Methods:

We have decided to follow a CSP approach because sudoku solving rules fit into it very easily. However, this algorithm doesn’t need a set of rules to be defined rather the domain of the variable is derived such a way that no conflicting values exists in the domain. To be more specific, we didn’t start by defining a domain for every variable (in this case a cell in sudoku matrix) like a traditional CSP, instead we started with domains of each row, column and unit box (3x3 regions) and used these as parent for the cell domains. Initially, the algorithm fills each row, column and unit box domains with all allowable values (1 to 9) then removes ones that are already defined in the problem description from the respective domains. When an empty cell is picked, the domain is derived from its parent row, column and box. See the equation for further clarification

Furthermore, this algorithm adopts recursive behavior in order to implement backtracking and eliminating complexity in terms of memory usage. If a partial failure is occurred this algorithm backtracks to root of the problem and tries using a different value. Hence, this will always find an answer if there one.

We have tested this algorithm with a subset of “1 MILLION SUDOKU PROBLEMS” dataset. We have separated a subset of 10000 problems from it and analyzed generated output by matching it with the given output. The results are astonishing!

**Variables:**

*prob[][] 🡺* The problem stored in a 2D array *rowDom[] 🡺* Array of domains of all rows by index *colDom[] 🡺* Array of domains of all columns by index  
*boxDom[] 🡺* Array of domains of all unit boxes by index *rowDomList 🡺* Sorted list of row domains by size (ascending)  
*colDomList 🡺* Sorted list of column domains by size (ascending)  
*rem* *🡺* number of remaining cells to be filled

**function** solve() **returns** success **or** failure **{**  
 **if**(*rem* = 0) **return** *success*  
 *pos[]* 🡨 *getCellPosition()* //returns position of the cell with least domain size  
 *dom* 🡨 *getDomain(pos)* //returns domain of the cell  
 *cell 🡨 prob[pos]*   
 **while**(*domIsNotEmpty()*)**{**  
 *num* 🡨 *dom.pop()  
 cell 🡨 num* *popNumFromAllParents(num,pos)*  
 *sortAllLists()  
 rem = rem – 1  
 result 🡨 solve()* //recursive call**if***(result* = *success)* **return** *success  
 pushNumToAllParents(num,pos)  
 rem = rem + 1* **}**  
 *cell* = 0  
 **return** *failure***}**

**function** getCellPosition() **returns** row, column & unit box position of a cell **{  
 for** *i* = 0 **to** *smallestDomainListSize()***{**  
 **for** *j* = 0 **to** *largestDomainListSize()***{** *ri\_cj 🡨 rowDomList(i)* + *colDomList(j)   
 rj\_ci 🡨 rowDomList(j)* + *colDomList(i)*   
 *min* 🡨 *findMinDomSize(ri\_cj, rj\_ci)  
 pos[] 🡨 {min\_row, min\_col, min\_box}* **if**(*prob[pos]* = 0) **return** pos  
 **}  
 }** *pos[]* 🡨 {-1,-1,-1}  
 **return** *pos***}**

**function** getDomain(*pos*) **returns** a Domain **{** *dom 🡨 rowDom[pos\_row] ∩ colDom[pos\_col] ∩ boxDom[pos\_box]* **return** *dom***}**

Result:

Since we are using CSP which is an algorithm that finds a solution rather than predicting it, there are no false positive results. After we tested our program on 10000 problems, we found that there were no inaccurate results. Every single generated output matched with the given one. As a result, the algorithms accuracy is 100%

Moreover, as our problem is Sudoku, the answer is a configuration, meaning there are no negative outcomes. Hence, precision is also 100%

Discussion:

Any constraint problem like Sudoku can be tackled using CSP approach. It is much more efficient and accurate compared to other approaches. We have tried solving sudoku with genetic algorithm at first. But the algorithm was stuck in a local minima. Furthermore, the evaluation function was much more complex. CSP can deal with these type of problems very easily