CSCI 3901 Winter 2021

Assignment 4

-Dinesh Kumar Baalajee Jothi – B00861292

Problem -1

Test cases for Problem-1

❖ **<u>Input Validation:</u>**

➢ **public boolean loadPuzzle(BufferedReader stream):**

1. **Stream is null: Return False**
2. **The input contains empty space at the start : Return False**

   a. Section 1:
      1. Input size is 0: Return False
      2. Input is Null: Return False
      3. Input size is between 3-9: Return True

   b. Section 2:
      1. Input line characters lesser than size(n): Return False
      2. Input line characters greater than size(n): Return False
      3. Input line characters equals size(n): Return True

   c. Section 3:
      1. The input line has more than 3 values: Return False
      2. The input has a negative result: Return False
      3. The input has more than one space between grouping: Return True

➢ **public boolean validate() :**
The following are checked for validation purpose which is given in the assignment pdf:

1. If the input is not an nxn puzzle: Return False

2. If the input is an nxn puzzle: Return true at last

3. If the grouping does not form connected set of cells: Return False

4. If the grouping forms connected set of cells: Return true at last

5. If the grouping does not have a value and an operator: Return False

6. Every grouping with the = operator has more than one cell: Return False

7. Every grouping with the − or / operator does not have exactly two cells: Return False

8. Every grouping with the + or ∗ operator must have less than two cells: Return False

- **public boolean solve():**
  No input validation since there are no input parameters

- public String print():
  No input validation since there are no input parameters

- public int choices():
  No input validation since there are no input parameters

❖ <u>Boundary Cases:</u>

- **public boolean loadPuzzle(BufferedReader stream):**
  a. Input size is lesser than 3: Return False
  b. Input size is greater than 9: Return False
- **public boolean solve():**
  a. Call solve() without loading/Validating : Return False
  b. Call solve() without validating wrong input: Return false
  c. Call solve() and again call solve(): Return true
- **public String print():**
  a. Call print() without solving : Return empty string
- **public int choices():**
  a. Call choices() without solving : Return 0

❖ <u>Control Flow:</u>

❖ **public boolean loadPuzzle(BufferedReader stream):**

a. Get the input stream -> Get the size of the puzzle -> If the size is not between 3-9 -> Return False

b. Get the input stream -> Get the size of the puzzle -> If the size is between 3-9 -> Get the cell grouping character of the nxn puzzle square -> If the grouping is not a nxn puzzle -> Return False.

c. Get the input stream -> Get the size of the puzzle -> If the size is between 3-9 -> Get the cell grouping character of the nxn puzzle square -> If the grouping is a nxn puzzle -> Get the grouping constraint -> If grouping is not correct -> -> Return False

d. Get the input stream -> Get the size of the puzzle -> If the size is between 3-9 -> Get the cell grouping character of the nxn puzzle square -> If the grouping is a nxn puzzle -> Get the grouping constraint -> If grouping is not correct -> Load the puzzle -> Return True

❖ <u>Data Flow:</u>

➢ **Normal flow (Expected order):**

❖ **public boolean loadPuzzle(BufferedReader stream) -> public boolean validate() -> public boolean solve() -> public String print() <-> public int choices()**
Steps:

1. The puzzle is loaded
2. The input to the puzzle are validated
3. The puzzle is solved
4. Printing the puzzle (or) Printing the choices

➢ **Un-expected order:**

1. Call validate() before loadPuzzle()
2. Call solve() before loadPuzzle()
3. Call solve() before validate()
4. Call print() before solve()
5. Call choices() before solve()
6. Call solve() continuously
7. Call loadPuzzle() two times
8. Call validate() two times

Solution Explanation:

1. We first load and validate the puzzle.

2. The following are the steps taken to solve the puzzle:
   a. First a puzzle is created with “0” in all the cells.
   b. The puzzle is then filled with result value of the “=” operator.
   c. Using the above filled puzzle, we then iterate through every row & column and we get the specific cell of every row and column.
   d. If the cell has “0” we then get the grouping alphabet of the particular cell with the grouping operator and the result.

   e. We use the operator to find the correct operator to do the operation.
   f. We then try all the possible values with the operation to find the desired result. When

we get the desired result, we compare it in row and column to confirm it does not violate the row/column rule since we have filled the puzzle with “=” operator already.

g. If the value we got is correct, We then add it to a hashset of the cell.

h. We do this for all the cells in the puzzle.

i. At the end of the iteration, we will have a set of values for every cell which satisfies the grouping operation and result.

j. We then use the **SolvePuzzle()** method, to try solving the puzzle using recursion and backtracking.

   a. We get the cell which is empty from the puzzle we created. The row and column of the unfilled cell is got from the emptycell.

   b. We try all the values from 1 to n to be placed in the particular cell.

   c. We use the method **Valid()** to check if the values is valid to be placed in the cell

Valid():

   a. This method checks if the value is in the particular row

   b. If not in the row, It checks if it is in the column already

   c. If not in the column, it checks if the value satisfies the grouping constraint and is present in the set of value we got from **ExpectedValue()** method using the **CanPlace()** method

CanPlace() :

   a. We get the particular cell using the row and column.

   b. We check if the value is in the set of values of the cell which we have already got using the ExpectedValue() function.

   c. If the value is present we then check if the value form the correct grouping using **CorrectGrouping()** method:

      i. The correct grouping method gets the particular operator and the result of the cell using the row and column.

      ii. It checks if it satisfies the grouping operation and result.

      ii. If it satisfies the condition, true is returned.

   h. If all the above conditions are satisfied then the value is placed in the particular cell.

   h. Then the SolvePuzzle() method is called recursively.

   i. If a value is not set in a cell then we return false, and we set 0 in the previous cell and go back to try different value until a correct value is set for this cell thus backtracking until we get the correct value.

   j. If all the values are set True is returned and when we can’t set the correct value to solve the puzzle false is returned.

Steps taken to provide efficiency:

1. After creating a puzzle we fill the cells of the puzzle with the result of the “=” operators, since there is no choice in choosing the value for the “=” operator.

2.When placing an order if it is not the last choice in the group, it won’t be checked for the operation, rather we check only if the value is the last element to be placed. (E.g. If a has 3 cells with addition operation, we won’t check any addition condition for the first two values, we check the addition only for the last values like “ if v1+v2+v3(the value we are trying to insert) == result, so that we can assume most of the times the group to be correct which limits dead-end (i.e. backtracking)).

3. After filling the puzzle with the result of the “=” operator, we use that puzzle to get the possible values of all the cells, so that when we try to fill the solve the puzzle, we check only the valid possible values rather than checking for every values. When getting these values, we also check if the value is already in the row or column because we might have some values already in the puzzle when we filled the cells using the “=” operator. It provides efficiency in choosing the values to solve the puzzle.

4. When solving the puzzle, we choose only the empty cells to be filled (i.e. we neglect the cells whose value is already placed which might be the somewhat filled puzzle with the “=” opeartor), we check if the value we are trying to place is in the row or the column already and also check if it satisfies the grouping condition before placing the value in the cell. By, doing this we only set the possible values from the set we have eliminating the row and column ambiguity.

Reference:

1. Referred to https://www.geeksforgeeks.org/sudoku-backtracking-7/ to get more understanding about backtracking.