A study of solving sudoku:
Using combination of various inference rules

Abstract

This report contains analysis of using various inference rules for solving sudoku generated in different environments by following a fixed-order baseline and most constrained variable heuristic. The inference rules used are: naked singles, hidden singles, naked pairs, hidden pairs, naked triples, hidden triples in the order presented. While determining rules for selection either simple backtracking is used or a subset of rules in a predetermined order are used.

I. Introduction

The experiments were carried out for different baseline and different inference rules. First, we explain the meaning of the baselines used and describe the inference rules incorporated in the algorithm to obtain the objective. Pseudocode for the algorithms is provided and following it we describe the experimental set up. We discuss the results in detail mentioning any constraints encountered by our algorithms, any surprises for us and if the performance can be improved by incorporating more variables.

II. Description

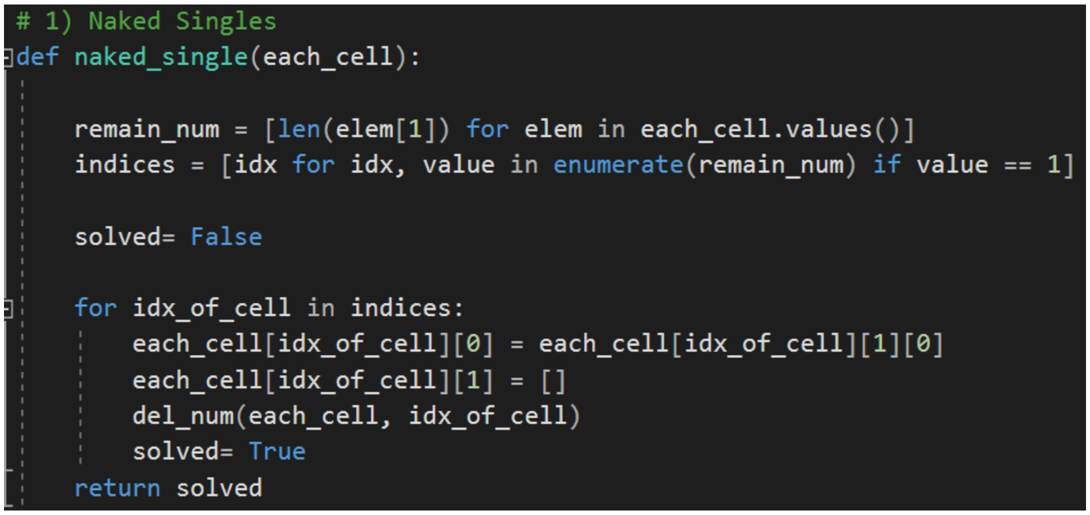
- **Fixed Baseline:**
  A baseline normally provides a measure for comparison. It's normally an easy to implement algorithm or procedure that may not be optimal and is very often far from optimal but helps in providing an idea for the solution or rather gives a upper bound for the performance of any other metric.In solving sudoku, the fixed baseline solution visited the nodes in top to bottom manner in a fixed manner.

- **Most constrained Variable**
  Most constrained variables are used in a search algorithm to ensure fail first i.e if a certain assignment is unsatisfiable for a problem, let it fail first. At every step in the search algorithm, a MCV is chosen to expand and the domains are updated for each variable after an expansion.
- **Inference Rules**

Inference rules are used to make a problem satisfiable by following the given constraints. Following inference rules were used for solving the sudoku problem.

1) **Naked Singles**

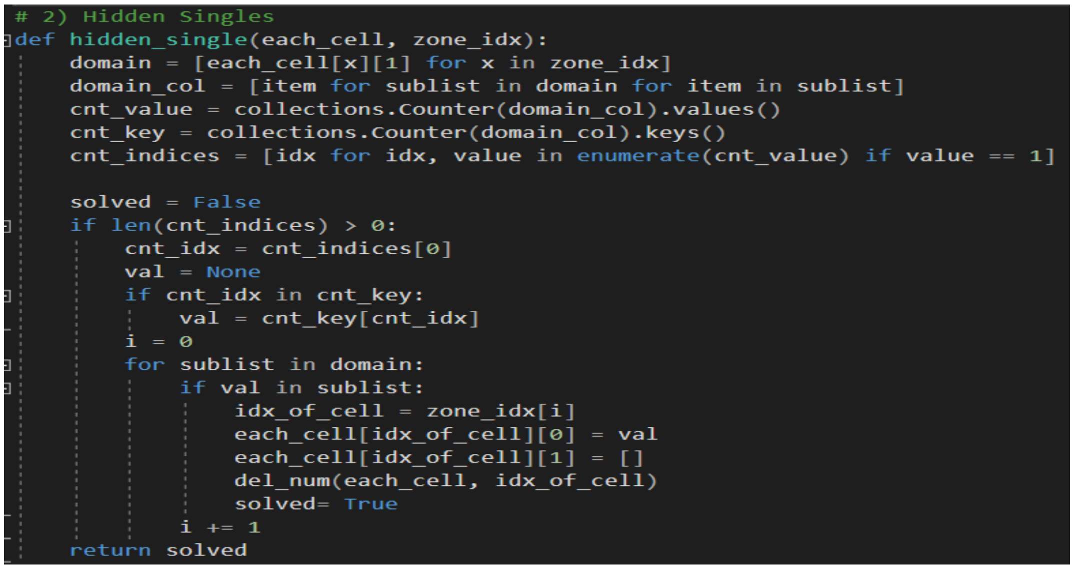
   Naked Single means that in a specific cell only one digit remains possible The last remaining candidate has no other candidates to hide behind and is thus ***naked***. The digit must then go into that cell.



```
# 1) Naked Singles
def naked_single(each_cell):

    remain_num = [len(elem[1]) for elem in each_cell.values()]
    indices = [idx for idx, value in enumerate(remain_num) if value == 1]

    solved= False

    for idx_of_cell in indices:
        each_cell[idx_of_cell][0] = each_cell[idx_of_cell][1][0]
        each_cell[idx_of_cell][1] = []
        del_num(each_cell, idx_of_cell)
        solved= True
    return solved
```

2) **Hidden Singles**

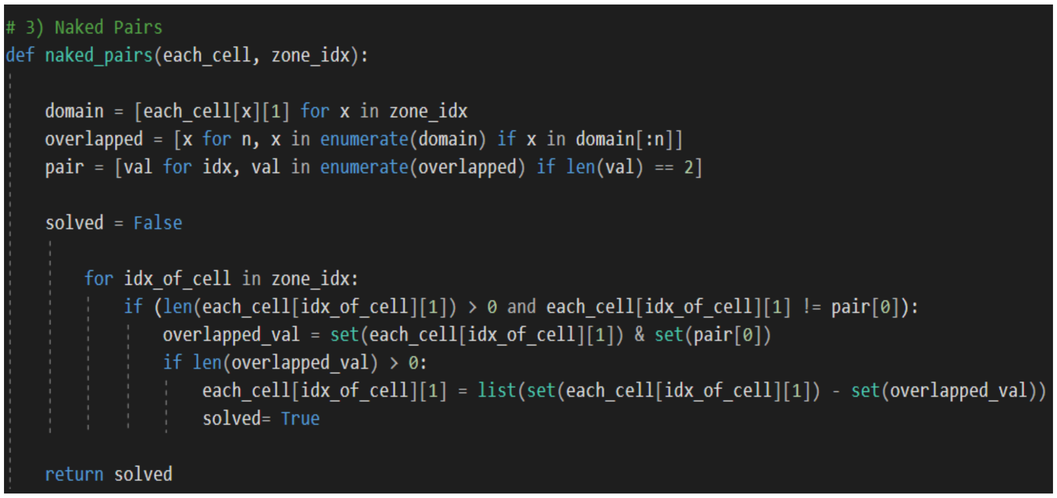
   Hidden Single means that for a given digit and house only one cell is left to place that digit. The cell itself has more than one candidate left, the correct digit is thus *hidden* amongst the rest.



```
# 2) Hidden Singles
def hidden_single(each_cell, zone_idx):
    domain = [each_cell[x][1] for x in zone_idx]
    domain_col = [item for sublist in domain for item in sublist]
    cnt_value = collections.Counter(domain_col).values()
    cnt_key = collections.Counter(domain_col).keys()
    cnt_indices = [idx for idx, value in enumerate(cnt_value) if value == 1]

    solved = False
    if len(cnt_indices) > 0:
        cnt_idx = cnt_indices[0]
        val = None
        if cnt_idx in cnt_key:
            val = cnt_key[cnt_idx]
        i = 0
        for sublist in domain:
            if val in sublist:
                idx_of_cell = zone_idx[i]
                each_cell[idx_of_cell][0] = val
                each_cell[idx_of_cell][1] = []
                del_num(each_cell, idx_of_cell)
                solved= True
            i += 1
    return solved
```

3) **Naked Pairs**

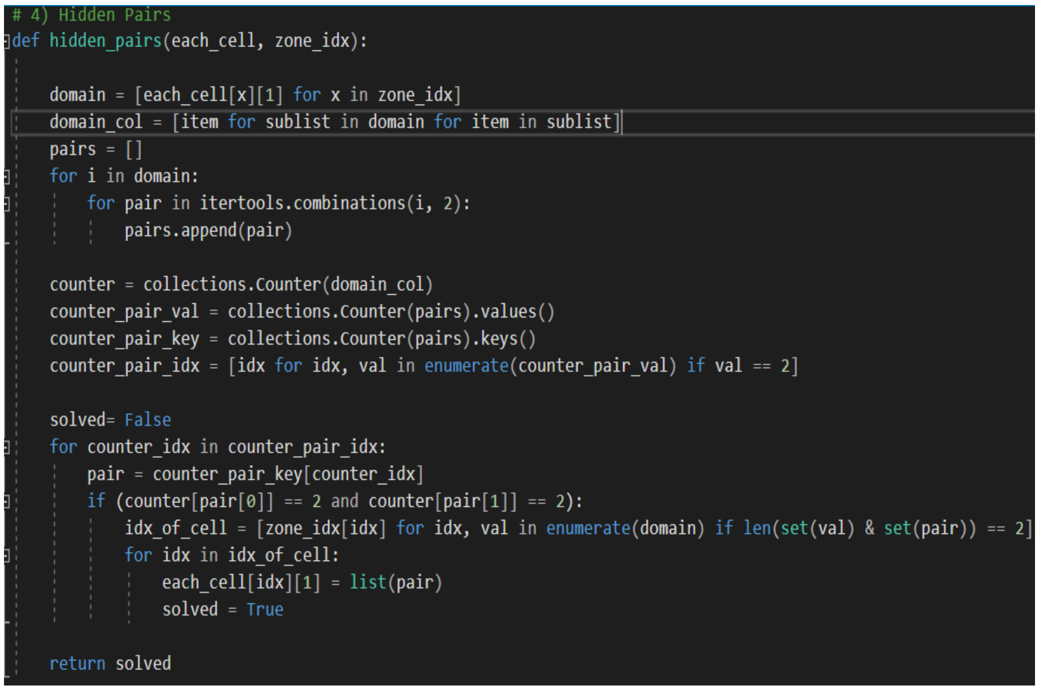
   The "naked pair" solving technique is an intermediate solving technique. In this technique the Sudoku is scanned for a pair of cells in a row, column or box containing only the same two candidates. Since these candidates must go in these cells, they can therefore be removed from the candidate lists of all other unsolved cells in that row, column or box. Reducing candidate lists may reveal a hidden or naked single in another unsolved cell, generally however the technique is a step to solving the next cell.



```
# 3) Naked Pairs
def naked_pairs(each_cell, zone_idx):

    domain = [each_cell[x][1] for x in zone_idx
    overlapped = [x for n, x in enumerate(domain) if x in domain[:n]]
    pair = [val for idx, val in enumerate(overlapped) if len(val) == 2]

    solved = False

        for idx_of_cell in zone_idx:
            if (len(each_cell[idx_of_cell][1]) > 0 and each_cell[idx_of_cell][1] != pair[0]):
                overlapped_val = set(each_cell[idx_of_cell][1]) & set(pair[0])
                if len(overlapped_val) > 0:
                    each_cell[idx_of_cell][1] = list(set(each_cell[idx_of_cell][1]) - set(overlapped_val))
                    solved= True

    return solved
```

4) **Hidden pairs**

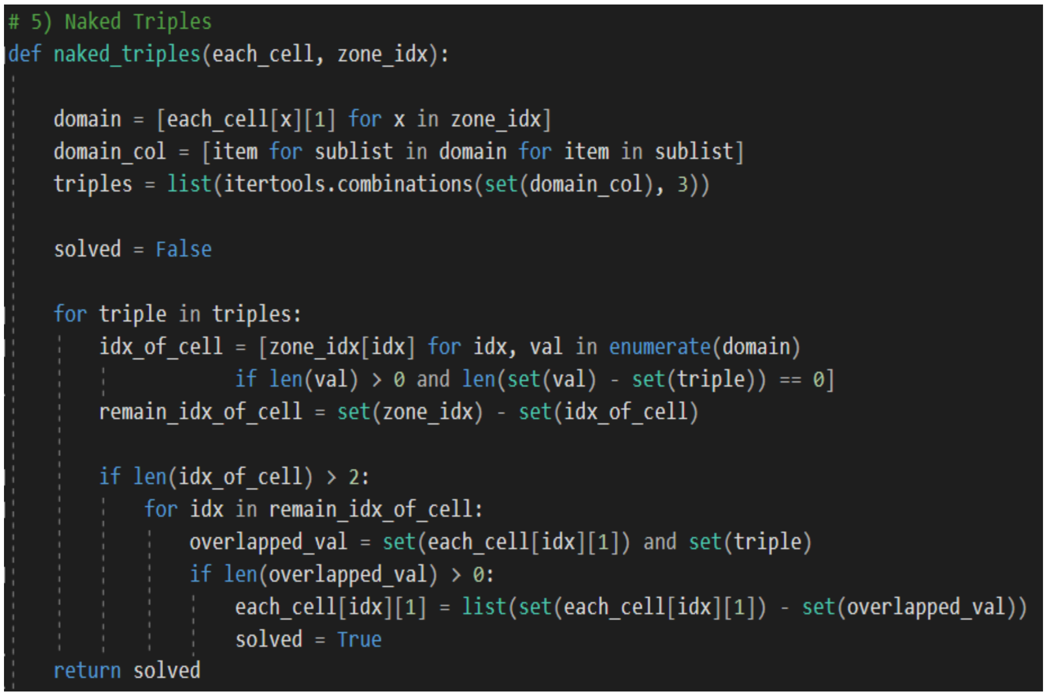
A Hidden Pair is basically just a “buried” Naked Pair. It occurs when two pencil marks appear in *exactly two cells* within the same house (row, column, or block).This time, however, the pair is not "Naked" - it is buried (or hidden) among other pencil marks.



```
 # 4) Hidden Pairs
def hidden_pairs(each_cell, zone_idx):

    domain = [each_cell[x][1] for x in zone_idx]
    domain_col = [item for sublist in domain for item in sublist]
    pairs = []
    for i in domain:
        for pair in itertools.combinations(i, 2):
            pairs.append(pair)

    counter = collections.Counter(domain_col)
    counter_pair_val = collections.Counter(pairs).values()
    counter_pair_key = collections.Counter(pairs).keys()
    counter_pair_idx = [idx for idx, val in enumerate(counter_pair_val) if val == 2]

    solved= False
    for counter_idx in counter_pair_idx:
        pair = counter_pair_key[counter_idx]
        if (counter[pair[0]] == 2 and counter[pair[1]] == 2):
            idx_of_cell = [zone_idx[idx] for idx, val in enumerate(domain) if len(set(val) & set(pair)) == 2]
            for idx in idx_of_cell:
                each_cell[idx][1] = list(pair)
                solved = True

    return solved
```

5) **Naked Triples**

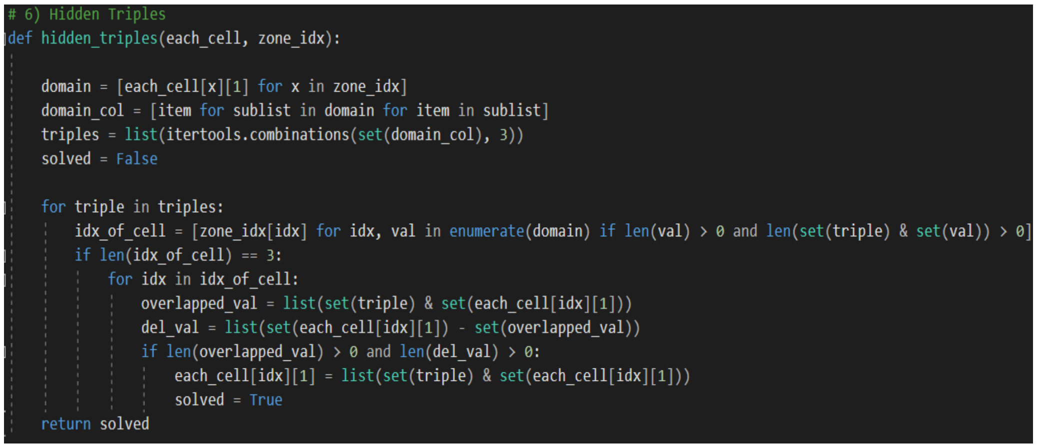
Naked triples like the name suggests are three numbers that do not have any other numbers residing in the cells with them. Unlike naked pairs, naked triples do not need all of the three candidates in every cell. Quite often only two of the three candidates will be shown.



```python
# 5) Naked Triples
def naked_triples(each_cell, zone_idx):

    domain = [each_cell[x][1] for x in zone_idx]
    domain_col = [item for sublist in domain for item in sublist]
    triples = list(itertools.combinations(set(domain_col), 3))

    solved = False

    for triple in triples:
        idx_of_cell = [zone_idx[idx] for idx, val in enumerate(domain)
                  if len(val) > 0 and len(set(val) - set(triple)) == 0]
        remain_idx_of_cell = set(zone_idx) - set(idx_of_cell)

        if len(idx_of_cell) > 2:
            for idx in remain_idx_of_cell:
                overlapped_val = set(each_cell[idx][1]) and set(triple)
                if len(overlapped_val) > 0:
                    each_cell[idx][1] = list(set(each_cell[idx][1]) - set(overlapped_val))
                    solved = True
    return solved
```

6) **Hidden Triples**

Hidden triples are much harder to spot. They will occur in harder puzzles. Hidden triples like naked triples are restricted to three cells in a row, column, or region. Hidden triples like hidden pairs have additional digits that camouflage the three candidates.



```python
# 6) Hidden Triples
def hidden_triples(each_cell, zone_idx):

    domain = [each_cell[x][1] for x in zone_idx]
    domain_col = [item for sublist in domain for item in sublist]
    triples = list(itertools.combinations(set(domain_col), 3))
    solved = False

    for triple in triples:
        idx_of_cell = [zone_idx[idx] for idx, val in enumerate(domain) if len(val) > 0 and len(set(triple) & set(val)) > 0]
        if len(idx_of_cell) == 3:
            for idx in idx_of_cell:
                overlapped_val = list(set(triple) & set(each_cell[idx][1]))
                del_val = list(set(each_cell[idx][1]) - set(overlapped_val))
                if len(overlapped_val) > 0 and len(del_val) > 0:
                    each_cell[idx][1] = list(set(triple) & set(each_cell[idx][1]))
                    solved = True
    return solved
```

III. Experimental Setup

We chose python as our programming language as it is easy to use and has a lot of support in developer community. This one turned out to be hardest of all the assignments as it had more work and required very careful implementation of the inference rules and baselines. A lot of the code required careful debugging for hours and had to be re-implemented to get things running. Owing to memory and time constraints, the search steps were restricted to 1000 for all the inference rules. Sometimes this was enough to obtain a solution while some other times it wasn't;

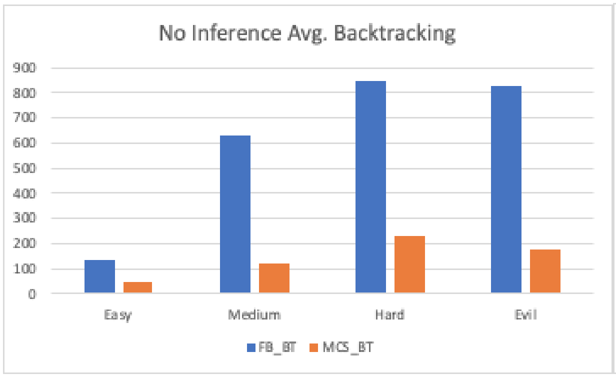
nevertheless all the information like CPU time and number of backtracks were recorded for both the cases. We ran the python program in Ubuntu 16.04 docker environment with quad-core 2.2Ghz Intel Core i7 CPUs and 16 GB memories. The process of the experiment is described as following:

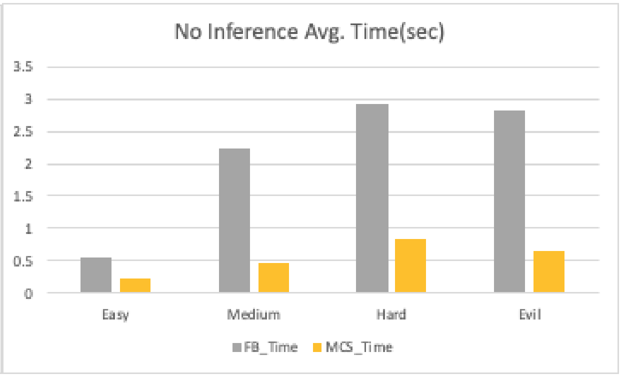
```
def run_experiment:
    For following inference rules record everything:
        ● No inference
        ● Naked and Hidden Singles.
        ● Naked and Hidden Singles and Pairs
        ● Naked and Hidden Singles, Pairs, and Triples
    Data = sudoku_solvers(inferences)
    All_info[inference] = Data
```

IV. Results

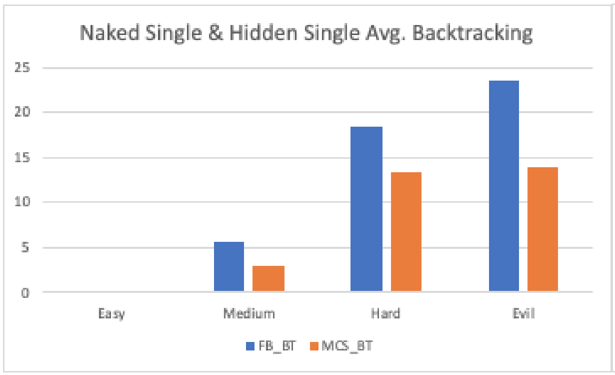
Following plots are the results of the experiment. Details will be discussed in Discussion and Conclusion section.

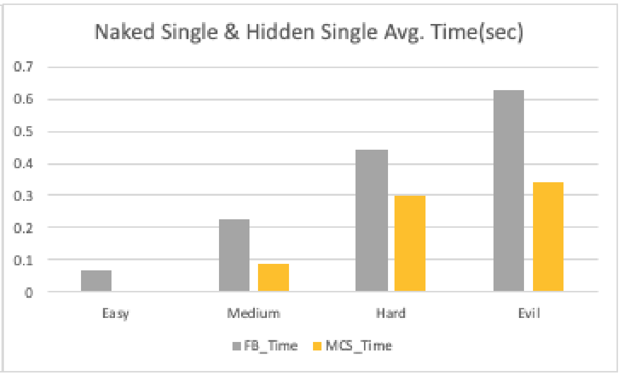
1. No Inference



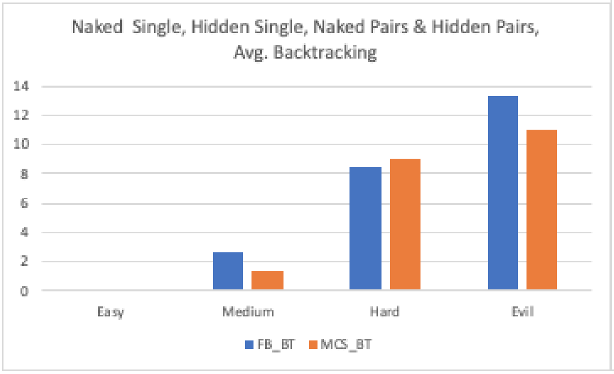


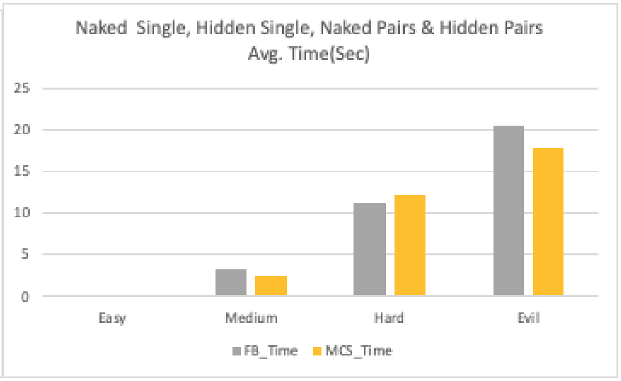
2. Naked Single and Hidden Single



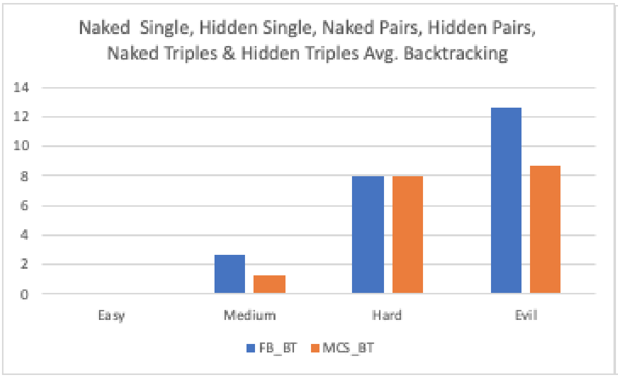


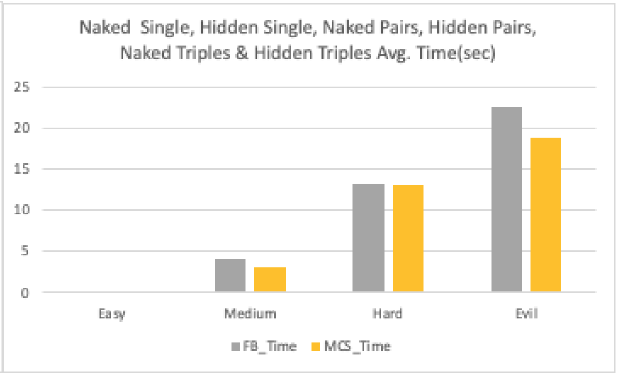
3. **Naked Single, Hidden Single, Naked Pairs, and Hidden Pairs**



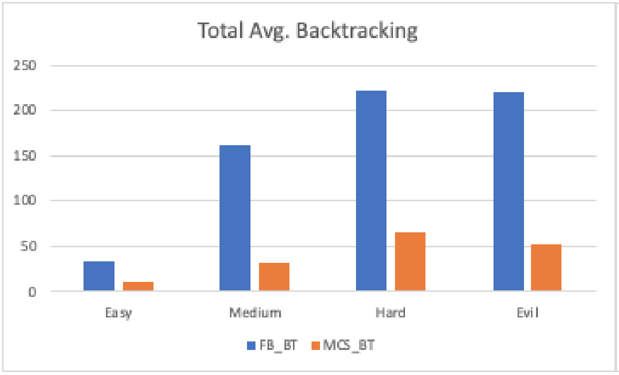


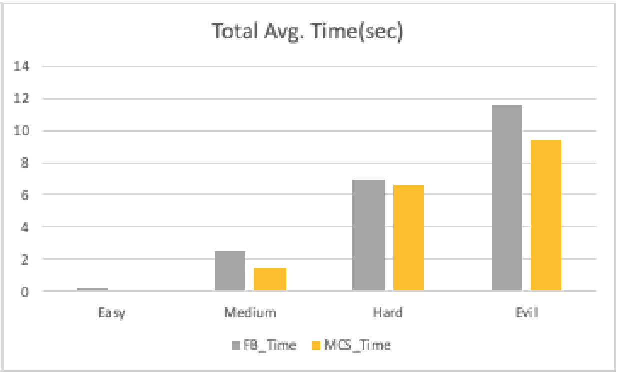
4. **Naked Single, Hidden Single, Naked Pairs, Hidden Pairs, Naked Triples, and Hidden Triples**





5. **Total Average Backtracks and Average CPU Time**





V. Discussion and Conclusion

1. Report the number of problems solved and the number of backtracks with each problem.

   As shown in output.xls attached in the submission of this project, All problems were solved except 33 failures occurred in No Inference fixed baseline condition.

2. Experts appear to grade the problems by the complexity of rules needed to solve them without backtracking. Is this conjecture roughly correct?

   This seems roughly correct as higher the complexity of rules, more the number of constraints in a CSP and lesser the state space to search for a solution

3. Grade each problem, by the set of rules used in solving it. Report also the average number of filled-in numbers (in the beginning) for each of these types of problems. Would this accurately reflect the difficulty of the problem?

   Referring to figures above, it seems that they properly reflect the difficulty of the problem since the number of backtracks and CPU time are linearly increases.