Uniform: A Probabilistic SAT Solving Algorithm

Charles Dana
charles.dana@hec.edu

Abstract

SAT solving algorithms mainly focus on improving the backtracking approach. In this paper, we present a probabilistic alternative to backtracking that uses only the uniform heuristic as a choice function: Uniform, a markov chain that converges to the solution when it exists. This algorithm will be presented in Python and C with a set of benchmarks, as well as a conjecture that a class of non-trivial SAT problems can be solved in quadratic time. We will also compare the performance of the algorithm with instances of SATLIB: Uniform Random-3-SAT. The goal of this paper is to present an alternative to CDCL that does not rely on Boolean constraint propagation or clause learning to compete with the most advanced SAT solvers.

1 Introduction

SAT problem solving started in 1960 when Davis and Putnam presented a procedure [1] to solve any SAT instance with promising results: they managed to beat computers working by hand. The only problem with their algorithm is that it creates an exponential amount of clauses when working on large SAT instances. It is therefore unusable in practice. What remains of the DP procedure is the DPLL algorithm [2]. This is an intelligent backtracking algorithm, whose features, such as Boolean constraint propagation and pure literals, are still relevant. Boolean constraint propagation consists of adding variables to a clause where only one variable remains available for the solution. Pure literals consist of adding a variable when there is no occurrence of its negation in the remaining constraints. The DPLL process can be summarized as follows. Make a decision based on a heuristic, add all possible information without making any further assumptions, and repeat until you reach a contradiction. If you reach a contradiction, go back to your last decision, and reverse the associated variable, before repeating the process until you reach a solution or realize that you have searched the entire space of partial solutions and proven that there is no solution. In 1999, Schöning introduced WalkSAT: a local search approach for solving SAT with great performance at the time. Instead of trying to deduce a solution using logic and backtracking, WalkSAT consists of exploring the field of complete assignments, starting from an initial guess and a random walk on $3n$ neighbors, with variable flips. There are many variants of WalkSAT using features such as Tabu search [4], no aggravating flip that would increase the number of unsatisfied clauses. WalkSAT had better theoretical performance than DPLL at that time but it lacks an important feature, it cannot prove the unsatisfiability of the SAT instance. In 2005, Chaff [7] was introduced as a breakthrough for backtracking as an SAT solver. It retained the features that made DPLL successful, but added clause learning: by studying the implication graph, the algorithm can actually learn new clauses when it reaches a contradiction without changing the solution of the problem. This has reduced the solution time considerably, and current solvers are conflict-oriented: they make mistakes as fast as possible to learn as much as possible about the problem, so as not to explore the search space too deeply before reaching a solution. We call these solvers CDCL (Conflict Driven Clause Learning). This is the current state of the art of SAT solvers [8].

2 Generating non-trivial SAT problems

According to the definition of SATLIB [6] you can generate hard random SAT instances with the following piece of code.

```
from random import randint, choice
from time import time

def generate(n):
  t = time()
  problem = []
  while len(problem) < 4.3 * n:
    clause = [choice([-1, 1]) * randint(1, n) for dummy_x in range(3)]
    if len(set(clause)) == 3:
      if [x for x in clause if -x in clause] == []:
        if [x for x in clause if x > 0] != []:
          problem += [clause]
  orientation = {abs(x) : choice([-1, 1]) for clause in problem for x in clause}
  problem = [[x * orientation[abs(x)] for x in clause] for clause in problem]
  print("generation duration " + str(time() - t))
  return problem
```

This is called an instance of uniform random 3-SAT with the solution planted, with a clause/variable ratio of 4.3. This means that there is an equal probability between the integers to be in the clause as long as there are three distinct integers, two integers are not opposite and one of them must be positive. The "all positive" solution is trivial in this form, but as soon as we change the sign of the variables, the problem becomes very difficult. Since a SAT solver cares whether a variable is positive or negative, it is necessary to change the sign. Finally, this ratio of 4.3 has been identified by the scientific community as difficult to solve. Intuitively, a smaller ratio would imply fewer constraints, thus a simpler problem to solve, and too large a ratio would make the solution planted by a CDCL obvious [8].

3 Verifying a solution

```
def verify(solution, problem):
  t = time()
  for x in solution:
    if -x in solution:
      print("boolean error")
  for clause in problem:
    if [x for x in clause if x in solution] == []:
      print("clause error")
  print("verify duration " + str(time() - t))
```

This piece of code runs in linear time $O(n)$ and a solution to the problem would only print the verification time in the console. If "boolean error" is printed, it means that your solution contradicts itself, there is a contradiction (x and $-x$) in your solution. If "clause error" is printed, it means that there is a clause/constraint such that no element of the clause is contained in the solution. Intuitively, finding a solution is tricky because you can't contradict yourself and you have to satisfy all constraints simultaneously.

4 Uniform: Solving non-trivial SAT instances

```
from random import choice
from time import time

def solve(problem):
    t = time()
    solution = set()
    constraints = [tuple(clause) for clause in problem]
    assigned = {x : [] for clause in problem for x in clause}
    for clause in problem:
        for x in clause:
            assigned[x] += [tuple(clause)]
    while len(constraints) > 0:
        clause = choice(constraints)
        undefined = [x for x in clause if not -x in solution]
        if len(undefined) > 0:
            x = choice(undefined)
            solution.add(x)
            for clause in assigned[x]:
                if clause in constraints:
                    constraints.remove(clause)
            if -x in assigned:
                for clause in assigned[-x]:
                    if not clause in constraints:
                        if [y for y in clause if y in solution] == []:
                            constraints += [clause]
        if len(undefined) == 0:
            x = choice(clause)
            solution.remove(-x)
            for clause in assigned[x]:
                if not clause in constraints:
                    if [y for y in clause if y in solution] == []:
                        constraints += [clause]
            for clause in assigned[-x]:
                if not clause in constraints:
                    if [y for y in clause if y in solution] == []:
                        constraints += [clause]
    print("solution duration " + str(time() - t))
    return solution
```

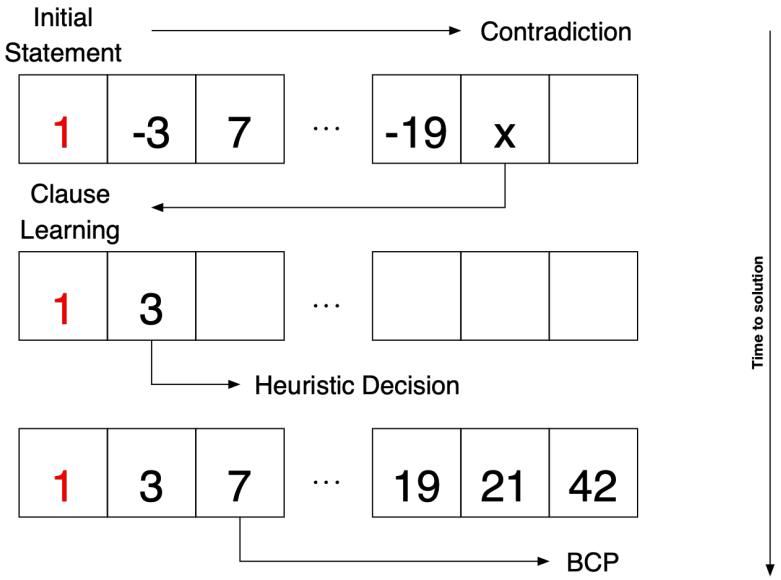
This algorithm uses only the uniform heuristic as a choice function, which returns an element from the list at random. The hold dictionary is useful for reducing the computation time of an iteration of the while loop, which is initially $O(n)$, and as time goes on $o(n)$. If you follow the length of the constraints, which is the number of clauses not satisfied by the current solution, it generally decreases, and oscillates around $1 - 20$ for a problem with 1000 clauses. The closer you are to the solution, the faster the algorithm is. Apart from this performance note, Uniform works as follows. It starts by initializing the solution to the empty set, and the constraints as all the clauses of the problem. It chooses a constraint at random, and computes the set of variables that are not assigned in the solution, meaning that they are not already defined in the other direction. If this set is not empty, it selects one at random and adds it to the solution. If this set is empty, it takes one of the variables from the clause and removes its opposite from the solution. It computes the new constraints as the clauses that are not already satisfied by the current solution. This process is repeated until there are no more constraints, and the solution is returned.

Conjecture Let T_n be the time to solution of the problem of size n.

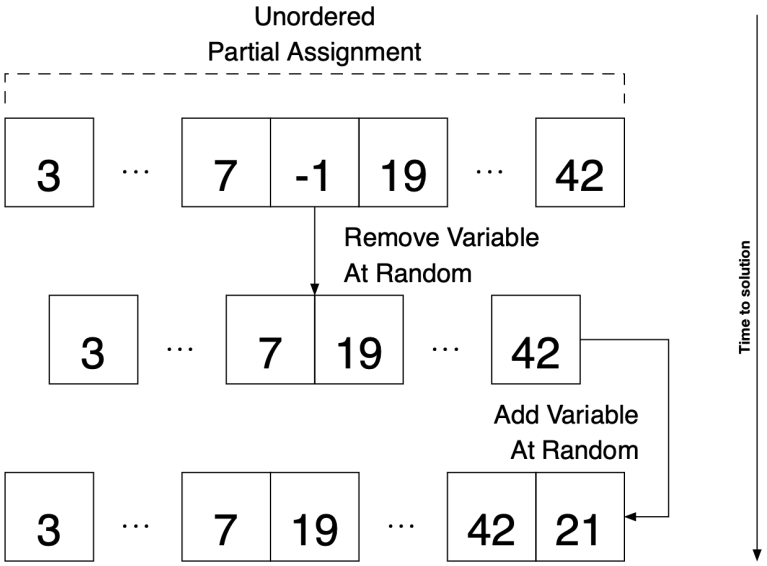
$$\frac{\ln T_n}{\ln n} \xrightarrow[n \to +\infty]{\mathbb{P}} 2$$

5 Paradigm shift

To better understand the innovation associated with Uniform, we will describe the paradigm shift by first proposing a simplified model of how CDCL manages to find an existing solution.



The main reason that SAT CDCL solvers have trouble tackling random instances is that before proceeding to learn the clauses, there is little or no information about what a satisfactory initial assumption would be. In this scheme, the initial assumption that 1 is in the solution is correct, but if no solution containing 1 is associated with the problem, the solver must traverse the graph in search of a contradiction involving 1 that is strong enough to revert to its initial assignment and finally be able to reach a solution. The main problem with CDCL is that the variable assignment is ordered, and although you can skip some of the variable assignment with Boolean constraint propagation and clause learning, the back and forth generates an exponential amount of computation.

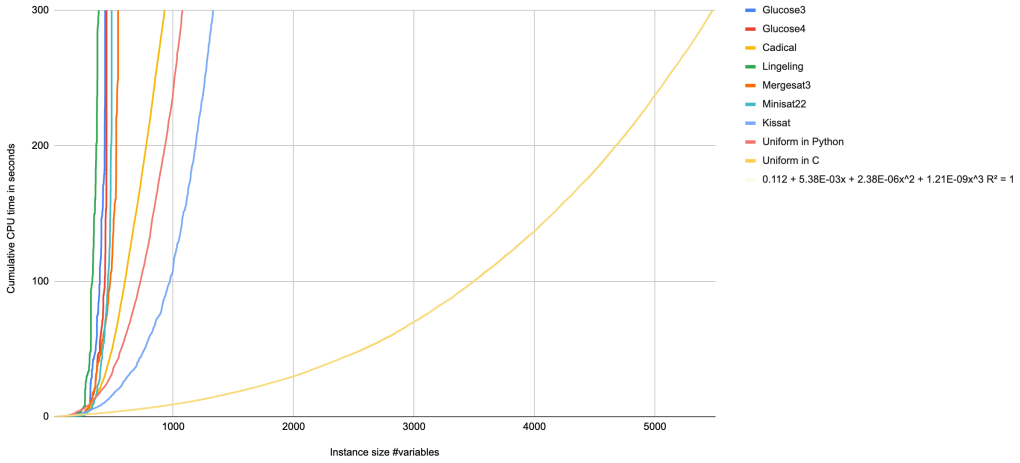


The paradigm shift introduced by Uniform is the following: the search for a solution can be seen as a random walk on the finite space of partial assignments (3^n possibilities). An unordered partial assignment is a set of variables. At each iteration of the while loop, the algorithm adds/removes a single variable from the set. This is done randomly using the uniform heuristic on 1) the clauses that are not satisfied by the current assignment 2) the variables within that clause. Intuitively, if your partial assignment has a lot of variables that should not be part of the solution, this will create a lot of unsatisfied constraints that will result in a higher probability of variable deletion. The uniform heuristic used in the implementation as a choice function may be questionable, but we will see that this simplistic logic shows potential on randomly generated instances.

6 Benchmark

6.1 Solving tailor-made instances

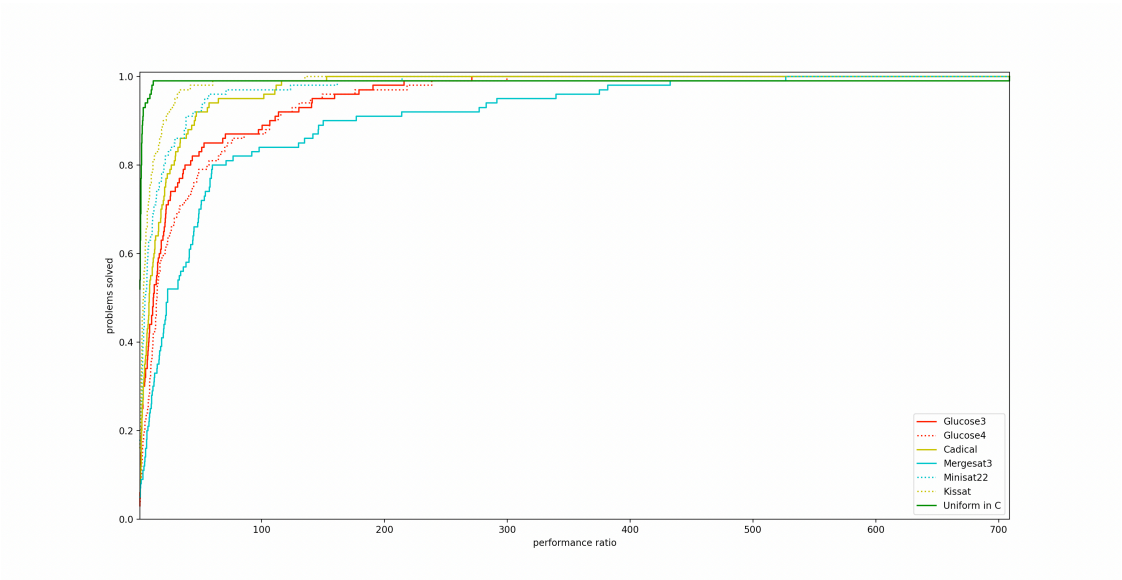
The Benchmark consists of the instances generated in Section 2. We gave 300 seconds of computational time to the best CDCL solvers on the market on a 2017 15-inch MacBook Pro, generating consecutive $10, 11, 12, 13$ problems up to 6000 by increasing the number of variables one by one. This should give an incentive on the solving speed, and the associated complexity of Uniform in Python and Uniform in C. The source code of Uniform in C is available here, the Python script that performed the benchmark is available here.



The results place Uniform in C in first place, with a trend polynomial of degree 3, which seems to highlight the conjecture of Section 4. Second place goes to Kissat, followed by Uniform in Python, and CaDiCaL in fourth place.

6.2 Solving uf250-1065 instances from SATLIB

The second Benchmark is made of the uf250-1065 instances of SATLIB. The methodology remains the same, but this time the problems are known in the scientific community for their difficulty. The cumulative computation time is not relevant since these problems have the same number of variables, we used the performance profile methodology instead.



Uniform in C shows excellent performance on most instances, except the last one. We tested the algorithm on other SATLIB instances, and the results are not good: Uniform in C only works on instances where the positive/negative ratio of the occurrence of problem variables is close to 1. This leads to the conclusion that the uniform heuristic is limited in solving most SAT instances.

7 Conclusion

In this paper, we have presented an alternative to CDCL on a specific class of SAT problems called Uniform. The purpose of this algorithm is to present a conjecture about non-trivial SAT problems that can be solved in polynomial time. Behind the benchmark results, this raises a question mark as to whether the random walk as an SAT solver can lead to a breakthrough in real-life problems. The real weakness in solving SAT problems and NP-complete problems more generally lies in the scale factor. Large instances are beyond the reach of the CDCL approach, and Uniform's ability to solve the described subclass of NP-complete problems in quadratic time could pave the way for solving large problems with sufficient computational power. Based on empirical experiments, Uniform does not perform well on instances where the ratio of positive/negative occurrences is not close to 1. This is a first result, adding a heuristic on the choice of clauses and variables should be the next step to make this type of solver effective on a wider range of SAT problems.

References

[1] Martin Davis, Hilary Putnam
Journal of the ACM Volume 7 Issue 3 July 1960
A Computing Procedure for Quantification Theory
https://dl.acm.org/doi/10.1145/321033.321034

[2] Martin Davis, Hilary Putnam, George Logemann, Donald Loveland, 1962
A machine program for theorem-proving
https://dl.acm.org/doi/10.1145/368273.368557

[3] Stephen A. Cook, May 1971
The complexity of theorem-proving procedures
https://dl.acm.org/doi/10.1145/800157.805047

[4] Bertrand Mazure, Lakhdar Saïs, Éric Grégoire, 1997
Tabu Search for SAT
https://www.aaai.org/Papers/AAAI/1997/AAAI97-044.pdf

[5] Uwe Schöning, 1999
A probabilistic algorithm for k-SAT and Constraint Satisfaction Problems
https://homepages.cwi.nl/~rdewolf/schoning99.pdf

[6] Holger H. Hoos, Thomas Stützle, 2000
SATLIB: An Online Resource for Research on SAT
https://www.cs.ubc.ca/~hoos/Publ/sat2000-satlib.pdf

[7] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik, May 2005
Chaff: Engineering an Efficient SAT Solver
https://www.princeton.edu/~chaff/publication/DAC2001v56.pdf

[8] Handbook of Satisfiability,
Chapter 4: Conflict-Driven Clause Learning SAT Solvers
Joao Marques-Silva, Ines Lynce and Sharad Malik
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsch, 2008
https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf

[9] Jussi Rintanen, 2010
Heuristics for Planning with SAT
https://link.springer.com/chapter/10.1007/978-3-642-15396-9_34

[10] Hadi Katebi, Karem A. Sakallah, João P. Marques-Silva, 2011
Empirical Study of the Anatomy of Modern Sat Solvers
https://link.springer.com/chapter/10.1007/978-3-642-21581-0_27

[11] Alexey Ignatiev, Antonio Morgado, Joao P. Marques-Silva, June 2018
PySAT: A Python Toolkit for Prototyping with SAT Oracles
https://link.springer.com/chapter/10.1007/978-3-319-94144-8_26