Report on Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls

mahibuet045

September 2018

Paper link: https://www.comp.nus.edu.sg/~meel/Papers/ijcai16_counting.pdf

Probabilistic inference is a technique which is growingly being used to justify reasoning about huge unknown data sets arising from practical applications. But, the exact probabilistic inference is almost impossible due to the hardness of analyzing and organizing data in high-dimensional spaces. So, experts have studied approximate techniques (e.g., Markov Chain Monte Carlo, variational approximations, interval propagation and randomized branching choices) to solve real-world instances of this problem.

A favourable alternative to probabilistic inference is to reduce the problem to discrete integration or constrained counting, in which we count the models of a given set of constraints. Although constrained counting is computationally hard, recent advances in hashing-based techniques for approximate counting have given strength to this approach. But early approaches were suffered from scalability problems. Finally, a scalable approximate counter with strict theoretical guarantees named **ApproxMC** was reported.

In subsequent work, this approach has been extended to finite domain discrete integration. There have been also several recent efforts to design efficient universal hash functions to compromise the gap between scalability and providing strict guarantees. But, all these approaches to model counting via hashing use a linear search to identify the right values of parameters for the hash functions. Consequently, the number of calls to the **SAT solver** increases proportionately with the number of variables n, in the input constraint. Unfortunately, **SAT solver** is the most expensive step in these algorithms. So, the main contribution of this paper is to drop **linear** SAT solver calls to **sublinear** (in n) calls while providing strong theoretical guarantees. It presents a new hashing-based approximate counting algorithm, called **ApproxMC2**, for CNF formulas, that reduces the number of SAT solver calls from linear in n to logarithmic in n and also provides strongly probably approximately correct (SPAC) guarantees. It calculates a model count within a prescribed tolerance

ϵ of the exact count, and with a prescribed confidence of at least $1 - \delta$. It also ensures that the approximate value of the model count matches the exact model count. Experimental results demonstrate that **ApproxMC2** outperforms **ApproxMC** by $1 - 2$ orders of magnitude in running time, given that the same family of hash functions are used.

There are several similarities and dissimilarities between **ApproxMC2** and **ApproxMC**. The main similaries are in **ApproxMC2** and **ApproxMC** function. **ApproxMC** uses 2-universal hash functions to randomly partition the solution space of the original formula into enough small cells. The sizes of randomly chosen cells are then determined using calls to a specialized SAT solver **CryptoMiniSAT** and the count is **nSols**. Then, this count is multiplied by total number of cell **nCells** to get the approximate model count (**nSols** $*$ **nCells**). For gaining the desired confidence of $1-\delta$, both **ApproxMC2** and **ApproxMC** invoke this subroutine repeatedly $\mathcal{O}(\log(1/\epsilon))$ times and collecting the **nSols** and **nCells** estimates in a list C. Finally, both algorithms compute the median of the estimates in C to obtain the desired estimate of model count.

The main dissimilarities of **ApproxMC2** and **ApproxMC** is the dependency injection in hash functions when searching for the "right" way of partitioning original search space within successive invocation of **ApproxMC2Core**. Due to the lack of dependence in **ApproxMCCore**, it starts from $m = 1$ and ends to $|S|$ for finding m^* such that number of solutions in a randomly chosen cell is lesser than $thresh$. So, **ApproxMCCore** makes $\mathcal{O}(thresh * |S|)$ SAT solvers calls. But dependence injection in **ApproxMC2Core** makes $\mathcal{O}(thresh * \log |S|)$ SAT solvers calls given that the improvement in the number of calls comes without pushing complication to the separate calls. The trick of reduction comes from the function **LogSATSearch** which uses a galloping search to zoom down to the right value of m. It was observed that repeated invocations of **LogSATSearch** with the same input formula F often output similar values of m. So, output value of previous invocation was also provided (**mPrev**) and tried to linearly search a small neighbourhood of **mPrev** before starting galloping search. After that, **LogSATSearch** works as a binary search algorithm bisecting the interval between $loIndex$ and $hiIndex$ expect that if the number of the solution of randomly chosen is lesser than the thresh then m will be doubled. It ensures that the search requires $\mathcal{O}(\log m^*)$ (instead of $\mathcal{O}(\log |S|)$) calls to BSAT where m^* is the value of m where search will stop. According to mathematical calculation, the upper bound of failure probability of **ApproxMC2** is 0.36.

Report on Balancing Scalability and Uniformity in SAT Witness Generator

Paper link: https://www.comp.nus.edu.sg/~meel/Papers/DAC2014.pdf

Constrained-random simulation (constrained-random verification, or CRV) is a leading approach used in the production for checking functional justification of complex digital designs. The preliminary goal of functional verification is to uncover design bugs before production. In CRV, the testing engineer declaratively lists a set of constraints on the possible values of circuit inputs. Typically, the source of these constraints is usage requirements, environmental constraints, constraints on operating conditions. A constraint solver is then used to list random values for the circuit inputs satisfying these constraints. Since it is not possible to estimate where bugs will be found, every solution to the set of constraints has an equal possibility to show an anomaly. So, the only solution is to sample the solution space uniformly or almost-uniformly at random. Solution to this problem has been mentioned in the literature. These solutions fall in one of two categories: those that achieve theoretical guarantees of uniformity but scale poorly in practice, and those that fit large problem instances with weak or no guarantees of uniformity. So, promise to guarantees of uniformity and scale to large instance was two conflicting goals. This paper describes an algorithm for generation solutions satisfying a set of Boolean constraints providing stronger guarantees on uniformity and higher scalability in practice.

Sampling techniques based on hashing helps to bridge between two conflicting goals. The core idea in hashing-based sampling is to use r-wise independent hash functions (for a suitable value of r) to randomly partition the space of witnesses into **small** cells (which can be enumerated) of roughly equal size, and then randomly pick a solution from a randomly chosen cell.

Firstly, **XORSample**‘ a hashing-based near-uniform generator of SAT witnesses was presented. The core of this generator was 3-wise independent linear hash functions. But to understand the guarantee of near-uniformity, their algorithm requires the user to provide **difficult-to-estimate** input parameters. In the question of scalability, **XORSample**‘ has been shown to scale to constraints involving a few thousand variables.

A new hashing-based SAT witness generator, called **UniWit** represented a slight but effective step towards achieving the conflicting goals of scalability and guarantees of uniformity. Similar to **XORSample**‘, the **UniWit** algorithm uses 3-wise independent linear hashing functions. But, the guarantee of near-uniformity of witnesses generated by **UniWit** does not depend on difficult-to-estimate input parameters. In the question of scalability, **UniWit** has been shown to scale to formulas with several thousand variables. The lower bound success probability of **UniWit** was 0.125.

A hashing-based algorithm called **PAWS** can sample from a distribution defined over a discrete set using a graphical model. The algorithm presented in

this paper has some similarities and dissimilarities with **PAWS**. But, **UniGen** scales to hundreds of thousands of variables besides preserves the theoretical guarantees.

While **UniGen** inherits some characteristics from earlier hashing-based algorithms such as **XORSample**$^{'}$, **UniWit** and **PAWS**, but there are major distinctness that allows **UniGen** to significantly outperform these earlier algorithms, both in terms of theoretical guarantees and measured performance. The family of hash function used in **UniGen** is 3-independent. Though the hash function is used in **XORSample**$^{'}$, **UniWit** and **PAWS** there are some difference in **UniGen** about its usages. The algorithms **XORSample**$^{'}$, **UniWit** and **PAWS** partition R_F by randomly choosing $h \in H_{xor}(|X|, m, 3)$ and $\alpha \in \{0,1\}^m$. Each conjunctive constraint of the form $(h(x_1, \ldots, x_{|X|})[i] \leftrightarrow \alpha[i])$ is an xor of a subset of variables of X and $\alpha[i]$, and is called an xor-clause. It is observable that the expected number of variables in each such xor-clause is approximately $|X|/2$. It is clear that the difficulty of checking satisfiability of a CNF formula with xor-clauses grows significantly with the the number of variables per xor-clause. As a result, It is therefore extremely difficult to scale **XORSample**$^{'}$, **UniWit** and **PAWS** when there are hundreds of thousands of variables. **UniGen** resolves this problem by independent support (often far smaller than $|X|$) of a boolean formula and clearly dependent support of a boolean formula can be uniquely determined by the values of variables (`sampling variable`) S of independent support. This implies that **UniGen** accepts a subset of the support of boolean formula as an additional input. Without loss of generality, let $S = \{x_1, \ldots x_{|S|}\}$, where $|S| \leq |X|$. The set R_F can now be partitioned by randomly choosing $h \in H_{xor}(|S|, m, 3)$ and $\alpha \in \{0,1\}^{|S|}$.If $|S| \leq |X|$, the expected number of variables per xor-clause is significantly reduced. This not only makes satisfiability checking easier but also allows scaling to much larger problem size. The problem to find out independent support of a boolean formula is easy to solve. The effectiveness of a hashing-based probabilistic generator depends on m used in the choice of the hash function family. In **UniGen**, an approximate model counter is first used to estimate $|R_F|$ within a specified tolerance and with a specified confidence. Estimation on $|R_F|$ helps to determine a small range of candidate values of m. **UniGen** has access to random binary number and two subroutine: 1. `BSAT(F, N)`: for every $N > 0$, returns $min(|R_F|, N)$ distinct witnesses of F and 2. It is used to calculate the number of the approximate model within a small cell. `ApproxMC`: an approximate model counter $ApproxModelCounter(F, \epsilon, 1-\delta)$. It is used to determine an estimation of m.

UniGen starts with calculating *pivot* and κ which is used to calculate *loThresh* and *hiThresh* that defines the expected size of a `small` cell. The estimate C (approx. count) is then used to determine a range of candidate values for m. The range of candidate values for m is justified and it is proved that the probability of m fallen in this range is at least 0.8. According to theoretical prove, if $\epsilon > 1.71$ then the success probability of **UniGen** is at least 0.62.

Report on Counting Answer Sets via Dynamic Programming

Paper link: https://arxiv.org/pdf/1612.07601.pdf

Day by day the effectiveness of efficient SAT solvers are increasing and gradually model counting problem for the propositional satisfiability problem (#SAT) has been getting avoidable attention. Because, counting the number of models of a propositional formula is an effective measurement and can be used in the areas of machine learning, probabilistic reasoning, statistics etc.

Similar efforts for ASP solvers have been also made. As a result, efficient ASP solvers are now available. Evaluation of any ASP program is usually a two-step process. Firstly, `grounder` instantiates the program which replaces all variables by domain constants, and then `solver` evaluates the resulting program and computes all the answer sets which are all possible solutions to the combinatorial. Due to this fact, the answer set counting problem (#ASP) has received less attention than the #SAT problem. However, enumerating all answer sets is computationally hard and is not really necessary for answer sets counting.

Answer set existence problem can be decided in linear time in input size besides the enumeration of answer sets can be calculated with linear delay. Tree decomposition technique exploits the structure of a given problem which accomplishes the goal of answer set calculation. Tree decomposition is the technique where any graph is re-arranged into a tree by combining cyclic parts of the graph within a single tree node. Tree decomposition is an effective technique to apply dynamic programming on any graph. The size of these tree nodes is bounded by a (small) constant. Because this constraint allows us that any problem can be evaluated efficiently by traversing the tree decomposition in a bottom-up manner.

This paper presents a dynamic programming-based answer set counting algorithm utilizing the structure of the given input ASP program bypassing the expensive enumeration of all answer sets. In this paper, it is described that how tree decomposition can be used for counting without answer set materialization. Also proposed the further adaptation of the algorithm depending on different graph representations of the program.

Traversal on tree decomposition should be started from the leaf nodes to the root node. At each node, a subproblem is solved which belongs to the part of the problem instance that is generated by the content of the bag of the current node. This results in a set of partial solutions which is propagated from the child nodes to the parent node. From these, the parent node then calculates the partial solutions from its child subproblems. Finally, we will reach the root node where there is a correspondence between the partial solutions of the root node and the solutions of the whole problem instance.

To build tree decompositions for ground answer set programs $\pi = (A, R)$, there are two types of graph representations studied. `Incidence graph:` The incidence graph $G_{inc}(\pi)$ of π is an undirected, bipartite graph $(A \cap R, E)$, where E contains an edge (a, r), iff atom a occurs in rule r of π. `primal graph:` $G_{prim}(\pi)$ of π is an undirected graph (A, E), where E contains an edge (a_i, a_j), iff there exists an $r \in R$, such that both a_i and a_j appear in r. A tree decomposition of such a graph representation of a program π is called a tree decomposition of π.

For normalized tree decompositions of programs, there six types of nodes: leaf (**LEAF**), join (**JOIN**), atom introduction (**AI**), atom removal (**AR**), rule introduction (**RI**), and rule removal (**RR**) node. The leaf node has no child node. Join node has two child nodes. Atom introduction node is a node whose bag contains one additional atom compared to the bag of its child node. Similarly, atom removal node is a node whose bag contains one less atom compared to the bag of its child node. Rule introduction and rule removal nodes are defined similarly to AI and AR nodes, where the difference between whose bags is an added or removed rule instead of an atom. Notice that primal graph tree decompositions can't contain **RI** and **RR** nodes. At each node t of the tree decomposition, the algorithm will compute a set τ of tuples that represent the partial solutions under the subtree. Given a node t and its two childrenś tuple are denoted by τ_1 and τ_2 respectively. obviously, the set τ is derived from the sets τ_1 and τ_2.

Counting procedure is propagated from the child node to the root node. Counting procedure for different types of nodes are as follows: for the leaf node, the count is 1. For the join node, the count is the product of both of its child nodes. For introduction and removal nodes, the count is the total number of surjective function mapping each tuple for the child node to a parent node.