Advanced Machine Learning

February 17, 2025

Spring 2025

CS 726: Programming Assignment **Submitted by:** Pinak Mahapatra , Danish Siddiqui and Aansh Samyani

Contents

1 Problem Statement **2**

2 Triangulation **2**
2.1 Checking if a Vertex is Simplicial . . . 3
2.2 Finding the Vertex with Minimum Degree . . . 3
2.3 Making a Vertex Simplicial . . . 4
2.4 Adding Edges to the Graph . . . 4

3 Junction Tree Construction **5**
3.1 Finding the Minimum Spanning Tree . . . 5

4 Assigning Potentials to Cliques **6**
4.1 Index Mapping . . . 7

5 Message Passing Algorithm **7**
5.1 Initialization of Messages . . . 7
5.2 Message Passing Procedure . . . 8
5.3 Computing Clique Marginals . . . 9
5.4 Computing the Partition Function (Z-value) . . . 9
5.5 Computing Marginals for Variables . . . 10
5.6 Extracting Marginals . . . 10

6 Computing the Top-K Most Probable Assignments **11**

7 Conclusion **13**

8 Resources **13**

9 Contributions **13**

1 Problem Statement

- **Triangulation**: Explain the process of triangulating the graph. Include diagrams if necessary.
- **Junction Tree Construction**: Describe how to construct a junction tree from the triangulated graph and how to assign potentials to each clique.
- **Marginal Probability**: Show how to calculate the marginal probability of each variable using the junction tree.
- **MAP Assignment**: Define the MAP assignment and explain how to find it in the context of message passing algorithms.
- **Top k Assignments**: Discuss how to find the top k assignments of probability values and their significance.

2 Triangulation

In this section, we describe the algorithm for triangulating an undirected graph and extracting maximal cliques. The goal is to convert the graph into a chordal (triangulated) graph by processing vertices iteratively.

Algorithm 1 Triangulate and Extract Maximal Cliques

```
triangulated_graph ← deep_copy(adj_list)
induced_graph ← deep_copy(adj_list)
vertices ← {0, 1, ..., n − 1}                          ▷ n is the number of vertices
remaining_vertices ← vertices
ordering ← []
while remaining_vertices ≠ ∅ do
    simplicial_vertex ← NONE
    for each v ∈ remaining_vertices do
        if is_simplicial(v, induced_graph) then
            simplicial_vertex ← v
            break
        end if
    end for
    if simplicial_vertex ≠ NONE then
        chosen_vertex ← simplicial_vertex
    else
        chosen_vertex ← vertex in remaining_vertices with minimum degree in induced_graph
    end if
    if not is_simplicial(chosen_vertex, triangulated_graph) then
        new_edges ← make_simplicial(chosen_vertex, induced_graph)
        add_edges(new_edges, triangulated_graph)
        add_edges(new_edges, induced_graph)
    end if
    remaining_vertices ← remaining_vertices \ {chosen_vertex}
    Append chosen_vertex to ordering
    remove_vertex(chosen_vertex, induced_graph)
end while
maximal_cliques ← get_maximal_cliques(triangulated_graph)
```

The algorithm utilizes helper functions like `is_simplicial`, `min_deg`, `make_simplicial`, and `add_edges`. It first removes simplicial vertices from `rem`. If none exist, the vertex with the minimum degree is selected and made simplicial by adding necessary edges in the induced graph.

These edges are then incorporated into both the induced and triangulated graphs. Finally, once the triangulation process is complete, the maximal cliques can be extracted using the `get_maximal_cliques` function. Once the graph is triangulated, the maximal cliques can be efficiently extracted, which is useful for later steps such as junction tree creation. Below is a list of all the helper functions that we have used

2.1 Checking if a Vertex is Simplicial

A vertex is simplicial if all its neighbors form a complete subgraph, meaning each pair of neighbors is connected by an edge. The function below checks whether a given vertex is simplicial by verifying that all its neighbors are interconnected.

Algorithm 2 Check if a Vertex is Simplicial

```
function IS_SIMPLICIAL(vertex, adj_list)
    if vertex ∉ adj_list then
        return False
    end if
    neighbors ← list of neighbors of vertex from adj_list
    for i ← 0 to |neighbors| − 1 do
        for j ← i + 1 to |neighbors| − 1 do
            if neighbors[j] ∉ adj_list[neighbors[i]] then
                return False
            end if
        end for
    end for
    return True
end function
```

2.2 Finding the Vertex with Minimum Degree

This function selects the vertex with the minimum degree from the induced graph. It iterates through the remaining vertices, keeping track of the vertex with the lowest degree.

Algorithm 3 Find the Vertex with Minimum Degree

```
function MIN_DEG(adj_list, remaining)
    mn_deg ← ∞
    min_ver ← None
    for each vertex in remaining do
        degree ← |adj_list[vertex]|
        if degree < mn_deg then
            mn_deg ← degree
            min_ver ← vertex
        end if
    end for
    return min_ver
end function
```

2.3 Making a Vertex Simplicial

Algorithm 4 Make a Vertex Simplicial

```
function MAKE_SIMPLICIAL(vertex, adj_list)
    edges ← []
    neighbors ← list of neighbors of vertex from adj_list
    for i ← 0 to |neighbors| − 1 do
        for j ← i + 1 to |neighbors| − 1 do
            if neighbors[j] ∉ adj_list[neighbors[i]] then
                Append (neighbors[i], neighbors[j]) to edges
            end if
        end for
    end for
    return edges
end function
```

If a vertex is not already simplicial, this function ensures it becomes simplicial by adding the necessary edges among its neighbors.

2.4 Adding Edges to the Graph

This function modifies the graph by adding the specified edges to both the induced and triangulated graphs. When a new edge (u, v) is introduced, it ensures that v is added to the adjacency list of u and vice versa. This step maintains symmetry in the graph representation, ensuring that connections between nodes are properly established. By updating both graphs, it helps maintain consistency throughout the triangulation process, ensuring that any structural modifications are reflected in both versions of the graph.

Algorithm 5 Add Edges to the Graph

```
function ADD_EDGES(edges, adj_list)
    for each (u, v) in edges do
        if v ∉ adj_list[u] then
            Append v to adj_list[u]
        end if
        if u ∉ adj_list[v] then
            Append u to adj_list[v]
        end if
    end for
end function
```

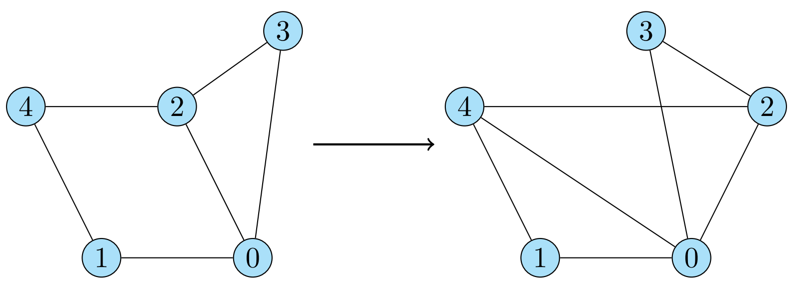

Figure 1: Graph before and after triangulation. Additional edges were added to form a chordal graph.

3 Junction Tree Construction

The junction tree is formed from maximal cliques of the triangulated graph. Edges connect cliques sharing elements, weighted by the number of shared elements. A spanning tree is then built, ensuring the running intersection property.

Algorithm 6 Construct the Junction Tree

```
function GET_JUNCTION_TREE
    edge_list ← []
    n ← number of maximal cliques
    for ind1 ← 0 to n − 1 do
        for ind2 ← ind1 + 1 to n − 1 do
            wt ← |maximal_cliques[ind1] ∩ maximal_cliques[ind2]|  ▷ Weight is the number of common elements
            if wt > 0 then
                Append (ind1, ind2, −wt) to edge_list  ▷ Negative weight for maximum spanning tree
            end if
        end for
    end for
    (mst, _) ← MSTFIND(n, edge_list)  ▷ Find the maximum spanning tree
    mst_adj ← empty adjacency list of size n
    for each (u, v, w) in mst do
        Append v to mst_adj[u]
        Append u to mst_adj[v]
    end for
end function
```

Below is a list of helper functions that we used with the pseudocode

3.1 Finding the Minimum Spanning Tree

Algorithm 7 Find the Minimum Spanning Tree

```
function MSTFIND(V, edges)
    Sort edges in increasing order of weight
    dsu ← Initialize Disjoint Set Union for V vertices
    mst ← []  ▷ List to store the edges of MST
    cost ← 0  ▷ Total cost of the MST
    for each (u, v, w) in edges do
        if dsu.find(u) ≠ dsu.find(v) then  ▷ Check if u and v belong to different components
            dsu.unionbysize(u, v)
            Append (u, v, w) to mst
            cost ← cost + w
            if |mst| = V − 1 then  ▷ Stop when we have V − 1 edges
                break
            end if
        end if
    end for
    return (mst, cost)
end function
```

This function finds the minimum spanning tree (MST) of a graph using Kruskal's algorithm. The edges are first sorted by weight, and a Disjoint Set Union (DSU) data structure is used to manage connected components efficiently. The algorithm selects the smallest weighted edges that do not form cycles, constructing the MST.

4 Assigning Potentials to Cliques

.

Algorithm 8 Assign Potentials to Cliques

function ASSIGN_POTENTIALS_TO_CLIQUES
  Initialize $junction\_potentials$ with all values set to 1
  for $i \leftarrow 0$ to $|\text{maximal\_cliques}| - 1$ **do**
    $junction\_potentials[i] \leftarrow$ table of size $2^{|\text{maximal\_cliques}[i]|}$ initialized with 1
  end for
  for each clique x in original cliques **do**
    $l \leftarrow []$
    $ind \leftarrow -1$
    $mx \leftarrow \infty$
    ▷ Find the smallest junction clique that contains x
    for each junction clique y in maximal_cliques **do**
      if $x \subseteq y$ **and** $|y| < mx$ **then**
        $l \leftarrow y$
        $ind \leftarrow$ index of y
        $mx \leftarrow |y|$
      end if
    end for
    if $ind = -1$ **then**
      continue
    end if
    $potential_x \leftarrow$ potential of clique x
    $index\_mapping \leftarrow imap(l, x)$ ▷ Maps indices from original clique to junction clique
    for $pot\_idx \leftarrow 0$ to $|potential_x| - 1$ **do**
      $clique\_idx \leftarrow index\_mapping[pot\_idx]$
      for each index k in $clique\_idx$ **do**
        $junction\_potentials[ind][k] \leftarrow junction\_potentials[ind][k] \times potential_x[pot\_idx]$
      end for
    end for
  end for
end function

The function assigns potentials to cliques in the junction tree to ensure proper message passing in probabilistic graphical models. It first initializes `junction_potentials`, setting all values to 1, and assigns a potential table of size $2^{|\text{maximal\_cliques}[i]|}$ to each maximal clique. Then, for each original clique, it finds the smallest junction clique that contains it, ensuring efficient mapping. Using the `imap()` function, it establishes index mappings between original and junction cliques. Finally, it assigns potential values by iterating through the original clique's potential table and updating the corresponding indices in the junction clique, maintaining consistency for belief propagation and probabilistic inference.

4.1 Index Mapping

The `imap` function is designed to map the index positions of a **given clique** within a **maximal clique** in the context of probabilistic graphical models. It first converts both `maxim_clique` and `givenClique` into sorted lists (`max_vars` and `orig_vars`, respectively) to ensure consistency in indexing. Then, it identifies the positions of the given clique's variables within the maximal clique and stores them in the list `positions`. This step ensures that variables in the smaller clique can be correctly aligned with those in the larger maximal clique.

Next, the function constructs a mapping between the indices of the given clique and its corresponding positions in the maximal clique's probability table. It iterates over all possible binary states of `orig_vars` and `max_vars`, represented using binary bit vectors (`i_bits` for the given clique and `j_bits` for the maximal clique). It checks whether the bits in `j_bits` match the expected positions from `i_bits` using the precomputed `positions` list. If a match is found, the corresponding index from `max_vars` is added to the mapping. The function ultimately returns a dictionary where each binary state of `orig_vars` is mapped to a list of compatible states in `max_vars`, ensuring correct alignment of probability distributions for inference and message passing in junction trees.

5 Message Passing Algorithm

The goal of our implementation is to compute the **marginal probabilities** of variables in a probabilistic graphical model. We achieve this through **message passing** on a **junction tree**, which allows for efficient inference. Our approach follows these steps:

1. **Message Passing:** Exchange information between cliques in the junction tree.
2. **Computing Clique Marginals:** Use the received messages to compute clique marginals.
3. **Computing the Partition Function (Z-value):** Normalize clique marginals.
4. **Computing Marginals of Variables:** Extract marginal probabilities for individual variables.

5.1 Initialization of Messages

We initialize a dictionary `messages` where each key is a tuple (i, j) representing message exchange between cliques, and each value is a table of size $2^{|\text{common variables}|}$, initialized as:

$$M_{(i \to j)}(X_{\text{common}}) = 1$$

for all possible binary assignments of the common variables.

Algorithm 9 Initialize Messages

function MESSAGES_INIT($j\_tree, maxim\_clique$)
  $messages \leftarrow$ empty dictionary
  for each i in j_tree **do**
    for each j in $j\_tree[i]$ **do**
      $common\_vars \leftarrow |maxim\_clique[i] \cap maxim\_clique[j]|$
      $messages[i, j] \leftarrow$ list of size 2^{common_vars} initialized to 1
    end for
  end for
  return $messages$
end function

5.2 Message Passing Procedure

The function `message_passing()` runs iteratively to allow information exchange:

1. **Track Received Messages:** Maintain a `receivedSet` dictionary to track received messages.
2. **Process Nodes Based on Received Messages:**
   - If a node has received **all but one** of its expected messages, it can now send its own message. If it has received **all** messages, it sends messages to all neighbors.
3. **Call `sendMessage()` to Compute Messages:** Updates the message dictionary.
4. **Repeat for Several Iterations:** Until convergence is achieved.

Mathematically, a message from clique C_i to C_j is:

$$M_{(i \to j)}(X_{\text{common}}) = \sum_{X_{C_i} \setminus X_{\text{common}}} \Psi_i(X_{C_i}) \prod_{k \in \text{neigh}(i) \setminus j} M_{(k \to i)}(X_{\text{common}})$$

where:

- X_{common} represents the shared variables between C_i and C_j. $\Psi_i(X_{C_i})$ is the potential function of clique C_i. The product term represents messages received from other neighbors.

Algorithm 10 Message Passing in Junction Tree

```
function MESSAGE_PASSING(j_tree, pot, maxim_clique)
    messages ← MESSAGES_INIT(j_tree, maxim_clique)
    receivedSet ← dictionary mapping each node to an empty set
    nodes ← |j_tree|
    iterations ← 0
    while iterations < 10 do
        for i ← 0 to nodes − 1 do
            neigh ← j_tree[i]
            diff ← neigh − receivedSet[i]
            if |diff| > 1 then
                continue                    ▷ Node needs more messages before sending
            else if |diff| = 1 then
                x ← element of diff
                receivedSet[x] ← receivedSet[x] ∪ {i}
                SENDMESSAGE(i, x, pot, messages, maxim_clique, j_tree)
            else
                for each v in neigh do
                    receivedSet[v] ← receivedSet[v] ∪ {i}
                    SENDMESSAGE(i, v, pot, messages, maxim_clique, j_tree)
                end for
            end if
        end for
        iterations ← iterations + 1
    end while
    return messages
end function
```

Algorithm 11 Send Message Between Cliques

```
function sendMessage(x, y, pot, messages, max_cliques, j_tree)
    C_x ← sorted variables of max_cliques[x]
    C_y ← sorted variables of max_cliques[y]
    new_msg ← pot[x]                                  ▷ Start with the clique potential
    common ← C_x ∩ C_y
    for each i in j_tree[x] do
        if i = y then
            continue
        end if
        msg ← messages[i, x]
        common2 ← max_cliques[i] ∩ max_cliques[x]
        mp ← IMAP(C_x, common2)
        for each (t, v) in mp do
            for each j in v do
                new_msg[j] ← new_msg[j] × msg[t]
            end for
        end for
    end for
    ans_msg ← list of size 2^|common| initialized to 0
    mp2 ← IMAP(C_x, common)
    for each (k, v) in mp2 do
        for each i in v do
            ans_msg[k] ← ans_msg[k] + new_msg[i]
        end for
    end for
    messages[x, y] ← ans_msg
end function
```

5.3 Computing Clique Marginals

After **message passing** is complete, we compute **clique marginals** using:

$$P(X_C) = \Psi_C(X_C) \prod_{j \in \text{neigh}(C)} M_{(j \to C)}(X_C)$$

Steps:

1. **Initialize clique marginals** as copies of junction potentials.
2. **Multiply all received messages** for each clique.
3. **Store the results** in `clique_marginals`.

This ensures that each clique potential is properly **normalized**.

5.4 Computing the Partition Function (Z-value)

The **partition function** Z is used to normalize probability distributions:

$$Z = \sum_X P(X)$$

Since the graphical model ensures global consistency, Z can be computed using any clique's marginal:

$$Z = \sum_{X_{C_1}} P(X_{C_1})$$

where C_1 is any clique. In our implementation:

```
self.z_value = sum(self.clique_marginals[0])
```

This provides the **normalization constant** for all marginal probability calculations.

Algorithm 12 Compute the Partition Function Z

```
function GET_Z_VALUE
    messages ← MESSAGE_PASSING(mst_adj, junction_potentials, maxim_cliques)    ▷ Obtain messages using message passing
    clique_marginals ← dictionary initialized with copies of junction_potentials    ▷ Initialize clique marginals with junction potentials
    for each clique c in maxim_cliques do
        for each neighboring clique d in mst_adj[c] do
            cliq ← maxim_cliques[c]
            a ← maxim_cliques[d]
            b ← maxim_cliques[c]
            common ← sorted set of common variables between a and b
            mp ← IMAP(cliq, common)    ▷ Map common variables to indices
            msg ← copy of message sent from d to c
            for each (k, v) in mp do
                for each l in v do
                    clique_marginals[c][l] ← clique_marginals[c][l] × msg[k]
                end for
            end for
        end for
    end for
    z_value ← sum of all elements in clique_marginals[0]    ▷ Compute partition function Z by summing over clique marginal
    return z_value
end function
```

5.5 Computing Marginals for Variables

With clique marginals computed, obtaining **marginal probabilities for individual variables** is straightforward.

5.6 Extracting Marginals

For each variable X_i:

1. **Find a clique** containing X_i.
2. **Marginalize over all other variables**.
3. **Normalize using** Z.

Algorithm 13 Compute Marginal Probability of a Variable

```
function MARG_XI(potential, clique_vars, xi)
    l ← list of size 2 initialized with 0
    mp ← IMAP2(xi, clique_vars)            ▷ Get index mapping for variable X_i
    for each (k, v) in mp do
        for each index j in v do
            l[k] ← l[k] + potential[j]     ▷ Sum over potential values
        end for
    end for
    return l
end function
```

Mathematically, the marginal probability of a variable is:

$$P(X_i) = \sum_{X_C \setminus X_i} P(X_C)$$

where:

- $P(X_C)$ is the clique marginal containing X_i.
- The summation marginalizes over all other variables.

This is implemented as:

```
ans = marg_xi(self.clique_marginals[j], l, i)
ans = [x/self.z_value for x in ans]  # Normalize
self.marginals.append(ans)
```

6 Computing the Top-K Most Probable Assignments

Algorithm 14 Compute Top-K Assignments

```
function COMPUTE_TOP_K
    topk_messages ← TOPK_MESSAGE_PASSING(mst_adj, junction_potentials, maxim_cliques, k)
    newmsg ← initialize empty lists for each clique assignment
    for each clique assignment i do
        Initialize newmsg[i] with empty assignment and probability from clique potential
    end for
    for each neighbor i in j_tree[x] do
        Retrieve msg from topk_messages[i, x]
        for each common variable mapping do
            Multiply probabilities and merge consistent assignments
            Keep only top-k assignments
        end for
    end for
    Normalize probabilities using Z and return top-k assignments
end function
```

The function `compute_top_k` finds the top-k most probable assignments in the graphical model using **message passing**. Instead of summing probabilities like in marginals computation, it keeps track of the top-k highest probability assignments for each clique. Messages are processed iteratively, ensuring consistent variable assignments while maintaining the top-k highest probability configurations. Finally, the results are normalized using the partition function Z and returned in the required format.

- **Step 1: Initialize Message Passing**
  - Calls `topk_message_passing` to retrieve messages containing the top-k assignments.
  - Initializes required variables such as `max_cliques`, `junction_potentials`, and `mst_adj`.
- **Step 2: Initialize Assignment Storage**
  - Creates a list `newmsg` to store potential assignments.
  - For each possible assignment, initializes probability values from clique potentials.
  - Converts integer indices into binary representations for variable assignment tracking.
- **Step 3: Process Incoming Messages**
  - Iterates through neighboring cliques in the junction tree.
  - Retrieves messages and extracts common variables between cliques.
  - Uses the `imap` function to align variable indices.
- **Step 4: Merge Assignments from Messages**
  - Iterates over assignment pairs from the message and the current clique.
  - Merges assignments while ensuring consistency (no variable conflicts).
  - Computes the new probability by multiplying the corresponding values.
  - Stores only the **top-k** highest probability assignments.
- **Step 5: Aggregate Assignments and Select Top-K**
  - Groups assignments based on shared variables.
  - Sorts them by probability in descending order.
  - Keeps only the **top-k** most probable assignments.
- **Step 6: Normalize Probabilities and Return Results**
  - Normalizes probabilities using the partition function Z.
  - Formats the results into a list containing:
    - **Assignment:** The binary variable configuration.
    - **Probability:** The normalized probability.
  - Returns the final list of the top-k most probable assignments.

7 Conclusion

In this assignment, we implemented a **graphical model inference framework** using **message passing** on a **junction tree**. We began by **triangulating the graph** and extracting **maximal cliques**, forming the foundation of the **junction tree structure**. We then implemented the **message passing algorithm**, allowing efficient **probabilistic inference** by exchanging information between cliques. Using these messages, we computed **clique marginals**, which helped derive the **partition function** (Z), a key normalization constant.

Additionally, we extracted **marginal probabilities** for individual variables by marginalizing over clique potentials. To extend our approach, we implemented a **top-k assignment algorithm**, identifying the most probable variable configurations while maintaining consistency in assignments. Overall, the assignment provided a **comprehensive understanding of graphical models**, junction tree algorithms, and probabilistic inference techniques, reinforcing **theoretical concepts through practical implementation**. We got better idea of working of algorithm by actually implementing them.

8 Resources

- Maximal Clique Problem - Recursive Solution (GeeksforGeeks)
- Variable Elimination Annotated Notes (Carnegie Mellon University)
- Belief Propagation Lecture Notes (Carnegie Mellon University)
- Junction Tree Algorithm (Stanford Institute)
- ChatGpt : Used for understanding some core concepts in detail, very limited code usage.

9 Contributions

- Pinak Mahapatra: Triangulation, Junction Tree Construction and Report Writing.
- Danish Siddiqui: Message Passing algorithm implementation, Clique potential assignment, marginal probability and Report Writing
- Aansh Samyani: Top K Assignments and Report Writing