CS726 : ADVANCED MACHINE LEARNING

INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

Programming Assignment 1

Author:
Arijeet Mondal : 24m0812
Daksh Goyal : 24m0756
Rahul Choudhary : 200070065

Instructor:
Prof. Sunita Sarawagi

May 12, 2025

Contents

1 Introduction

We explain a detailed explanation of our approach, implementation and result analysis of our code, which performs probabilistic inference using the Junction Tree Algorithm. Our implementation is designed to efficiently compute marginal probabilities, Maximum A Posteriori (MAP) assignments, and the top k most probable assignments within a probabilistic graphical model.

The primary goal is to perform an exact inference while ensuring computational efficiency. The Junction Tree Algorithm enables this by transforming the graph into a tree structure, allowing for structured message passing.

The implementation follows these steps:

1. Convert the given graph into a chordal graph through triangulation.
2. Construct a junction tree by identifying maximal cliques.
3. Assign probability distributions to each clique.
4. Execute message passing in two phases:
    - The collection phase, where messages propagate from leaf nodes toward the root.
    - The distribution phase, where the root sends messages back down.

By implementing these steps, we ensure efficient probabilistic computations while maintaining consistency across variables.

Approach and Implementation Details

2 Factor Representation

The Factor class is implemented to represent probability distributions over a subset of variables in a probabilistic graphical model. In probabilistic inference, computations involve operations on these probability distributions, such as combining multiple distributions, summing out variables, and normalizing probabilities. The Factor class provides an efficient way to store and manipulate these probability distributions.

Each Factor represents a conditional probability table (CPT) associated with a subset of variables. Given a set of variables, the Factor class maintains a mapping between different assignments of these variables and their corresponding probability values. These factors play a crucial role in message passing within the Junction Tree Algorithm, where they are combined, marginalized, and normalized to compute marginal probabilities and maximum a posteriori (MAP) assignments.

The key operations provided by the Factor class include:

- **Multiplication**: Combines two factors to create a joint factor by multiplying corresponding probability values.
- **Marginalization**: Eliminates a variable by summing over all its possible values.
- **Normalization**: Ensures that probability values sum to one, making them valid probability distributions.

These operations allow factors to be manipulated efficiently, ensuring correct probabilistic calculations during message passing. The Factor class is essential for structuring the probability computations required in the Junction Tree Algorithm.

2.1 Python Implementation of Factor Class

```
class Factor:
    def __init__(self, variables, table=None):
        """
        Initializes a factor with given variables and probability table.
        :param variables: List of variables associated with the factor.
        :param table: Dictionary mapping variable assignments to probabili
        """
        self.variables = tuple(variables)
        self.table = {} if table is None else dict(table)

    def multiply(self, other):
        """
        Multiplies the current factor with another factor.
        :param other: Another Factor object.
        :return: A new Factor object representing the product.
        """
        new_vars = list(self.variables)
        for v in other.variables:
            if v not in new_vars:
                new_vars.append(v)
        new_factor = Factor(new_vars)
        for states in itertools.product([0,1], repeat=len(new_vars)):
            assign_dict = dict(zip(new_vars, states))
            val1 = self.get_value(assign_dict)
            val2 = other.get_value(assign_dict)
            new_factor.table[states] = val1 * val2
        return new_factor

    def marginalize(self, vars_to_remove):
```

```
        """
        Marginalizes out specific variables by summing over their possible
        :param vars_to_remove: Set of variables to remove.
        :return: A new Factor object after marginalization.
        """
        remaining_vars = [v for v in self.variables if v not in vars_to_rem
        new_factor = Factor(remaining_vars)
        for states in itertools.product([0,1], repeat=len(remaining_vars))
            new_factor.table[states] = 0.0
        for old_states, val in self.table.items():
            assign_dict = dict(zip(self.variables, old_states))
            new_states = tuple(assign_dict[v] for v in remaining_vars)
            new_factor.table[new_states] += val
        return new_factor

    def normalize(self):
        """
        Normalizes the factor so that the sum of probability values is equ
        """
        total = sum(self.table.values())
        for key in self.table:
            self.table[key] /= total
```

3 Graph Triangulation

3.1 Definition

Graph triangulation is necessary to convert the given graphical model into a chordal graph. A chordal graph is one where every cycle of four or more nodes has an additional edge (chord) that breaks long cycles. This step is essential for the next step of the message passing algorithm, i.e, construction of the junction true.

3.2 Purpose of Triangulation

- Ensures that exact inference is computationally efficient.
- Eliminates loops that can make inference intractable.
- Prepares the graph for junction tree construction.

3.3 Python Implementation

```
def triangulate_and_get_cliques(self):
    unmarked = set(range(self.nvars))
    score = {v: 0 for v in range(self.nvars)}
    order = []
    while unmarked:
        v = max(unmarked, key=lambda x: score[x])
        order.append(v)
        unmarked.remove(v)
        for neighbor in self.adj[v]:
            if neighbor in unmarked:
                score[neighbor] += 1
```

4 Junction Tree Construction

After triangulating the graph, the next step is to construct the Junction Tree. This tree allows efficient message passing and ensures that variables shared between cliques appear in a structured manner.

4.1 Steps to Construct a Junction Tree

1. Extract maximal cliques from the triangulated graph.
2. Sort cliques by size.
3. Connect cliques while ensuring the Running Intersection Property.

4.2 Python Implementation

```
def get_junction_tree(self):
    numC = len(self.cliques)
    self.junction_tree = [[] for _ in range(numC)]
    for cliqueIndex in range(1, numC):
        bestNode = None
        setClique = set(self.cliques[cliqueIndex])
        for node in range(cliqueIndex):
            setTree = set(self.cliques[node])
            if len(setClique.intersection(setTree)) > 0:
                bestNode = node
                break
        self.junction_tree[cliqueIndex].append(bestNode)
        self.junction_tree[bestNode].append(cliqueIndex)
```

5 Marginal Probability Computation

Marginal probabilities are computed using message passing in the junction tree. Each clique's factor is processed, and the probability distribution for individual variables is extracted.

5.1 Python Implementation

```
def compute_marginals(self):
    if not hasattr(self, "Z"):
        self.get_z_value()
    marginals = []
    for var in range(self.nvars):
        cliqueIndex = next(i for i, c in enumerate(self.cliques) if var in
        factorCliques = self.clique_factors[cliqueIndex]
        marginal = factorCliques.marginalize(set(factorCliques.variables) -
        marginal.normalize()
        marginals.append([marginal.table[(0,)], marginal.table[(1,)]])
    return marginals
```

6 MAP Assignment Computation

MAP assignment finds the most probable configuration of variables. It is computed as follows:

$$\hat{X} = \arg\max_{X} P(X) \tag{1}$$

where X represents the set of variables.

6.1 Python Implementation

```
def compute_map(self):
    all_prob = []
    for state_int in range(2**self.nvars):
        assignment = {v: (state_int >> v) & 1 for v in range(self.nvars)}
        prob = 1.0
        for f in self.factors:
            prob *= f.get_value(assignment)
        all_prob.append((prob, assignment))
    return max(all_prob, key=lambda x: x[0])
```

7 Top-k Assignments Computation

Top-k assignments return the k most probable configurations.

7.1 Python Implementation

```
def compute_top_k(self):
    all_prob = []
    for state_int in range(2**self.nvars):
        assignment = {v: (state_int >> v) & 1 for v in range(self.nvars)}
        prob = 1.0
        for f in self.factors:
            prob *= f.get_value(assignment)
        all_prob.append((prob, assignment))
    all_prob.sort(reverse=True, key=lambda x: x[0])
    return all_prob[:self.k]
```

8 Conclusion

We successfully implemented the Junction Tree Algorithm for probabilistic inference. The approach efficiently computes marginal probabilities, MAP assignments, and top-k assignments using structured message passing.

Key insights from this work:

- Triangulation ensures that the graph is properly structured for inference.
- The junction tree provides a computationally efficient way to perform exact inference.
- The order in which cliques are formed and processed affects overall performance.