2018

Cross Numbers

ASSIGNMENT 2

CARL HUMPHRIES

INTRODUCTION

Cross Numbers is a game concept where given a grid, every second row and every second column would represent equations. And the end elements would be the totals shown in Figure 1.

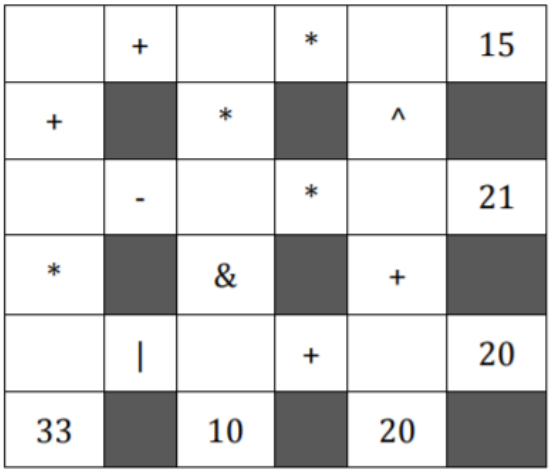


Figure 1

The grid in Figure 1 would have random numbers with some if not all missing. The user would then have to work out what numbers would be needed in each element. The game would be completed once the user has filled out the grid and they would be given a score based on time take and other factors of the game. The user could ask for hints (which would fill in the grid) and enter numbers to try in empty spots.

ALGORITHM DESIGN

Performance analysis

Generating a grid is a trivial task and is done by simply randomising numbers and operators then calculating totals. The grid then would be stripped, by removing numbers from the grid.

While generating a grid is quick and easy, making sure the users move is valid is quite difficult. Also determining a "hint" requires a similar difficulty. To do this the game would need to know all the solutions for a given grid. Resulting in a NP-Hard problem. It is required to brute force to some extent to find all solutions to the given grid.

To find all the solutions given no restriction other than the number range. The size of the state space to search is quite large. Given empty elements n:

$$state\ space = 9^n$$

Not only is the state space large, every combination tried needs to re calculate the equations, which with the best possible evaluator function would be:

$$O(o) \text{ Where o is the operators}$$

So just to brute force every possible solution the time complexity would be:

$$O(o \times 9^n)$$

The state space was reduced by with the addition of limitations to the game, for example; limiting the rows to have no duplicates, and limiting each row to only have x missing elements. This would bring down the brute force method to roughly:

$$O\left(o \times \left(\frac{n!}{(n-x)!}\right)^{row}\right)$$

Where:

- row = amount of rows/columns
- x = missing from each row
- n = 9-(row-x)

Therefore, the algorithm needed would need to follow these restrictions.

Finding a solution

The way this was achieve was with the following method.

1. Generate a grid.
2. For each row find all the number combinations that will lead to a valid row.
3. Check how big the state space would be by using these row solutions (product of lengths of the row solutions). If it is greater than 50000 go to step 1.
4. For all the combinations of row solutions. Check that the columns are valid. If the columns are valid store the solution.

This was effective due to generating a grid is a lot quicker than searching a large state space. The time complexity for generating a grid is roughly:

$O(size^2)$ Where size is number of rows | columns

Therefor the time complexity for this is roughly:

$$O\big((bad+1) \times O(size^2) + (statespace \times size^2)\big)$$

Where:

- bad = number of invalid grids
- statespace = product of row solution lengths
- size = number of rows | columns

Space complexity is given by $O(statespace \times size^2)$

Pseudocode

The following pseudocode outlines how this algorithm was achieved. This is only a summary and source code is well documented allowing for further inspection.

```
Algorithm find_grid(size, diffculty):
//Input: int size size of grid
//Input: int diffculty Difficulty level
//Output: Grid representation String[size][size]
current_grid <- self.gen_grid(size, diffculty)
grid <- copy(current_grid)
while get_solutions(grid) returns false do
    current_grid <- gen_grid(size, diffculty)
    grid <- copy(current_grid)
end while
```

```
Algorithm get_solutions(grid):
//Input: grid to get solutions
//Output: bool to say if grid was "bad"
row_possibles <- []
for i <- 0 grid.size//2 do
    numbers <- list of numbers 1-9 shuffled
    row_possibles.append(permute(numbers, len(missing)))
row_solutions <- list of empty lists length grid.size//2
for i <- 0 grid.size step 2 do
    for each j in row_possibles[i//2] do
        entry <- empty dictionary
        for k <- 0 to len(missing in row i//2) do
            missing <- first missing index
            grid[i][missing] <- j[k]
            entry[missing] = j[k]
        if entry in grid is valid:
            row_solutions[i//2].append(entry)
state_space = product of all row_solutions lengths
if state_space > 50000 or state_space == 0:
    return False
for each combination of row_solutions do
    grid.solutions <-
         check(combination of row_solutions)
return True
```

Evaluate Function

Because of the complexity with equations in a grid structure. An evaluate function was needed, that takes a row and returns the result, this is done recursively by:

1. If there is one element return that value Otherwise…
2. Find the operator that would resolve last. Use this as a splitting point.
3. Recurse on LHS and RHS then resolve LHS operator RHS.

Pseudocode

Below is the Pseudocode for the evaluate function:

```
Algorithm evaluate(e, low, high):
//Input: Array/Str e[] ordered equation
//Output: Evaluated equation result
if low = high: return e[low]
lowest = low+1
for I <- low+1 to high step 2 do
    if getOrder(e[i]) <= getOrder(e[lowest]):
        lowest <- i
operator <- e[lowest]
LHS <- evaluate(e, low, lowest-1)
RHS <- evaluate(e, lowest+1, high)
return doOperation(LHS, operator, RHS)
```

A representation of the grid structure shown in Figure UML. This is where the bulk of the algorithm is held.

Game representation

The difficulty setting determines how many numbers to remove from each row. The Base setting determines how the user will see the results (which number system). And the Size is just how big the game is.

Once the file is loaded the algorithms takes arguments to generate the grid. Then it takes 0.5-5.0 seconds to find all the solutions for that grid. The grid is then displayed to the user as seen below:

```
Size 4|6|8|10|12|14|16|18? 4
Diffculty 1:(Easy)|2:(Medium)|3:(Hard)? 3
Base? 2|8|10|16? 10
Total Proc. Time: 0.08, Start Game Grid (4 unknown, 1 solutions)
     0    1    2    3
0:   ?    +    ?    14
1:   |         &
2:   ?    +    ?    5
3:   13        1
```

If the user wants to enter a move they would type “m”, then enter their desired move. What the algorithm does is check all the solutions to see if the given move is in at least one of them. Then it updates the solutions, grid and unknowns.

```
Current Time: 10.97, Current Game Grid (3 unknown, 1 solutions)
     0    1    2    3
0:   9    +    ?    14
1:   |         &
2:   ?    +    ?    5
3:   13        1
```

If the user would like a hint they would simply type “h”, then the algorithm picks a random solution and then picks a random move from that solution updating the solutions, grid and unknowns.

```
Value 4 added at 2 0
Current Time: 16.99, Current Game Grid (2 unknown, 1 solutions)
     0    1    2    3
0:   9    +    ?    14
1:   |         &
2:   4    +    ?    5
3:   13        1
```

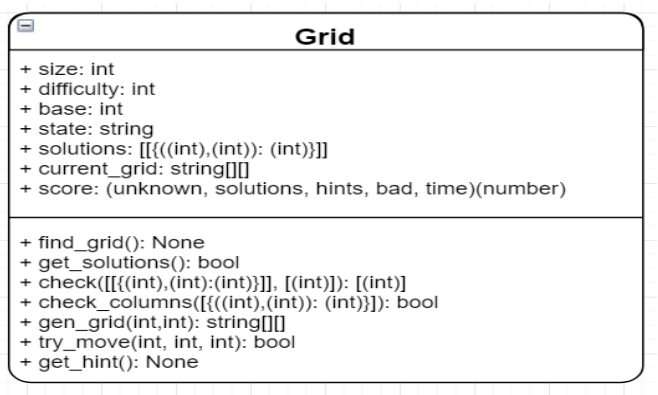
Finishing a game would look like so:

```
Finish Time: 42.29, Finish Game Grid (Score 146)
     0    1    2    3
0:   9    +    5    14
1:   |         &
2:   4    +    1    5
3:   13        1
```

Once the user is finished they will receive a score and the time it took them to complete. The score is calculated by:

$$\frac{\left(unknowns \times 1000 - badmoves - \frac{solutions}{2}\right)}{timetaken + hintsgiven \times 10 \times size} \times difficultylevel$$

Figure UML



EXPERIMENTAL RESULTS

All tests were done on a grid of size 18x18, and hard with most tests done over 1000 new games.

With the assumption that it is quicker to find a grid with reasonably sized state space, then it is to check a large state space. The following test results were analysed:

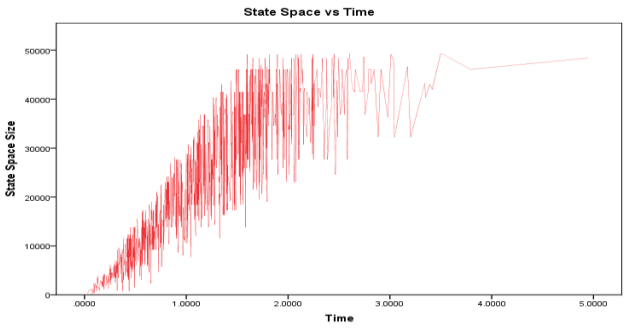


Figure 2

From the results in Figure 2 it is quite noticeable that the size of the state space increases the time quite drastically. Therefore, it is better mitigating the state space before searching it.

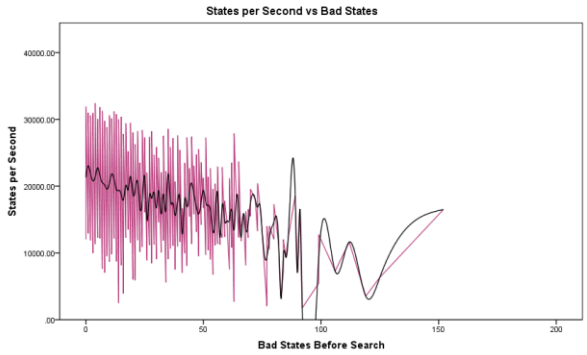


Figure 3

Because of the risk that finding a state space may take many iterations of grid generation, the test in Figure 3 was conducted. It aims to see how much slower the algorithm would be the larger the iterations to find a state space under 50,000. While there is a slight declining trend, this is too small to conclude the regeneration is slower than searching larger spaces.

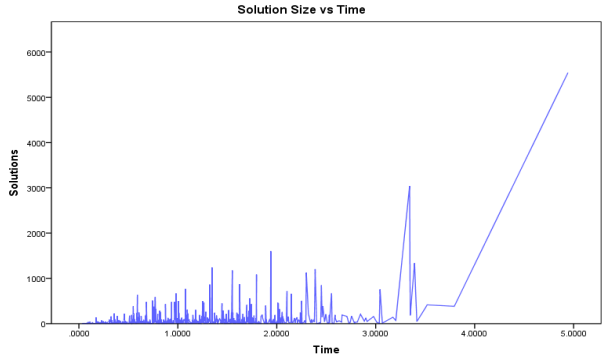


Figure 4

Another factor that could weigh in on the time efficiency of the algorithm is the amount of solutions. From the results in Figure 4, it is quite apparent that there is an increase in time as the amount of solutions increase. There are too many inconsistencies to say for certain that; because there are many solutions, it will take longer.

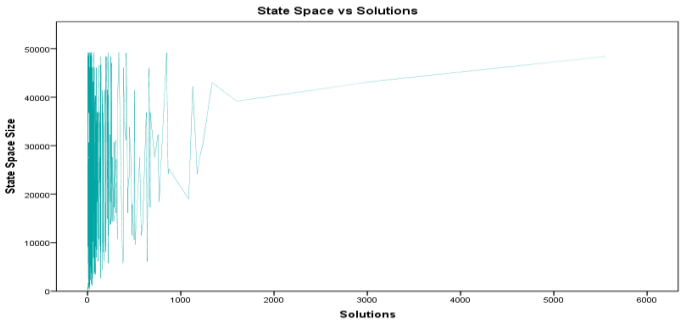


Figure 5

One thing that could be disadvantage of limiting the state space to 50,000. Was that if the state space is small there would be less solutions. But Figure 5 shows that there is no connection in the amount of solutions vs the size of the state space.

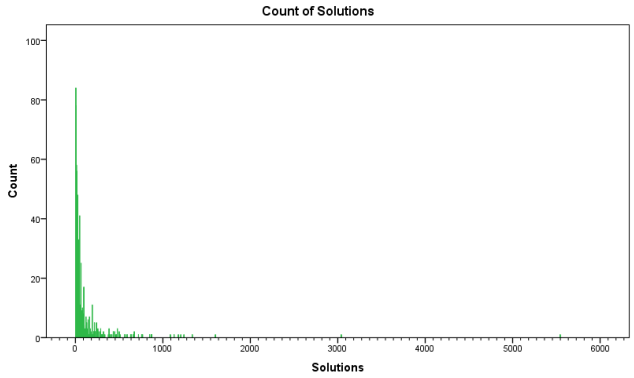


Figure 6

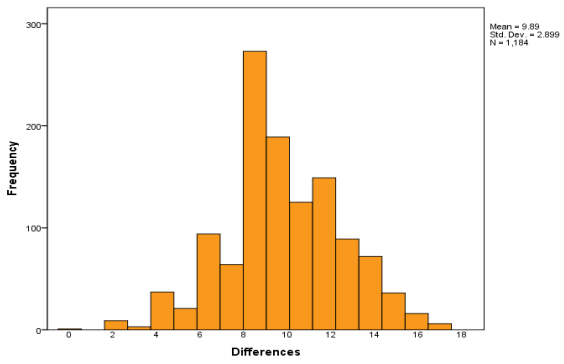


Figure 7

The number of solutions was collected for 1000 trials and plotted against the occurrences Figure 6. Lower number solutions seemed to be a lot more common leading further to the belief that a large state space isn't that important or necessary.

Finally, all the grids solutions were compared to the original grid (for a grid with large solution size). The number of differences was put into a histogram showing interesting results Figure 7. It resembles a normal distribution with a slight difference. Even numbers seemed to be more common. This could be due to that most of the operators' pairs (LHS and RHS) are the same if you swapped the pair.

CONCLUSION

The algorithm used seemed to be quite effective in fulfilling the task. While it has restrictions the game still "seems random" and playable.

The assumption that regenerating the grid until a small state space is found proved to be effective. Given that there was no direct connection to speed with generating larger amounts of grid. And that the amount of solutions wasn't dependant on the state space.

The biggest factor for time complexity was the state space shown in performance analysis and in experimental results. And with the implementation of restrictions to mitigate this, the results seemed good.

One thing to note that the algorithm could potentially take a while finding a grid with small enough state space. But the experimental result showed that the most amount of bad generations was 152 with a time of 2.44 seconds. Which was close to half of the longest time at 4.94 seconds.

Another note is that because the amount of solutions shown to be usually small Figure 6, the results in Figure 4 wouldn't be as much of a factor.

In addition, ~5 seconds is an acceptable wait time for a game of this complexity. With most times being under 2 seconds.