Report

# Test Setup

The central part of the test setup is done via the framework provided, for which we write a separate independent test method to test the running time, number of backtracks and number of assignments for each different problem, which are all logged and measured for further analysis. Since there might be variations with CPU execution time, repeated tests are done for each combination, and an average is calculated for each performance measure to ensure the maximum accuracy.

For the number of combinations to test, we choose to test all combinations of all possible permutation, this result in 390 different combinations for each 15 different problems. Since we were lucky enough to finish all the codes 3 days after the project is released, we have sufficient time test out all possible combinations without worrying about narrowing the number of combinations. The second reason to test on such a large scale is that it provides us with more comprehensive data for analysis.

However, due the nature of such a large scale test, number of tests and timeout is chosen differently for each difficulty level. Below is a glimpse of the test setup

---------------------------------------------

NumTest | Difficulty | Timeout

---------------------------------------------

20 | 'E' | 30/60

5 | 'M' | 60/120

1 | 'H' | 300/600

---------------------------------------------

The test machine spec is i7 6700hq, with 16GB of ram, running on Windows 10 Education.

# 1.2 Interpretation of raw data set

Four different data are collected for each test, which are id, running time, number of backtracks and number of assignments for each different problem. For Example, the following dataset in PE.txt, in section PE1.txt

id numBacktracks numAssignments avgtime

16 1 5 0.026801705360412598

First value is the id of the combination. The name of the combination algorithm can be obtained from id\_list.txt, from which we can obtain:

16 ['NKD', 'MRV', 'LCV']

Thus, we can interpret this data by saying, the algorithm with Naked Double, MRV and LCV, is able to solve PE1.txt in 26.8ms, with 1 backtracks and 5 assignment nodes.

If the combination cannot solve the Sudoku puzzle in given timeout limit, the runtime is recorded as infinity, indicating failure in solving the puzzle.

# Combinations

There are a total of 390 different combinations in our test, each of which correspond to a specific ordering of the algorithms. For each example, with naked triple and naked double, two different combination can be generated , as illustrated below

NKD, NKT, None, None

NKT, NKD, None, None

In our experiments, we find that the order the algorithms do matter in terms of the runtime performance. This possibly due to the number of hints available to NKD and NKT.

# 1.4 Comparison Methods

Multiple aspects of comparisons can be obtained by comparing the raw dataset’s three performance measure. The following sections describe those aspects in more detail

* + 1. Solving

The solving ability of a combination of algorithm is defined by its ability to solve a Sudoku puzzle in given set of time. If the algorithm cannot solve the problem in given time limit, we consider it as failure. However, solving ability of each combination could vary greatly across different Sudoku puzzle. In some cases, some combination is able to solve PH1 in 10 seconds, but fail to solve PH2 in 300 seconds time limit. To address this problem, we introduce penalty time.

* + 1. Penalty time

Recall that if a combination cannot solve in given limit time, runtime is recorded as infinity. In order to measure the mean the runtime of each algorithms, but also consider failure cases, we introduce penalty time. If a combination fails to solve the puzzle, a penalty time which is twice the timeout is added to the running time. E.g. if timeout is 300, penalty time would be 600.

Then the average is calculated by considering penalty time during the calculation. However, it should be stressed that penalty time does not depict a precise view of the performance of the combination, but only allows us to compare the performance of different algorithms.

The mean of the runtime of a combination should not be interrupted as a precise running time measure, but only a mere indicator for comparing itself with other combination.

* + 1. Puzzle Difficulty

In this report, all x-axis are categorized into three different parts, PE, PM and PH, which in turn means the average of 5 sub problems.

* + 1. Number of Backtracks and Assignments

The number of backtracks records the number of deadend during backtracking search. The number of assignments records the number of non-deadend nodes during backtracking search that lead to the correct solution.

Both backtracks and assignments measure the space complexity of the algorithms. In general cases space used is associated with difficulty level, since the harder the problem is, the more search it needs to do. Both of them are also positively associated with runtime of the combination. This is also due to the fact that good combination algorithms try to terminate deadend as early as possibly to prevent backtracks. Those combinations that cannot solve the problem always generate huge number of backtracks and assignments.

# 1.5 Defining the Best Combination

When considering the best combinations out of the 390 combos, combinations that cannot solve all puzzles are excluded from the list. Because the best combination should always be able to solve a puzzle in given time constraint.

The second factor taken into account is the average runtime. The less time it runs, the better it is. The third factor is the space complexity.

There are two reasons we favour runtime over space complexity, the first reason is that our test machine, which has 16GB of ram, so memory is large enough that space used do not necessarily affect the performance. The second reason is that runtime is a better indicator that shows fast the combination is.

# Evaluation

We first start off by looking at how heuristics and the other of other consistency checking affect each consistency checking

Looking at the

looking at the above chart, for PE problem, all combinations merely make a difference. But as the problem grow harder, the differences among combinations in terms of runtime start to emerge. From the graph we can see that, the most efficient subsequent combinations out of all three difficulties are:

AC NKD NKT MRV LCV

AC NKT NKD MRV LCV

AC NKT NKD MRV

AC NKD NKT MRV

AC NKT NKD FC MRV LCV

The most efficient combinations starting with FC are:

FC NKD NKT MRV LCV

FC NKT NKD AC MRV LCV

FC NKD NKT AC MRV LCV

FC NKT NKD MRV LCV

The most efficient combinations starting with NKD are:

NKD NKT MRV LCV

NKD NKT AC MRV LCV

NKD AC NKT MRV LCV

NKD NKT FC MRV LCV

NKD FC NKT MRV LCV

The most efficient combinations starting with NKT are:

NKT NKD MRV LCV

NKT NKD AC MRV LCV

NKT AC NKD MRV LCV

NKT NKD FC MRV LCV

NKT FC NKD MRV LCV

Looking at the above four different charts, we can see that by adding NKT and NKD to FC or AC, the runtime improves dramatically, on average increases by 214.3%. Both NKT and NKD boosts the performance significantly.

Interestingly, the top five most efficient combinations all contain variable length heuristics LCV. LCV tries to avoid failure by assigning values that leave the maximum number of choices. The idea of LCV is to search the whole state space and find a solution as soon as possible. Both NKD and NKT are heuristics that eliminate possible candidates by inferencing, it is fast enough that it solves