CS 726 Assignment 2

Joel Anto Paul(210070037), Jay Chaudhary(210070022), Aditya Kumar(210070003)

February 2025

1 Triangulation of the Graph

Triangulation ensures that the graph is **chordal** (every cycle of four or more nodes has a **chord**, i.e., an extra edge that prevents long cycles). This is required for efficient inference.

1.1 Steps in Triangulation

1. Compute an **elimination order** (using **minimum degree heuristic**).
2. Iteratively remove nodes while adding **fill-in edges** to preserve chordality.
3. Extract the maximal cliques from the triangulated graph.

1.2 Pseudocode for Triangulation

Algorithm 1 Triangulation Algorithm

```
Input: Graph G = (V, E)
Output: Triangulated graph G'
function TRIANGULATE(Graph G)
    Compute elimination order using minimum-degree heuristic
    for each node v in elimination order do
        Identify neighbors of v in current graph
        Connect all pairs of neighbors (add fill-in edges)
        Remove v from graph
    end for
    Return modified graph G'
end function
```

2 Junction Tree Construction

A **junction tree** (also known as a clique tree) is a tree structure where:

1. Each node represents a **maximal clique** from the triangulated graph.
2. Edges connect cliques with shared variables (separators).
3. The tree satisfies the **running intersection property**: If a variable appears in two cliques, it must appear in all cliques along the path connecting them.

2.1 Steps to Construct the Junction Tree

1. Find all **maximal cliques** in the triangulated graph.
2. Create a **weighted graph** where nodes are cliques, and edge weights are the size of their intersection.
3. Compute the **maximum spanning tree** (MST) using Prim's or Kruskal's algorithm.

2.2 Pseudocode for Junction Tree Construction

Algorithm 2 Junction Tree Construction

Input: Triangulated graph G'
Output: Junction tree T
function ConstructJunctionTree(Graph G')
    Find all maximal cliques in G'
    Construct a weighted graph where:
        Nodes = maximal cliques
        Edges = shared variables between cliques
        Weights = size of shared variable set
    Compute the **Maximum Spanning Tree** (MST)
    Return MST as the Junction Tree
end function

2.3 4. Maximum Spanning Tree Construction (Kruskal's Algorithm)

Algorithm 3 Maximum Spanning Tree (Kruskal's Algorithm)

function MaximumSpanningTree(*graph*)
    edges ← sorted list of all $(weight, node1, node2)$ in descending order
    parent ← dictionary mapping each node to itself (for union-find)
    function Find(*node*) ▷ Find root representative
        if *parent*[*node*] == *node* **then**
            return *node*
        end if
        parent[*node*] ← Find(*parent*[*node*]) ▷ Path compression
        return *parent*[*node*]
    end function
    function Union($node1, node2$) ▷ Merge two sets
        $root1$ ← Find($node1$)
        $root2$ ← Find($node2$)
        parent[$root1$] ← $root2$
    end function
    mst ← empty adjacency list
    for each edge $(weight, node1, node2)$ **do**
        if Find($node1$) ≠ Find($node2$) **then** ▷ Ensure no cycles
            Union($node1$, $node2$)
            Add edge $(node1, node2)$ to *mst*
        end if
    end for
    return *mst*
end function

3 Assigning new clique potentials

The algorithm assigns potentials to newly formed cliques by either directly using known potentials or computing them through the merging of relevant subcliques. The process involves sorting, reordering, and pairwise merging of clique potentials.

3.1 Processing of Original Cliques

The function first creates a dictionary of original cliques and their corresponding potential values. The potential values are stored as lists, indexed by tuples representing the cliques.

3.2 Iterating Over Cliques in the Junction Tree

For each clique in the junction tree, we compute the potential of the new cliques by multiplying the potentials of the subcliques comprising the clique

3.3 Finding Overlapping Subcliques

To construct the potential for a new clique, we:

- Identify all subcliques in the original cliques that are subsets of the new clique.
- Instead of removing them, set their potential values to 1 to prevent double usage.

3.4 Pairwise Merging of Subcliques

Once we have identified all subcliques, we:

- Merge them sequentially, treating the merged output as a new subclique.
- Convert the potential tables into binary form.
- Multiply only those elements where the variable values are equal across subcliques.

4 Computing Marginals

4.1 Junction Tree Initialization

We initialize the necessary data structures before running belief propagation:

- **Messages**: A dictionary to store messages between cliques.
- **Junction Tree**: The structure used for message passing.
- **Potentials**: Initial potential tables associated with each clique.
- **Cliques Set**: A mapping of cliques to sets for efficient intersection operations.
- **Visited Set**: Tracks visited nodes during traversal.

4.2 Forward and Backward Passes

Belief propagation consists of two main traversals:

- **Forward Pass**: Messages are propagated from the root (node 0) to the leaves.
- **Backward Pass**: Messages are propagated from leaves back to the root.

Each message represents the marginalization of a clique over the variables not shared with its neighbor.

4.3 Computing Clique Potentials

After message passing, the updated potentials for each clique are computed as follows:

1. Start with the original potential of the clique.
2. For each neighbor in the junction tree:
    - If a message exists from the neighbor to the current node, update the potential by multiplying it with the message.
    - The multiplication is performed over the intersection of variables in both cliques.
3. Store the final potential in a dictionary.

4.4 Computing Node Marginals

Once clique potentials are determined, the marginal probability of each node is calculated:

1. Iterate through each clique and extract individual nodes.
2. If a node has not been processed, sum out all other variables in the clique to obtain its marginal.
3. Store the marginal probability in a sorted list to maintain consistency.

The marginalization process ensures that the final marginals are normalized and reflect correct beliefs over individual variables.

4.5 Final Output

The function returns a list of computed marginals for each node, sorted in increasing order of node indices:

$$\texttt{self.marginals} = P(X_i)_{i=1}^{n} \tag{1}$$

where $P(X_i)$ represents the marginal probability distribution for variable X_i.

5 Computing the Partition Function

The partition function, denoted as Z, is computed as follows:

1. Sum the first two values of the first marginal probability distribution:

$$Z = \texttt{self.marginals}[0][0] + \texttt{self.marginals}[0][1] \tag{2}$$

2. Normalize all marginal distributions by dividing each value by Z:

$$\texttt{self.marginals} = \left[\frac{P(X_i)}{Z}\right]_{i=1}^{n} \tag{3}$$

This ensures that the final marginal distributions sum to one, making them valid probability distributions.

5.1 Final Output

The function returns the computed partition function Z and the normalized marginals:

$$(Z, \texttt{self.marginals}) \tag{4}$$

where Z is the normalizing constant and `self.marginals` contains the final probability distributions for each variable.

6 Compute Top K

6.1 Approach

The process follows these steps:

1. **Generate All Possible Assignments:** Use Cartesian product to create all possible variable assignments.
2. **Compute Joint Probability:**
   - For each assignment, compute the probability by multiplying clique potential values.
   - Convert the assignment into an index to access potential table values.
3. **Sort Assignments by Probability:** Sort all computed assignments in descending order of probability.
4. **Select Top-K Assignments:** Normalize probabilities using a partition function Z and extract the top-k assignments.

6.2 Pseudocode

Algorithm 4 Compute Top-K Most Probable Assignments

Input: Variable domains, cliques, clique potentials, number of top results k, normalization factor Z
Output: Top-k most probable assignments
function ComputeTopK
    Initialize an empty list `assignments_with_probs`
    for each assignment in the Cartesian product of all variable domains **do**
        Set `probability` to 1.0
        for each clique in the model **do**
            Retrieve the potential table for the clique
            Map variables in the clique to indices in the assignment
            Extract values of the clique variables from the assignment
            Convert the extracted values into an index for the potential table
            Multiply `probability` by the corresponding potential value
        end for
        Append the assignment and its probability to `assignments_with_probs`
    end for
    Sort `assignments_with_probs` in descending order of probability
    Extract top-k assignments and normalize probabilities using Z
    Return the top-k assignments
end function

6.3 Algorithm

The computational complexity is driven by the number of variable assignments and cliques.

1. **Initialization:** Prepare the list to store assignments and probabilities.
2. **Assignment Generation:** Iterate through the Cartesian product of all variable domains, leading to $O(2^N)$ complexity for binary variables.
3. **Joint Probability Calculation:** For each assignment:
    - Iterate over all cliques, extract relevant values, compute potential index, and multiply probabilities.
    - The complexity depends on clique sizes but is generally bounded by $O(K)$ where K is the number of cliques.
4. **Sorting:** Sorting $O(2^N)$ assignments takes $O(2^N \log(2^N))$ time.
5. **Top-K Extraction:** Extracting the top-k assignments runs in $O(k)$.

Thus, the overall complexity is approximately $O(2^N \log(2^N))$ due to sorting.

7 Individual Contributions

Joel Anto Paul (210070037) : triangulate and get cliques, get junction tree, compute top k

Aditya Kumar (210070003) : init, assign potentials to cliques, compute top k

Jay Chaudhary (210070037) : get z values, compute marginals, compute top k

8 References

Chatgpt, Other References