CS726 Scribe Notes

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

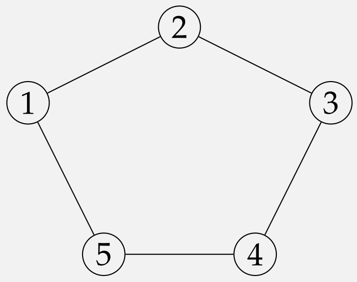
```
cycles ← Find all cycles in graph
for each cycle in cycles do
    if length of cycle $\geq 4$ then
        if there is no shortcut (vertices connected by non-cycle edge) then
            return False
        end if
    end if
end for
return True
```

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:

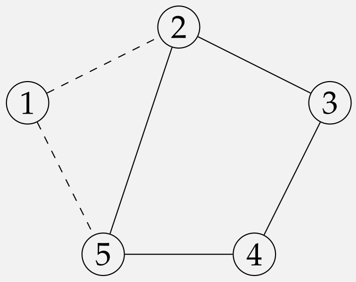
Algorithm 2 Triangulation Process

```
vertices_left ← Set of all vertices
while vertices_left is not empty do
    vertex ← Vertex in vertices_left with minimum degree
    for each pair $(i, j)$ of neighbours of vertex do
        if the graph does not contain an edge between $i$ and $j$ then
            Add edge $(i, j)$ to the original graph
        end if
    end for
    Remove vertex from vertices_left
    Update graph by removing vertex and updating degrees and edges
end while
```

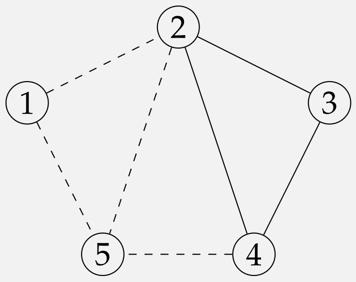
The figure below shows the run of the triangulation algorithm on an example graph:



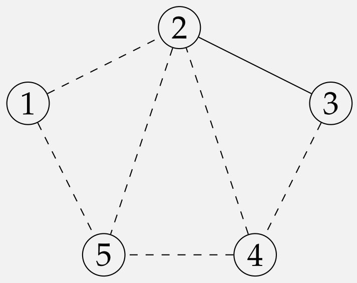
Step 1: Initial Graph



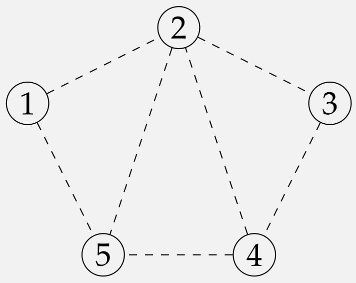
Step 2: Select 1, add (5,2), remove 1



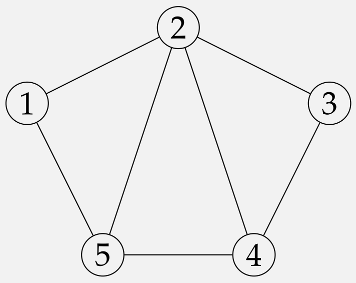
Step 3: Select 5, add (4,2), remove 5



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

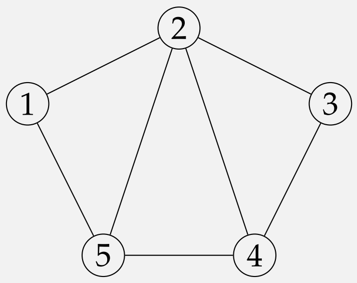
```
Require: Set of maximal cliques $\mathcal{C}$
Ensure: Junction tree satisfying the running intersection property
Initialize an empty priority queue $T$
for all pairs $(C_1, C_2)$ in $\mathcal{C}$ do
    Compute the intersection set $S = C_1 \cap C_2$
    if $S \neq \emptyset$ then
        Compute weight $w = |S|$
        Push $(-w, C_1, C_2)$ onto $T$ (negative weight for max heap)
    end if
end for

Initialize $parent[C] = C$ and $rank[C] = 0$ for all $C \in \mathcal{C}$

function FINDPARENT(C)
    if $parent[C] \neq C$ then
        $parent[C] \leftarrow \text{FindParent}(parent[C])$
    end if
    return $parent[C]$
end function

function UNION$(C_1, C_2)$
    $root_1 \leftarrow \text{FindParent}(C_1)$
    $root_2 \leftarrow \text{FindParent}(C_2)$
    if $root_1 \neq root_2$ then
        if $rank[root_1] > rank[root_2]$ then
            $parent[root_2] \leftarrow root_1$
        else if $rank[root_2] > rank[root_1]$ then
            $parent[root_1] \leftarrow root_2$
        else
            $parent[root_2] \leftarrow root_1$
            $rank[root_1] \leftarrow rank[root_1] + 1$
        end if
        return True
    end if
    return False
end function

Initialize an empty set $MST$ (Minimum Spanning Tree)
while $T$ is not empty do
    Pop $(w, C_1, C_2)$ from $T$
    if UNION$(C_1, C_2)$ then
        Add edge $(C_1, C_2)$ to $MST$
    end if
end while
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:

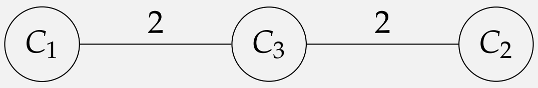
$$(C_1, C_3, 2), \quad (C_3, C_2, 2), \quad (C_1, C_2, 1)$$

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.

Firstly, we show how to calculate the Z value for the given graph.

Here is the pseudocode for computing the marginal probabilities in the graphical model using message passing.

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 $Z$
```

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

Algorithm 7 Compute Top-K Most Probable Assignments

```
procedure COMPUTETOPK( )
    $junction\_tree \leftarrow$ GetJunctionTree()
    $junction\_tree\_adj\_list \leftarrow \emptyset$
    for each $edge$ in $junction\_tree$ do
        $(a, b) \leftarrow edge$
        Add $b$ to $junction\_tree\_adj\_list[a]$
        Add $a$ to $junction\_tree\_adj\_list[b]$
    end for
    $root \leftarrow$ tuple of maximal cliques[0]
    $depth\_map[root] \leftarrow 0$
    procedure DFS($node, parent, depth$)
        for each $child$ in $junction\_tree\_adj\_list[node]$ do
            if $child \neq parent$ then
                $depth\_map[child] \leftarrow depth$
                DFS($child, node, depth + 1$)
            end if
        end for
    end procedure
    DFS($root$, None, 1)
    procedure SENDMESSAGE($from\_clique, to\_clique, parent\_map, clique\_potentials, messages$)
        $variables\_seen \leftarrow$ Set of variables in $from\_clique$
        for each $neighbor$ in $parent\_map[from\_clique]$ do
            if $neighbor \neq to\_clique$ and $(neighbor, from\_clique) \in messages$ then
                $variables\_seen \leftarrow variables\_seen \cup neighbor$
            end if
        end for
        Initialize $message\_to\_send$ with ones of size $2^{|variables\_seen|}$
        Convert $variables\_seen$ to list $list\_variables\_seen$
        $from\_potential \leftarrow clique\_potentials[from\_clique]$
        for each $i$ in range $2^{|variables\_seen|}$ do
            Compute binary assignment for $i$
            Compute $from\_potential\_index$ based on $from\_clique$
            Multiply $message\_to\_send[i]$ by $from\_potential[from\_potential\_index]$
            for each $neighbor$ in $parent\_map[from\_clique]$ do
                if $neighbor \neq to\_clique$ and $(neighbor, from\_clique) \in messages$ then
                    Multiply $message\_to\_send[i]$ by incoming message
                end if
            end for
        end for
        $messages[(from\_clique, to\_clique)] \leftarrow message\_to\_send$
    end procedure
    $messages \leftarrow \emptyset$
    $clique\_potentials \leftarrow self.clique\_potentials$
    $max\_depth \leftarrow \max(depth\_map.values())$
    for depth from $max\_depth$ to 0 do
        for each $(clique, d)$ in $depth\_map$ do
            if $d == depth$ then
                Find parent $p$ of $clique$
                for each $p$ do
                    SENDMESSAGE($clique, p, junction\_tree\_adj\_list, clique\_potentials, messages$)
                end for
            end if
        end for
    end for
    SENDMESSAGE($root$, None, $junction\_tree\_adj\_list, clique\_potentials, messages$)
    $message\_final \leftarrow messages[(root, None)]$
    $assignments \leftarrow$ all possible binary assignments of size $num\_variables$
    $probabilities \leftarrow message\_final[1]$
    for each $i$ in $probabilities$ do
        $probabilities[i] \leftarrow probabilities[i]/self.z$
    end for
    $assignment\_prob\_pairs \leftarrow$ Zip(assignments, probabilities)
    Sort $assignment\_prob\_pairs$ in decreasing order of probability
    return first $k\_value$ elements of $assignment\_prob\_pairs$
end procedure
```