Lecturer: Kuldeep Meel (Sem 2 AY20/21)

CS4244 Project Stage 1
SAT Solver

Evelyn Yi-Wen Chen (A0184698E), Yang Zi Yun (A0184682U)
{e0313687, e0313671}@u.nus.edu

This report explains our implementation of CDCL SAT Solver, experiments and analysis on different branching heuristics, solving Einstein Puzzle with our SAT Solver and our learning points in this project.

1. Solver Instructions

Our solver is written in Python3 and mainly consists of 2 files: 'mySATSolver.py' and 'experiments.py'.

1.1. mySATSolver.py

'mySATSolver.py' will run SAT solver on input cnf files (based on input heuristic), then append the result to a text file in 'result/'.

Input:

1. A path to a cnf file or a path to a directory containing cnf files.
2. Branching heuristic

Output:

1. A solved.txt file in ./result/ directory: the results are appended to the text file.

Usage: (3 input parameters required)

1. python3 mySATSolver.py [file/dir path] [heuristic choice: ['random', 'two_clause', 'all_clause', 'max_freq', 'vsids'] or 'all'] [allow debug: 0 or 1]
2. For example: python3 mySATSolver.py CS4244_project/sat/uf20-91/uf20-01.cnf two_clause 0

1.2. experiments.py

'experiments.py' will run SAT solver in 'mySATSolver.py' on all cnf files (on all heuristics), then output the results in a csv file.

Input:

1. A path to a directory containing cnf files.
2. A path to write the output csv file.

Output:

1. A csv file with experiment results:
    a. Each row corresponding to a cnf file, information including:
        i. *filename, #variables, #clauses,*
        ii. *SAT result and time taken (by cryptominisat),*
        iii. *SAT result, time taken, #branches, #implications (by each heuristic).*

Usage: (2 input parameters required)

- python3 experiments.py [cnf_dirname] [output_filename]
- For example: python3 experiments.py CS4244_project/sat/ experiments.csv

2. CDCL SAT Solver implementation

Our implementation of CDCL SAT solver followed closely the CDCL handbook [1], mainly the algorithm as shown in Figure 1:

Algorithm 1 Typical CDCL algorithm

```
CDCL(φ, ν)
 1  if (UnitPropagation(φ, ν) == CONFLICT)
 2     then return UNSAT
 3  dl ← 0                                          ▷ Decision level
 4  while (not AllVariablesAssigned(φ, ν))
 5        do (x, v) = PickBranchingVariable(φ, ν)                ▷ Decide stage
 6           dl ← dl + 1          ▷ Increment decision level due to new decision
 7           ν ← ν ∪ {(x, v)}
 8           if (UnitPropagation(φ, ν) == CONFLICT)              ▷ Deduce stage
 9              then β = ConflictAnalysis(φ, ν)                  ▷ Diagnose stage
10                   if (β < 0)
11                      then return UNSAT
12                      else Backtrack(φ, ν, β)
13                           dl ← β          ▷ Decrement decision level due to backtracking
14  return SAT
```

Figure 1. Typical CDCL algorithm [1]

3. Heuristics

From Figure 1, we can see that the major functions in the CDCL algorithm are: PickBranchingVariable, UnitPropagation and Conflict Analysis. We discuss the heuristics for Conflict Analysis and PickBranchingVariable.

3.1. ConflictAnalysis

ConflictAnalysis is invoked each time the solver reaches a conflict in unit propagation, it will analyse the conflict and return with new learnt clauses and backtracking decision level.

We've discussed a basic version of conflict analysis in class, which creates a new clause using decision variables and assigns literals at decision levels less than the current level. For the project, we implemented a slightly better heuristic, which is the first Unit Implication Points (UIP) as described in the SAT Handbook [1].

Starting from the conflict clause, we get the antecedent clauses of the literals at the current decision level and perform resolution. This propagates in the implication graph until the resolved clause contains only one literal from the current decision level. This allows ConflictAnalysis to terminate earlier and also backtracking to jump to a decision level which has a greater impact on the solver assignments.

3.2. PickBranchingVariable

The order of assignments of variables can affect the performance of the solver in the time taken to reach a conflict or a satisfying assignment. Hence, using heuristic can help the solver to reduce the search space and reach the solution faster.

Based on our observation, a good branching heuristic should greedily satisfy as many clauses for SAT instances, whereas lead to a conflict as soon as possible for UNSAT instances. It is also desirable to have heuristic with low computational overhead, which means requires less computational effort each time trying to run the heuristic during PickBranchingVariable execution.

We implemented four heuristics, namely:

1. **Random heuristic:**
   Decision variable is chosen at random, and assigned to a random truth value.

2. **Two-clause:**
   Decision variable is chosen from propositions with maximum occurrences in 2-clauses, i.e., with two unassigned literals, and break ties randomly. The variable is assigned to a random truth value.

3. **Maximum occurence (max freq):**
   Decision variable is chosen from propositions with maximum occurrences in all input clauses, and break ties randomly. The variable is assigned to a random truth value.

   This heuristic only calculates the frequency of each proposition once at the beginning of the program, and does not update the frequency table throughout the program. We think that this heuristic has low overhead without recomputing the frequency for every decision.

4. **Dynamic Largest Combined Sum (DLCS)**
   Decision variable is chosen from propositions with maximum occurrences in all unresolved clauses, and break ties randomly. As described in [2], let the count of variable x appears as positive literal be CP, negative literal as CN. We select the variable with the largest CP + CN, assign it to True if CP ≥ CN, False otherwise.

   Unlike Heuristic 3, DLCS recompute the occurrence of each positive literal and negative literal for every decision, hence is named as 'dynamic'. The difference between DLCS and two-clause is that DLCS considers all unresolved clauses, but two-clause only considers two clauses (unresolved clauses with two unassigned literals). However, both of them will need to iterate through all clauses to recompute the occurrence of literals or propositions.

5. **VSIDS (Variable State Independent Decaying Sum):**
   As implemented in Chaff Solver [3], the VSIDS heuristic chooses the literal with the highest score. The score is maintained for each literal (for both positive and negative propositions), initial score is the number of occurrences of the literal in the input clauses. The score is incremented for all literals in conflict clause. After every 256 conflicts, the score is divided by half to give priority to recently learned clauses.

   Since Heuristics 2 to 4 are only counting occurences of literals or propositions, their goal is to eliminate or satisfy as many clauses as possible. Those approaches do not consider the impact of unit propagation and conflicts. VSIDS attempts to address this issue by increasing the score of literals based on more recent conflicts.

   We found out that [4] developed a hybrid heuristic, namely Variable State Aware Decaying Sum (VSADS) that takes into account both occurrences and conflicts, by designing the score such that it is weighted on both DLCS and VSIDS. However, we didn't implement this approach due to limited computational resources to perform subsequent experiments.

4. Experiments

To evaluate and compare the performance of different branching heuristics, we designed the following experiments and measured the performance of the solver based on the total compute time and total number of invocations of *PickBranchingVariable* subroutine.

We wrote a random CNF generator, which takes in 3 parameters: namely N (the number of variables), K (the number of distinct literals per clause) and r (the ratio of the number of clauses to the number of variables).

We fix the variables k and r to be K = 3, r = 4.3. We choose r = 4.3 because according to our result in Project 2 and also the [5] paper, when K = 3, r = 4.3 the probability of a random CNF generator giving a SAT or UNSAT instance is around 50%. Also, the time taken for solving this

category (r = 4.3) is relatively high, which indicates the hardness or complexity of these instances.

The random variable N takes the values: 50, 75, 100, and 125, which corresponds to the number of clauses (L = upper bound of r * N) of 215, 323, 430, and 538. For each value of N, we generate 100 SAT instances and 100 UNSAT instances. (Timeout of 10 minutes is set if using random heuristic.)

5. Results and Analysis

We have included detailed experiment results for each heuristics on each instance in the Appendix section plotted in graphs. Note that we did not run experiments for N=125 SAT and N=100/125 UNSAT (for the Random heuristic) because it will take too long to run due to frequent timeouts which will eventually take too long to complete 100 instances.

Table 1 shows the average time taken to solve 100 3-CNF instances for each N.

Table 1. Average *Time Taken* to solve 100 3-CNF instances (seconds)

	N	random	two clause	max freq	DLCS	VSIDS
SAT	50	0.16	0.03	0.04	0.04	0.03
	75	4.64	0.18	0.27	0.24	0.15
	100	99.53	0.95	2.50	1.40	1.04
	125	NA	4.40	12.75	8.82	7.03
UNSAT	50	0.53	0.07	0.09	0.09	0.10
	75	16.80	0.41	0.57	0.66	0.68
	100	NA	2.73	5.36	5.17	5.69
	125	NA	17.42	48.05	40.75	40.13

As shown in Table 1, the two-clause heuristic takes the shortest time to solve 100 3-CNF instances on average, except for the (SAT, N=75) instance, where the VSIDS heuristic took 0.03 seconds shorter. However, if we examine closely at Figure A1 in Appendix, we can spot a few instances where the two clause heuristic took longer than the other heuristics. For a large number of variables (N >= 100), two-clause heuristic is the best choice which clearly outperforms the other heuristics.

For SAT instances, all heuristics except for random take a reasonably small amount of time. VSIDS appears to be second fastest after two-clause for N=100/125. This result shows that conflict clauses provide a good indication of literals to be assigned in order to satisfy the formula, and counting heuristics such as max_freq or DLCS might be too greedy, so that a lot of time is taken in unit propagation or conflict analysis as a result.

For UNSAT instances, max_freq, DLCS and VSIDS show similar performances, where VSIDS is slightly better when the number of variables is large. We also observed that max_freq heuristic is able to solve both SAT and UNSAT with a little more amount of time than DLCS

and VSIDS heuristics. This indicates that static heuristics can have similar performance as dynamic heuristics.

Table 2 shows the average #PickBranchingVariable subroutine to solve 100 3-CNF instances.

Table 2. Average *#PickBranchingVariable* subroutine to solve 100 3-CNF instances

	N	random	two clause	max freq	DLCS	VSIDS
SAT	**50**	109.12	25.61	31.61	29.93	31.25
	75	731.90	63.14	101.56	80.93	69.89
	100	3458.46	165.18	352.15	219.99	205.08
	125	NA	393.78	787.45	599.02	552.53
UNSAT	**50**	269.46	39.94	55.36	49.84	64.04
	75	1850.07	117.00	177.72	178.69	214.23
	100	NA	381.41	642.22	610.90	717.25
	125	NA	1033.04	1946.73	1789.60	1912.39

As shown in Table 2, the random heuristic invocates triple to quadruple the branches as compared to the other heuristics at SAT, N=50, and this becomes approximately 10 times the other heuristics at SAT, N=100. For solving UNSAT formulas, the random heuristic performs even worse, with approximately 5 times the other heuristics at UNSAT, N=50; and approximately 10 times the other heuristics at UNSAT, N=75. This shows that random heuristics which have no idea of the search space can result in exponential effort in searching for the solution.

It is interesting to note that although the VSIDS heuristic almost always invocates the most branches for UNSAT instances, it still takes the second shortest time to solve for 100 3-CNF instances on average. There is an exception at the SAT, N=75 case, where the VSIDS heuristic invoked the second least branches, which then allowed it to have the fastest solving time on average in this category.

The smallest numbers of #PickBranchingVariable invoked by two-clause heuristics shows that it effectively prunes away non-beneficial branches and chooses better variables to be assigned. We also notice that the time taken to solve the CNF formulas is proportional to the number of #PickBranchingVariable subroutine.

6. Einstein Puzzle

In this puzzle, we have 5 houses and the owner of each house has 5 attributes: Color, Nationality, Drink, Cigarette and Pet. We assigned each value of the attributes to a reference number, as shown in Table X below.

Table X. Assigning reference number (id) to each value of each attribute

Color	Nationality	Drink	Cigarette	Pet
Yellow (id: 0)	Norwegian (id: 5)	Water (id: 10)	Dunhill (id: 15)	Cat (id: 20)

Blue (id: 1)	Dane (id: 6)	Tea (id: 11)	Blends (id: 16)	Horse (id: 21)
Red (id: 2)	Brit (id: 7)	Milk (id: 12)	Pall Mall (id: 17)	Bird (id:22)
Green (id: 3)	German (id: 8)	Coffee (id: 13)	Prince (id: 18)	Fish (id: 23)
White (id: 4)	Swede (id: 9)	Beer (id: 14)	Blue Masters (id: 19)	Dog (id: 24)

Each value can belong to one of the 5 houses, hence we encode the idea of “Yellow (id: 0) is for House 1” as the variable 0*5 + 1 = 1, “Norwegian (id: 5) is for House 3” as the variable 5*5 + 3 = 28. This totals to **125 variables**, each indicating the truth value of a specific value in the respective house.

The 15 given rules can be divided into 4 categories.

Category 1: A value is found at a house.
The rules are encoded as the variable itself, which must be satisfied.

1. The man living in the center house drinks milk.
2. The Norwegian lives in the first house.

Category 2: Two values are found at the same house.
For each house, the two values are found in the house, which can be expressed as $(p \leftrightarrow q)$ thus translates to $(\neg p \vee q) \wedge (p \vee \neg q)$.

1. The Brit lives in the red house. ($\forall house_i$, Brit in $house_i \leftrightarrow$ Red in $house_i$)
2. The Swede keeps dogs as pets.
3. The Dane drinks tea.
4. The green house’s owner drinks coffee.
5. The person who smokes Pall Mall rears birds.
6. The owner of the yellow house smokes Dunhill.
7. The owner who smokes Bluemasters drinks beer.
8. The German smokes Prince.

Category 3: One value is on the left of another value.
Only one rule falls into this category, which here we make the assumption that the green house is adjacent to the white house on the left.

1. The green house is on the left of the white house.

Category 4: Two values are neighbours.
The rest of the given rules fall into this last category, where we need to take care of both sides as well as the edge cases (leftmost and rightmost house).

1. The man who smokes Blends lives next to the one who keeps cats.
2. The man who keeps the horse lives next to the man who smokes Dun-Hill.
3. The Norwegian lives next to the blue house.
4. The man who smokes Blends has a neighbor who drinks water.

There are however more implicit rules that we need to feed to the SAT solver.

Each color, nationality, drink, cigarette and pet must exist in at least one house.

For example,
$P_1 = Pet1$ in a specific house, $P_2 = Pet2$ in the same specific house,etc.
$P_1 \vee P_2 \vee P_3 \vee P_4 \vee P_5 = 1$

Now that we have made sure each of the items must exist, we need to ensure they only exist once, i.e. For each unique item, it cannot exist in two houses.
For example, $C_1 = Cat\ is\ in\ House\ 1,\ C_2 = Cat\ is\ in\ House\ 2,\ etc$
Cat cannot exist in both House 1 and House 2:$(\neg(C_1 \wedge C_2)) = (\neg C_1 \vee \neg C_2)$
To expand the formula for Cat:
$(\neg C_1 \vee \neg C_2) \wedge (\neg C_1 \vee \neg C_3) \wedge (\neg C_1 \vee \neg C_4) \wedge (\neg C_1 \vee \neg C_5) \wedge (\neg C_2 \vee \neg C_3) \wedge (\neg C_2 \vee \neg C_4)$
$\wedge (\neg C_2 \vee \neg C_5) \wedge (\neg C_3 \vee \neg C_4) \wedge (\neg C_3 \vee \neg C_5) \wedge (\neg C_4 \vee \neg C_5)$

For each house, there only exist 1 item of the same group.
For example, $Y_1 = House\ 1\ is\ Yellow,\ B_1 = House\ 1\ is\ Blue,\ etc$
A house cannot be both yellow and blue: $(\neg(Y_1 \wedge B_1)) = (\neg Y_1 \vee \neg B_1)$
The expansion for House 1 follows similarly as above.

We've generated 655 clauses in total. Lastly, in order for our SAT solver to answer the question "who owns the fish?". We add the last constraint, constraining the fish in each of the five houses, and ask our SAT solver whether this is SAT or UNSAT. If, for example, the constraint of fish in House 1 results in a SAT, it means that the fish is in fact in House 1. The next step is to find out who lives in House 1, and this person will be the owner of the fish. To do this, we add another constraint, constraining house 1 with each of the 5 nationalities and check which results in a SAT.

Our SAT solver took about 0.1 seconds to solve the Einstein Puzzle using the two-clause heuristic. Giving the answer that the Fish is in House 4, German is in House 4 and so the fish is owned by the German.

7. Conclusion

In conclusion, we've designed a SAT solver based on CDCL algorithm to check satisfiability of CNF formulas for this project. We've implemented 5 heuristics: random, two clause, maximum occurence (max freq), Dynamic Largest Combined Sum (DLCS) and VSIDS (Variable State Independent Decaying Sum). From our results, the two clause heuristic takes the shortest time and invocates the least amount of *#PickBranchingVariable* subroutine on average. Our SAT solver managed to solve the Einstein Puzzle problem using the two clause heuristic at around 0.1 seconds and gave the answer that the German owns the fish.

8. Learning points & Reflection

Evelyn Yi-Wen Chen: I learnt the importance of a SAT solver in computer science, and now that I've worked through this project, I can truly appreciate the effort computer scientists have

put in to make the first SAT solver work. I've also learnt the importance of correct encoding, as we need to include not only the explicit constraints but also the implicit ones so that our SAT solver will have the same knowledge as us. Lastly, I regret starting the project too late, so that we missed out on the valuable feedback from the feedback session.

Yang Zi Yun: I learned the importance of implementation of correct and efficient data structures in the process of converting algorithms into working code. I also found out that different heuristics can be designed from different perspectives of the solver, which all aims to exploit the problem structure to reduce the search space. The process of encoding a problem in CNF and solving it by SAT solver is also interesting.

We will grade ourselves A- for this project. There is still a lot of room for improvement such as using Object-Oriented design for our solver which will improve the testability and scalability of the solver. Besides that, unit propagation can be improved by implementing two-watch literals and lazy data structure. Also, we wish to come out of our own heuristic but only used existing ones due to time constraints. However, we've done a comprehensive testing of our solver, comparing the different heuristics extensively.

References

[1] Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch. (2008) Handbook of Satisfiability, https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf

[2] J. P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence, pages 62–74, 1999.

[3] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th Design Automation Conference, pages 530–535, Las Vegas, NV, June 2001. ACM/IEEE

[4] Tian Sang, Paul Beame, and Henry Kautz. 2005. Heuristics for fast exact model counting. In Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing (SAT'05). Springer-Verlag, Berlin, Heidelberg, 226–240. DOI:https://doi.org/10.1007/11499107_17

[5] Scott Kirkpatrick and Bart Selman. (1994) Critical Behavior in the Satisfiability of Random Boolean Expressions. *Science*, New Series, Vol. 264, No. 5163 (May 27, 1994), pp. 1297-1301

[6] Raymond Hettinger. (2019) SAT Solvers, https://rhettinger.github.io/einstein.html

Appendix

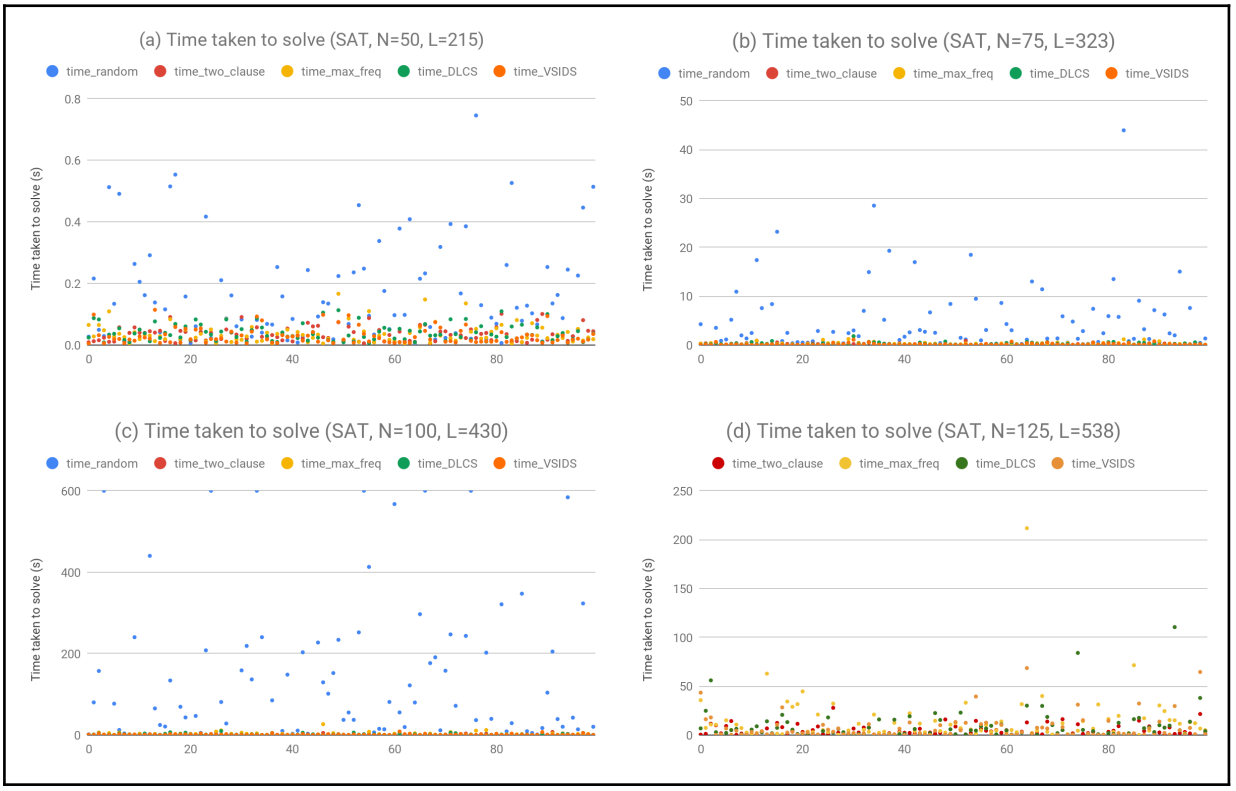

Figure A1. **Time taken** to solve 100 3-CNF instances (**SAT**, r=4.3, N=50/75/100/125)

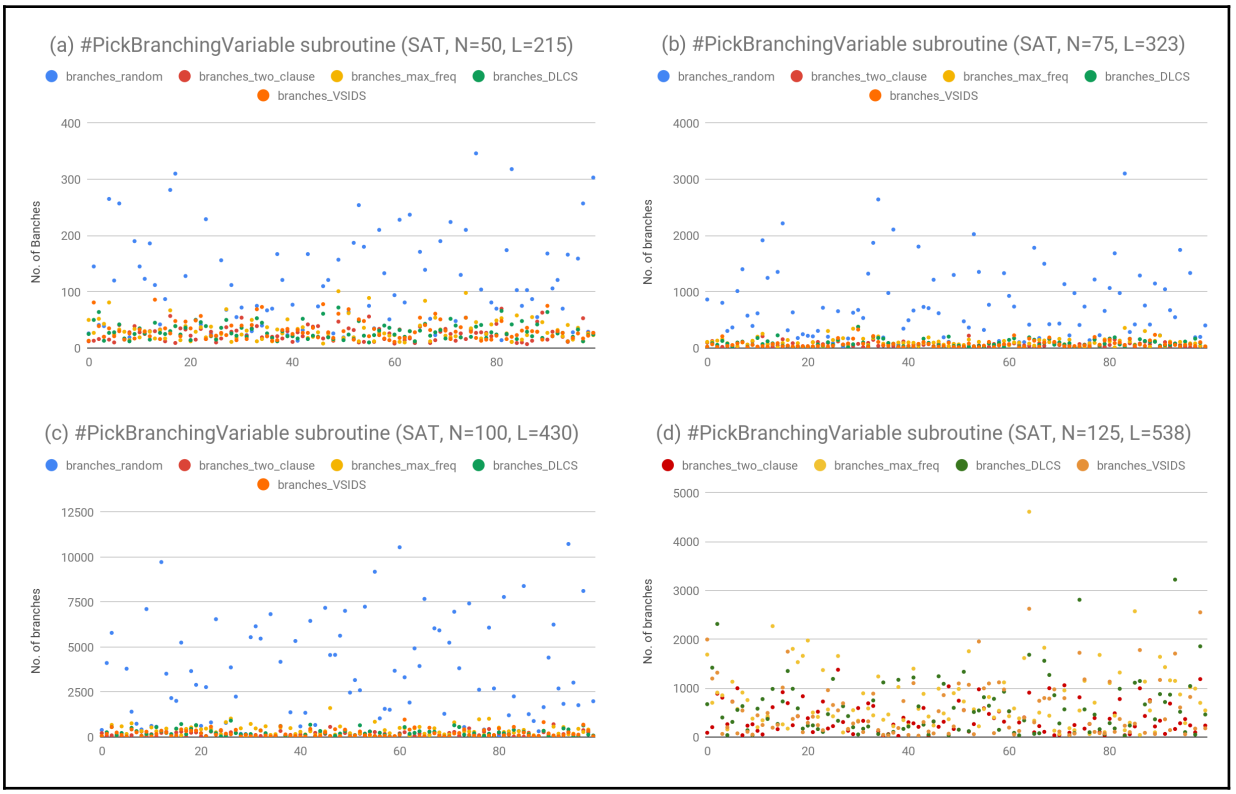

Figure A2. **#PickBranchingVariable subroutine** to solve 100 3-CNF instances (**SAT**, r=4.3, N=50/75/100/125)

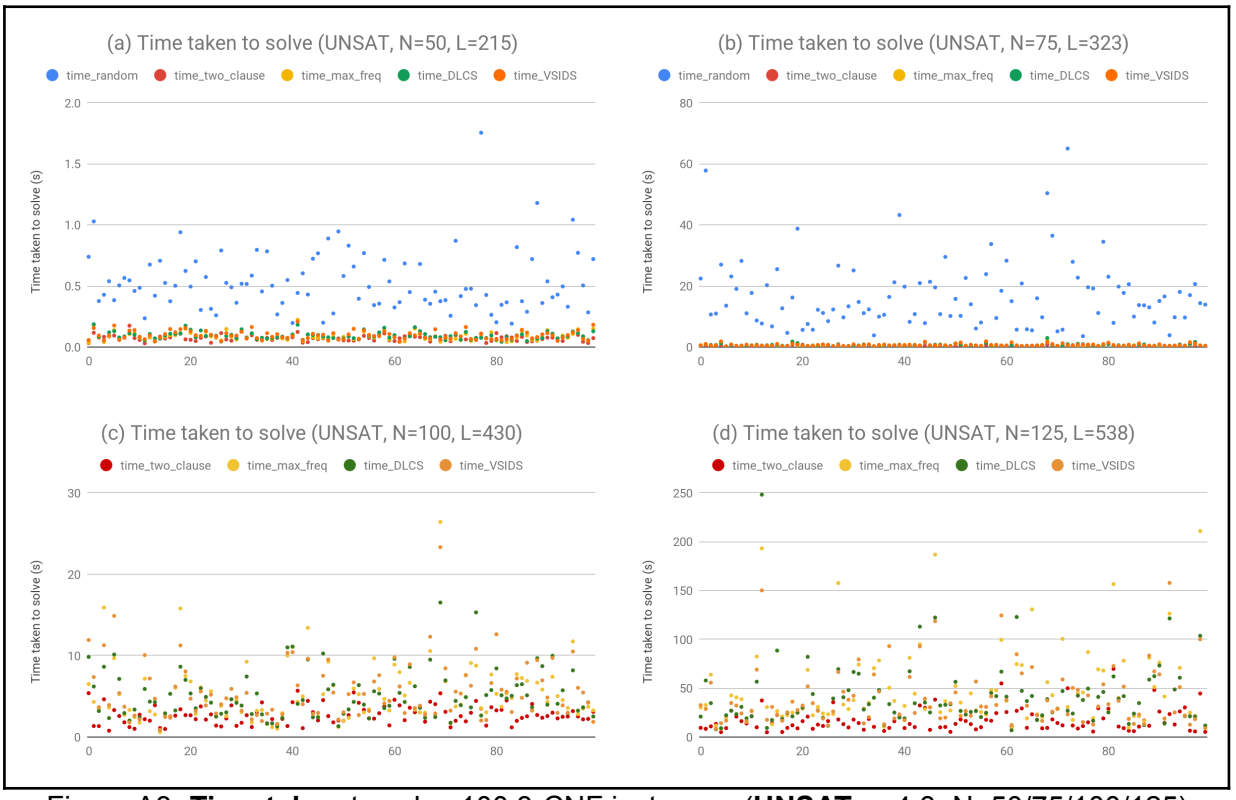

Figure A3. **Time taken** to solve 100 3-CNF instances (**UNSAT**, r=4.3, N=50/75/100/125)

Figure A4. **#PickBranchingVariable subroutine** to solve 100 3-CNF instances (**UNSAT**, r=4.3, N=50/75/100/125)