Report draft

Arijit Pramanik

May 12, 2018

Abstract

Given, a boolean formula, generating a satisfying assignment or counting the no. of such satisfying assignments (referred to as model counting) is of immense interest. However, owing to computational complexity, since SAT is a NP-complete problem, we are interested in "near-uniform" or "almost-uniform" generation of a satisfying assignment, as well as approximate counting of the models (a satisfying assignment). The other part is exploring the use of Low density Parity Constraints and leveraging counting techniques to make use of pairwise independent hash functions, and thus explore Average Universal Hash functions. Though a weaker class of hash functions, they retain the desirable statistical guarantees needed by most probabilistic inference methods.

1 Introduction

SAT is a NP-complete problem as shown by Cook, and can be reduced to the basic graph coloring problem, which is coloring each and every vertex of a graph, given some constraints. SAT is widely employed in model-checking and formal verification for generating test-stimuli based on a set of constraints. For finding a satisfying assignment for a Boolean formula, DNF can be converted to CNF in polynomial-time using **Tseitin transformation**, which is *equisatisfiable, not equivalent(which is otherwise NP-hard with an exponential blowup in the size of variables)* Similarly, although exact counting for DNF and CNF formulae are polynomially inter-reducible, there is no known polynomial reduction for the corresponding approximate counting problems.

2 Key points

Model counting is the the no, of satisfying assignments to a propositional Boolean formula, subjected to parity or XOR constraints. One of the leading approaches is to use randomized hashing. Larger constraints involving more than half of the original variables, provide probabilistic accuracy guarantees, however, shorter constraints are easier for SAT solvers, though their statistical properties is not so well understood. Typically, $\{H^{1/2}_{i\times n}\}$ is used corresponding to XORs, where each variable is added with probability 1/2 and hence an average length of $n/2$. Unlike a parity constraint of length 1, a parity constraint of length k can be propagated only after $k-1$ variables have been set.
Ermon et.al. has proposed that long parity constraints are not strictly necessary, and same accuracy guarantees can be obtained with shorter XORs. For a formula with n variables, θ(n) constraints are added and θ(logn) constraint length is necessary and sufficient, over θ(n) constraints in standard long XORs.
Constant factor approximation : To find approximate solutions to NP-hard optimization problems, with provable guarantees on distance of returned solution to optimal one. These APX algorithms find answer within some multiplicative factor of the optimal answer.
Using approximate probabilistic algorithms to compute |S| (set of solutions to a Boolean formula over n variables) - Partition S into 2^m cells, and select a lower dimension cell, and compute whether there is atleast one element in the cell(query to an NP-Oracle : An Oracle is machine that can solve a certain decision problem in a polynomial no. of operations). Repeating this for a small no. of times gives a constant factor approximation to |S| with high probability. We randomly generate these parity constraints (m parity constraints generate 2^m equivalence classes based on whether an odd/even no. of the subset of variables take the value 1). Equivalently, we can use a hash function to hash $\{0,1\}^n$ into 2^m hash bins.

For the algorithm demonstrated in the paper, we can omit the argument of the family of hash functions, by randomly generating a i x n matrix A, where each entry is a Bernoulli R.V. with parameter f_i, and $b \in \{0,1\}^i$ is chosen uniformly, at random, independently from A, we generate the family of hash functions as $h_{A,b}(x) = Ax + b(mod2)$
We need the Δ parameter to get at least $1 - \Delta$ bound on probability. We choose a $c \geq 2$, and then uniformly sample α from $2(min(\epsilon, 1/2 - 1/2^c))^2 ln2$ for a 2^{c+1} approximation of |S| with probability at least $1 - \Delta$. As we increase c, $1/2 - 1/2^c$ increases, so we might need to fix a suitable, sufficiently large ϵ so as to make α independent from ϵ, and strictly, a function of c.

3 Hash functions

We have already seen model counting and discrete integration techniques that are based on the universal hash functions and require an NP optimization oracle for decision-making. Construction of hash functions often utilize modular arithmetic, which maybe interpreted as XOR and parity constraints in the case of model counting problems. The NP oracle is implemented using a SAT solver. The new class of Average Universal Hash functions, are statistically weaker than original ones, and for large-enough sets, size of each hash bucket is sufficiently concentrated around its mean (these are implemented using LDPC codes).

ϵ-Strongly Universal hash functions $H = \{h : \{0,1\}^n \to \{0,1\}^m\}$ if when h is a hash function sampled uniformly randomly, then, $\forall x \in \{0,1\}^n$, the RV $h(x)$ is uniformly distributed in $\{0,1\}^m$ and $\forall x_1, x_2 \in \{0,1\}^n, x_1 \neq x_2, \forall y_1, y_2 \in \{0,1\}^m, P[h(x_1) = y_1, h(x_2) = y_2] \leq \epsilon/2^m$. The case when $\epsilon = 1/2^m$ corresponds to *pairwise independent hash-functions* where $h(x_1), h(x_2)$ are independent.

Statistically optimal hash functions can be potentially constructed by considering h to be all possible functions from $\{0,1\}^n \to \{0,1\}^m$, which are *fully independent* hash functions. However, these are not space-efficient unlike *pairwise independent hash functions* are based on modular arithmetic constraints (XOR constraints) of the form $Ax = b$ mod 2. So, as in week 1, let $A \in \{0,1\}^{m \times n}, b \in \{0,1\}^m$ The family, $H = \{h_{A,b}(x) : \{0,1\}^n \to \{0,1\}^m\}$, s.t. $h_{A,b}(x) = Ax + b(mod2)$ is a family of pairwise independent hash functions.
So, we want to look into a very large(high-dimensional) set S by randomly dividing it into cells using a hash function h, and looking at the properties of a randomly chosen, lower-dimensional set, $h^{-1}(y) \cap S$. This is utilized in model counting problems by adding to the model a set of randomly generated parity constraints. Fully independent hash functions are desired but computationally heavy. There could be high correlation among $\{h(x)\}_{x \in S}$, where all the RVs $h(x)$ are identical, and breaking of S is uneven to the order of a single cell entirely containing S.
We call $H : \{0,1\}^n \to \{0,1\}^m$ to be universal hash function if for $\forall x, y \in \{0,1\}^n . x \neq y, P[h(x) = h(y)] \leq 1/2^m$.

4 Differences between SPARSE-COUNT and ApproxMC

4.1 Overview

SAT solvers are used in both the frameworks where they are fed a set of parity constraints. Longer parity constraints involve more than half of the variables, and hence shorter constraints are generally preferred by these solvers. This is intuitively because for a parity constraint of length k, it can be propagated after all the other variables are set, unlike a single variable constraint, which can be set right away.

The hash functions that are being used in the ApproxMC framework belong to $H_{xor}(n, m, 3)$ (*where n is the no. of variables, and m is the no. of parity constraints*). Each such constraint consists of choosing $y \in \{0,1\}^n$ and taking their XOR. Hence, these m parity constraints are used to break down the solution set S into 2^m cells for further analysis. A calculated threshold or *pivot* is used to partition a cell further by adding one more parity constraint, if it has more than *pivot* elements. Here, we observe random sampling, i.e. each of the n variables are added to the constraint with a probability of 0.5.

For *SPARSE-COUNT*, an optimal constraint density f^* is calculated. Hence, here *f-sparse* *hash functions* are used for partitioning S into cells, and check for existence of at least one element that has a even parity with the sampled hash function, where f^* dictates the probability of a variable appearing in the parity constraint. So for each step out of T steps, a hash function is picked from the family and it is seen how many elements satisfy the original Boolean formula, along with the sampled hash function. **So, SPARSE-COUNT somewhat subsumes ApproxMC in the sense that putting $f^* = 0.5$, we can recover the family of hash functions used in ApproxMC. So, SPARSE-COUNT provides a trade-off between the quality and accuracy of the bounds against the computational resources and time.**

4.2 Statistical Guarantees

ApproxMC takes as input δ, ϵ and hence can produce results accordingly, but would require extra computation time. Here, $1 - \delta$ represents the confidence, and ϵ represents the margin of approximation. These are required to calculate the *pivot*. However, SPARSE-COUNT, under optimal constraint density conditions, produces a *constant factor approximation*, with a tunable confidence parameter.

5 Code review

5.1 Methods modified

Originally, the `AddHash` with added params `iterNo` and f_i function generates a no. of clauses so that the solution set can be decomposed into cells containing a no. of solutions less than pivot. Now, it is used to generate i clauses (i is the iteration index) every iteration, which is equivalent to sampling $h^i_{A,b}$ from the family of hash functions $H^{f_i}_{i \times n}$. For this, two new functions using the old rd (Random device) has been used.

`SetHash` added params `iterNo` and f_i is now just used to invoke `AddHash`,

`GenerateBernoulliBits` generates Bernoulli bits with parameter f_i $i \times n$ times, which gives us the matrix A.
`GenerateUinformBits` generates integers uniformly from {0,1} using the uniform_distribution<int> to generate b, independently and uniformly from A.
`fact`(to be optimized to be tail-recursive) hs been introduced to calculate $n!$,
`choose` - calculate $\binom{n}{k}$,
`epsilonEstimate` - get the value of ϵ, as a function of the no. of variables n, no. of clauses m, $q = 2^{m+2}$, and f_i,
`fOptimizer` have been introduced to calculate the optimal value of f_i for that particular iteration, with `f_initial` $= 0.5$ with a decrease of 0.01 per iteration and decrement of 0.001 per iteration beyond $f_i = 0.01$, with the bound to check for optimality as $4/(2^{m+2} - 1)$,
constant factor approximation `c` is set to 2.
α is set to $(\epsilon)^2 ln(2)$ for $\epsilon = 3/10$ and $epsilon = 5/9 - 1/2 = 1/18$ respectively.

5.2 Modifications to ApproxMC → SPARSE-COUNT

`numExplored` here is the iteration counter for the main loop, which loop till the no. of variables n. $T < -ceil(\frac{log(1/\delta)}{\alpha} logn)$. The inner loop loops from 1 till T, where `SetHash` is called to sample a hash function using f_i = `fOptimizer(..)`. `BoundedSATCount(1, solver, assumptions` is invoked and if it returns a nonzero value, then 1 is pushed to `medianComputeList`, else a 0 is pushed.

Out of the T iterations, if more than $\frac{T}{2}$ of the `medianComputeList` is zero, then the outer loop is termniated and `solCount.finalcellCount` $= \lfloor 2^{i-1} \rfloor$ is returned as the final answer.

5.3 Remarks

For the original code, `val` $= 0$ has been set to raise error against non-existence of ProbMap_.txt files in the build directory

For the `AddHash` function, the iteration indices of the inner loop for setting literals was offset by 1, which has been fixed. The Makefile has been modified to incorporate GMP and MPFR

6 References

6.1 Papers

- `https://arxiv.org/pdf/1404.6682.pdf`
- `http://proceedings.mlr.press/v32/ermon14.pdf` - SPARSE-WISH algorithm

6.2 Additional Reading

Abstract

We have a high-dimensional discrete set $\chi = \{0,1\}^n$ and its underlying probability distribution. We embed the set onto a higher dimensional space and randomly project it onto a lower-dimensional space and leverage combinatorial optimization tools for sampling.

6.3 Brief introduction

We have a weight function, $w(x) : \mathbb{R} \to \mathbb{R}$, from which we calculate its probability by normalizing the weights, using the partition function Z. We might also consider factor graphs, which are useful for performing Variable Elimination and Belief Propagation in Bayesian Networks. Now, we want to (approximately) sample from $p(x)$.

- We generate p' from p, where the new weight function now takes only a discrete set of geometrically increasing weights, which calls for discretization into disjoint buckets, $B_i = \{x|w(x) \in (\frac{M}{r^i}, \frac{M}{r^{i+1}})\}$ while the last bucket (say l^{th}) bucket is $B_l = \{x|w(x) \in (0, \frac{M}{r^l})\}$.
- From p', a new probability distribution p'' is defined over a higher dimensional embedding of χ, from which sampling is performed.
- We indirectly sample from p, by uniformly sampling form p'' (i.e. project the embedding into lower-dimensional subspace, using universal hash functions.

So, this reduces the weighted sampling problem to that of solving the MAP query and a polynomial number of feasibility queries. We effectively want to reduce this weighted sampling problem to uniformly sampling from a higher-dimensional discrete set where χ is embedded.
We define the embedding $S(w,l,b)$ of χ in $\chi \times \{0,1\}^{(l-1)b}$ where w is the weight function, l is the no. of buckets, and $b = \log_2 \frac{r}{r-1}$. We include tuples of $\{x, y_1^1, y_1^2, ..., y_{l-1}^{b-1}, y_{l-1}^b\}$, where for each $i \in \{1,2,...,l-1\}, w(x) \leq \frac{M}{r^i}, \sum_{k=1}^b y_i^k \geq 1$. Let p'' denote a uniform probability distribution over $S(w,l,b)$.
We constrain the space using the family of universal hash functions, search for P "surviving" configurations, and if lesser than P survive, we use the statistical method of acceptance-rejection sampling, to choose one of them. Combinatorial optimization is used to choose the maximum weight M through MAP inference. P is a no. chosen since it is a base case, where we know how to produce uniform samples via enumeration.

7 Brief overview of PAWS algorithm

First, we want to compute k, the no. of constraints/factors(which encodes a randomly chosen hash function) to add. The hash function is similar to what we used in SPARSE-COUNT. We perform this for T trials, until we get lesser than P surviving configuration for more than half of those trials, or we have performed the experiment of T trials, n' times.

In the PAWS procedure, we perform a MAP inference to get the maximum weight. We also perform a higher-dimensional embedding of χ, increasing its dimension by $(l-1)b$, and the no. of constraints included in the randomly sampled hash function is $k + \alpha$. , where we uniformly sample Am and c where a hash function as we studied earlier, is $Ax + c(mod 2)$. We can then use the NP-oracle, to check all $\{x, y\}$ s.t. $h^i_{A,c}(x, y) = 0$. If the set is empty, or has more than P distinct "surviving configurations", then we return bottom. Now we want to uniformly choose one of the configurations by acceptance-rejection sampling using uniform sampling.