CS726 Programming Assignment – 1 Report

Saksham Rathi (22B1003)
Sharvanee 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

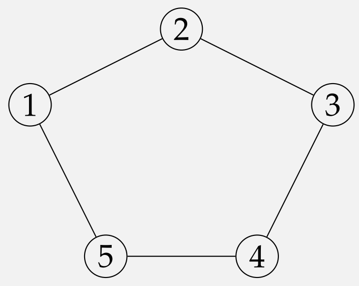
```
1: cycles ← Find all cycles in graph
2: for each cycle in cycles do
3:     if length of cycle $\geq 4$ then
4:         if there is no shortcut (vertices connected by non-cycle edge) then
5:             return False
6:         end if
7:     end if
8: end for
9: return True
```

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:

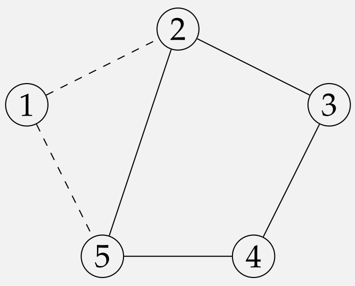
Algorithm 2 Triangulation Process

```
1: vertices_left ← Set of all vertices
2: while vertices_left is not empty do
3:     vertex ← 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
```

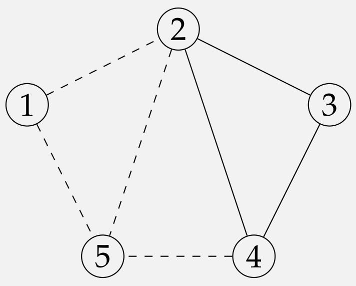
The figure below shows the run of the triangulation algorithm on an example graph:



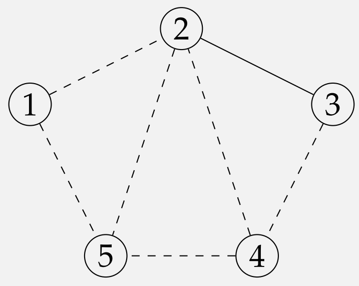
Step 1: Initial Graph



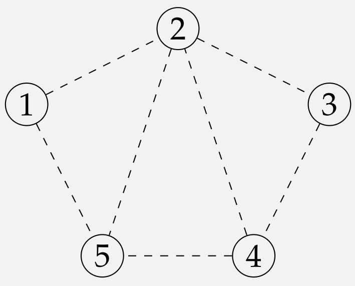
Step 2: Select 1, add (5,2), remove 1



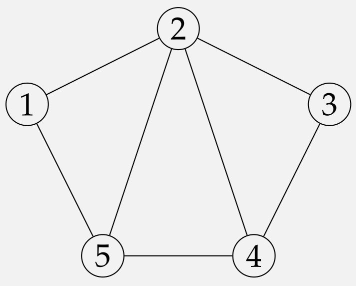
Step 3: Select 5, add (4,2), remove 5



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

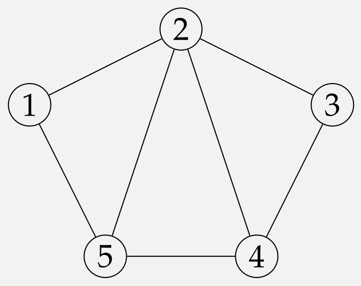
```
procedure Bron_Kerbosch(current_clique, candidates, excluded, maximal_cliques)
    if candidates is empty and excluded is empty then
        Add current_clique to maximal_cliques
        return
    end if
    for each vertex $v$ in candidates do
        Bron_Kerbosch(current_clique $\cup \{v\}$,
                candidates $\cap$ Neighbors($v$),
                excluded $\cap$ Neighbors($v$),
                maximal_cliques)
        Remove $v$ from candidates
        Add $v$ to excluded
    end for
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:

$$\begin{aligned} C_1 \cap C_3 &= \{2,5\}, \quad |C_1 \cap C_3| = 2 \\ C_1 \cap C_2 &= \{2\}, \quad |C_1 \cap C_2| = 1 \\ C_3 \cap C_2 &= \{2,4\}, \quad |C_3 \cap C_2| = 2 \end{aligned}$$

We construct edges with weights corresponding to intersection sizes:

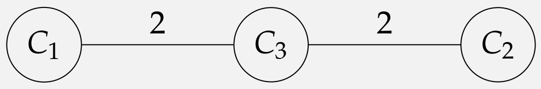
$$(C_1, C_3, 2), \quad (C_3, C_2, 2), \quad (C_1, C_2, 1)$$

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:

$$C_1 \leftrightarrow C_3 \leftrightarrow C_2$$



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 $M(C_{from} \to C_{to})$
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 $M(C' \to C)$
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.

Optimization Step: We maintain a set of extra variables, which are present in the clique from which a message is sent as compared to the clique to which the message is being sent. For all possible assignments of the extra variables, we only pick the top-K assignments for all other variables. This optimizes the algorithm by reducing the number of assignments being sent in the message.

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 Assignments in Graphical Model

```
1: Input: Graphical model with junction tree, clique potentials, $k$ value
2: Output: Top-$k$ assignments with highest probabilities
3: $junction\_tree \leftarrow$ get the junction tree
4: Initialize $junction\_tree\_adj\_list$ as empty dictionary
5: for each edge $(a, b)$ in $junction\_tree$ do
6:     Add $b$ to adjacency list of $a$ and vice versa
7: end for
8: $root \leftarrow$ any maximal clique as the root
9: Initialize $depth\_map$ with $root$ having depth 0
10: procedure DFS($node, parent, depth$)
11:     for each child in adjacency list of $node$ do
12:         if child $\neq$ parent then
13:             Set depth of $child$ and call DFS recursively
14:         end if
15:     end for
16: end procedure
17: Call DFS($root$, None, 1)
18: procedure SENDMESSAGE($from\_clique, to\_clique, parent\_map, clique\_potentials, messages$)
19:     Compute variables seen and neighboring variables
20:     Initialize message with uniform probability distribution
21:     for each assignment in possible values of seen variables do
22:         Compute potential index and update probability
23:         Multiply with incoming messages from neighboring cliques
24:     end for
25:     For each of the extra variables in the current clique (as compared to the clique to which we
    are sending the message, we pick only the top-K assignments for all other variables, this is
    essentially the optimization of the algorithm)
26:     Store the computed message in $messages$
27: end procedure
28: Initialize $messages$ as an empty dictionary
29: $clique\_potentials \leftarrow$ get clique potentials
30: $max\_depth \leftarrow$ maximum depth in $depth\_map$
31: for $depth = max\_depth$ down to 0 do
32:     for each clique at depth $depth$ do
33:         Identify parent cliques
34:         for each parent clique do
35:             Call SendMessage with $(clique, parent, ...)$
36:         end for
37:     end for
38: end for
39: Call SendMessage with $(root, \text{None}, ...)$
40: $message\_final \leftarrow$ computed message from root
41: Normalize probabilities using partition function $Z$
42: Sort assignments by probability in descending order
43: Return top-$k$ assignments
```