Constraint Satisfaction Solver for Hitori using Backtracking

Anant Bhide

1. INTRODUCTION

In the following project, I am going to implement a Constraint Satisfaction Solver for a logical puzzle named Hitori. Hitori is a number elimination grid game played using the following rules:

- We will be given a grid of squares as input, where each cell contains a number and we are supposed to eliminate the squares.
- The purpose of the game is to modify the initial grid wherein these following constraints be satisfied:
    - No number in a row or column appears more than once.
    - The eliminated cells cannot be horizontally or vertically adjacent, but they can be diagonal.
    - The remaining cells must horizontally or vertically connect with each other. That is all the cells that were not eliminated must be contiguous.

2. METHODS

- **Reading Puzzle**

I am using a text file that contains different input grids. Each line represents the initial numbers for a specific grid. There will be 64 characters in a single line, one for each variable of the puzzle. I am using the following code snippet to read from file.txt

```
file = open ('file.txt', 'r')
problems = file.readlines()
for p in problems:
    g = HitoriGrid ()
    g.read_file (p)
```

The above code creates a HitoriGrid object in memory and the method read_file() takes an input line as a grid and saves the domain of each cell variable to {initial number assigned to that cell + 'X'}. For example, if input number for a cell is 8, the domain of the cell variable be saved as '8X'.

- **Printing Puzzle**

I have defined a method in the HitoriGrid class to print the grid, that would be helpful while debugging the implementation. If the g.print() method is called, it will print the current state of the grid showing which of the cells were eliminated till the function call.

- **Making copies of Grid**

The search function will be easier to implement if we make a copy of grid for each recursive call. In each call, we can fix the value of a certain variable and proceed to check if that path tends to the goal state. If not, we can return from that path and the initial grid won't be altered as we fixed the variable in the grid's copy.

- **Checking if Puzzle has been solved**

I have defined two functions single_variable () and partition that would be used together to check if the final grid is the goal state.

- Single_variable (): returns true if every cell variable has been assigned a single value
- partition (): this method takes the row and column, of an arbitrary cell of the current grid state, as parameters and returns True if there exists a partition (cut) made by the eliminated cells ('X' in the grid), that is white cells have been divided in two parts such that all cells cannot be traversed horizontally or vertically from a random cell.

I use the above two functions using a conjunction to check if we got the final required state.

single_variable() and not partition()

Examples of partition:

```
   |   |   |   |   |   |   |
-------------------
   | 1 | 2 | 7 | 5 | 3 | X |
-------------------
   | 3 | 6 | 5 | 1 | X | 4 |
-------------------
   | X | 1 | 3 | 6 | 5 | X |
-------------------
   | 4 | 5 | X | 3 | 2 | 7 |
-------------------
   | 7 | X | 2 | 4 | X | 1 |
-------------------
```

Consider above is a subset (right down corner) of a grid that our algorithm is working on. Assume that all cell variables have been assigned a single value. Thus, the select_variable function will return true. All the rows and columns have unique numbers (they are not repeated in any row or column) and none of the eliminated cells ('X') are horizontally or vertically next to each other. Even though, the first 2 constraints are satisfied here, there exists a cut that separates 4 from other assigned numbers as we can only access the numbers horizontally or vertically. In this case, partition will return True.

And hence, single_variable () and not partition () will be False, so it is not the goal state.

To check if there is a partition in the grid, I have defined the following helper functions:

- available_moves (self, row, col): returns neighboring cells (horizontal or vertical), of the cell located at position (row, col), that don't have 'X' and have a number assigned.
- remove_eliminated_cells (self, cells): The list "cells" is a list of tuples (i, j), the position of a specific cell. Returns a list of only those cell locations (i, j) from the list cells that don't have a 'X' at their cell location.
- white_cells(self): returns total number of cells in the current grid that have a number assigned.

In the partition function, I perform BFS. The closed list collects all the nodes that can be accessed horizontally or vertically from an arbitrary cell.

I choose the middle cell of the grid (grid[width//2][width//2]) as the start state for BFS so that it can expand in all directions.

If the grid returns true for single_variable, I check if length of the closed list is equal to that of the number of white cells. If they are equal, it implies that all cells can be traversed horizontally or vertically from the middle cell and there is no partition in the grid. If they are not equal, we are not able to access all the cells in the grid horizontally or vertically from the middle cell which means that a partition of the eliminated cells is present in the grid.

- **Selection of variables for each iteration**

I have defined a function called select_variable that returns a tuple (i, j) with index of first cell variable on the grid whose domain is greater than 1.

- **Search function**

I am following the Backtracking pseudocode taught in the lectures to implement the search function.

The function selects next variable as the first cell with domain size greater than 1. It traverses through the domain set and if the current value is consistent, it creates a grid copy, updates the variable's value in the copy and continues in the same pattern.

If all variables are assigned, it starts BFS from the middle cell. If the middle cell is already eliminated, it takes one of the neighbors as the start state. If a partition of eliminated cells is not encountered then the grid has not been solved.

(I have used 13 random examples from the link provided below. H8 contains all the grids and the h8_answers contain the solutions to the questions).

REFERENCES

1. Wikimedia Foundation. (2021, December 30). *Hitori*. Wikipedia. Retrieved December 9, 2022, from https://en.wikipedia.org/wiki/Hitori
2. *Daily hitori*. BrainBashers. (n.d.). Retrieved December 9, 2022, from https://www.brainbashers.com/showhitori.asp
3. L. Lelis – CMPUT 366 assignment 4, 2022