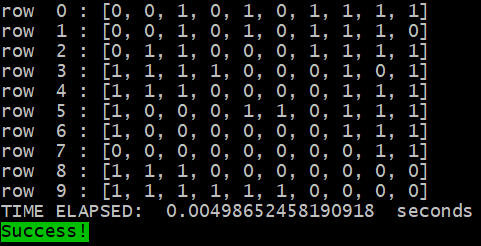
**Solutions**

**Part I: Smaller Inputs**

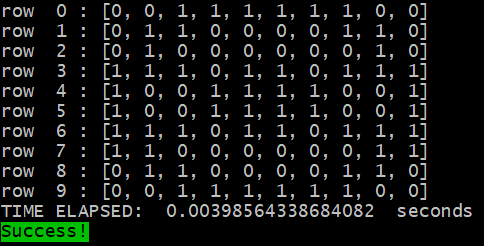
*10\_5*



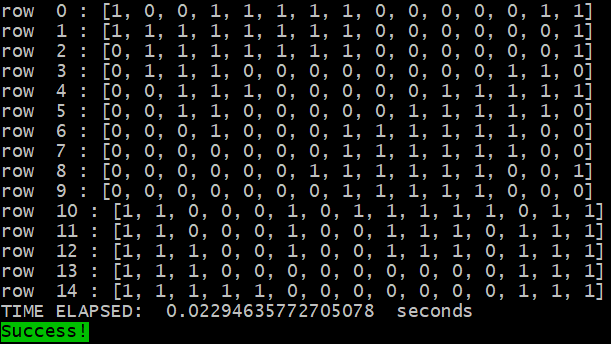
*10\_10\_1*



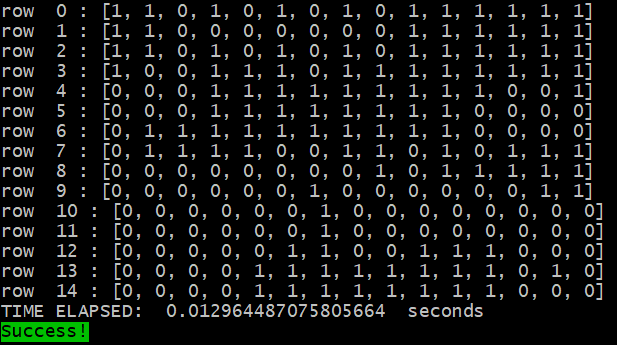
*10\_10\_2*



*15\_15\_1*

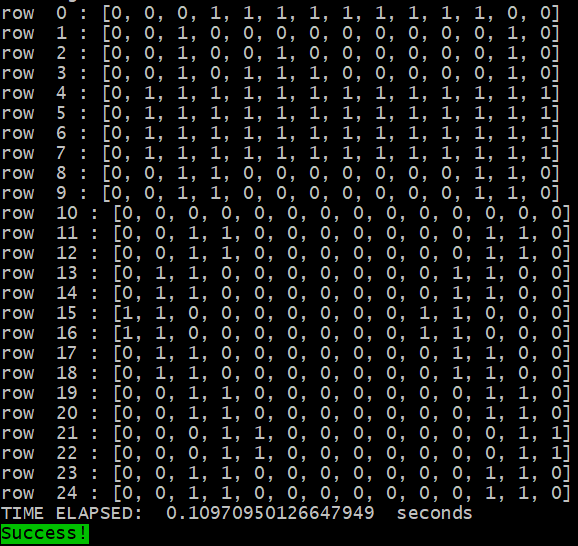


*15\_15\_2*

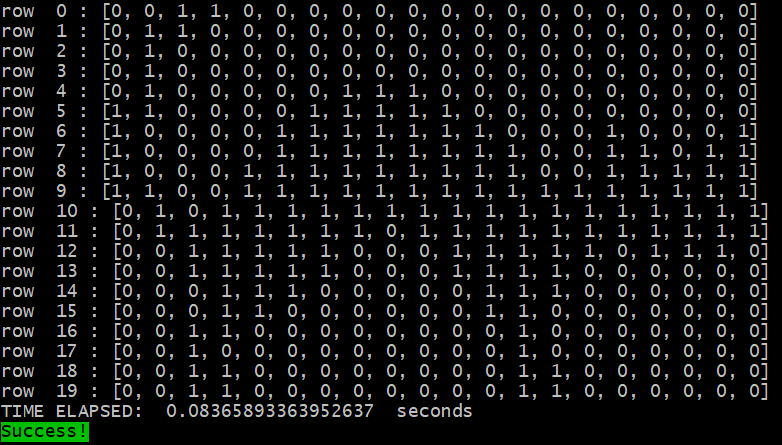


**Part II: Bigger Inputs**

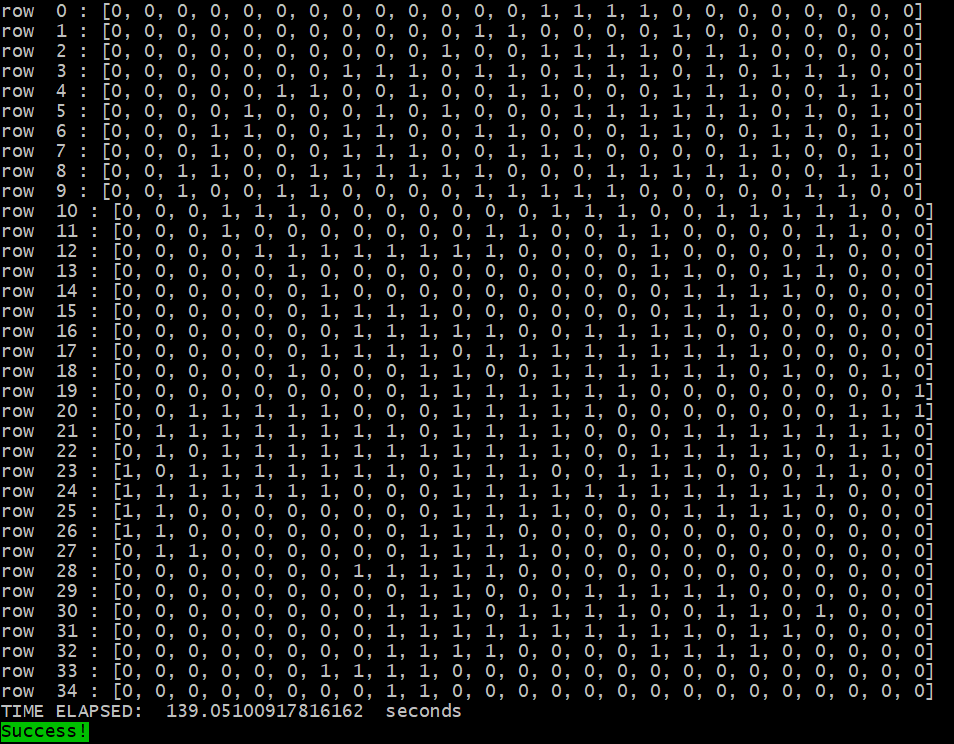
*14\_25*



*20\_20*

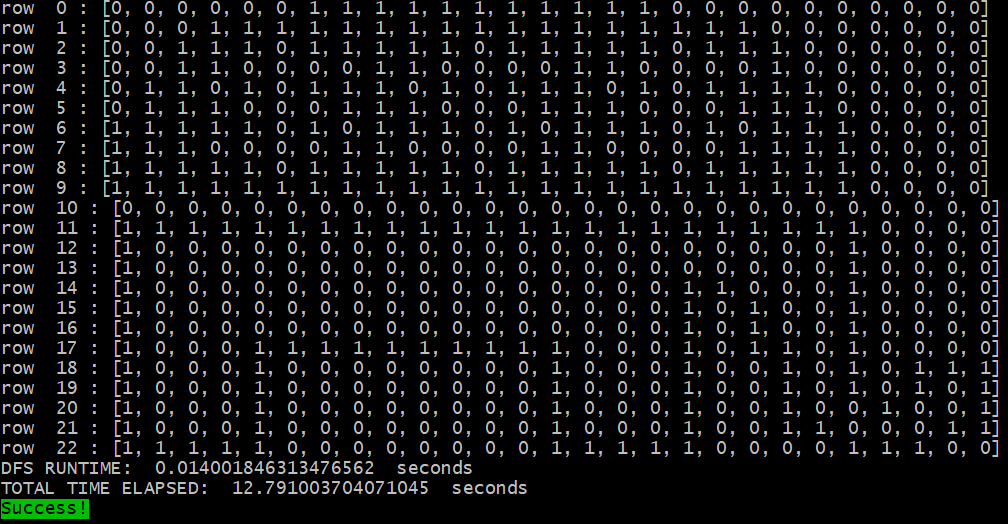


*35\_25*



**Extra Credit:**

*23\_27*

****

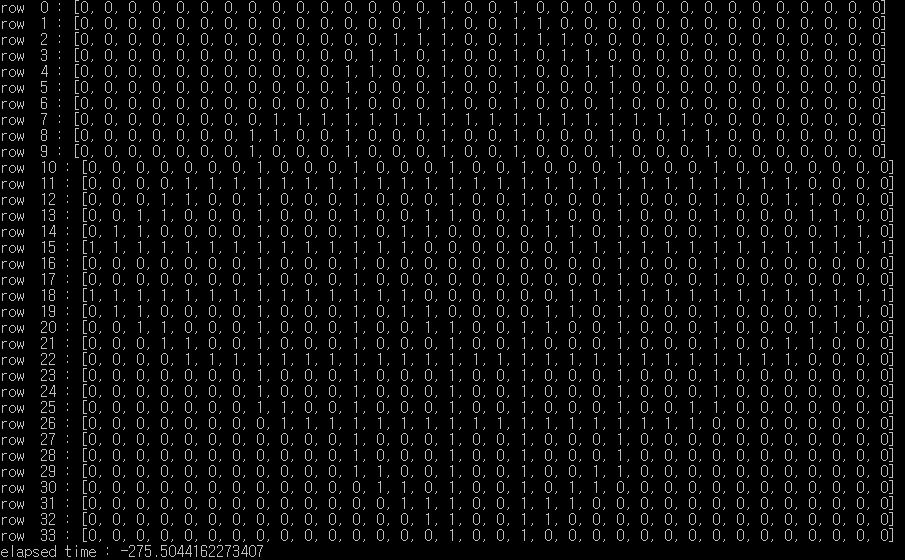
*34\_34*

****

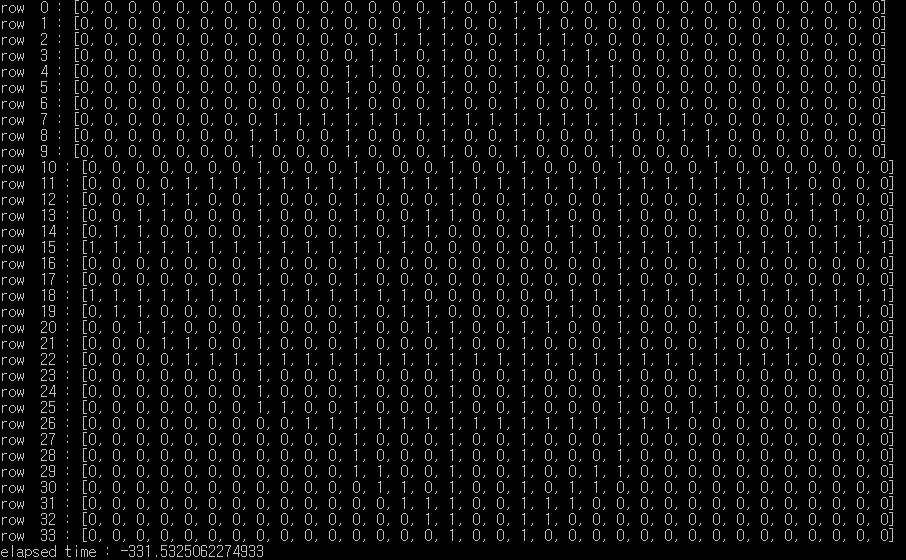
**The images below show runtime on a different machine(Desktop)**

Negative time was due to calculating start time - cur time. Runtime is abs val of shown elapsed time.

*34\_34, with initial constraint propagation.*

****

*34\_34 with simple DFS only.*

****

*35\_25 with initial constraint propagation and DFS*

****

*35\_25 with simple DFS only*

****