MIN-MAX-MOST (3M) heuristic for DPLL algorithm

Project Details

1 Introduction

In this project, we implement a DPLL algorithm for SAT solving. Let us first summarize the DPLL procedure. DPLL explores the space of truth assignments to find an assignment that holds. We use the word "active" proposition in a clause if it is still not assigned any value in the evaluation. At the beginning of the algorithm all propositions are active.

DPLL procedure has two main components,

- Unit preference Rule : Whenever only 1 proposition is active in any clause, we choose that proposition and the value assignment for this proposition is the one that satisfies that singular clause.
- Splitting Rule: when all clauses have atleast 2 propositions, we choose a predicate and then set that to true and false separately and recursively call the DPLL method with a reduced set of active propositions.

The lot of effort has gone into devising intelligent ways to choose this splitting predicate. We evaluate the heuristic of 2-occur and our own heuristic of MIN-MAX-MOST heuristic (3M heuristic) which is inspired by 2-occur.

2 How heuristics work (also learning from this project)

Before delving into the details of our heuristic we want to spend some time discussing the working of heuristics.

Unit Preference is the obvious rule to apply

First, we see that unit preference rule is the most obvious rule and is the best possible action you can take given a particular formula. The reason for this is that , no assignment that assigns the incorrect value to this proposition is ever going to lead to SAT. Thus, the only possible choice for this case is defined by this unitary clause. Choosing this proposition first, however, reduces the number of clauses to evaluate thus affecting the total computation and the number of active propositions as will be discussed in next part of this section.

Does choosing the splitting predicate matter ?

Let us investigate this question. Now evaluating the formula of m clauses with n active propositions where each clause if of length (number of literals) bounded by say l for all possible solutions is

$$cost \propto m \times 2^n \times l \tag{2.1}$$

Now note that choosing a proposition p vs q at the splitting step affects these quantities. Let us see how,

- m for the next step is the number of clauses which remain undecided. For example if we choose proposition p and set it to true, then all the clauses which have p are no longer required to be evaluated in next steps

$$m' = m - \text{ \# clauses which have literal p (if p = true) or } \neg \text{ p (if p = false)} \tag{2.2}$$

- n for the next step is the number of active propositions that remain from the m' undecided clauses.

- l for the next step is the updated length for the m' clauses that are remaining

From the above equations, it is clear that choosing the correct propositions to evaluate the satisfiability can affect the computation time drastically. However, this view point is geared towards reaching UNSAT. There might also be some heuristics that are geared towards quickly reaching SAT especially in choosing the order of trying false or true as values. To conclude,

1. Order of splitting predicates matter.
2. Order of choosing the values (true or false) for splitting predicates matter.

3 Heuristics that did not work vs 2-Occur

We tried a bunch of heuristics. However most of them don't work very well. For example if we choose the proposition that appears maximum number of times in the formula, it does not work better than 2-occur. It should be noted that this might work better than random selection. We compare with 2-occur which is especially strong when it comes to 3-CNF formula. We also tried choosing the proposition which has maximum number of occurances either as positive or negative literal. Here we count p and $\neg p$ separately. Even in this case, we could not beat the 2-occur heuristic.

Why 2-occur works the best is because it quickly gives the opportunity to apply unit-preference rule in the next step. The unit-preference rule is, as discussed above, the most cheap and useful rule.

4 MIN-MAX-MOST (3M) heuristic

We now describe the 3M heuristic. The heuristic is to choose the proposition which appears maximum number of times (MAX) in the clauses which have the minimum length (MIN) and assign it the value corresponding to the literal that appears most (MOST) in this set of minimum-length clauses. The idea behind this heuristic is again to try and push most clauses in the formula towards unitary clause which seems to perform well. By choosing the literal that appears most, we ensure that more number of clauses are decided in this step thus affecting future computation.

In terms of choosing the predicate this is a generalization of 2-occur rule. The 3M heuristic will show similar choices on the smaller length CNF formulas . However, things are different when we go to higher length CNF formulas like 4-CNF,5-CNF and so on.

5 Implementation details

The implementation is written in cpp and is available at https://github.com/apd10/comp509project. We implement the following splitting strategies.

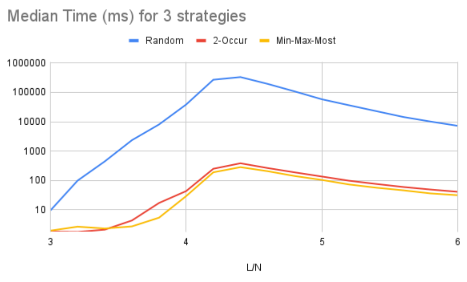
```
FIRST  0
RANDOM  1
2-OCCUR 2
3M 3
```

In order to compile the code run the make file. This will create a binary solve. In order to run the code use

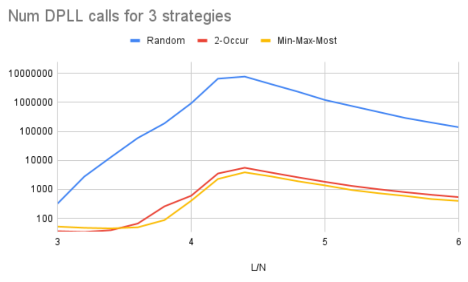
```
./solve <problem.file> <strat number>
```

For example to run a problem file problem.150.450 with strategy 3M use

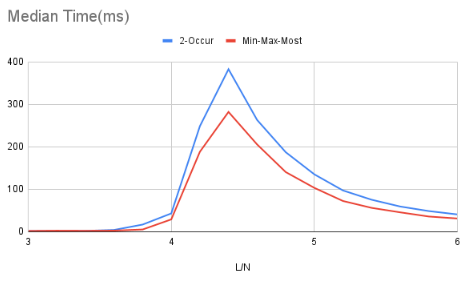
```
./solve problem.150.450 3
```



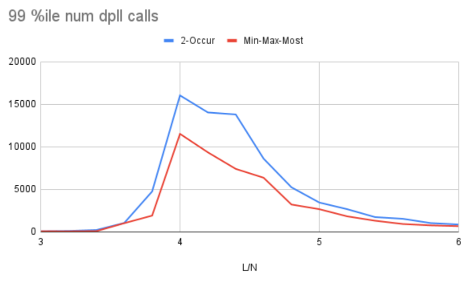
(a) Median time



(b) Median DPLL calls



(c) Median times



(d) 99%-ile number of DPLL calls

Figure 1: Time and number of DPLL call results for different strategies - Random, 2-Occur and Min-Max-Most

Random problem file generator We have added scripts to generate the random CNF problems. These scripts are present in the eval folder. In order to generate a file of 100 problems with N=150, L=450 and K=3, run

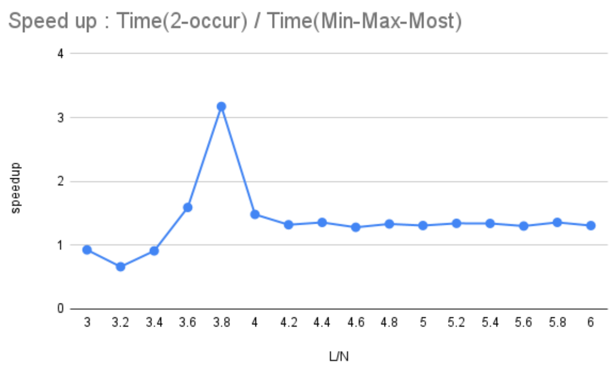
```
python3 ./eval/generate_random_problem.py 150 450 3 100 &> problem.N150.L450.K3.num100
```

6 Evaluation

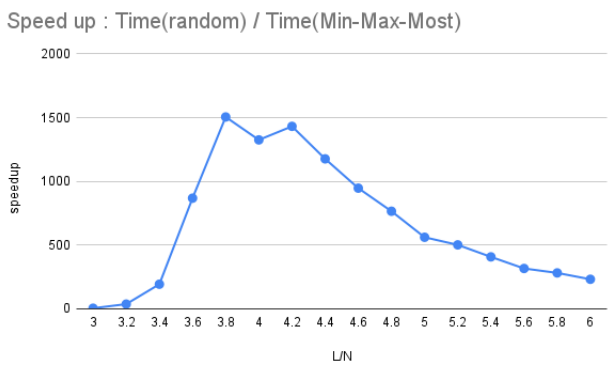
We use N=150 and K=3 and vary the value of L from 450 to 900 as per the instructions. The result comparing time, number of split calls, number of dpll calls and number of unit preference applications are shown in the figures 1. We can see that 2-Occur and 3M strategy both outperform random strategy by a large margin. We can see this in the domain of number of DPLL calls in total which, in turn, translates into the time as well. Among 2-occur and 3M, 3M consistently beats 2-occur in this case of 3-CNF formulas both in terms of DPLL calls and time.

We can observe the comparison between 3M and other strategies in the plots of the ratio of the performance as shown in figure 2. As can be seen, 3M achieves over 1000x speedup in the interesting region (why interesting? see probability plots in following discussion). Also, it gets a consistent performance boost over 2-occur which is around 1.35×.

We can see from the figure 3, that the probability of a random formula with K=3 and N=150 being satisfiable depends on the ratio L/N. When $L/N \leq 3.8$, the formula is satisfiable with high probability (almost 1) and when $L/N \geq 4.8$ the formula is almost always UNSAT. The region between $L/N = 3.8$ to 4.8 is interesting. We expect the probability to go down as number of clauses increase. This is because as number of clauses increase, the number of constraints increase. However, there is something fundamental about L/N around 3.8 and 4.8 since something completely breaks here. It would be interesting to see if we can identify this fundamental shift at these two points.



(a) Speed up of 3M vs 2-occur (Median times)



(b) Speed up of 3M vs random (Median times)

Figure 2: Speed up plots. As we can see 3M achieves over 1000x speed up when compared to random strategy and 1.3x average speed up when compared to 2-Occur

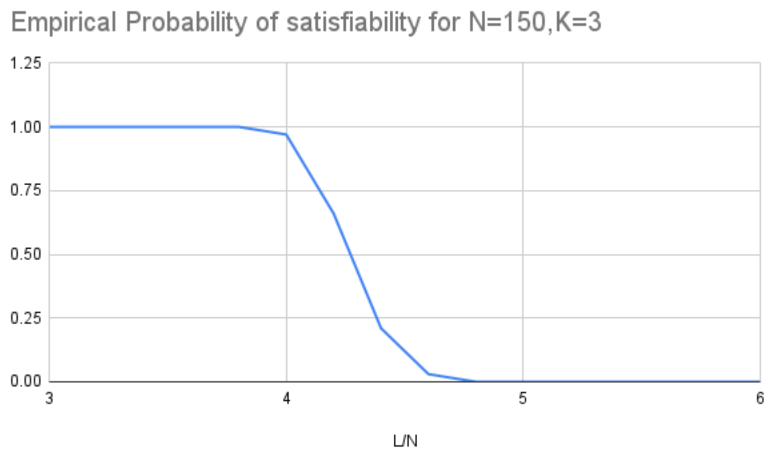


Figure 3: Empirical probability of satisfiability of random formulas

7 Conclusion

To summarize this project, we learned that for a random CNF formula, there is a specific region of L/N where the formula can be both SAT or UNSAT with reasonable probability. In the region on the left, the problem is almost always SAT whereas on the right the problem is almost always UNSAT. There is a fundamental shift of paradign across these boundaries. To solve the problem of CNF-SAT, we see that heuristics are quite effective and can give over $1000\times$ of speed up as compared to randomly exploring the space of truth assignments. We also saw that 2-Occur is a really good heuristic for 3-CNF formulas. We showed that 3M which is an extension of 2-Occur along with choosing the order of value assignment consistently outperforms the 2-Occur heuristic.

8 Interim Report

Implementation : We implement a DPLL with unit-preference and picking the first proposition for splitting. The code is present at https://github.com/apd10/comp509project files described below

- Code for splitting method

- encoding generating script in generate problem.py
- puzzle.in file with the puzzle
- solution can be generated by using ./solve puzzle.in.
- the mapping from integer variables to worded variables can be seen in log (generated by generate_problem.py)

To run the code

- run ‘make‘
- ./solve puzzle.in

The solution is that fish is owned by German.