# SUDOKU SOLVER USING CSP

## Abstract:

In this project the Backtracking algorithm with its variations for the propagation of constraints is implemented in Python language, in particular we will use a pure Backtracking (without inference), BT with Forward checking and BT with MAC (Maintaining Arc Consistency), in the final part of the exercise we will compare their application on different instances of Sudoku.

## 1. Introduction

The developed software mainly implements the backtracking algorithm with its abstract representation in which the way in which:

- Choose the variable to assign a value to.
- Choose the value to assign to the chosen variable.
- Make inferences about other variables.

The code provided was taken into consideration as implementation basis, this was slightly modified in order to improve the performance of applying the algorithms to different instances of CSP, however the code remained general enough in so as to allow the solution of a generic problem, which however must specify:

- The variables to take into consideration.
- The domain of variables.
- The relationship that binds the variables.
- Variables linked to a certain variable by a relationship.

## 2. Backtracking

As has already been anticipated in Section 1, the backtracking algorithm has been implemented in this project, which however will present different representations, for the choice of the variable the heuristic has been taken into consideration Minimum Remaining Values (MRV) which allows us to choose the variable with the smallest domain in order to make backtrack quickly (also called fail-first), for the choice of the value to assign instead no heuristic was taken into consideration, this because normally in the Sudoku problem there is only one solution therefore it is in different to try to find the first solution (in case of failure the developed program will show an error message since it is not possible to find the solution), instead it is useful to try to realize that we have chosen the wrong variable with the MRV, this is also very intuitive, for example in the Sudoku game if we see a row with only one free box (Domain with a single element) the first thing we do is assign a value to this box, instead for regarding the propagation of constraints we have 3 different options:

## 2.1 Backtracking Pure

In the case of pure backtracking, i.e. without inference, when choosing a value to assign to a certain variable v1, one proceeds by calling the conflicts method which returns the number of violations that would occur when assigning this value to v1, if the number of violations is 0 we proceed with the assignment of v1 (without informing the other variables of this choice, for this reason their domain will continue to be unchanged) to then choose a new variable recursively, however when a variable v2, (with v2 = v1) does not find a value for which the number of violations (with respect to the assignments already made) is 0, the algorithm realizes this and proceeds with the backtrack by changing the value assumed for v1, so it is natural to expect a large number of backtracks in solving any CSP problem. Max

## 2.2 Backtracking with Forward-checking

Unlike pure backtracking, with forward-checking backtracking, when a value is assigned to a variable v1, the variables (linked to this by a constraint) are informed so that they reduce their domain and then decrease the number of possible states to analyse, i.e. we try to cut the tree so that it is much faster to find the solution, these cuts do not cancel the solution since they are cuts of arcs that are not consistent with the assignment made.

For example, in the case of Sudoku when a box takes on a value, it informs the variables of the same row, column and square box so that they do not have this value in their domain, otherwise there would be an inconsistency between the variables, so in this case it is natural to expect a smaller number of possible analysed states, i.e. a much smaller tree than in pure backtracking, and therefore also a smaller number of backtracks.

## 2.3 Backtracking with MAC

As already described in Section 2.2 forward-checking allows us to infer variables related to a variable v1, and if it is found that giving a certain value to v1 causes an empty domain in a variable v2 (related to v1 from a constraint) then go back and change the value, but it is useful to note that this could have been predicted even before assigning the value to v1, this solution is called Maintaining arc-consistency, after assigning a value to one variable v0, the AC-3 is called on all neighbours of v0 (among which there will also be v1), the values in D1 are removed for which there would be a domain void between its neighbours (among which there will also be v2), and if |D1| = 0 then there is an inconsistency in assigning the value to v0 and therefore we proceed by changing the value for v0, in this way we are able to look ahead when we assign a value to a variable.

As far as backtracking with MAC is concerned, a lower number of backtracks is expected than with FC but with a time that could be slightly high since the backtracking steps with MAC are longer, i.e. they require a higher computational cost.

| Strategies | Easy 0 | Easy 1 | Easy 2 | Hard 1 | Hard 2 | Evil 1 |
|---|---|---|---|---|---|---|
| BT puro | 44 | 165 | 254 | 294 | 697 | 818 |
| BT FC | 01 | 0 | 0 | 875 | 170 | 890 |
| BT MAC | 0 | 0 | 0 | 587 | 19 | 775 |

## 3. Sudoku

Going back to the final part of Paragraph 1, we can model Sudoku so as to be able to apply the different backtracking algorithms described above, so we should specify:

- Variables: The set made up of the 81 cells, V = {v0, ..., v80}.

- Domain of variables: Di = {1, 2, 3, 4, 5, 6, 7, 8, 9} ÿ vi empty and Dj = {value in vj} IS not empty.

- Constraint: Two cells in the same row, column and 3x3 matrix must not have the same value.

- Variables related to a certain variable v: the boxes in the same row, column and 3x3 matrix.

In addition to the implementation of the algorithms, a file has been created that allows you to graphically display the Sudoku, its solution and choose which inference method you want to use, for more information on graphical development in python see: [5], [6 ] and [7].

## 4. Execution of tests

In the gui.py file there are 6 Sudoku instances with 2 different levels (The examples were taken randomly , later the results obtained by running the algorithms on each Sudoku instance will be shown (5 tests for each algorithm).

### 4.1 Experimental results

As can be seen in Table 1, the number of backtracks that occurs with pure backtracking is always higher than with forward-checking and MAC, this is completely normal since with pure backtracking we do not make inferences, moreover it is also observed how our theoretical expectations have been confirmed, namely that using forward-checking implies having a greater number of states to analyze (compared to the MAC) and therefore a greater number of backtracks.

From Table 2 instead we observe the times that each algorithm takes, it is therefore observed that pure backtracking needs more time to find the solution to the problem, this is completely normal given the high number of states to be analyzed and therefore also confirmed by the number of backtracks in Table 1,

but we also observe a very interesting thing, we observe that forward-checking takes less time than the MAC, as had already been anticipated in Paragraph 2.3, this confirms our theoretical suspicions since the MAC performs much more demanding steps by propagating the arc-consistency among the variables and therefore also requires more computational time even if it analyzes a smaller number of states.

To reproduce the experimental results it is sufficient to run the test.py file, modifying the inf variable in the main() method: inf = no inference refers to pure back tracking, inf = forward checking to backtracking with FC and inf = mac to backtracking with MAC.

Time required by various instances

| Strategies | Easy 0 | Easy 1 | Easy 2 | Hard 1 | Hard 2 | Evil 1 |
|---|---|---|---|---|---|---|
| BT pure | 0.00874 | 0.014484 | 0.01868 | 14.46901 | 1.69020 | 5.546142 |
| BT FC | 0.00534 | 0.005116 | 0.00659 | 0.105089 | 0.011204 | 0.4636 |
| BT MAC | 0.0315 | 0.02603 | 0.02526 | 0.39495 | 0.05872 | 1.9644 |

Table 2: Time (in seconds) taken by each algorithm to find the solution of the various Sudoku instances

To select the Sudoku instance it is sufficient to change the values of the level and which variables with the values indicated by the comments in the code, furthermore (as was said in Paragraph 3) the possibility of displaying the state start and end of the Sudoku with the respective times used by the algorithms, to start the graphical interface it is sufficient to execute the file main.py.

In the default graphical interface, the easiest level is selected but to change the level it is sufficient to click once on the

”Level/Hard” menu (to change the instance continuing with the Hard level one must click a second time on ”Level/ Hard”, the same applies to the Easy level).

## Implementation:

```
"""CSP (Constraint Satisfaction Problems) problems and solvers. (Chapte
r 6)."""

# CLASS FROM https://github.com/aimacode/aima-
python/ slightly modified for performance (see tag @modified)
# the proof about performance can be found in the files original_result
s.txt and modified_results.txt
# @modified: removed unused imports
class CSP:
    """This class describes finite-
domain Constraint Satisfaction Problems.
    A CSP is specified by the following inputs:
        variables   A list of variables; each is atomic (e.g. int or st
ring).
        domains     A dict of {var:[possible_value, ...]} entries.
        neighbors   A dict of {var:[var,...]} that for each variable li
sts
                    the other variables that participate in constraints
.
        constraints A function f(A, a, B, b) that returns true if neigh
bors
                    A, B satisfy the constraint when they have values A
=a, B=b

    In the textbook and in most mathematical definitions, the
    constraints are specified as explicit pairs of allowable values,
    but the formulation here is easier to express and more compact for
    most cases. (For example, the n-Queens problem can be represented
    in O(n) space using this notation, instead of O(N^4) for the
    explicit representation.) In terms of describing the CSP as a
    problem, that's all there is.

    However, the class also supports data structures and methods that h
elp you
    solve CSPs by calling a search function on the CSP. Methods and slo
ts are
    as follows, where the argument 'a' represents an assignment, which
is a
    dict of {var:val} entries:
        assign(var, val, a)     Assign a[var] = val; do other bookkeepi
ng
        unassign(var, a)        Do del a[var], plus other bookkeeping
        nconflicts(var, val, a) Return the number of other variables th
at
                                conflict with var=val
        curr_domains[var]       Slot: remaining consistent values for v
ar
                                Used by constraint propagation routines
.
    The following methods are used only by graph_search and tree_search
:
        actions(state)          Return a list of actions
        result(state, action)   Return a successor of state
        goal_test(state)        Return true if all constraints satisfie
d
    The following are just for debugging purposes:
        nassigns                Slot: tracks the number of assignments
made
        display(a)              Print a human-readable representation
    """

# added a variable to save the number of backtracks
# in my opinion it is better to show the backtracks instead of the assi
gnments
    def __init__(self, variables, domains, neighbors, constraints):
        """Construct a CSP problem. If variables is empty, it becomes d
omains.keys()."""
        variables = variables or list(domains.keys())
        self.variables = variables
        self.domains = domains
        self.neighbors = neighbors
        self.constraints = constraints
        self.initial = ()
        self.curr_domains = None
        self.nassigns = 0
        self.n_bt = 0

    def assign(self, var, val, assignment):
        """Add {var: val} to assignment; Discard the old value if any."
""
        assignment[var] = val
        self.nassigns += 1

    def unassign(self, var, assignment):
        """Remove {var: val} from assignment.
        DO NOT call this if you are changing a variable to a new value;
        just call assign for that."""
        if var in assignment:
            del assignment[var]

# @Modified: the original used a recursive function, in my opinion this
 one looks better
#            and is easier to understand
```

```
    def nconflicts(self, var, val, assignment):
        """Return the number of conflicts var=val has with other variab
les."""
        count = 0
        for var2 in self.neighbors.get(var):
            val2 = None
            if assignment.__contains__(var2):
                val2 = assignment[var2]
            if val2 is not None and self.constraints(var, val, var2, va
l2) is False:
                count += 1
        return count

    def display(self, assignment):
        """Show a human-readable representation of the CSP."""
        # Subclasses can print in a prettier way, or display with a GUI
        print('CSP:', self, 'with assignment:', assignment)

    def goal_test(self, state):
        """The goal is to assign all variables, with all constraints sa
tisfied."""
        assignment = dict(state)
        return (len(assignment) == len(self.variables)
                and all(self.nconflicts(variables, assignment[variables
], assignment) == 0
                        for variables in self.variables))

    # These are for constraint propagation

    def support_pruning(self):
        """Make sure we can prune values from domains. (We want to pay
        for this only if we use it.)"""
        if self.curr_domains is None:
            self.curr_domains = {v: list(self.domains[v]) for v in self
.variables}

    def suppose(self, var, value):
        """Start accumulating inferences from assuming var=value."""
        self.support_pruning()
        removals = [(var, a) for a in self.curr_domains[var] if a != va
lue]
        self.curr_domains[var] = [value]
        return removals

    def prune(self, var, value, removals):
        """Rule out var=value."""
        self.curr_domains[var].remove(value)
        if removals is not None:
            removals.append((var, value))

    def choices(self, var):
        """Return all values for var that aren't currently ruled out.""
"
        return (self.curr_domains or self.domains)[var]

    def restore(self, removals):
        """Undo a supposition and all inferences from it."""
        for B, b in removals:
            self.curr_domains[B].append(b)


# ______________a_______________________________________________________
__________
# Constraint Propagation with AC-3

def AC3(csp, queue=None, removals=None):
    """[Figure 6.3]"""
    if queue is None:
        queue = [(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbo
rs[Xi]]
    csp.support_pruning()
    while queue:
        (Xi, Xj) = queue.pop()
        if revise(csp, Xi, Xj, removals):
            if not csp.curr_domains[Xi]:
                return False
            for Xk in csp.neighbors[Xi]:
                if Xk != Xi:
                    queue.append((Xk, Xi))
    return True


def revise(csp, Xi, Xj, removals):
    """Return true if we remove a value."""
    revised = False
    for x in csp.curr_domains[Xi][:]:
        # If Xi=x conflicts with Xj=y for every possible y, eliminate X
i=x
        if all(not csp.constraints(Xi, x, Xj, y) for y in csp.curr_doma
ins[Xj]):
            csp.prune(Xi, x, removals)
            revised = True
    return revised

# _____________________________________________________________________
_________
```

```python
# CSP Backtracking Search

# Variable ordering


# @Modified: we just want the first one that haven't been assigned so r
eturning fast is good
def first_unassigned_variable(assignment, csp):
    """The default variable order."""
    for var in csp.variables:
        if var not in assignment:
            return var






# @Modified: the original used a function from util files and was harde
r to understand,
#            it also apparently used 2 for loops: one to find the minim
um and
#            other one to create a list (and a lambda function)
def mrv(assignment, csp):
    """Minimum-remaining-values heuristic."""
    vars_to_check = []
    size = []
    for v in csp.variables:
        if v not in assignment.keys():
            vars_to_check.append(v)
            size.append(num_legal_values(csp, v, assignment))
    return vars_to_check[size.index(min(size))]






# @Modified: the original used a function count and a list, in my opini
on it is faster to
#            just count with a loop 'for' without calling external func
tions
def num_legal_values(csp, var, assignment):
    if csp.curr_domains:
        return len(csp.curr_domains[var])
    else:
        count = 0
        for val in csp.domains[var]:
            if csp.nconflicts(var, val, assignment) == 0:
                count += 1
        return count


# Value ordering


def unordered_domain_values(var, assignment, csp):
    """The default value order."""
    return csp.choices(var)


def lcv(var, assignment, csp):
    """Least-constraining-values heuristic."""
    return sorted(csp.choices(var),
                  key=lambda val: csp.nconflicts(var, val, assignment))





# Inference


def no_inference(csp, var, value, assignment, removals):
    return True


def forward_checking(csp, var, value, assignment, removals):
    """Prune neighbor values inconsistent with var=value."""
    for B in csp.neighbors[var]:
        if B not in assignment:
            for b in csp.curr_domains[B][:]:
                if not csp.constraints(var, value, B, b):
                    csp.prune(B, b, removals)
            if not csp.curr_domains[B]:
                return False
    return True


def mac(csp, var, value, assignment, removals):
    """Maintain arc consistency."""
    return AC3(csp, [(X, var) for X in csp.neighbors[var]], removals)

# The search, proper

# @Modified: we should notice that with MRV it works good since the par
tial initial state
#            leaves some variables with unitary domain so we will start
 to assign these variables.
```

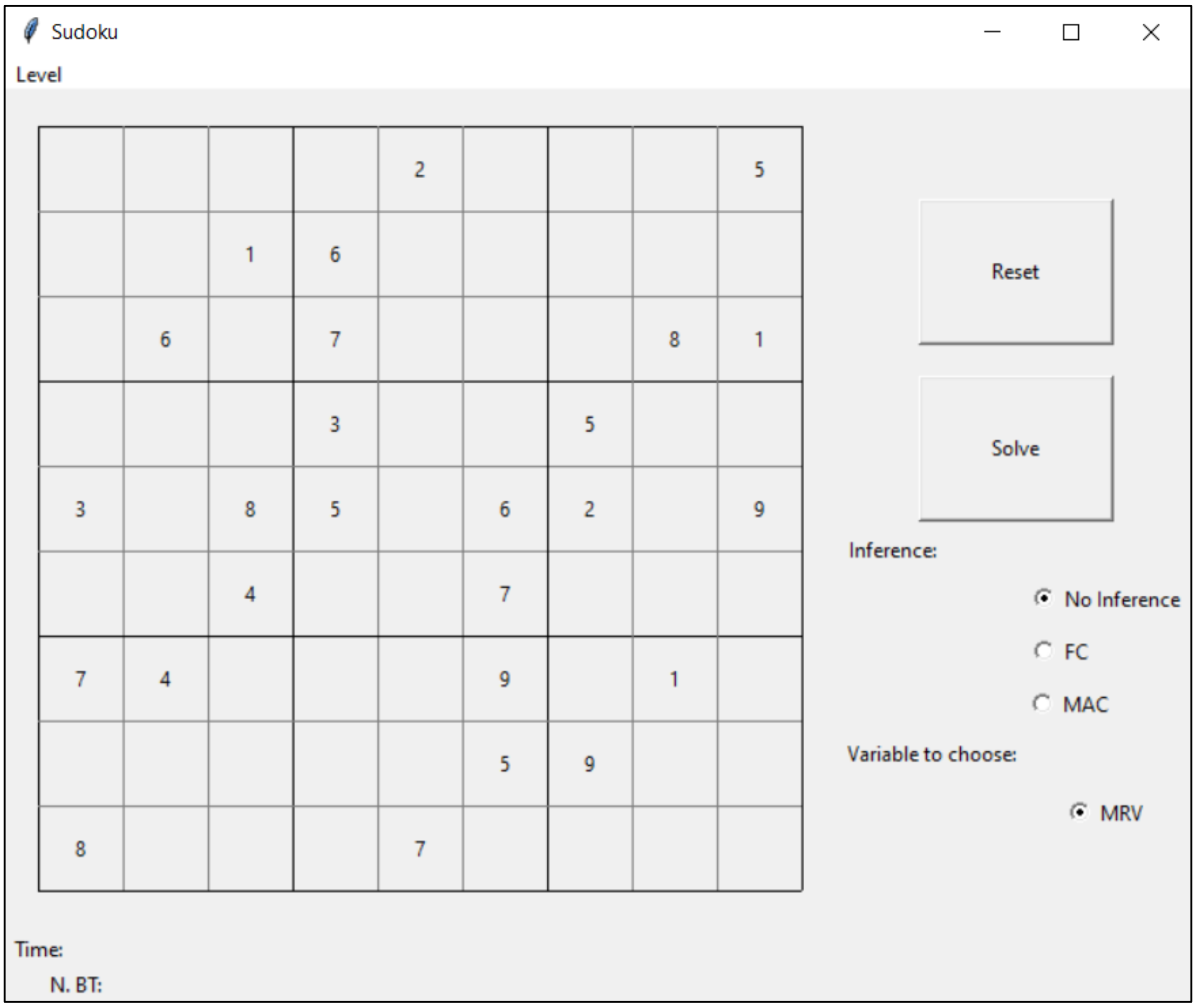
```
#             Added csp.n_bt+=1


def backtracking_search(csp,
                        select_unassigned_variable,
                        order_domain_values,
                        inference):
    """[Figure 6.5]"""
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if 0 == csp.nconflicts(var, value, assignment):
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                    else:
                        csp.n_bt += 1
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result

def different_values_constraint(A, a, B, b):
    """A constraint saying two neighboring variables must differ in val
ue."""
    return a != b
```

Output:



|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  |  | 2 |  |  |  | 5 |
|  |  | 1 | 6 |  |  |  |  |  |
|  | 6 |  | 7 |  |  |  | 8 | 1 |
|  |  |  | 3 |  |  | 5 |  |  |
| 3 |  | 8 | 5 |  | 6 | 2 |  | 9 |
|  |  | 4 |  |  | 7 |  |  |  |
| 7 | 4 |  |  |  | 9 |  | 1 |  |
|  |  |  |  |  | 5 | 9 |  |  |
| 8 |  |  |  | 7 |  |  |  |  |

Inference:

[✓ No Inference]

FC

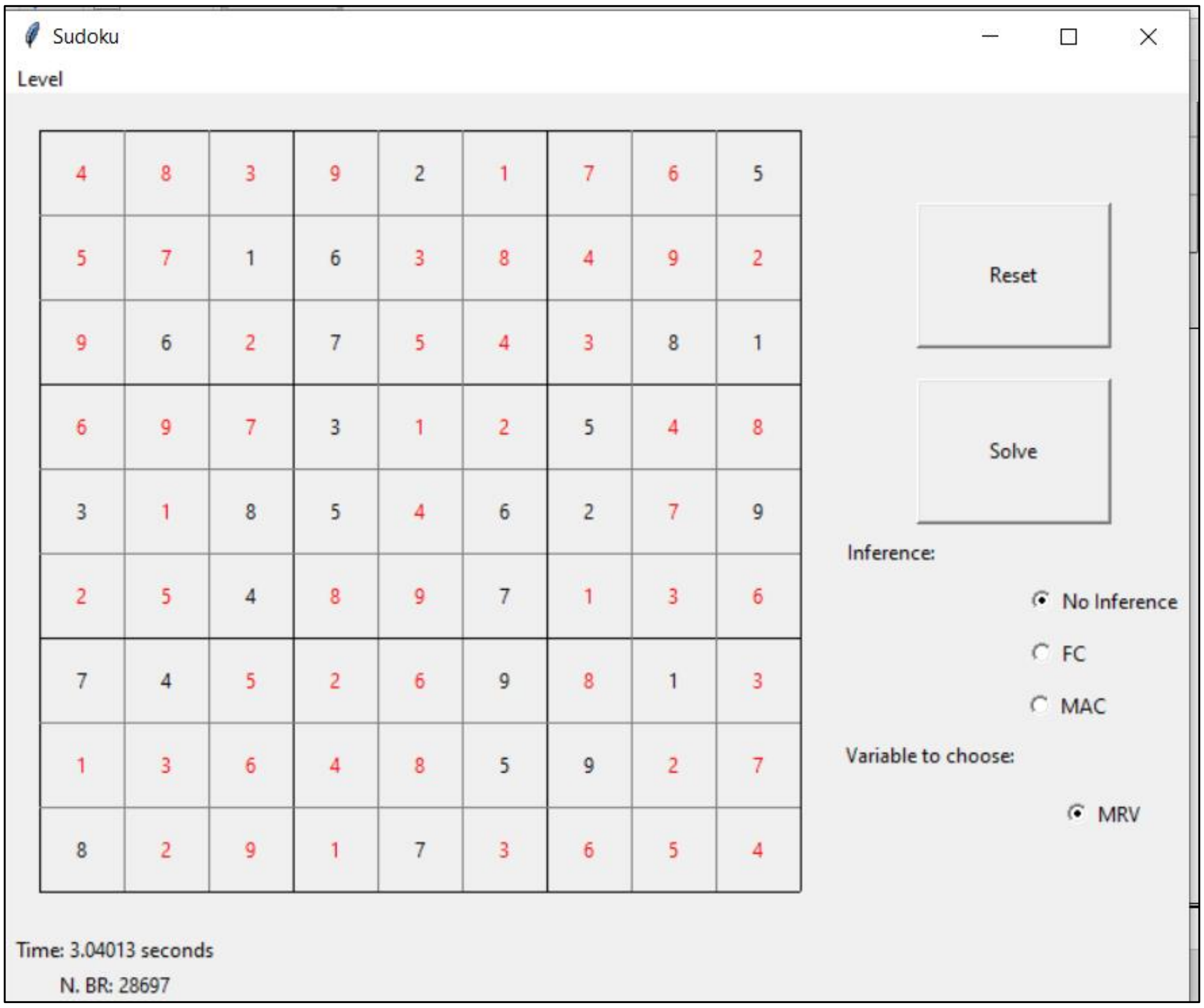
MAC

Variable to choose:

[✓ MRV]

Time:

N. BT:



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 8 | 3 | 9 | 2 | 1 | 7 | 6 | 5 |
| 5 | 7 | 1 | 6 | 3 | 8 | 4 | 9 | 2 |
| 9 | 6 | 2 | 7 | 5 | 4 | 3 | 8 | 1 |
| 6 | 9 | 7 | 3 | 1 | 2 | 5 | 4 | 8 |
| 3 | 1 | 8 | 5 | 4 | 6 | 2 | 7 | 9 |
| 2 | 5 | 4 | 8 | 9 | 7 | 1 | 3 | 6 |
| 7 | 4 | 5 | 2 | 6 | 9 | 8 | 1 | 3 |
| 1 | 3 | 6 | 4 | 8 | 5 | 9 | 2 | 7 |
| 8 | 2 | 9 | 1 | 7 | 3 | 6 | 5 | 4 |

Inference:

[✓ No Inference]

FC

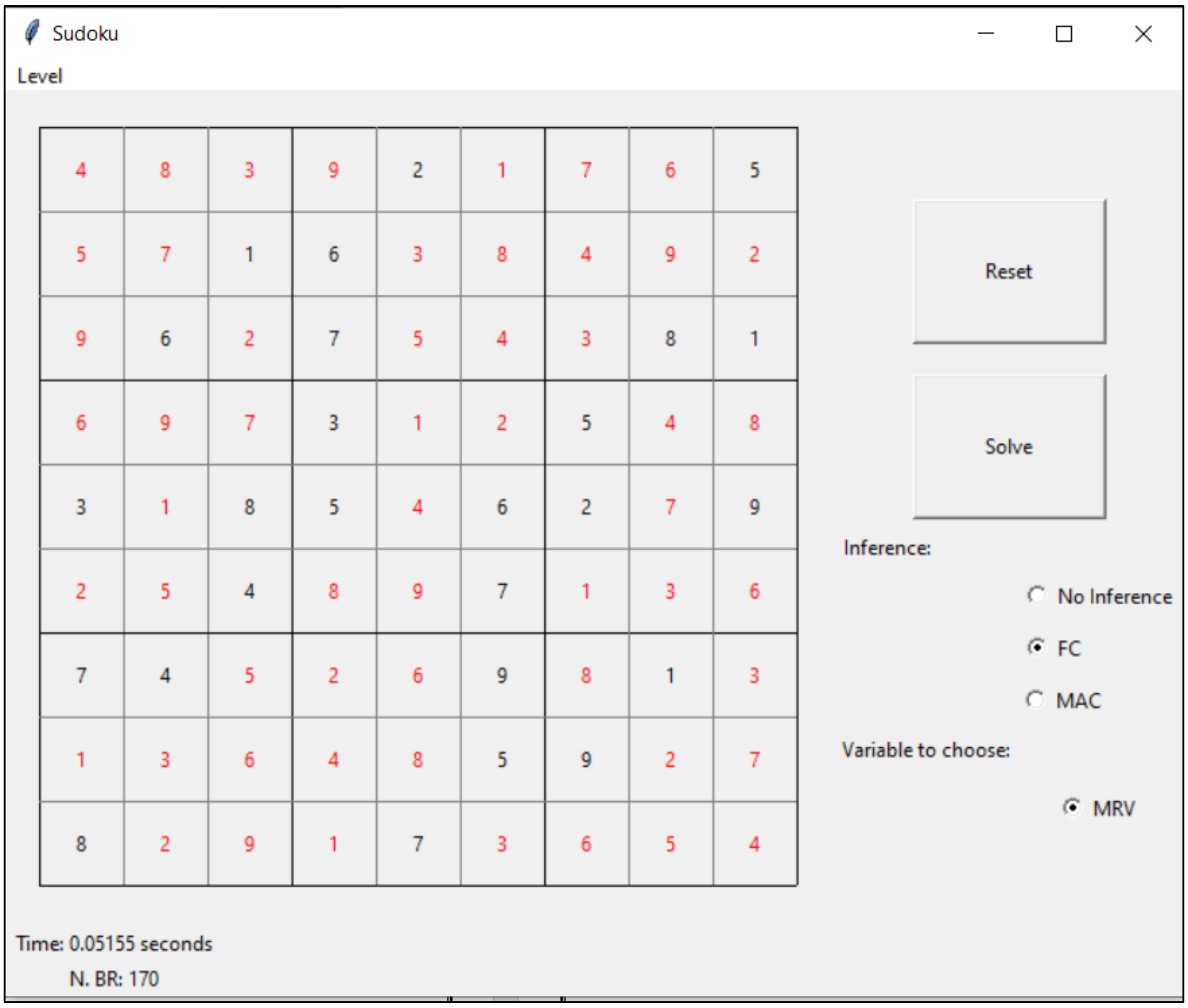
MAC

Variable to choose:

[✓ MRV]

Time: 3.04013 seconds

N. BR: 28697



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 8 | 3 | 9 | 2 | 1 | 7 | 6 | 5 |
| 5 | 7 | 1 | 6 | 3 | 8 | 4 | 9 | 2 |
| 9 | 6 | 2 | 7 | 5 | 4 | 3 | 8 | 1 |
| 6 | 9 | 7 | 3 | 1 | 2 | 5 | 4 | 8 |
| 3 | 1 | 8 | 5 | 4 | 6 | 2 | 7 | 9 |
| 2 | 5 | 4 | 8 | 9 | 7 | 1 | 3 | 6 |
| 7 | 4 | 5 | 2 | 6 | 9 | 8 | 1 | 3 |
| 1 | 3 | 6 | 4 | 8 | 5 | 9 | 2 | 7 |
| 8 | 2 | 9 | 1 | 7 | 3 | 6 | 5 | 4 |

Inference:

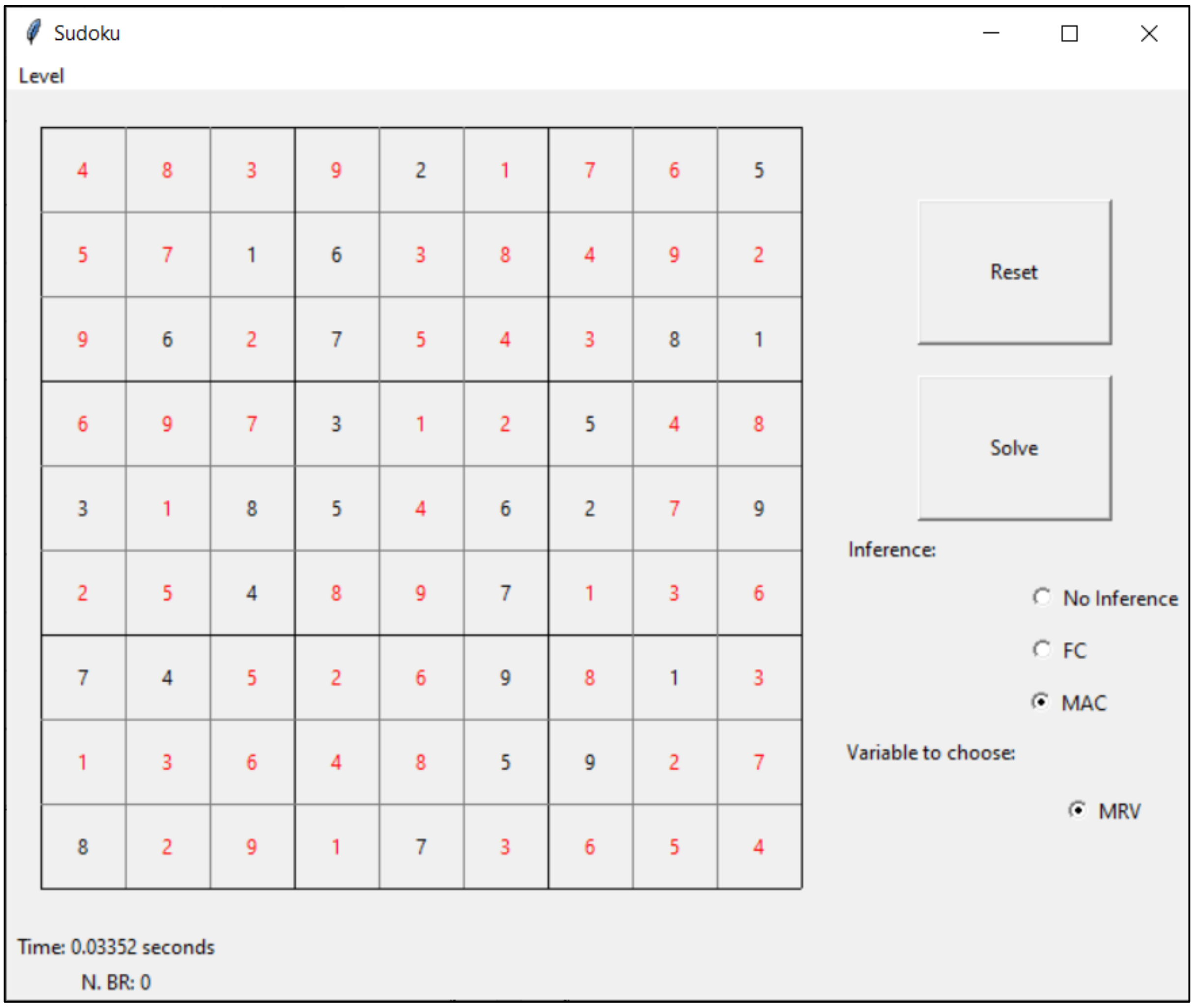
No Inference [✓ FC] MAC

Variable to choose:

[✓ MRV]

Time: 0.05155 seconds

N. BR: 170



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 8 | 3 | 9 | 2 | 1 | 7 | 6 | 5 |
| 5 | 7 | 1 | 6 | 3 | 8 | 4 | 9 | 2 |
| 9 | 6 | 2 | 7 | 5 | 4 | 3 | 8 | 1 |
| 6 | 9 | 7 | 3 | 1 | 2 | 5 | 4 | 8 |
| 3 | 1 | 8 | 5 | 4 | 6 | 2 | 7 | 9 |
| 2 | 5 | 4 | 8 | 9 | 7 | 1 | 3 | 6 |
| 7 | 4 | 5 | 2 | 6 | 9 | 8 | 1 | 3 |
| 1 | 3 | 6 | 4 | 8 | 5 | 9 | 2 | 7 |
| 8 | 2 | 9 | 1 | 7 | 3 | 6 | 5 | 4 |

Inference:
No Inference FC [✓ MAC]

Variable to choose:
[✓ MRV]

Time: 0.03352 seconds
N. BR: 0

## 4.2 Conclusion

- From the previously seen experimental results we can conclude that: in general, if you want to have the least number of states analysed, the best choice is backtracking with MAC;
- however, if the goal is to find the solution in the shortest possible time, the best option could be forward-checking since this takes less time even if the number of analyzed states is slightly higher than the MAC;
- moreover, if the problem does not have such a high state set, such as for example the Sudoku instance" Easy2" given the experimental results in Tables 1 and 2, it could return pure backtracking is also useful if you want to apply a simple and clean algorithm (without too many calls to other methods).

## 4.3 References

- Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. 3rd edition. Pearson, 2010.
- Code for the book"Artificial Intelligence: A Modern Approach" https://github.com/aimacode/aima-python
- The telegraph: World's hardest sudoku ever: http://bit.ly/1EN89Ke
- Evil Sudoku generator: https://www.websudoku.com/
- The new Boston: Python GUI with Tkinter https://youtu.be/RJB1Ek2Ko_Y
- Effbot: An Introduction to Tkinter http://effbot.org/tkinterbook/
- New Coder: Graphical User Interfaces http://newcoder.io/gui/

**Group members:**

1. Kiran Patil 211070904
2. Mayuresh Murudkar 211070903
3. Shrishail Gouragond 211070906
4. Sankalp Vasave 211070909