

CS726 Programming Assignment – 1 Report

Saksham Rathi (22B1003)

Sharvaneer Sonawane (22B0943)

Deeksha Dhiwakar (22B0988)

Department of Computer Science,
Indian Institute of Technology Bombay

1 Triangulation

This step is implemented in the function `triangulate_and_get_cliques`. We first check if the graph is already triangulated, using the function `whether_triangulated` described below:

Algorithm 1 Check if Graph is already Triangulated

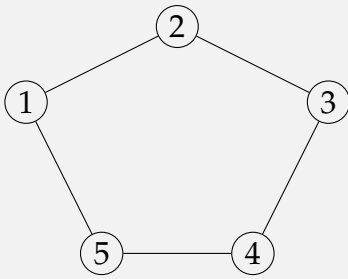
```
1: cycles  $\leftarrow$  Find all cycles in graph
2: for each cycle in cycles do
3:   if length of cycle  $\geq 4$  then
4:     if there is no shortcut (vertices connected by non-cycle edge) then
5:       return False
6:     end if
7:   end if
8: end for
9: return True
```

If the graph is already triangulated, we directly proceed with extracting the maximal cliques. If the graph is not triangulated, we first triangulate it using the minimum degree heuristic, as described in the pseudocode below:

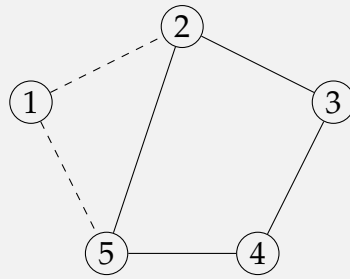
Algorithm 2 Triangulation Process

```
1: vertices_left  $\leftarrow$  Set of all vertices
2: while vertices_left is not empty do
3:   vertex  $\leftarrow$  Vertex in vertices_left with minimum degree
4:   for each pair  $(i, j)$  of neighbours of vertex do
5:     if the graph does not contain an edge between  $i$  and  $j$  then
6:       Add edge  $(i, j)$  to the original graph
7:     end if
8:   end for
9:   Remove vertex from vertices_left
10:  Update graph by removing vertex and updating degrees and edges
11: end while
```

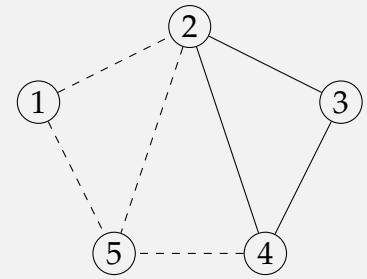
The figure below shows the run of the triangulation algorithm on an example graph:



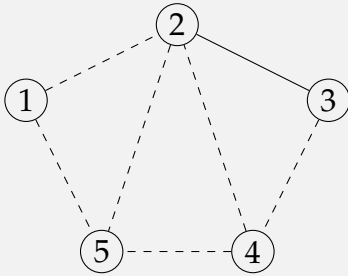
Step 1: Initial Graph



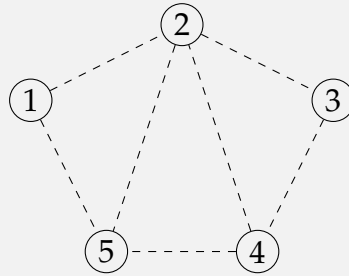
Step 2: Select 1, add (5,2), remove 1



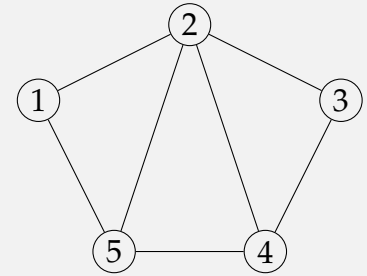
Step 3: Select 5, add (4,2), remove 5



Step 4: Select 4, remove 4



Step 5: Select 2, remove 2



Final Triangulated Graph

Once we obtain the triangulated graph, we extract the maximal cliques from it using the function `get_maximal_cliques`, which uses the Bron-Kerbosch algorithm described below:

Algorithm 3 Bron-Kerbosch Algorithm for Finding Maximal Cliques

```

1: procedure BRON_KERBOSCH(current_clique, candidates, excluded, maximal_cliques)
2:   if candidates is empty and excluded is empty then
3:     Add current_clique to maximal_cliques
4:     return
5:   end if
6:   for each vertex  $v$  in candidates do
7:     Bron_Kerbosch(current_clique  $\cup \{v\}$ ,
8:       candidates  $\cap$  Neighbors( $v$ ),
9:       excluded  $\cap$  Neighbors( $v$ ),
10:      maximal_cliques)
11:   Remove  $v$  from candidates
12:   Add  $v$  to excluded
13:   end for
14: end procedure

```

2 Junction Tree Construction

This step is implemented in the `get_junction_tree` function. We use the maximal cliques obtained after the triangulation process to create the junction tree while maintaining the running intersection property. Each clique in this tree retains its assigned potential values. Consider the given triangulated graph:

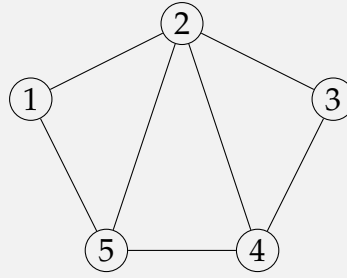
Algorithm 4 Constructing a Junction Tree from Maximal Cliques

Require: Set of maximal cliques \mathcal{C} **Ensure:** Junction tree satisfying the running intersection property

```

1: Initialize an empty priority queue  $T$ 
2: for all pairs  $(C_1, C_2)$  in  $\mathcal{C}$  do
3:   Compute the intersection set  $S = C_1 \cap C_2$ 
4:   if  $S \neq \emptyset$  then
5:     Compute weight  $w = |S|$ 
6:     Push  $(-w, C_1, C_2)$  onto  $T$  (negative weight for max heap)
7:   end if
8: end for
9:
10: Initialize  $parent[C] = C$  and  $rank[C] = 0$  for all  $C \in \mathcal{C}$ 
11:
12: function FINDPARENT( $C$ )
13:   if  $parent[C] \neq C$  then
14:      $parent[C] \leftarrow \text{FindParent}(parent[C])$ 
15:   end if
16:   return  $parent[C]$ 
17: end function
18:
19: function UNION( $C_1, C_2$ )
20:    $root_1 \leftarrow \text{FindParent}(C_1)$ 
21:    $root_2 \leftarrow \text{FindParent}(C_2)$ 
22:   if  $root_1 \neq root_2$  then
23:     if  $rank[root_1] > rank[root_2]$  then
24:        $parent[root_2] \leftarrow root_1$ 
25:     else if  $rank[root_2] > rank[root_1]$  then
26:        $parent[root_1] \leftarrow root_2$ 
27:     else
28:        $parent[root_2] \leftarrow root_1$ 
29:        $rank[root_1] \leftarrow rank[root_1] + 1$ 
30:     end if
31:     return True
32:   end if
33:   return False
34: end function
35:
36: Initialize an empty set  $MST$  (Minimum Spanning Tree)
37: while  $T$  is not empty do
38:   Pop  $(w, C_1, C_2)$  from  $T$ 
39:   if UNION( $C_1, C_2$ ) then
40:     Add edge  $(C_1, C_2)$  to  $MST$ 
41:   end if
42: end while
43: return  $MST$ 

```



Final Triangulated Graph

From the triangulated graph, we obtain the maximal cliques:

$$C_1 = \{1, 2, 5\}, \quad C_2 = \{2, 3, 4\}, \quad C_3 = \{2, 4, 5\},$$

The intersection sets are as follows:

$$C_1 \cap C_3 = \{2, 5\}, \quad |C_1 \cap C_3| = 2$$

$$C_1 \cap C_2 = \{2\}, \quad |C_1 \cap C_2| = 1$$

$$C_3 \cap C_2 = \{2, 4\}, \quad |C_3 \cap C_2| = 2$$

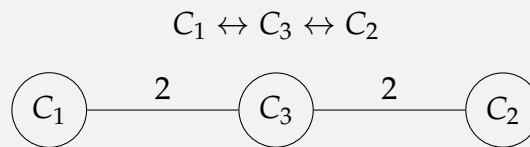
We construct edges with weights corresponding to intersection sizes:

$$(C_1, C_3, 2), \quad (C_3, C_2, 2), \quad (C_1, C_2, 1)$$

Using Kruskal's algorithm:

1. Sort edges: $(C_1, C_3, 2)$, $(C_3, C_2, 2)$, $(C_1, C_2, 1)$.
2. Add (C_1, C_3) (weight 2).
3. Add (C_3, C_2) (weight 2).
4. Ignore (C_1, C_2) (weight 1) as it would form a cycle.

Final Junction Tree:



For any two cliques C_i and C_j containing the same variable X , all cliques along the unique path in the tree must contain X . This holds for all intersections.

3 Marginal Probability

Here is the pseudocode for sharing messages between the maximal cliques of the graph.

Computing the partition function

Firstly, we show how to calculate the Z value for the given graph. So, we first select a root node and then perform a depth-first search (DFS) to calculate the depth of each maximal clique (node in the junction tree). Then, we send messages from the deepest leaves till the root node. Finally, we calculate the partition function using the root clique potential and the received messages. (Messages are being sent again from root to the internal nodes and finally to the leaves, to be used in the next step of computing marginal probabilities.)

Algorithm 5 Computation of Partition Function Z

Require: Graphical Model with maximal cliques and potentials**Ensure:** Partition function Z

```

1: Construct the junction tree  $JT$  from maximal cliques
2: Initialize adjacency list  $JT_{adj}$  from  $JT$ 
3: Select a root clique  $C_{root}$ 
4: Initialize depth map with  $C_{root}$  at depth 0
5: function DFS( $node, parent, depth$ )
6:   for each child in  $JT_{adj}[node]$  do
7:     if child  $\neq$  parent then
8:       Update depth map
9:       Call DFS on child with depth +1
10:    end if
11:  end for
12: end function
13: Perform DFS from  $C_{root}$ 
14: function SENDMESSAGE( $C_{from}, C_{to}$ )
15:   Compute separator set  $S = C_{from} \cap C_{to}$ 
16:   Initialize message vector  $M$  of size  $2^{|S|}$ 
17:   Modify clique potential based on incoming messages
18:   for each state assignment in  $C_{from}$  do
19:     Compute corresponding separator index
20:     Aggregate message value
21:   end for
22:   Store message  $M(C_{from} \rightarrow C_{to})$ 
23: end function
24: Initialize messages dictionary
25: Initialize clique potentials
26: for each clique from deepest to root do
27:   Send messages to parent cliques
28: end for
29: for each clique from root to leaves do
30:   Send messages to child cliques
31: end for
32: Compute partition function  $Z$  using root clique potential and received messages
33: return  $Z$ 

```

Computing the marginal probabilities

Here is the pseudocode for computing the marginal probabilities in the graphical model using message passing. We get the partition function Z from the previous step and use it to normalize the marginal probabilities. We iterate over each variable in the graphical model and find the maximal clique containing it. We then extract the potential function for the clique (using the messages received from neighboring cliques) and compute the marginal probability for the variable. (For every variable, we need to do this only for one clique containing it, as the marginal probability will be the same in all maximal cliques containing the variable.)

Algorithm 6 Computation of Marginal Probabilities

Require: Graphical Model with maximal cliques, clique potentials, and messages

Ensure: Marginal probabilities for each variable

```

1: Initialize adjacency list for junction tree
2: Retrieve partition function  $Z$  using previously computed values
3: Initialize marginal probability list  $M$  with zeros
4: for each variable  $X_i$  in the graphical model do
5:   Find a maximal clique  $C$  containing  $X_i$ 
6:   Extract the potential function for clique  $C$ 
7:   for each neighboring clique  $C'$  of  $C$  do
8:     Compute separator set  $S = C \cap C'$ 
9:     Retrieve message  $M(C' \rightarrow C)$ 
10:    for each assignment in  $C$  do
11:      Identify corresponding index in  $S$ 
12:      Multiply message values with clique potential
13:    end for
14:  end for
15:  Compute marginal probability for  $X_i$ 
16:  Normalize values using  $Z$ 
17: end for
18: return  $M$ 

```

4 Finding the Most Probable Assignment

Here is the pseudocode for finding the top-K most probable assignments in the graphical model. The value of the partition function Z is used from the previous step. Similar to the previous step, we do a DFS to calculate the depth of each maximal clique. We then send messages from the deepest leaves to the root node. This time the messages which are being send are of a different format, and they contain the following two values:

- The union of all sets of variables seen so far (from its descendants and including itself).
- Mapping between the assignments of the variables in the union set, and the corresponding product of potentials.

Moreover, to optimize, we are not sending all the assignments in the message, but only the top-K most probable assignments, where top-K is being performed over all the variables present in the current maximal clique, but not in the descendants. Finally, we send a message from the root node to “None”, which basically collects the assignments of all the variables in the graphical model. We then normalize the probabilities and return the top-K most probable assignments.

Algorithm 7 Compute Top-K Most Probable Assignments

```

1: procedure COMPUTETOPK()
2:   junction_tree  $\leftarrow$  GetJunctionTree()
3:   junction_tree_adj_list  $\leftarrow \emptyset$ 
4:   for each edge in junction_tree do
5:     (a, b)  $\leftarrow$  edge
6:     Add b to junction_tree_adj_list[a]
7:     Add a to junction_tree_adj_list[b]
8:   end for
9:   root  $\leftarrow$  tuple of maximal cliques[0]
10:  depth_map[root]  $\leftarrow$  0
11:  procedure DFS(node, parent, depth)
12:    for each child in junction_tree_adj_list[node] do
13:      if child  $\neq$  parent then
14:        depth_map[child]  $\leftarrow$  depth
15:        DFS(child, node, depth + 1)
16:      end if
17:    end for
18:  end procedure
19:  DFS(root, None, 1)
20:  procedure SENDMESSAGE(from_clique, to_clique, parent_map, clique_potentials, messages)
21:    variables_seen  $\leftarrow$  Set of variables in from_clique
22:    for each neighbor in parent_map[from_clique] do
23:      if neighbor  $\neq$  to_clique and (neighbor, from_clique)  $\in$  messages then
24:        variables_seen  $\leftarrow$  variables_seen  $\cup$  neighbor
25:      end if
26:    end for
27:    Initialize message_to_send with ones of size  $2^{|variables\_seen|}$ 
28:    Convert variables_seen to list list_variables_seen
29:    from_potential  $\leftarrow$  clique_potentials[from_clique]
30:    for each i in range  $2^{|variables\_seen|}$  do
31:      Compute binary assignment for i
32:      Compute from_potential_index based on from_clique
33:      Multiply message_to_send[i] by from_potential[from_potential_index]
34:      for each neighbor in parent_map[from_clique] do
35:        if neighbor  $\neq$  to_clique and (neighbor, from_clique)  $\in$  messages then
36:          Multiply message_to_send[i] by incoming message
37:        end if
38:      end for
39:    end for
40:    messages[(from_clique, to_clique)]  $\leftarrow$  message_to_send
41:  end procedure
42:  messages  $\leftarrow \emptyset$ 
43:  clique_potentials  $\leftarrow$  self.clique_potentials
44:  max_depth  $\leftarrow$  max(depth_map.values())
45:  for depth from max_depth to 0 do
46:    for each (clique, d) in depth_map do
47:      if d == depth then
48:        Find parent p of clique
49:        for each p do
50:          SENDMESSAGE(clique, p, junction_tree_adj_list, clique_potentials, messages)
51:        end for
52:      end if
53:    end for
54:  end for
55:  SENDMESSAGE(root, None, junction_tree_adj_list, clique_potentials, messages)
56:  message_final  $\leftarrow$  messages[(root, None)]
57:  assignments  $\leftarrow$  all possible binary assignments of size num_variables
58:  probabilities  $\leftarrow$  message_final[1]
59:  for each i in probabilities do
60:    probabilities[i]  $\leftarrow$  probabilities[i] / self.z
61:  end for
62:  assignment_prob_pairs  $\leftarrow$  Zip(assignments, probabilities)
63:  Sort assignment_prob_pairs in decreasing order of probability
64:  return first k.value elements of assignment_prob_pairs
65: end procedure

```