Belief Propagation with Local Structure and Its Applications in Program Analysis

Yiqian Wu, Yifan Chen, Yingfei Xiong, Xin Zhang*
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
School of Computer Science, Peking University, Beijing, China
wuyiqian@pku.edu.cn, yf_chen@pku.edu.cn, xiongyf@pku.edu.cn, xin@pku.edu.cn

***Abstract*—In program analysis, there is an emerging trend to apply probabilistic reasoning. In general, these approaches build their models based on probabilistic graphical models because they can express local correlations through factors in a compositional manner, which is suitable for program analysis. These models commonly use the loopy belief propagation algorithm to infer the marginal probability distribution for efficiency. However, the efficiency of loopy belief propagation is still affected by large factors. To address this challenge, our insight is that we can exploit the local structure of probabilistic constraints to speed up the inference. To realize this idea, we use if-then rules to encode the factors with local structures and propose an efficient loopy belief propagation algorithm based on it. We also discuss the inference algorithm complexity and prove some applicable conditions of our approach. Our approach is evaluated on two existing program analysis works based on probabilistic graphical models. The results show that our approach can be 5.11 and 2.31 times faster than the original loopy belief propagation algorithm on average, respectively.**

***Index Terms*—Program analysis, Probabilistic graphical model, Belief propagation**

I. Introduction

There is an emerging trend in program analysis to incorporate probabilistic reasoning to express uncertainties and enable new use cases. For instance, approaches like BINGO [1] and DynaBoost [2] leverage Bayesian networks and user feedback to prioritize bug reports. Drake [3] introduces a probabilistic framework for static analysis of continuously evolving programs, while BayeSmith [4] automates the learning of probabilistic models for static analysis alarms. SmartFL [5] employs probabilistic constraints to model program semantics for fault localization. A common thread weaving through these advancements is the crucial role of *probabilistic constraints*, which encode uncertain relationships between program properties and may inspire new directions for many traditional program analysis problems.

In these applications, probabilistic constraints are typically represented using probabilistic graphical models [6] with discrete random variables. A probabilistic graphical model captures the dependencies between random variables through a graph structure, effectively encoding a multi-dimensional probability distribution in a factorized form. In program analysis, random variables often signify whether certain program properties hold, and analysis rules naturally translate into probabilistic constraints defining correlations between these variables. The inherent suitability of graphical models lies in their ability to describe these local correlations through factors, unify all constraints into a joint probability distribution, and perform marginal probability inference to estimate the likelihood of program facts.

Despite the expressive power of probabilistic graphical models, a significant bottleneck hindering the widespread adoption of this paradigm is the computational cost of *probabilistic inference*. Exact inference on these models is a counting problem and thus does not scale to the large models generated from real-world program analysis problems. Consequently, most existing approaches employing probabilistic reasoning in program analysis rely on approximate inference algorithms, most notably, loopy belief propagation(LoopyBP) [7]. LoopyBP is an iterative message-passing algorithm that approximates the marginal probabilities of each random variable by exchanging messages based on the probabilities of its adjacent variables. While the number of message updates can be bounded polynomially with respect to the graph size to achieve feasible approximations, the cost of each individual update remains substantial. Specifically, the algorithm requires enumerating all possible assignments for variables involved in a probabilistic constraint. This enumeration is exponential in the number of variables within a constraint. In program analysis, it is common to encounter large constraints involving multiple variables, leading to considerable inefficiency in LoopyBP.

To overcome this challenge, our key insight is that while a direct enumeration of variable assignments is exponential, the probabilistic constraints arising in program analysis often possess inherent *local structures* that can be exploited for more efficient updates. The concept of exploiting local structure has been explored in exact inference algorithms for probabilistic graphical models [8]. However, these existing techniques cannot be directly applied to LoopyBP because they are tightly coupled with the specific operations of exact inference algorithms, such as transforming and simplifying constraint representations (e.g., using Algebraic Decision Diagrams (ADDs) [9] in variable elimination [10], [11]). LoopyBP, in contrast, does not manipulate constraint representations but rather queries probabilities and performs summations and products over message terms. Therefore,

*Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00132

simplifying representations in isolation offers no direct benefit.

To address this challenge, our key observation is that while LoopyBP cannot directly benefit from simplifying the representations, local structures can be exploited to simplify the computation on a constraint by grouping different variable assignments of the same probability together. Concretely, probability values that are shared by multiple assignments can be extracted as common terms in message updating. For instance, we find that there is a widely used probabilistic variant of *Horn* constraints in program analysis, i.e., constraints in the form of $p : x_1 \wedge ... \wedge x_n \rightarrow y$, which indicates that if the condition evaluates to true, Boolean variable y has a probability of p to hold, otherwise, y must be false. An optimized LoopyBP algorithm can combine the evaluation of $x_1 \wedge ... \wedge x_n$ rather than enumerating all 2^n assignments of x_i, thereby significantly reducing complexity.

To realize this idea, we represent probabilistic constraints using the "If-then rules" [12] (also known as "rule CPDs" [13]) encoding to reflect local structures and propose an improved LoopyBP using it. We have proved several theorems that if the complexity of the encoding satisfies a given requirement, the improved algorithm has a guarantee in its time complexity. Our approach is general and can be applied to various local structures. As an instance, we show that our approach enables efficient inference for Horn constraints with linear time complexity, which often appears in program analysis applications. We have evaluated our approach on a fault localization approach based on probabilistic constraints [5] and a Bayesian program analysis approach [1] to demonstrate the effectiveness of our approach in practical program analysis problems. We compared our approach with the original LoopyBP algorithm. The experimental results show that our approach can be 5.11 and 2.31 times faster on average, respectively. This demonstrates that our approach is valuable and necessary for many practical program analysis works, and it can be the basis of more future program analysis works with probabilistic inference.

In summary, this paper makes the following contributions:

- An improved LoopyBP algorithm specifically designed to exploit local structures within probabilistic constraints, significantly enhancing inference efficiency.
- A theoretical analysis providing a rigorous proof of the computational complexity and applicable conditions for our improved inference algorithm.
- A comprehensive experimental evaluation on real-world fault localization and Bayesian program analysis problems, demonstrating the practical effectiveness and efficiency gains of our approach.

The rest of the paper is organized as follows. Section II discusses related research. Section III motivates our approach with an illustrative example. Section IV presents a basic mathematical background on factor graphs and loopy belief propagation. Section V describes our approach in detail. Section VI shows the evaluation and the experiment results. Section VII concludes the paper.

II. RELATED WORK

The modeling and inference of probabilistic graphical models have been an area of intensive research over the past few decades, with comprehensive surveys available in various textbooks [6], [14], [15]. This section contextualizes our work by comparing it with other inference algorithms, discussing its connection to probabilistic programming, and reviewing probabilistic constraint encodings.

A. Exact Inference Algorithms

The most common exact inference algorithms are variable elimination [10], [11] and junction tree algorithm [16], [17]. These algorithms offer precise solutions but come with a steep cost: their complexity is exponential in the treewidth of the graphical model [18]. Given that many practical graphical models in program analysis exhibit high treewidth (due to complex interdependencies rather than tree-like structures), these algorithms generally do not scale.

While advancements like Ace [19] and techniques leveraging conditioning with local structure [8] exploit intrinsic local structures to optimize exact inference, they fundamentally rely on operations that modify or decompose the network structure (e.g., CNF simplification, d-DNNF compilation, network decomposition). These operations are where local structures yield computational gains. Our key distinction is that LoopyBP operates differently; it does not modify the network structure but rather iteratively queries probabilistic constraints and computes message products at individual nodes. This difference means existing exact inference optimizations based on structural modifications cannot be directly applied to LoopyBP. Our work introduces a novel approach to leverage local structures within the LoopyBP message-passing mechanism itself, a gap not addressed by prior exact inference methods.

Furthermore, although these algorithms can avoid the influence of high treewidth caused by large factors with local structure, they are still affected by cycle structures in the graph, which also result in high treewidth. This property makes LoopyBP exploiting local structures irreplaceable in many practical scenarios. Our experiments in Section VI confirm that many large-scale probabilistic graphs from program analysis, while benefiting from local structure, still cannot be handled by exact solvers like Ace or ProbLog [20] due to their inherent cyclicity and high treewidth. This reinforces why program analysis applications frequently turn to approximate methods like LoopyBP. Our approach is thus valuable and necessary as it specifically enhances LoopyBP, making it more efficient for the types of large, cyclic graphs prevalent in probabilistic program analysis.

B. Approximate Inference Algorithms

Loopy belief propagation (LoopyBP) [7] is a widely used approximate inference algorithm. It is also the core focus of this paper. As we discussed in Section IV, LoopyBP iteratively passes messages to eliminate the impact of high treewidth caused by cycle structures in the graph. However, the efficiency of LoopyBP is still affected by the size of individual

factors. To the best of our knowledge, our work is the first to systematically exploit local structure in LoopyBP.

Other approximate inference algorithms mainly include variational methods [21] and sampling-based methods [22], [23]. However, these methods are generally not suitable for high-dimensional discrete probability spaces, which is common in program analysis as the number of variables in our benchmark is usually between 10^4 to 10^5. Furthermore, these algorithms do not involve exploiting local structures to our knowledge.

C. Inference in Probabilistic Programming

Modern probabilistic programming languages are typically more expressive than probabilistic graphical models. Therefore, we can leverage inference engines of existing probabilistic programming languages to solve the inference problem of probabilistic graphical models. However, their inference algorithms are essentially generalized versions of the inference algorithms mentioned above for probabilistic graphical models and still suffer from efficiency issues with large models. These algorithms include algebraic inference algorithms [24]–[26], weighted model counting [20], [27], [28], sampling-based algorithms [29]–[31], variational inference [32], [33]. Moreover, their performance on probabilistic graphical models is often inferior to the algorithms that are designed specifically for probabilistic graphical models. For example, the empirical evaluation of Dice [28], a state-of-the-art discrete probabilistic programming language, shows that the Bayesian network solver Ace [19] performs better than Dice on large Bayesian networks. On the other hand, a large number of probabilistic programming languages perform inference by converting the program into a probabilistic graphical model [34], [35]. Our approach can directly serve as an efficient approximate inference engine for these languages.

D. Encoding Frameworks for Probabilistic Factors

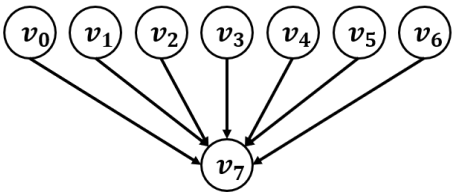
The choice of encoding framework for probabilistic factors is fundamental, influencing both input scale and algorithm design. Most techniques without local structures use the default tabular encoding framework directly, which lists all function values, leading to exponential storage and computational complexity for large factors. In our approach, we mainly discuss our algorithm based on If-then rules [12] (also known as rule CPDs [13]), an encoding specifically designed to capture local structures efficiently. Other notable frameworks include decision trees [12] (tree CPDs [13]) and their generalization, algebraic decision diagrams (ADDs) [9], which use directed acyclic graph structures to avoid redundancy. Extending our methodology to these alternative structured encodings represents a promising direction for future work.

III. Overview

In this section, we informally introduce our approach through an example from SmartFL [5], a fault localization technique that uses probabilistic constraints to model program semantics and performs probabilistic inference to identify likely incorrect program statements.

```
area.setRect(area.getX() + l, area.getY() + t,
    w - l - r, h - t - b);
```

(a) The corresponding statement of the factor



(b) A factor from the statement

Figure 1: A Motivating Example for Loopy Belief Propagation without Local Structure

A. The Efficiency Bottleneck of LoopyBP

SmartFL employs LoopyBP, a message-passing algorithm, due to its scalability for the enormous models extracted from real-world programs. However, even with LoopyBP's efficiency, some applications within SmartFL can still take significant computation time. This primarily stems from the inefficiency of computing messages sent between variables within the same constraint. For simplicity, we will not describe the full LoopyBP algorithm but focus on calculating messages that are related to one particular probabilistic constraint.

In probabilistic graphical models, a probabilistic constraint is represented as a probabilistic factor. Figure 1 shows an example factor from fault localization for program "Chart-15" (from the Defects4j dataset [36]). SmartFL tracks test execution traces through instrumentation and parses them to capture dynamic dependencies, converting them into factor graphs. One execution of the statement in Figure 1(a) is converted into the factor in Figure 1(b). In SmartFL, each random variable represents the correctness of a statement or a runtime variable value. A written variable's correctness depends on its associated statement, the read variable values, and the control predicate values. In Figure 1, node "v_0" denotes the statement's correctness, "v_1" denotes the correctness of the variable `area`, and "v_2" to "v_5" represent four calculated parameters'. Node "v_6" indicates the correctness of the object pointed to by `area` (distinct from v_1, which refers to the address). Node "v_7" indicates the correctness of the object pointed by `area` after the function call. All random variables here follow a Bernoulli distribution. Intuitively, if the statement and all variables contributing to a computation are correct, the written variable should also be correct. Otherwise, there is a high probability the written variable is incorrect. SmartFL assigns the following conditional probability to the factor:

$$p(v_7 = 1 \mid v_0 \wedge \cdots \wedge v_6) = \begin{cases} 1, v_0 \wedge \cdots \wedge v_6 = 1 \\ 0.01, v_0 \wedge \cdots \wedge v_6 = 0 \end{cases}.$$

The corresponding factor function can be written as:

$$f_a(v_0,\ldots,v_7)=\begin{cases}1, v_7=1, v_0\wedge\cdots\wedge v_6=1\\ 0, v_7=0, v_0\wedge\cdots\wedge v_6=1\\ 0.01, v_7=1, v_0\wedge\cdots\wedge v_6=0\\ 0.99, v_7=0, v_0\wedge\cdots\wedge v_6=0\end{cases}$$

SmartFL applies LoopyBP to calculate marginal probability distributions of individual statements to rank them according to their likelihood of being incorrect. During LoopyBP, each factor receives messages from incident variables and sends messages back. A message is a real-valued function mapping a variable's domain value to a real value. For example, let us set $V=\{v_0,v_1,v_2,v_3,v_4,v_5,v_6\}$, the message from the factor a to the random variable v_7 be $\mu_{a\to v_7}$, then $\mu_{a\to v_7}(1)$ can be computed as:

$$\mu_{a\to v_7}(1)=\sum_V\left(f_a(V,1)\prod_{v^*\in V}\mu_{v^*\to a}(x_{v^*})\right)$$

The right side means enumerating each case in V and summing the results. For example, the first case should be $\{v_i=0,\forall v_i\in V\}$ and the term summed is $f_a(0,\ldots,0,1)\cdot\mu_{v_0\to a}(0)\ldots\mu_{v_6\to a}(0)$. There will be 2^7 terms in total.

The LoopyBP algorithm traditionally represents factor functions in a tabular way, which explicitly lists function values for all variable assignments. *Crucially, it must enumerate this table during message calculation.* In this example, the calculation requires enumerating all cases in set V, a total of 2^7 terms, and each requires 7 multiplications. For factors with more variables, this leads to an exponential time complexity of $O(n2^n)$, where n is the number of incident random variables. This is the main challenge faced by current LoopyBP.

B. Our Approach

While more memory-efficient factor representations exist, they do not inherently reduce time complexity if the algorithm still enumerates all variable assignments. *Our key idea is to exploit local structures to group assignments of the same factor function value. By extracting these as common terms, we can reduce redundant computations.*

To achieve this, we first re-encode the factor instead of enumerating every case of a tabular factor encoding. For the example factor, we introduce four condition functions to represent the four distinct cases. For a single random variable x, we introduce an indicator function $g_c(x)$, which equals 1 if $x=c$ and 0 otherwise. Then, we compose these into condition functions: $G_1=g_1(v_7)\prod_{v^*\in V}g_1(v^*)$, $G_2=g_0(v_7)\prod_{v^*\in V}g_1(v^*)$, $G_3=g_1(v_7)$, and $G_4=g_0(v_7)$. For simplicity, we omit the domain of each indicator function and factor function, as G_i means $G_i(V,v_7)$ and f_a means $f_a(V,v_7)$. The factor function can now be written as:

$$f_a=\begin{cases}1, G_1=1\\ 0, G_2=1\\ 0.01, G_3(1-G_1)=1\\ 0.99, G_4(1-G_2)=1\end{cases}$$

Since these conditions are mutually exclusive and exhaustive, we can transform f_a as a sum of products:

$$f_a=1\cdot G_1+0\cdot G_2+0.01\cdot G_3(1-G_1)+0.99\cdot G_4(1-G_2)$$

Because only one of the four sets of conditions is 1 and the rest are 0 for any input, we can add these terms to get the factor function. After expanding each condition function, the factor function simplifies to:

$$\begin{aligned}f_a=0.99\cdot g_1(v_7)\prod_{v^*\in V}g_1(v^*)-0.99\cdot g_0(v_7)\prod_{v^*\in V}g_1(v^*)\\ +0.01\cdot g_1(v_7)+0.99\cdot g_0(v_7)\end{aligned}$$

Next, we modify the message-updating algorithm to exploit this local structure. After substituting f_a into $\mu_{a\to V_7}(1)$, the calculation can be processed separately as:

$$\begin{aligned}\mu_{a\to v_7}(1)=&\sum_V\left(0.99*g_1(1)\prod_{v^*\in V}g_1(v^*)\prod_{v^*\in V}\mu_{v^*\to a}(x_{v^*})\right)\\ &-\sum_V\left(0.99*g_0(1)\prod_{v^*\in V}g_1(v^*)\prod_{v^*\in V}\mu_{v^*\to a}(x_{v^*})\right)\\ &+\sum_V\left(0.01*g_1(1)\prod_{v^*\in V}\mu_{v^*\to a}(x_{v^*})\right)\\ &+\sum_V\left(0.99*g_0(1)\prod_{v^*\in V}\mu_{v^*\to a}(x_{v^*})\right)\end{aligned}$$

For the first term, $g_1(1)$ is 1 and $\prod_{v^*\in V}g_1(v^*)$ is non-zero only when all $v^*=1$, thus the sum becomes $0.99\cdot\prod_{v^*\in V}\mu_{v^*\to a}(1)$. The second term is 0 because $g_0(1)=0$. Similarly, the last term is 0. For the third term, since $0.01*g_1(1)=0.01$ is a constant, the result is $0.01*\prod_{v^*\in V}(\mu_{v^*\to a}(1)+\mu_{v^*\to a}(0))$. The equation simplifies to:

$$\mu_{a\to v_7}(1)=0.99\cdot\prod_{v^*\in V}\mu_{v^*\to a}(1)+0.01\cdot\prod_{v^*\in V}(\mu_{v^*\to a}(1)+\mu_{v^*\to a}(0))$$

This new calculation process requires only 2×7 multiplications, achieving a linear time complexity of $O(n)$ with respect to the number of random variables n, significantly improving upon the original exponential complexity. By reorganizing computation to exploit local structures, we achieve a linear-time calculation. The detailed algorithm is presented in Section V.

IV. Background

Before introducing our approach formally, we describe background information about factor graphs [37] and LoopyBP [7] first, as the general probability inference process in program analysis is usually transformed into LoopyBP based on factor graphs and our approach aims to optimize this computation process.

A. Factor Graphs

A factor graph [37] is a bipartite graphical model that visually represents the factorization of a probability distribution over a set of random variables. It consists of two types of nodes: variable vertices ($V = \{v_1, \ldots, v_n\}$), each representing a random variable x_i, and factor vertices ($A = \{a_1, \ldots, a_m\}$), each representing a multivariate function f_j. Edges connect variable vertices to factor vertices, indicating which variables are part of a factor's domain. The joint probability distribution is defined as the product of all factor functions, normalized to sum to one:

$$p(x_1, \ldots, x_n) = \frac{\prod_{j=1}^{m} f_j(X_j)}{\sum_{x_1,\ldots,x_n} \prod_{j=1}^{m} f_j(X_j)}$$

where X_j is the set of random variables connected to factor a_j. Factor graphs are a general probabilistic graphical model representation, allowing other models like Bayesian networks to be directly converted. Defining a probability distribution using a factor graph can factorize the joint probability distribution into a product of factors, which enables efficient inference algorithms, particularly approximate ones like LoopyBP.

B. LoopyBP

LoopyBP is a multi-round iterative message-passing algorithm. It iteratively updates messages that go between a factor vertex and a variable vertex that are incident, which are in turn used to calculate the marginal probabilities in the end. Each message is a real-valued function that maps the domain of the associated random variable (the set of values that can be taken by the random variable) to a real value. A message $\mu_{v\to a}(x_v)$ from variable v to factor a is the product of messages received by v from all other incident factors ($a^* \in I(v) \setminus \{a\}$) in the previous iteration:

$$\mu_{v\to a}(x_v) = \prod_{a^* \in I(v)\setminus\{a\}} \mu_{a^*\to v}(x_v)$$

where $I(v)$ denotes the set of incident factors of v.

Conversely, a message $\mu_{a\to v}(x_v)$ from factor a to variable v is computed using the factor function $f_a(\mathbf{x}_a)$ and messages from all other incident variables in the previous iteration. This involves summing over all possible assignments of variables in X_a under the condition that v is fixed to x_v:

$$\mu_{a\to v}(x_v) = \sum_{\mathbf{x}_a \setminus x_v} \left(f_a(\mathbf{x}_a) \prod_{v^* \in I(a)\setminus\{v\}} \mu_{v^*\to a}(x_{v^*}) \right) \quad (1)$$

where $I(a)$ denotes the set of incident variables of a, $\mathbf{x}_a$ denotes the values of the set of random variables that are incident to a (including x_v), and the summation $\sum_{\mathbf{x}_a \setminus x_v}$ means to sum over all possible values in $\mathbf{x}_a$ while the value of v is fixed to x_v.

The algorithm iterates until messages converge or the iteration exceeds a certain limit. Finally, the marginal probability of each variable $p(x_v)$ is computed as the normalized product of all incoming messages from its incident factors:

$$p(x_v) = \frac{\prod_{a\in I(v)} \mu_{a\to v}(x_v)}{\sum_{x_v} \prod_{a\in I(v)} \mu_{a\to v}(x_v)}$$

V. Our Approach

This section details our approach. In Section V-A, we introduce an if-then-else encoding framework for probabilistic factors and automatically convert a factor into an algebraic expression. Section V-B describes the optimization process for this encoding, leading to a simplified algebraic expression. Section V-C then explains how to perform LoopyBP using this simplified algebraic factor. Section V-D analyzes the time complexity of our approach and presents related theorems.

A. Encoding Factor Functions

For a given factor, let n be the number of associated random variables, denoted $x_1, \ldots, x_n$. Each node v in the factor can be indexed in $1, \ldots, n$. We use $\mathbf{x} = \{x_1, \ldots, x_n\}$ for the set of all variable assignments. For simplicity, we assume all variables share the same domain size r, where values are indexed $1, \ldots, r$. We treat r as a small constant since discrete random variables in program analysis typically have limited value ranges. While our current application uses Bernoulli variables ($r = 2$), this generalized formulation highlights our approach's extensibility to multi-valued variables without fundamental changes. Additionally, we use m to denote the number of if-conditions encoded in our framework, indexed $1, \ldots, m$, and represented as $C_1, \ldots, C_m$.

We encode a probabilistic factor $f(\mathbf{x})$ using an if-then-else structure with the following syntax:

$$\begin{aligned} S &\to \textbf{if } C \textbf{ then } A \textbf{ else } S \mid A \\ C &\to \mathrm{B} \textbf{ and } \mathrm{B} \mid \mathrm{B} \\ \mathrm{B} &\to x = c \\ A &\to \textbf{return } p \end{aligned}$$

Here, x is a random variable from $\mathbf{x}$, c is an integer value from $\{1, \ldots, r\}$, and p is a non-negative real factor value. B is an "atomic condition" (e.g., $x = c$), and C is a "composite condition" for an if-branch. We enforce that all x within the same condition C must be distinct, as each condition typically constrains only a single variable assignment.

To formally express these conditional statements, we introduce several functions. For each atomic condition "$x = c$", we define an indicator function $g_c(x)$: $g_c(x) = 1$ if $x = c$, and $g_c(x) = 0$ otherwise. For a composite condition "$x = c$ **and** $\ldots$ ", we define a condition function $G(\mathbf{x}) = \prod g_c(x)$ as the product of all indicator functions of "$x = c$" in the condition. $G(\mathbf{x}) = 1$ if all atomic conditions are met, and 0 otherwise. For simplicity, we denote G_i as $G_i(\mathbf{x})$ and f as $f(\mathbf{x})$.

Given m encoded conditions $C_1, \ldots, C_m$, each with a condition function G_i and corresponding factor value p_i, and a default factor value p_d (when no conditions are met), we can transform f into the following algebraic form:

$$f = p_d + \sum_{i=1}^{m} (p_i - p_d) G_i \prod_{j=1}^{i-1} (1 - G_j) \quad (2)$$

This formulation correctly captures the if-then-else logic: if the i-th branch is taken, it means $G_i = 1$ and $G_j = 0$ for all $j < i$, so f correctly evaluates to p_i. If no conditions are met, $f = p_d$. To simplify (2) and perform subsequent calculations, we need to expand these multiplicative $(1 - G_i)$ terms as:

$$\prod_{1\le j<i}(1-G_j) = 1 - \sum_{1\le j<i} G_j + \sum_{1\le j<k<i} G_jG_k - \sum_{1\le j<k<l<i} G_jG_kG_l + \cdots + (-1)^{i-1}G_1\ldots G_{i-1}$$

Substituting this back into (2), f can be expressed as:

$$f = p_d + \sum_{1\le i\le m}(p_i - p_d)G_i - \sum_{1\le i<j\le m}(p_j - p_d)G_iG_j + \cdots + (-1)^{m-1}(p_m - p_d)G_1G_2\ldots G_m \tag{3}$$

B. Encoding Optimization

Notice that in (3), there are 2^m terms on the right-hand, which can undermine the effectiveness of our approach. Considering an example factor function like:

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} p_a, & \text{only 1 } x_i \text{ is 1} \\ p_b, & \text{otherwise} \end{cases}$$

where $x_i \in \{0, 1\}$. This factor can be encoded with n conditions, one for each $x_i = 1$ while others are 0:

if $x_1 = 1$ **and** $x_2 = 0$ **and** $\ldots x_n = 0$ **then**
  return p_a
else if $x_1 = 0$ **and** $x_2 = 1$ **and** $\ldots x_n = 0$ **then**
  return p_a
  ...
else if $x_1 = 0$ **and** $x_2 = 0$ **and** $\ldots x_n = 1$ **then**
  return p_a
else
  return p_b

The corresponding condition functions are $G_i = g_1(x_i)\prod_{j\neq i} g_0(x_j)$ for $i = 1$ to n, with $p_i = p_a$ and default $p_d = p_b$. In this case, (3) would yield 2^n terms, leading to exponential complexity. However, we can find among these n conditions that no two conditions can be satisfied at the same time for any input $\mathbf{x}$ because there can only be one $x_i = 1$ to satisfy the conditions. Therefore we can deduce $i \neq j \Rightarrow G_iG_j = 0$, and then simplify (3) to $f = p_b + \sum_{1\le i\le m}(p_a - p_b)G_i$ with only $n + 1$ terms.

This example highlights how *mutual exclusivity* between conditions enables optimization. Specifically, if two conditions G_i and G_j contain conflicting atomic conditions (e.g., $g_{c_1}(x)$ in G_i and $g_{c_2}(x)$ in G_j for the same variable x where $c_1 \neq c_2$), then $G_iG_j \equiv 0$. We capture these relationships using an $m\times m$ lower triangular matrix $L = (l_{i,j})$. For $i \le j$, $l_{i,j} = 0$. For $i > j$, if G_i and G_j are mutually exclusive, $l_{i,j} = 0$, otherwise $l_{i,j} = 1$. This matrix forms the adjacency matrix for an *exclusive graph* EG, where each node represents a condition and each edge represents that the connected two conditions are non-exclusive. For each term in (3), if the set of conditions in the term forms a clique in EG, the term will be kept. Otherwise, we can remove this term from (3) because there are at least two conditions in the set with no edge between them in EG, which causes this term to be zero.

In this way, the optimization problem reduces to finding all cliques in EG. We use an algorithm [38] to list all maximal cliques in EG and take the union among the power sets of each maximal clique. The smallest clique is the empty set, representing the p_d term in (3). For any other clique $\{i_1, \ldots, i_o\}$ (with i_o being the maximum index), its corresponding term in (3) is $(-1)^{(o-1)}(p_{i_o} - p_d)G_{i_1}\ldots G_{i_o}$. Let l be the number of non-zero terms after this optimization. Each such term can be represented as t_k (denoting $t_k(\mathbf{x})$) for $k = 1\ldots l$. We can simplify (3) to (4) as there are only l instead of 2^m terms.

$$f = \sum_{k=1}^{l} t_k \tag{4}$$

C. Sending Messages

After encoding optimization, we obtain a simplified algebraic factor expression (4) that fully exploits local structures. We calculate each LoopyBP message based on this expression. Substituting $f(\mathbf{x})$ with (4) into the message update rule from (1), the message from factor a to node i becomes:

$$\mu_{a\to i}(x_i) = \sum_{k=1}^{l}\sum_{\mathbf{x}\setminus x_i}\left(t_k \prod_{j\in\{1,\ldots,n\}\setminus i}\mu_{j\to a}(x_j)\right) \tag{5}$$

where x_i is the fixed value for the target node i. This computation involves two main steps:

- Each inner summation $\sum_{\mathbf{x}\setminus x_i}\left(t_k\prod_{j\in\mathbf{x}\setminus i}\mu_{j\to a}(x_j)\right)$ computes a *sub-message* for the k-th term of f.
- The final message $\mu_{a\to i}(x_i)$ is the sum of all such sub-messages.

Therefore, the core of our algorithm is to efficiently calculate each sub-message and then sum up these sub-messages to the LoopyBP message. For each term $t = (-1)^{(o-1)}(p_{i_o} - p_d)G_{i_1}\ldots G_{i_o}$ in (4), we process it as follows:

- **Term Expansion**: Expand each condition function $G(\mathbf{x})$ into its product of indicator functions. For example, $G_{i_1}(\mathbf{x})\ldots G_{i_o}(\mathbf{x}) = (\prod g_c(x))\ldots(\prod g_c(x))$.
- **Term Representation**: Represent the expanded term compactly as a tuple $(p, \text{indicator}[n])$:
  - An array indicator$[n]$ where:

$$\text{indicator}[i] = \begin{cases} c, & \text{if } g_c(x_i) \text{ exists in the term} \\ -1, & \text{otherwise} \end{cases}$$

  - A coefficient $p = (-1)^{(o-1)}(p_{i_o} - p_d)$

To efficiently handle message computations, we represent all received messages to the factor using a two-dimensional array $\mu[n][r]$, where $\mu[i][j]$ stores the message $\mu_{a\to i}(j)$. Given the following inputs:

- Node count n and target node index i
- Fixed value x_i for the target node
- Term representation $(p, \text{indicator}[n])$

Algorithm 1 Sub-message Calculation for One Message

Input: n, i, x_i, p, indicator$[n]$, $\mu[n][r]$
Output: sub_message

```
1:  if indicator[i] ≠ x_i and indicator[i] ≠ −1 then
2:      sub_message ← 0
3:  else
4:      sub_message ← p
5:      for j ← 1, n do
6:          if j = i then
7:              continue
8:          c ← indicator[j]
9:          if c = −1 then
10:             sub_message ← sub_message × Σ_{k=1}^{r} μ[j][k]
11:         else
12:             sub_message ← sub_message × μ[j][c]
```

- All received messages $\mu[n][r]$

The sub-message can be computed using Algorithm 1. Lines 1-2 indicate that if the term's constraint on variable i conflicts with the fixed value x_i, the sub-message is zero. The algorithm then initializes the sub-message with the term's coefficient p. Lines 5-12 iterate through all nodes j except the target node i. If node j is not constrained by the term (indicator$[j] = -1$), the sub-message is multiplied by the sum of all incoming messages from j ($\sum_{k=1}^{r} \mu[j][k]$). If node j is constrained to a specific value c, the sub-message is multiplied by $\mu[j][c]$. Algorithm 1 has a time complexity of $O(n)$ (assuming constant r) due to a single loop with constant-time operations.

Next, we discuss an optimized algorithm for calculating multiple messages in LoopyBP. Considering the calculation process of every message sent by a factor in one iteration during LoopyBP, our approach can leverage shared calculations among sub-messages to reduce time complexity. For each term $(p, \text{indicator}[n])$ in (4), we use a two-dimensional array sub_message$[n][r]$ to represent all sub-messages from this term, where sub_message$[i][j]$ stores the sub-message contribution to $\mu_{a\to i}(j)$. Algorithm 2 shows the pseudocode. The core idea is that each sub-message is the sum-product of messages from all other nodes except itself. Thus, we can first compute a global sum-product mul of all received messages corresponding to the constraints of this term. A sub-message to node i is then obtained by dividing mul by the value from node i that contributed to mul. Notice that mul is inherently tied to the constraints of the term and remains invariant for each node i receiving the message, which allows us to share this sum-product across all sub-messages efficiently. In addition, special handling for $val = 0$ is crucial to prevent division by zero. Lines 7-10 show that if we first encounter a constrained node's value is zero, we store its index (zi) and constraint (zc), set a zero flag (zf), and skip multiplication to keep $mul \neq 0$. Subsequent zeros are not handled because after the first zero appears, the sub-messages for all other nodes become zero, and we only need to process the sub-message for node zi. The time complexity of Algorithm 2 is $O(n)$ (assuming constant r), as each node is processed once per loop with $O(1)$ operations. This shared calculation approach avoids the $O(n^2)$ complexity by calling Algorithm 1 n times.

Algorithm 2 Sub-message Calculation for All Messages

Input: n, p, indicator$[n]$, $\mu[n][r]$
Output: sub_message$[n][r]$

```
1:  mul ← p, zf ← 0, zi ← 0, zc ← 0
2:  for i ← 1, n do
3:      c ← indicator[i]
4:      if c = −1 then
5:          mul ← mul × Σ_{k=1}^{r} μ[i][k]
6:      else
7:          val ← μ[i][c]
8:          if val = 0 and zf = 0 then
9:              zf ← 1, zi ← i, zc ← c
10:             continue
11:         mul ← mul × val
12: for i ← 1, n do
13:     c ← indicator[i]
14:     for j ← 1, r do
15:         if i = zi and j = zc then
16:             sub_message[i][j] ← mul
17:         else if zf = 1 or c ≠ j then
18:             sub_message[i][j] ← 0
19:         else if c = −1 then
20:             sub_message[i][j] ← mul / Σ_{k=1}^{r} μ[i][k]
21:         else
22:             sub_message[i][j] ← mul / μ[i][j]
```

According to our approach's description, the transformations in each step are mathematically equivalent, so the calculated message values are mathematically the same as the traditional LoopyBP algorithm.

D. Discussion about Time Complexity

This section details the time complexity of our approach and proves key theorems guaranteeing its performance. We define n as the number of random variables, r as the constant value range of each variable, m as the number of encoding conditions, and l as the number of cliques after optimization. Our approach comprises five implementation steps:

1) **Parsing Input**: Linearly scanning m conditions, each with at most n constraints. Complexity: $O(mn)$.
2) **Constructing Adjacency Matrix L and Exclusive Graph EG**: Iterating over pairs of conditions (m^2 pairs), scanning up to n variables per pair to determine whether they are mutually exclusive. Complexity: $O(m^2 n)$.
3) **Finding Cliques in EG**: Worst-case complexity is $O(2^m)$ (iterating the power set). However, as shown later, this step typically does not become the bottleneck.
4) **Expanding Terms to indicator**$[n]$: For l terms, each with at most m condition functions, and each function having at most n constraints. Complexity: $O(lmn)$.
5) **Calculating and Combining Sub-messages**: For a single message, Algorithm 1 calculates one sub-message

in $O(n)$, leading to $O(ln)$ for l sub-messages. For all messages, Algorithm 2 still calculates all sub-messages for one term in $O(n)$, resulting in $O(ln)$ for l terms.

Since m and l depend on the factor's encoding, we now propose and prove some theorems as the applicable conditions of our encoding and complexity guarantees for our approach.

Theorem V.1. *If a factor can be encoded with m conditions and m is a constant, the total time complexity is $O(n)$.*

Proof. It is obvious that $l \leq 2^m$ because m is the number of nodes in EG. Therefore, m and l are constants, and all five steps can be processed in $O(n)$. □

Theorem V.1 shows that if a factor can be encoded with constant conditions, our approach can calculate LoopyBP messages in $O(n)$. Many common factors with local structure can be encoded by constant conditions in program analysis, such as **Horn constraints**, which makes this theorem a useful sufficient condition to ensure the complexity of our approach.

Theorem V.2. *If a factor can be encoded with m mutually exclusive conditions, the total time complexity is $O(mn)$.*

Proof. Mutual exclusivity implies $l = m + 1$ (each condition forms a clique of size 1, plus an empty set clique). Thus, steps 1 and 5 have $O(mn)$ complexity. Step 2 can be skipped as L is a zero matrix. Step 3 directly finds cliques in $O(m)$ since EG has no edges. Step 4 processes at most one condition function G_i per term, leading to $O(mn)$ complexity. Therefore, all steps can be processed in $O(mn)$. □

Theorem V.2 offers another sufficient condition for complexity assurance. For example, if a factor uses $m = O(n^k)$ mutually exclusive conditions, its complexity is $O(n^{k+1})$. The example in Section V-B falls under this theorem, yielding $O(n^2)$ complexity.

Theorem V.3. *For any probabilistic factor, the worst-case time complexity of our approach is $O(nr^n)$ by encoding the factor with r^n mutually exclusive conditions.*

Proof. For factors without any local structure, our encoding can degenerate into a tabular representation by enumerating all variable values. Such encoding uses $m = r^n$ mutually exclusive conditions, allowing direct application of Theorem V.2. □

Theorem V.3 confirms our approach is no worse than standard tabular LoopyBP if no local structure exists, though our primary focus is on exploiting such structure.

Theorem V.4. *For any factors and encoding conditions, if the number of terms after optimization is bounded by a polynomial in n, as $l = O(n^k)$ where k is a constant, the time complexity of our approach is $O(n^{3k+1})$.*

Proof. Since $m < l$ (each condition forms at least one clique of size 1), the complexities are: Step 1: $O(n^{k+1})$; Step 2: $O(n^{2k+1})$; Step 4: $O(n^{2k+1})$; Step 5: $O(n^{k+1})$. Let e be the number of edges in EG. Since $e < l$ (each edge corresponds to at least a clique of size 2), finding all maximal cliques (using [38]) takes $O(en) = O(n^{k+1})$ per clique. Because the number of maximal cliques is smaller than l, and the size of each power set of each maximal clique is smaller than l, the complexity of step 3 is $O(n^{3k+1})$. □

We classify factors meeting Theorem V.4 as optimizable factors by our approach, as it reduces exponential tabular complexity to polynomial time. Theorem V.4 ensures that for these factors, the optimization part of our approach will not lead to exponential time.

In summary, the above theorems guarantee the complexity of our approach under specific conditions. These theorems can also serve as a guide for encoding. For example, when dealing with the probabilistic model with Horn constraints, by encoding them using $m = 3$ conditions, the total time complexity should be $O(n)$ according to Theorem V.1, which will be discussed in the next section's experiments.

VI. EMPIRICAL EVALUATION

Probabilistic inference has been adapted in various usage scenarios of programming languages and software engineering research. Researchers have designed different probabilistic models [1]–[5] to encode the uncertainties in their tasks where logical constraints cannot be solved. The efficiency of probabilistic inference is critical for the performance of these methods. Crucially, many of these probabilistic models leverage inherent structures from underlying logic formulas such as CNF and Horn Clauses, making them suitable to be optimized by exploiting local structures.

Table I: Probabilistic Horn clauses

Inputs	Factor Value
$x_1 \wedge x_2 \wedge \cdots \wedge x_n = 1, y = 1$	p_1
$x_1 \wedge x_2 \wedge \cdots \wedge x_n = 1, y = 0$	p_2
$x_1 \wedge x_2 \wedge \cdots \wedge x_n = 0, y = 1$	p_3
$x_1 \wedge x_2 \wedge \cdots \wedge x_n = 0, y = 0$	p_4

Horn clauses are fundamental in logic programming, and their probabilistic variants are frequently used as probability constraints in program analysis. Generally, a Horn clause is an implication: $x_1 \wedge x_2 \wedge \cdots \wedge x_n \rightarrow y$, meaning if all x_i are true, then y is true. All variables are binary, taking values in $\{0, 1\}$. We define probabilistic Horn clauses as probabilistic constraints of the form presented in Table I, each characterized by four parameters (p_1, p_2, p_3, p_4).

We evaluate our approach using two applications that originally employed LoopyBP for probabilistic inference on probabilistic Horn clauses. SmartFL [5] locates program faults by building factor graphs from program semantics and inferring the probability of each statement being wrong. BINGO [1] resolves false alarms of program analysis by converting analysis rules into Bayesian networks and performing posterior

inference with user feedback interactively. For a fair comparison, we implement our approach in their artifact separately to replace the probabilistic inference part and compare it with their original LoopyBP implementation.

Additionally, we use Ace [19] and ProbLog [20] as baselines, representing state-of-the-art exact inference techniques. Ace is considered to be one of the SOTA Bayesian network solvers exploiting local structures, while ProbLog is a widely recognized probabilistic logic programming tool. We implement an adaptor to convert the probabilistic models generated by both SmartFL and BINGO to Ace networks and ProbLog programs. We use the default setting of Ace v3.0 and ProbLog 2.2.4 engine to perform inference on these converted networks and programs. Considering that the exact inference techniques they use cannot work for many large probabilistic models, we set a timeout of 10 minutes for each SmartFL case and 3 hours for each BINGO case.

A. RQ1: How effective is our approach on SmartFL?

Background. As detailed in Section III, SmartFL locates program faults by a probabilistic graphical model derived from program semantics. Each node represents the probability of a statement or a run-time variable value being correct, and the probability factors are encoded as probabilistic Horn clauses with parameters $(p_1 = 1.0, p_2 = 0.0, p_3 = q, p_4 = 1.0 - q)$, where $q = 0.01$ for statements like "+" and $q = 0.5$ for statements like "<". The inference time is hardly affected by parameter values, as multiplication and addition counts remain constant. Thus, we use SmartFL's default parameters. Theoretically, our approach calculates LoopyBP messages in $O(n)$ for each probabilistic constraint according to Theorem V.1.

Dataset. SmartFL's artifact was evaluated on the Defects4J benchmark [36]. We collect 198 valid factor graphs, as SmartFL fails to build the graph in some cases. The dataset averages 102k lines of code, with an average graph size of 258k variables and 812k edges. We compare SmartFL's original LoopyBP implementation (*orig*) against our approach (*opt*) through directly replacing the probabilistic inference component in SmartFL. We only compare the execution time of probabilistic inference. For Ace and ProbLog, we extract the probabilistic graph constructed by SmartFL and convert it into a Bayesian network input for Ace and a probabilistic logic program input for ProbLog.

Experiment Setup. We run each technique three times per case and report the average. All measurements are taken on an Ubuntu 18.04 machine with 2 Intel Xeon Gold 6320 CPUs and 512GB RAM. SmartFL (both *orig* and *opt*) is implemented in Java on JDK 1.8. Both algorithms are single-threaded, with message passing fixed at 100 iterations. We use the default setting of Ace v3.0 and ProbLog 2.2.4 engine. The timeout for Ace and ProbLog is 10 minutes(600 seconds).

Results. Figure 2 presents the SmartFL results. Each point represents a test program, with the x-axis showing the time of the original LoopyBP and the y-axis showing the time of our approach. Both axes are on a logarithmic scale

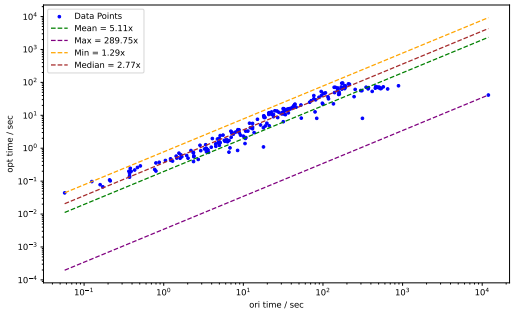


Figure 2: Comparison of Inference Time on SmartFL

due to wide variations. We focus on the acceleration rate $time_{orig}/time_{opt}$. From this figure, we can observe that our approach is faster than the original LoopyBP on all test programs. The least acceleration is **1.28x**, on the test program "Lang57", and the greatest acceleration is **289.75x**, on "Chart15". The average acceleration rate is **5.11x**, with a median of **2.77x**. These results confirm that our approach effectively optimizes the probabilistic inference in SmartFL over real-world benchmarks.

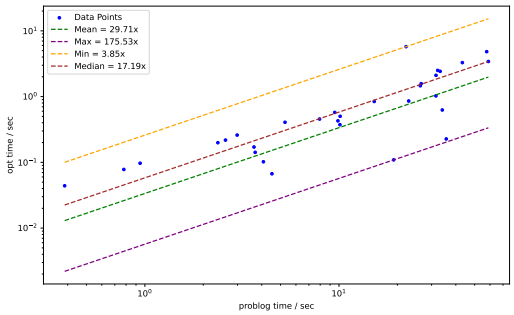


Figure 3: Comparison of Inference Time on ProbLog and optimized SmartFL

For ProbLog, we find it can only work on 31 cases, and cannot terminate within 10 minutes on the remaining 167 cases. Figure 3 displays ProbLog's performance for these 31 cases (x-axis: ProbLog execution time). These results indicate that ProbLog is generally unsuitable for SmartFL's scale. Similarly, Ace only succeeds on the 3 smallest cases, timing out on 195 larger ones within 10 minutes. Specifically, Ace fails on graphs with more than 700 factors. These results show Ace is also largely impractical for SmartFL.

Correctness. As shown in Section V-C, our approach is mathematically equivalent to the original LoopyBP algorithm, implying identical results. However, we observe minor differences in calculated marginal probabilities between the two versions of SmartFL. We speculate that the reason is the difference in floating-point errors under different implementations, which does not affect our conclusions. To quantify these differences, we use the relative error for calculated marginal probabilities. For instance, if a calculated marginal probability

is 0.5 in the original SmartFL and 0.51 in our optimized version, the relative error is 2%. We set different relative error thresholds to determine whether a calculated probability value has changed. For threshold 10^{-1}, no probability value has changed; For threshold 10^{-5}, the changed proportion is $5*10^{-4}$; For threshold 10^{-10}, the proportion is $5*10^{-3}$. It can be seen that 99.5% probability values have a relative error of less than 10^{-10}, which proves the correctness of our approach.

B. RQ2: How effective is our approach on BINGO?

Background. BINGO converts static analysis rules into Bayesian networks and performs probabilistic inference with user feedback as observations. In each iteration, BINGO requests user feedback on the highest likelihood alarm and then performs posterior inference with this feedback to recompute the confidence of remaining alarms. Efficient probabilistic inference is the key to faster true alarm identification.

BINGO's Bayesian network model uses two factor types. One is converted from logical-and clauses with probabilities, which can be presented in the form of probabilistic Horn clauses in Table I with $(p_1 = q, p_2 = 1.0 - q, p_3 = 0.0, p_4 = 1.0)$ (we set $q = 0.999$ according to BINGO). The other one is converted from logical-or clause like $y = x_1 \vee x_2 \vee \cdots \vee x_n$, which can be converted to its dual form $\neg y = \neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n$, and then presented in probabilistic Horn clause with parameters $(p_1 = 1.0, p_2 = 0.0, p_3 = 0.0, p_4 = 1.0)$. As with RQ1, Theorem V.1 indicates an $O(n)$ time complexity for each probabilistic constraint.

Dataset. We apply our approach to the data race analysis benchmark from BINGO, which comprises 8 Java programs from previous work [39], [40] and the DaCapo benchmark [41]. For each program, BINGO performs data race analysis, builds a Bayesian network, and runs an interactive loop with provided oracles. It iteratively examines ranked alarms until all true alarms are found or a time limit is reached. The average program size is 81k lines, with an average graph size of 62k variables and 159k edges. Similar to RQ1, we also convert the Bayesian network into the input form of Ace and ProbLog.

Experiment Setup. BINGO uses the LoopyBP algorithm implemented in LibDAI [42], a probabilistic inference library written in C++. We implement our approach within LibDAI as well and get it wrapped to be invoked by BINGO. Both implementations are single-threaded. The maximum message-passing iteration of each inference invocation is set to 1000, and the total timeout is set to 12 hours. Since the implementation of LoopyBP is randomized in LibDAI, we invoke BINGO with both algorithms simultaneously in pairs with the same random seed. We conduct three contrast experiments per test program. We use the default setting of Ace v3.0 and ProbLog 2.2.4 engine. The timeout for Ace and ProbLog is 3 hours(1800 seconds) for one iteration.

Results. Figure 4 (with detailed data in the Appendix) illustrates the results. Each line represents a BINGO resolution process, plotting accumulated probabilistic inference time (x-axis) against the number of true alarms found (y-axis). Solid

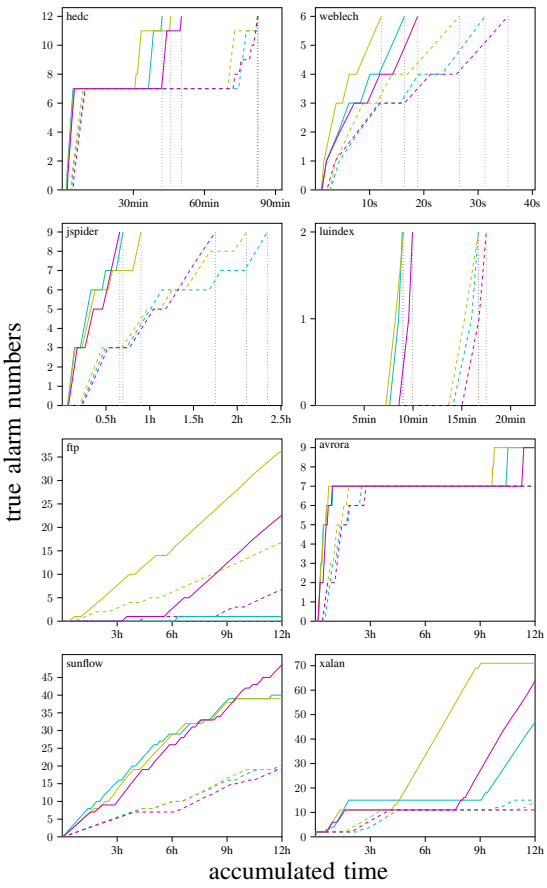


Figure 4: Comparison of Resolution Efficiency on BINGO

lines denote BINGO with our approach, while dashed lines represent BINGO with the original LoopyBP. For the first four programs (*hedc*, *weblech*, *jspider*, *xalan*), all true alarms are successfully filtered. On all programs that completed the process, BINGO enhanced with our approach runs faster than its counterpart with the original LoopyBP. The average acceleration rate of BINGO ($time_{orig}/time_{opt}$) on these four programs is **2.10x**. The other four programs (*ftp*, *avrora*, *sunflow*, *xalan*) time out before finding all true alarms. Even so, BINGO with our approach still discovers significantly more true alarms than the original BINGO. Across these four programs, our approach finds a total of 400 true alarms, which is **2.76x** more than the 131 true alarms found by the original BINGO. Both outcomes are directly attributable to the improved efficiency of LoopyBP. Data from the Appendix shows that our approach reduces the average time per single probabilistic inference invocation by $35\% \sim 70\%$, achieving an average speed-up of **2.31x**. This enhanced efficiency allows BINGO to either resolve all alarms faster or find more true alarms within a given time limit.

Regarding Ace and ProbLog, neither can complete prior inference for any iteration before the timeout, indicating they are entirely unsuitable for BINGO's models.

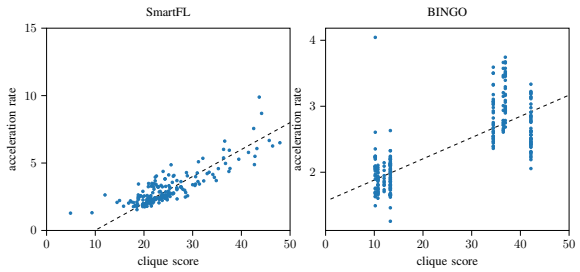


Figure 5: The relevance of acceleration rate and clique score

C. RQ3: What is the relationship between acceleration rate and the input model shape?

Back to Figure 2, we observe that larger test cases, which demand longer inference times, generally exhibit higher acceleration rates. According to Section V-D, our approach computes all factor messages in $O(n)$ time, where n is the number of nodes of the factor. In contrast, the original LoopyBP has an $O(nr^n)$ complexity. This suggests that factor graphs with a higher number or larger degree of factors should yield better acceleration. We build a *clique score* for each factor graph F to measure the number and sizes of high-degree nodes: $p_{clique}(F) = \Sigma_{f \in F}(|f| * r^{|f|})/\Sigma_{f \in F}|f|$, and on Figure 5. Figure 5 plots the clique scores against acceleration rates from our SmartFL and BINGO evaluations.

Linear regression analysis (dashed lines in Figure 5) reveals a positive correlation: higher clique scores correspond to greater acceleration rates. Notably, the "Chart-15" SmartFL case, with the highest clique score of 1460.2, also achieves the maximum acceleration of 289.75x (clipped from the graph due to its scale). This strong correlation indicates our approach delivers superior acceleration on models containing more and larger cliques.

VII. Conclusion

This paper proposes a novel approach to exploit local structure to speed up loopy belief propagation for program analysis. We use if-then rules to encode the factors with local structures and propose an efficient loopy belief propagation algorithm based on it. We also discussed the inference algorithm complexity and proved some theorems about time complexity guarantees. We experiment on two existing program analysis works based on loopy belief propagation, which shows the effectiveness of our approach.

To facilitate research, our implementation and data are available at **https://github.com/Neuromancer42/libdai**.

Acknowledgments

This research was partly supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902 and the National Natural Science Foundation of China under Grant No. 62172017 and Grant No. W2411051.

References

[1] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 722–735. [Online]. Available: https://doi.org/10.1145/3192366.3192417

[2] T. Chen, K. Heo, and M. Raghothaman, "Boosting static analysis accuracy with instrumented test executions," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1154–1165. [Online]. Available: https://doi.org/10.1145/3468264.3468626

[3] K. Heo, M. Raghothaman, X. Si, and M. Naik, "Continuously reasoning about programs using differential bayesian inference," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 561–575. [Online]. Available: https://doi.org/10.1145/3314221.3314616

[4] H. Kim, M. Raghothaman, and K. Heo, "Learning probabilistic models for static analysis alarms," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1282–1293. [Online]. Available: https://doi.org/10.1145/3510003.3510098

[5] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 958–969. [Online]. Available: https://doi.org/10.1145/3510003.3510073

[6] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886

[7] J. Pearl, *Probabilistic reasoning in intelligent systems - networks of plausible inference*, ser. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[8] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Inference with Local Structure, p. 313–339.

[9] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal methods in system design*, vol. 10, pp. 171–206, 1997.

[10] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Inference by Variable Elimination, p. 126–151.

[11] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009, ch. Exact Inference: Variable Elimination, p. 287–344.

[12] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Other representations of CPTs, p. 117–119.

[13] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009, ch. Context-Specific CPDs, p. 162–175.

[14] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. [Online]. Available: http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521884389

[15] R. E. Neapolitan *et al.*, *Learning bayesian networks*. Pearson Prentice Hall Upper Saddle River, 2004, vol. 38.

[16] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Inference by Factor Elimination, p. 152–177.

[17] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009, ch. Exact Inference: Clique Trees, p. 345–380.

[18] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Models for Graph Decomposition, p. 202–242.

[19] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *Artif. Intell.*, vol. 172, no. 6-7, pp. 772–799, 2008. [Online]. Available: https://doi.org/10.1016/j.artint.2007.11.002

[20] L. D. Raedt, A. Kimmig, and H. Toivonen, "Problog: A probabilistic prolog and its application in link discovery," in *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, M. M. Veloso, Ed., 2007, pp. 2462–2467. [Online]. Available: http://ijcai.org/Proceedings/07/Papers/396.pdf

[21] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, "An introduction to variational methods for graphical models," *Mach. Learn.*, vol. 37, no. 2, pp. 183–233, 1999. [Online]. Available: https://doi.org/10.1023/A:1007665907178

[22] A. Darwiche, *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009, ch. Approximate Inference by Stochastic Sampling, p. 378–416.

[23] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009, ch. Particle-Based Approximate Inference, p. 487–550.

[24] T. Gehr, S. Misailovic, and M. T. Vechev, "PSI: exact symbolic inference for probabilistic programs," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9779. Springer, 2016, pp. 62–83. [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_4

[25] P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov, "Probabilistic inference by program transformation in hakaru (system description)," in *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, ser. Lecture Notes in Computer Science, O. Kiselyov and A. King, Eds., vol. 9613. Springer, 2016, pp. 62–79. [Online]. Available: https://doi.org/10.1007/978-3-319-29604-3_5

[26] C. Dehnert, S. Junges, J. Katoen, and M. Volk, "A storm is coming: A modern probabilistic model checker," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10427. Springer, 2017, pp. 592–600. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_31

[27] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. D. Raedt, "Inference and learning in probabilistic logic programs using weighted boolean formulas," *Theory Pract. Log. Program.*, vol. 15, no. 3, pp. 358–401, 2015. [Online]. Available: https://doi.org/10.1017/S1471068414000076

[28] S. Holtzen, G. V. den Broeck, and T. D. Millstein, "Scaling exact inference for discrete probabilistic programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 140:1–140:31, 2020. [Online]. Available: https://doi.org/10.1145/3428208

[29] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of statistical software*, vol. 76, 2017.

[30] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. A. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," in *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, D. A. McAllester and P. Myllymäki, Eds. AUAI Press, 2008, pp. 220–229. [Online]. Available: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24

[31] V. K. Mansinghka, U. Schaechtle, S. Handa, A. Radul, Y. Chen, and M. C. Rinard, "Probabilistic programming with programmable inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 603–616. [Online]. Available: https://doi.org/10.1145/3192366.3192409

[32] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. A. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep universal probabilistic programming," *J. Mach. Learn. Res.*, vol. 20, pp. 28:1–28:6, 2019. [Online]. Available: http://jmlr.org/papers/v20/18-403.html

[33] A. Kucukelbir, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic variational inference in stan," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 568–576. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/352fe25daf686bdb4edca223c921acea-Abstract.html

[34] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain: a first-order type for uncertain data," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 51–66. [Online]. Available: https://doi.org/10.1145/2541940.2541958

[35] A. McCallum, K. Schultz, and S. Singh, "FACTORIE: probabilistic programming via imperatively defined factor graphs," in *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 1249–1257. [Online]. Available: https://proceedings.neurips.cc/paper/2009/hash/847cc55b7032108eee6dd897f3bca8a5-Abstract.html

[36] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[37] F. R. Kschischang, B. J. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, 2001. [Online]. Available: https://doi.org/10.1109/18.910572

[38] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM J. Comput.*, vol. 6, no. 3, pp. 505–517, 1977. [Online]. Available: https://doi.org/10.1137/0206036

[39] M. Eslamimehr and J. Palsberg, "Race directed scheduling of concurrent programs," *SIGPLAN Not.*, vol. 49, no. 8, p. 301–314, feb 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555263

[40] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133881

[41] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[42] J. M. Mooij, "libdai: A free and open source c++ library for discrete approximate inference in graphical models," *Journal of Machine Learning Research*, vol. 11, no. 74, pp. 2169–2173, 2010. [Online]. Available: http://jmlr.org/papers/v11/mooij10a.html