Infrence

In Probablistic Graphical Models

Addis Ababa University Institute of Technology
School of Information Technology and Engineering
Department of Artificial Intelligence

Submited BY:

Kereyu Banata

Mohammedfereje Suleyman

Rediet Girmay

Thomas Kitaba

Zehara Yassine

Submited To:

Dr Beakal Gizachew Assefa

Probabilistic Inference Methods

This Projectexplores key algorithms used for performing inference in probabilistic graphical models. Each group focuses on one major inference technique — from exact to approximate methods — highlighting their principles, mechanics, complexity, and theoretical underpinnings.

1: Variable Elimination (VE)

1.1 Topic Focus

Variable elimination is a general inference method for any factor graph it applies to both Bayesian Networks and Markov Random Fields (MRFs).

It works by systematically eliminating variables by summing them out or by mariginalizaing them.

1.2 Key Concepts & Sources

Core Idea: Variables are eliminated one at a time by summing over their possible values.

Often demonstrated through examples like computing the probability of getting a job by eliminating nuisance variables such as *Difficulty*, *Effort*, or *SATGrade*.

Conversion of a **Bayesian Network (BN)** to a **Markov Random Field (MRF)** through *moralization* is a relevant step for certain analyses.

1.3 Algorithm & Mechanics

Compute joint probability using factor products.

Sequentially **marginalize out non-query, non-evidence variables**.

When evidence is present, **clamp observed variables** and remove them from the elimination order.

In Simple words this is the exceution order

- Initialize with the initial set of factors.
- Choose the elimination order for the variables.
- Loop to join and sum out variables according to the order.

· Normalize the resulting factor to get the posterior probability.

1.4 Inference Types

Used for **marginal** and **conditional** probability computation.

Must be re-run independently for each marginal query.

1.5 Complexity & Limitations

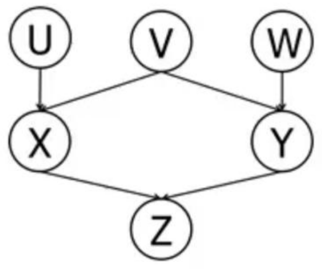
Time complexity: Exponential in the number of latent variables.

Depends heavily on the **elimination order** — large intermediate factors cause high cost.

No reuse of computation across multiple queries.

1.6 Practical Solution:

Assume we have this graph



Question: Query **P(U | +z)**

Step 1: Create factor

P(U) P(V) P(W) P(X | U,V) P(Y | V,W) P(+z | X,Y)

Step 2: Chose elimination ordering

W,Y,V,X

Step 3: loop to join and sumout Random variables.

3.1 Eliminate W :

Join P(W)) and P(Y | V,W) to form a factor over Y and V. then

Sum out W from this joint factor to get a new factor

f1(Y, V) = ∑ **P(W)P(Y | V,W)**

3.2 Eliminate Y :

Join **P(+z | X,Y)** and f1(Y, V) to form a factor over +Z and V then Sum out Y from this joint factor to get a new factor

f2(+Z, V, X) = ∑ **f1(Y, V)P(+z | X,Y)**

3.3 Eliminate V:

Join **P(V), P(X | U,V)** and **f2(+Z, V)** to form a factor over +Z , U and X then Sum out V from this joint factor to get a new factor

f3(+Z, U, X) = ∑ **f2(+Z, V, X)P(V)P(X | U,V)**

3.4 Eliminate X:

Join **f3(+Z, U, X)** to form a factor over +Z and U then Sum out X from this joint factor to get a new factor

f4(+Z, U) = ∑ **f3(+Z, U, X)**

Step 4: Normalize **f4(+Z, U)** to get a proper probablity distribution to get **P(U| +Z)**

2: Sum-Product Belief Propagation (Marginal Inference)

2.1 Introduction

Imagine a relay race where each runner passes local information to the next, building a complete picture of the race's outcome. Similarly, Sum-Product Belief Propagation computes marginal probabilities in tree-structured graphical models through localized message passing. This algorithm efficiently solves problems like estimating the probability of a specific event (e.g., a sunny Wednesday) by integrating over all possible states of other variables.

2.2 Core Concepts

2.2.1 Tree-Structured Graphs

Tree-structured graphs contain no cycles, enabling efficient recursive computation. Examples include:

- Markov chains: $X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_T$
- Rooted trees with hierarchical relationships

2.2.2 Factor Graphs

Factor graphs are bipartite graphs with:

- Variable nodes (circles): Represent random variables
- Factor nodes (squares): Represent interactions between variables A simple chain factor graph: $X_1 - f_1 - X_2 - f_2 - X_3$

2.3 Algorithm Foundation

2.3.1 Message Passing Intuition

The algorithm operates via localized message exchanges between nodes, where each message encapsulates probabilistic information from one part of the graph to another. This distributed approach transforms global inference into manageable local computations.

2.3.2 Formal Message Definitions

Variable-to-Factor Message:

$$\mu_{x\to f}(x) = \prod_{h\in ne(x)\setminus\{f\}} \mu_{h\to x}(x)$$

Factor-to-Variable Message:

$$\mu_{f\to x}(x) = \sum_{y} \left(\phi_f(x,y) \prod_{y\in ne(f)\setminus\{x\}} \mu_{y\to f}(y)\right)$$

2.3.3 Algorithm Steps

1. **Initialization**: Set all messages to 1 (neutral for multiplication).
2. **Forward Pass**: Propagate messages from leaves to root, computing variable-to-factor and factor-to-variable messages.
3. **Backward Pass**: Propagate messages from root to leaves using the same rules.
4. **Marginal Computation**:

$$P(x) \propto \prod_{f\in ne(x)} \mu_{f\to x}(x)$$

   Normalize by summing over states.

2.3.4 Numerical Considerations

In large graphs, multiplying small probabilities can cause numerical underflow. A log-space implementation (Log-Sum-Product) computes:

$$log\mu_{x\to f}(x) = \sum log\mu_{h\to x}(x)$$

This converts products to sums for improved numerical stability, similar to the Max-Sum variant used in MAP inference. The log-sum-exp trick further stabilizes computations by normalizing intermediate sums, ensuring robustness in large-scale models.

2.4 Algorithm Pseudocode

2.4.1 Sum-Product Belief Propagation Algorithm

Algorithm: Sum-Product Belief Propagation

```python function SumProductInference(Factors): # Initialize messages for each variable-factor pair for each variable x in Factors: for each neighboring factor f of x: message[x → f] = 1 # Neutral element for multiplication
```

```
# Forward pass: Compute messages from leaves to root
for each node in topological order:
    if node is variable x:
        for each neighboring factor f:
            # Variable-to-factor message: product of incoming messages
            message[x → f] = $\prod_{h \in neighbors(x) \setminus \{f\}}$ message[h → x]
    else if node is factor f:
        for each neighboring variable x:
            # Factor-to-variable message: sum over products
            message[f → x] = $\sum_{y} [\varphi_f(x,y) * \prod_{y \in neighbors(f) \setminus \{x\}}$ message[y → f]]

# Backward pass: Compute messages from root to leaves
for each node in reverse topological order:
    if node is variable x:
        for each neighboring factor f:
            message[x → f] = $\prod_{h \in neighbors(x) \setminus \{f\}}$ message[h → x]
    else if node is factor f:
        for each neighboring variable x:
            message[f → x] = $\sum_{y} [\varphi_f(x,y) * \prod_{y \in neighbors(f) \setminus \{x\}}$ message[y → f]]

# Compute marginal probabilities
marginals = {}
for each variable x:
    # Belief is product of all incoming messages
    belief(x) = $\prod_{f \in neighbors(x)}$ message[f → x]
    marginals[x] = normalize(belief(x))  # Ensure probabilities sum to 1

return marginals
```

2.5 Theoretical Analysis

2.5.1 Complexity Analysis

Method	Complexity	Example (T=30, K=2)
Brute Force	$O(K^T)$	$2^{30} \approx$ 1 billion
Sum-Product	$O(T K^2)$	30 * 4 = 120

2.5.2 Connection to Forward-Backward Algorithm

The Forward-Backward algorithm for Hidden Markov Models is a special case of Sum-Product Belief Propagation:

- **Forward messages** = Left-to-right factor-to-variable messages

- **Backward messages** = Right-to-left factor-to-variable messages

- **Marginal computation** = Product of forward and backward messages

2.6 Example Application: Weather Prediction

Consider a 3-day weather sequence with states {Sunny, Rainy} and transition probabilities:

- P(Sunny|Sunny) = 0.8, P(Rainy|Sunny) = 0.2
- P(Sunny|Rainy) = 0.3, P(Rainy|Rainy) = 0.7

Forward Pass Results:

- μ_{f→Tuesday}(Sunny) = 1.1, μ_{f→Tuesday}(Rainy) = 0.9
- μ_{f→Wednesday}(Sunny) = 1.15, μ_{f→Wednesday}(Rainy) = 0.85

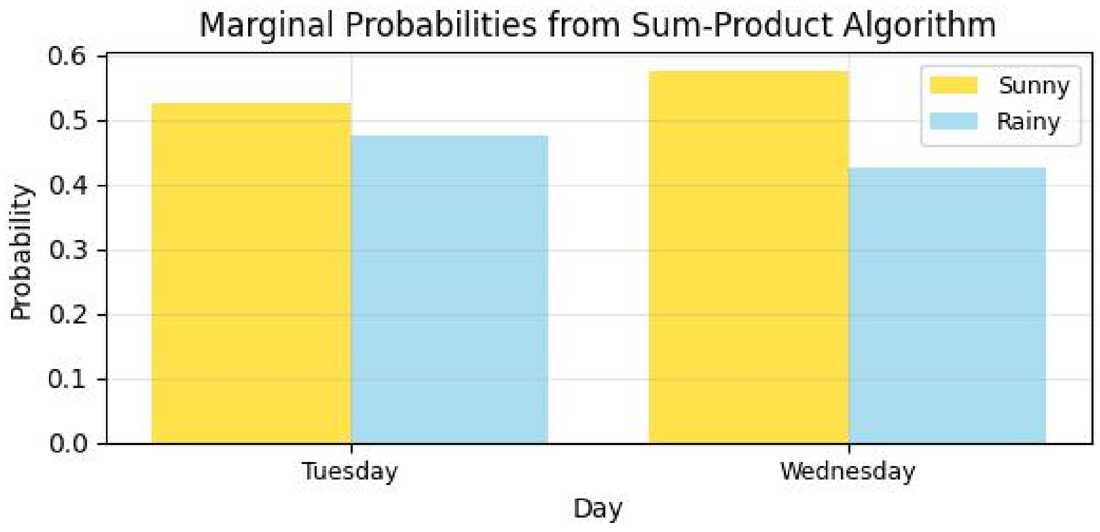
Backward Pass Results:

- All backward messages = 1.0 (uniform prior from future)

Final Marginal for Wednesday:

- P(Sunny) = 0.575, P(Rainy) = 0.425

Visualization of Results



2.7 Applications and Extensions

2.7.1 Real-World Applications

- **Speech Recognition**: Decodes phoneme probabilities in HMMs to transcribe audio signals into text.

- **Error-Correcting Codes**: Used in decoding algorithms for LDPC and turbo codes in communication systems.
- **Computer Vision**: Computes marginal distributions for pixel labels in image segmentation tasks.
- **Medical Diagnosis**: Reasons about disease probabilities given symptom observations in Bayesian networks.
- **Bioinformatics**: Infers gene regulatory networks from expression data using probabilistic models.

2.7.2 Beyond Trees: Loopy Belief Propagation

For graphs with cycles, the same message update rules can be applied iteratively (Loopy Belief Propagation). While not guaranteed to converge to exact marginals, it often provides good approximations in practice and has proven successful in applications like image processing and error-correcting codes. Convergence can be improved with techniques like damping or message scheduling, though results remain approximate.

2.8 Efficiency

Runs in **$O(TK^2)$** time for a chain with **T nodes** each having **K states**. Provides **exact inference** for tree-structured graphs.

2.9 Conclusion

The Sum-Product Belief Propagation algorithm provides an efficient, exact solution for marginal inference in tree-structured graphical models. By leveraging the tree structure and dynamic programming through message passing, it achieves exponential speedup over brute-force methods while maintaining theoretical guarantees.

Key Advantages:

- **Exact Inference**: Guaranteed correct marginals for tree-structured graphs.
- **Computational Efficiency**: Linear time complexity in graph size.
- **Distributed Computation**: Naturally parallelizable message updates.
- **Theoretical Foundation**: Generalizes well-known algorithms like Forward-Backward.

The algorithm's connection to fundamental special cases and its extensibility to approximate inference in loopy graphs demonstrate its central importance in probabilistic reasoning. This elegant framework transforms complex probability puzzles into a symphony of local computations, making probabilistic queries tractable for real-world applications.

3: Max-Product / Max-Sum Algorithm (MAP Inference)

3.1Max-Product / Max-Sum Algorithm (MAP Inference)

Maximum A Posteriori (MAP) inference is the task of finding the single most likely assignment or configuration for all the hidden variables in a model, given some observed evidence.

Formula: $x^*=\text{argmax}_x PX=x|E=e$

Since the evidence E=e is fixed, this is proportional to $\text{argmax}_x PX=x,E=e$, the joint probability.

The Max-Product (MAP) Inference algorithm is primarily concerned with Maximum A Posteriori (MAP) inference, which aims to identify the most probable configuration of a set of variables given observed data. Unlike marginalization, which sums over all possible configurations to compute probabilities, MAP inference seeks to maximize the joint posterior probability, denoted as maxxPx.

Key Concepts

Goal: The central objective of MAP inference is to maximize the joint posterior probability $P(x)$, which can be represented as:

$$P(x \mid y) \propto P(y \mid x) P(x)$$

where $P(y \mid x)$ is the likelihood of the observed data given the configuration x, and $P(x)$ is the prior probability of x.

Algorithm Transition

Transition from Sum-Product: The transition from the Sum-Product algorithm involves replacing the summation operations in message passing with maximization operations. This change enables the algorithm to focus on finding the configuration that yields the highest probability rather than calculating the overall probability of the observations.

The Viterbi Algorithm serves as a specific instance of Max-Product inference tailored for Hidden Markov Models (HMMs). In HMMs, the Viterbi Algorithm identifies the most likely sequence of hidden states (the configuration of variables) that results in the observed events.

Max-Product vs. Max-Sum

Max-Product: This variant operates directly in the probability space. It calculates the maximum product of probabilities, which can lead to numerical instability due to the multiplication of small probability values.

Max-Sum: To mitigate numerical issues, the Max-Sum algorithm employs the log-domain. Instead of calculating the product of probabilities directly, it computes the sum of their logarithms:

$$\max_x P(x) \dashrightarrow \max_x \sum \log(Px))$$

This approach provides greater numerical stability and is particularly useful in practical applications where probabilities can be exceedingly small.

Backtracking for Joint Maximizer

Once the forward maximization process is complete, a backward pass, akin to the Viterbi algorithm, is performed to reconstruct the optimal configuration of variables. This backtracking phase is crucial as it resolves any ties that may occur during the maximization process and ensures a consistent reconstruction of the MAP sequence. By tracing back through the computed messages, the algorithm can effectively determine the sequence of variables that corresponds to the maximum posterior probability.

This comprehensive structure of the Max-Product / Max-Sum algorithm not only highlights its theoretical underpinnings but also illuminates its practical applications in fields such as machine learning, computer vision, and natural language processing.

Algorithm

function MaxProductInference(Factors): // Initialize messages for each variable X in Factors: for each neighbor Y of X: message[X → Y] = 1 // Start with neutral element for maximization

```
// Forward pass: Compute messages
for each factor φ in Factors:
   // Extract variables involved in the factor
   variables = GetVariables(φ)

   // For each variable, compute messages
   for each variable X in variables:
      for each neighbor Y of X:
         // Update message from X to Y
         message[X → Y] = max_{x} (φ(x) * product(message[Z → X] for all Z ≠ Y))

// Backward pass: Find the MAP configuration
MAP_configuration = {}
for each variable X in Factors:
   // Determine the value that maximizes the joint probability
   MAP_configuration[X] = argmax_{x} (product(message[Y → X] for each neighbor Y)

return MAP_configuration
```

Explanation:

1. **Initialization:** Messages from each variable to its neighbors are initialized to 1, representing a neutral element for maximization.

2. Forward Pass:

- For each factor, the algorithm computes messages sent to neighboring variables by maximizing over the possible values of the variable involved in the factor.
- The messages are updated based on the factor values and the messages received from other neighbors. 3, Backward Pass:
- After the forward pass, the algorithm determines the configuration of variables that maximizes the joint probability by going back through the messages to find the values of the variables.

4. Return: The algorithm returns the MAP configuration, which is the most probable assignment of the variables.

Conclusion

The Max-Product/Max-Sum algorithm represents a fundamental paradigm shift from marginal inference to finding the most probable explanation (MPE) of all variables jointly. Through our exploration, several key insights emerge:

Fundamental Transition

The algorithm elegantly transforms the well-established Sum-Product framework by replacing summation with maximization, demonstrating how a simple operational change can redirect inference from computing distributions to finding optimal configurations.

Practical Superiority of Max-Sum

While Max-Product provides the conceptual foundation, Max-Sum emerges as the practically superior implementation due to its numerical stability. By operating in log-space, it converts multiplications into additions, eliminating the risk of numerical underflow that plagues probability-based computations in large graphs.

The Critical Role of Backtracking

A crucial realization is that the forward pass alone is insufficient - it only computes the value of the maximum probability, not the actual assignment that achieves it. The backtracking phase, inspired by the Viterbi algorithm, is essential for reconstructing the optimal configuration by tracing back through stored argmax decisions.

Algorithmic Generality

The framework's beauty lies in its generality:

- It specializes to the Viterbi algorithm for HMMs
- Applies to tree-structured graphs with guaranteed optimality
- Extends to loopy graphs via iterative message passing (though without optimality guarantees)

Computational Efficiency

For tree-structured graphs, the algorithm maintains the same favorable linear time complexity as its Sum-Product counterpart, making MAP inference tractable for large-scale problems where brute-force search over all possible configurations would be exponentially expensive.

Real-world Significance

MAP inference underpins numerous real-world applications including image segmentation, error-correcting codes, protein folding, and natural language processing, where finding the single best overall explanation is more valuable than understanding individual variable uncertainties.

In essence, the Max-Product/Max-Sum algorithm with backtracking provides a complete, efficient, and practical framework for maximum a posteriori inference, balancing theoretical elegance with computational feasibility across a wide spectrum of probabilistic graphical models.

4 : Junction Tree Algorithm (JTA)

Topic Focus

Extends **exact inference** to **graphs with cycles** using **clique clustering** into a *junction tree*.

Key Concepts & Sources

Overcomes the limitations of Variable Elimination:

- Can reuse computations for multiple queries.
- Handles general cyclic graphs.

Junction Tree algorithm

The Junction Tree algorithm can be broken down into these steps:

1. Initialize with the initial set of factors and construct a junction tree from the graph.

2. Choose the elimination order to form cliques and assign factors to cliques.

3. Loop to pass messages between cliques by marginalizing and combining factors, continuing until all messages are propagated.

4. Normalize the potentials in the cliques to obtain the marginal probabilities for the query variables.

Triangulation

The graph must be **triangulated (chordal)** to form a valid junction tree.

Triangulation defines the elimination order and clique structure.

Finding the **optimal triangulation** that minimizes maximal clique size is **NP-hard**.

JTA Inference

Messages are **sum-product updates** between clique nodes based on shared variables (separators).

The computational cost remains **exponential in clique size**, though often less than brute-force enumeration.

5: Loopy Belief Propagation (LBP)

5.1. Introduction and Topic Focus

Loopy Belief Propagation (LBP) is an approximate inference algorithm designed for probabilistic graphical models (PGMs), particularly factor graphs or Markov Random Fields (MRFs) that contain cycles (loops).

In graphs with cycles, exact inference methods such as the Junction Tree Algorithm (JTA), Variable Elimination, or Belief Propagation on trees become computationally intractable due to exponential growth in complexity from induced cycles.

Core Purpose:

Enables approximate marginal inference by extending the Sum-Product Belief Propagation algorithm to loopy graphs through iterative message passing.

Key Advantages:

Efficient and scalable for large graphs.

Linear time per iteration relative to the number of edges.

Limitations:S

Provides approximations; no formal convergence guarantee.

Works best in sparsely connected or "locally tree-like" graphs.

Theoretical Foundation:

Optimizes the Bethe free energy, a variational approximation to the true log-partition function.

5.2. Key Concepts

5.2.1 Graphical Model Context

Undirected Graphs (MRFs): Nodes = random variables; edges = dependencies.

Joint probability:

$$P(X) = \frac{1}{Z}\prod_{c} \psi_c (X_c)$$

Factor Graphs: Bipartite representation with variable nodes (circles) and factor nodes (squares). Messages pass between nodes.

Cycles: LBP treats the graph as tree-like, iterating messages around loops.

5.2.2 Message Passing Mechanism

Messages: Vectors summarizing beliefs from one node to another.

Initialization: Uniform or random.

Iteration: Repeated updates until convergence.

Belief Computation:

$$\mathbf{b}(\mathbf{x}) \propto \prod_{\mathbf{f}\in \mathbf{neighbors}(\mathbf{x})} \mathbf{m}_{\mathbf{f}\to\mathbf{x}}(\mathbf{x}), \quad \mathbf{with} \sum_{\mathbf{x}} \mathbf{b}(\mathbf{x}) = \mathbf{1}$$

5.2.3 Dependence on Markov Blanket

Node updates rely only on the local Markov blanket (neighbors + potentials).

Efficient: time complexity $O(|E| \cdot K^2)$.

Highly parallelizable.

5.3. Message Updates

5.3.1 Variable-to-Factor Message

$$m_{x\to f}(x) = \phi_x(x) \prod_{g\in \text{neighbors}(x)\setminus f} m_{g\to x}(x)$$

Aggregates incoming messages except from f.

5.3.2 Factor-to-Variable Message

$$m_{f \to x}(x) = \sum_{X_f \setminus x} \left[\psi_f(X_f) \prod_{y \in \text{neighbors}(f) \setminus x} m_{y \to f}(y) \right]$$

Sum over all configurations of neighbors except x.

5.3.3 Update Process

Initialize messages uniformly or randomly.

Iteratively update:

Synchronous: all nodes at once.

Asynchronous: one node at a time.

Convergence check:

$$\Delta = \sum \left| m^{new} - m^{old} \right|$$

Stop if $\Delta < \epsilon$ or after maximum iterations.

5.4. Convergence and Techniques

5.4.1 Convergence Properties

No guarantee: Fixed points may not exist; iterations may oscillate.

Empirical success: Works in practical cases.

Failure cases: Strongly frustrated graphs or short cycles.

5.4.2 Acceleration and Stabilization Techniques

Asynchronous Updates: Use latest messages immediately.

Message Damping: Blend new and old messages (e.g., $\lambda = 0.5$) to reduce oscillations.

Other Variants: Tree-Reweighted BP, Residual BP, Generalized BP.

5.5. How LBP Works (Example)

Consider a cycle of 4 binary variables (A, B, C, D):

Pairwise potentials:

$$\psi(x_i, x_j) = \begin{cases} 2, & \text{if} x_i = x_j \\ 0.5, & \text{otherwise} \end{cases}$$

Factor graph: Variables = circles, factors = squares.

Iteration:

Initialize messages $[1,1]$.

Messages propagate around the cycle; beliefs update iteratively.

Adding evidence (e.g., $\mathrm{D} = 1$) propagates favorability.

After damping (~10 iterations), beliefs converge to approximate marginals.

5.6. Applications

Error-Correcting Codes: LDPC decoding, turbo codes.

Computer Vision: Stereo matching, image denoising, segmentation.

Other Domains: NLP (dependency parsing), bioinformatics, SAT solvers.

LDPC Example:
LBP iteratively computes log-likelihood ratios on Tanner graph edges, achieving near-Shannon-limit performance.

7. Conclusion

LBP bridges the gap between exact inference (limited to trees) and practical approximate inference in cyclic graphs. Its message-passing paradigm, local updates, and empirical robustness make it indispensable despite approximations.