1. **Sudoku Solver using CSP**

   For 9*9 sudoku,

   Variable:
   Each slot of sudoku acts as one variable. So, there are 9*9 slots ie. 81 variables.

   Domain:
   Each variable in sudoku can take any value from 1 to 9.

   Constraints:
   These constraints are formulated as a constraint matrix of 81*81 variables. Here a value of 1, denotes constraint present between the variables.

   - **Row Constraints:**
     Each row must contain values from 1 to 9.
     If 1st slot is taken then 2 to 9 slots will conflict, setting Matrix[1][2], Matrix[1][3] and so on, all to be 1.

   - **Column Constraints:**
     Each column must contain values from 1 to 9.
     If 1st slot is taken then 10,19,28,37,46,55,64,73 slots will conflict, setting Matrix[1][10], Matrix[1][19] and so on, all to be 1.

   - **Box Constraints:**
     Each box must contain values from 1 to 9.
     If 1st slot is taken then 2,3,10,11,12,19,20,21 slots will conflict, setting Matrix[1][2], Matrix[1][3] and so on, all to be 1.

   "To get a consistent and complete solution to this CSP, every cell must be assigned and should follow all the above three constraints" .

   The search for sudoku solution differs in how an unassigned variable is chosen and how its next value is to be taken. Also, if constraints are propagated it will reduce more number of domain values giving faster results.

- **Backtracking Search(BT):**

  Selection of variable: Any first unassigned slot is chosen

Next value: Domain values are selected from the possible values

Constraint propagation: No constraint is propagated, it outputs false when a constraint is not followed and then backtracks.

- **Minimum Remaining Values(MRV):**

  Selection of variable: A variable with minimum remaining values

  Next value: Domain values are selected from the possible values

  Constraint propagation: No constraint is propagated, it outputs false when a constraint is not followed and then backtracks.

- **Least Constraining values(LCV):**

  Selection variable: Variable with minimum remaining values

  Next value: Values chosen with least constraints or more possibilities

  Constraint propagation: No constraint is propagated, it outputs false when a constraint is not followed and then backtracks.

- **Maintaining Arc Consistency(MAC):**

  Selection variable: Variable with minimum remaining values

  Next value: Values chosen with least constraints or more possibilities

  Constraint Propagation: Constraints are propagated with each assignment. So, with each assignment domain values are reduced to get consistent assignments. If there is no domain value left for a particular variable, then immediately, we backtrack unlike all the above searches described above.

Running Sudoku Solver:

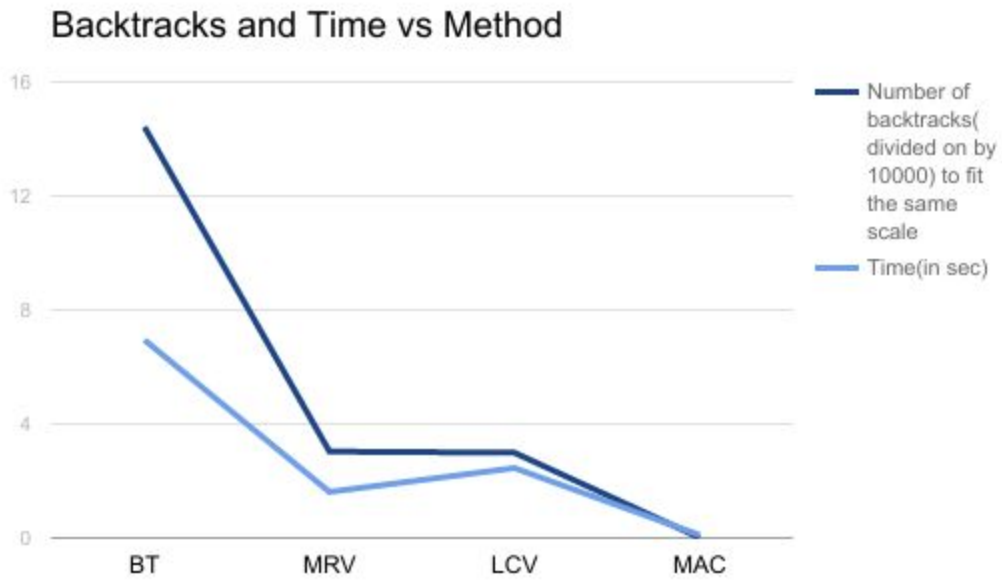
python l3.py p.txt>stats.txt

For running a particular method change line 7 with BT, MRV, LCV or MAC stats.txt gives time taken, number of backtracks and resources used by sudoku.

Observations:

Comparing the various search methods by running first sudoku solved in p.txt file.

	Time taken (in sec)	Number of Backtracks	Resources used (in MB)
Backtracking (BT)	6.9486	1,44,202	6
Minimum Remaining Values (MRV)	1.6165	30,323	6
Least Constraining Value(LCV)	2.4546	29,918	6
Maintaining Arc Consistency (MAC)	0.1062	146	7



- **Backtracking Search:**
  - ➔ We could clearly observe that **number of backtracks is maximum** for this case. As it does not use any heuristic to select a variable and to select the values out of that variable.

  - ➔ As, backtracking search has the maximum number of backtracks, it takes **maximum time** to solve sudoku.

  - ➔ Amount of resource used is almost equivalent to all the rest of the methods.

- **Minimum Remaining Values**
  - ➔ It uses a heuristic to select the most constraining variable ie. variable with minimum remaining values, due to which **number of backtracks reduces to 4 times than that of backtracking search**.

  - ➔ As number of backtracks reduces by 4 times, the time taken to execute MRV also reduces drastically from 6.9486 sec to 1.6165 sec.

  - ➔ Amount of resources utilised is almost same to that of Backtracking search.

- **Least Constraining Value(LCV):**

  ➔ It uses a heuristic to select the most constraining variable as well as that value is chosen for variable which has least constraints. As it combines both the heuristics, number of backtracks reduces drastically in comparison with backtracking search.

  But, there is a **very slight change in number of backtracks between MRV and LCV**.

  ➔ Time reduces drastically in comparison to that of BT but it slightly increases in comparison to MRV.

  This might be due to extra cost incurred while calculating one more heuristic for the values.

  ➔ Amount of resources utilised is same to that of BT and MRV.

- **Maintaining Arc Consistency(MAC):**

  ➔ **Number of backtracks are minimum** in the case of MAC.
  This is due to “constraint propagation” that happens at every assignment in case of MAC along with heuristics used for MRV and LCV.
  It reduces the number of domain values to least, leading to less domain values that needs to be checked.

  The number of backtracks are almost reduced to 987 times than that of Backtracking search.
  Number of backtracks are almost reduced to 200 times in comparison to MRV and LCV.

  ➔ As the number of backtracks are reduced to 987 times and approximately 200 times to BT and MRV respectively.
  The **time taken to solve a Sudoku problem reduces almost to fraction of a second** ie 0.1062 sec

➔ The **resources used in MAC although increases** to 7 MB.
This is due to extra usage of stack in MAC that is required to keep track of all the domain values of the variable selected.

- Despite the differences observed in above methods for computing the solutions to sudoku, all the methods **output the same solution**.
  This is due to unique solution of Sudoku.

- For all the sudoku's, we observed that **MAC takes the least number of backtracks and minimum amount of time**. Although extra resources used are more.
  Total time taken to solve all 150 sudokus using **MAC is 160.0578 sec~3 mins with resources~ 271 MB**

- **Backtracking search** for solving all 150 sudoku's took **11443.7773 secs~3 hrs 17 mins with resources ~24 MB**

- In some sudoku's we also observed that LCV encounters more number of backtracks than MRV.

  Like second sudoku problem observes 836 backtracks in case of LCV but only 114 backtracks in case of MRV which tells that choosing least constraint value may not be always be helpful but sometime randomly choosing any value for variable may help.

- Although we did observe that LCV takes more than MRV for almost all the cases. This might be due to extra cost incurred as one more heuristic is calculated.

- In most cases backtracking perform the worst, but in some rare cases sudoku puzzles performs better than when chosing with heuristic LCV or MRV. But always worse than that of MAC.

  <u>Conclusions:</u>
  All of the above methods give the same solution to sudoku but differ in terms of time space and backtracks.

  ➔ Backtracking search in most cases gives the worst performance but consumes less memory resources.

➔ MRV and LCV perform almost similar but generally better than Backtracking search.
➔ MAC gives the best performance in computing the solution to sudoku in terms of time but occupies more memory resources.

2. Sudoku Solver using Minisat

9*9 sudoku is a puzzle with 81 slots that could be filled up by 9 different values i.e for each slot we have 9 variables. So, total consisting of **81*9=729 variables**. The number of clauses that could be derived from Sudoku depends on the number of values already assigned to the sudoku.

Conversion of constraints to boolean sat problem:

There are 5 types of constraints present in sudoku that rises to various clauses :

- **Already assigned constraints:**
  For the already assigned values for slots,variable will always be true. So forming 1 clause for each of the variables assigned.
  So, it forms number of clauses equal to number of assigned variables.

  Given the index and value, it converts to variable by 9*index+value. This variable is true, so it forms a clause like 42 0. This is shown below:

```
for i in range(len(self.dict)):
    if self.dict[i]!=0:
        valuedVariables.append(i)
totalClauses=totalClauses+len(valuedVariables)
filestr=filestr+str(totalClauses)+"\n"

#already assigned constraints
for index in valuedVariables:
    filestr=filestr+str(index*size+self.dict[index])+" "+"0"+"\n"
```

- **Slot constraints:**
  Each of the slot in sudoku will always take atleast one value and atmost one value.
  So designing clauses for both atleast as well as atmost one value.
  1. **Atleast one value for each slot** implies either of 9 variables at one slot will be true, eg. for 1st slot either variable 1,2,3,4,5,6,7,8 or 9 will be true.
     This gives rises to 81 more clauses as every slot will form one new clause. It is done by printing this pattern by below lines of code

```
#clauses that atleast one should be true
string=""
for i in range(len(self.dict)):
    for j in range(size):
        string=string+str(size*i+j+1)+" "
    string=string+"0"+"\n"
```

2. **Atmost one value for each slot** implies for each slot either of two variables will be false out of nine variables. So, constraints will be of form -1 -2 0 and so on uptil -8 -9 0 for 1st slot.

   This would be done for 81 such slots giving rise to (81*9*8)/2 clause=2916 clauses. These clauses are formed by:

```
def getClauses(self):
    string=""
    for i in range(1,size*size*size,size):
        for j in range(i,i+size):
            for k in range(j+1,i+size):
                string=string+str(-j)+" "+str(-k)+" "+"0"+"\n"
    #print string
```

- **Row constraints:**
  In each row there should be atleast one value from 1 to 9.
  So, for each row atleast one variable with value 1 should be true i.e out of 1 valued variables of 1st row, one variable must be true. The clause for value 1 row 1 is:

  1 10 19 28 37 46 55 64 73 0

  Similarly, repeating for value 2 in 1st row and so on for all the rows.

  Thus 81 clauses each of length 9 could be formed by below function:

```
def getRowClauses(self):
    string=""
    count=0
    for i in range(0,size*size*size,size*size):
        count=0
        p=0
        while (count<size):
            count=count+1
            count2=0
            j=i+1+p
            while (count2<9):
                count2=count2+1
                string=string+str(j)+" "
                j=j+size
                #print(j)
            string=string+"0"+"\n"
            p=p+1
    #print string
    return string
```

Here outer loop runs to increment the variable from one row to another, inner loop corresponding to variable for each value from 1 to 9 and innermost for each slot.

- **Column constraints:**
  In each column there should be atleast one value from 1 to 9.
  So, for each column atleast one variable with value 1 should be true i.e out of 1 valued variables of 1st column, one variable must be true. The clause for value 1 column 1 is:

  1 82 163 244 325 406 487 568 649 0

  Similarly, repeating for value 2 in 1st column and so on for all the columns forming 81 clauses each of length 9 could be formed.

```
def getColumnClauses(self):
    string=""
    for i in range(1,size*size+1):
        count=0
        j=i
        while (count<size):
            count=count+1
            string=string+str(j)+" "
            j=j+size*size
        string=string+"0"+"\n"
    #print string
    return string
```

Here outer loop runs to increment the variable from one value of column to another and inner loop for each slot.

- **Box constraints:**
  In each 3*3 block, there should be atleast one value from 1 to 9. So, for each box atleast one variable with value 1 should be true. The clause for 1 box with value 1 is:

  1 10 19 82 91 100 163 172 181 0

  Repeating this, forms 81 clauses each of length 9.

```
def getBoxClauses(self):
    string=""
    for i in range(squareSize):
        for j in range(squareSize):
            for value in range(size):
                for k in range(squareSize):
                    for l in range(squareSize):
                        val= (k+i*squareSize)*size*size+(l+j*squareSize)*size
                        string=string+str(value+val+1)+" "
```

Total number of clauses formed= Number of already assigned clauses +
Number of slot clauses +
Number of Row clauses +
Number of Column clauses +
Number of Box clauses
= Number of assigned clauses +
81(atleast)+2916(atmost)+81+81+81+81
= Number of assigned clauses + 3240

For 1st sudoko,
Assigned number of variables=23
Total clauses formed are 3240+23=3263

- Getting constraints into Boolean CNF gives an intermediate file “minisat.txt” which is then used to call Minisat.

Running minisat:

python minisat.py p.txt /directory_to_minisat_executable/

Understanding output of Minisat to solve Sudoku:

- Running minisat creates a minisatoutput.txt file whose first line indicates Satisfiable or Unsatisfiable.
- If satisfiable it prints the value of all variables.Positive indicates variable to true and negative indicates variable to false.
- If variable is positive, value to Sudoku is assigned by:

  index=(variable-1)/size
  newValue=variable%size
  where size=size of sudoku=9

- Final output of all the solved sudokus are printed in finalminisat.txt

Observations:

- Total time taken to solve all 150 sudoku's using SAT solver=61.0155 secs
- The output of finalminisat.txt remains same as that of CSP solved.