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

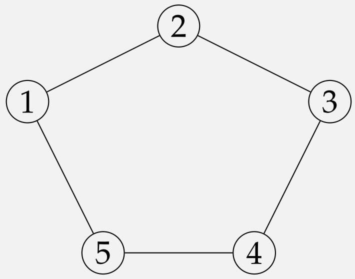
```
cycles ← Find all cycles in graph
for each cycle in cycles do
    if length of cycle ≥ 4 then
        if there is no shortcut (vertices connected by non-cycle edge) then
            return False
        end if
    end if
end for
return True
```

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:

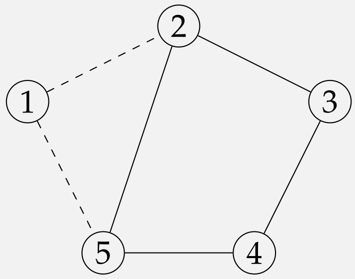
Algorithm 2 Triangulation Process

```
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
```

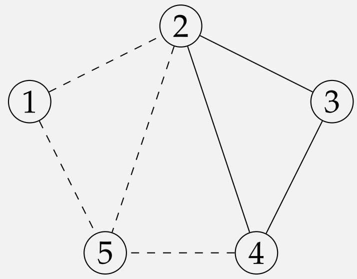
The figure below shows the run of the triangulation algorithm on an example graph:



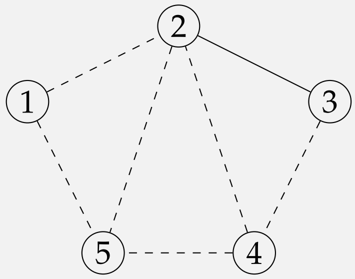
Step 1: Initial Graph



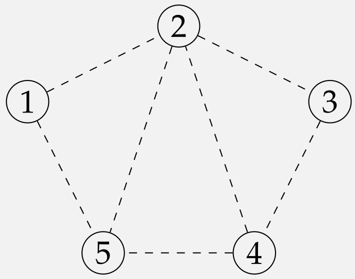
Step 2: Select 1, add (5,2), remove 1



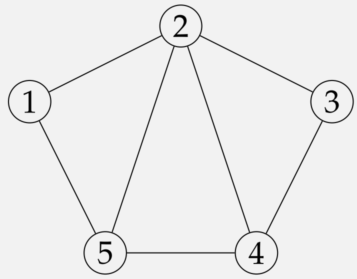
Step 3: Select 5, add (4,2), remove 5



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

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

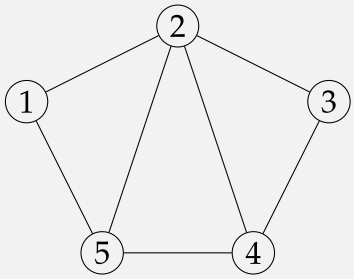
Initialize an empty priority queue T
for all pairs (C_1, C_2) in $\mathcal{C}$ **do**
    Compute the intersection set $S = C_1 \cap C_2$
    if $S \neq \emptyset$ **then**
        Compute weight $w = |S|$
        Push $(-w, C_1, C_2)$ onto T (negative weight for max heap)
    end if
end for

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:

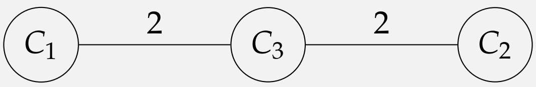
$$(C_1, C_3, 2), \quad (C_3, C_2, 2), \quad (C_1, C_2, 1)$$

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.

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

Construct the junction tree JT from maximal cliques
Initialize adjacency list JT_{adj} from JT
Select a root clique C_{root}
Initialize depth map with C_{root} at depth 0
function DFS($node, parent, depth$)
    for each child in $JT_{adj}[node]$ **do**
        if child $\neq$ parent **then**
            Update depth map
            Call DFS on child with depth $+1$
        end if
    end for
end function
Perform DFS from C_{root}
function SENDMESSAGE(C_{from}, C_{to})
    Compute separator set $S = C_{from} \cap C_{to}$
    Initialize message vector M of size $2^{|S|}$
    Modify clique potential based on incoming messages
    for each state assignment in C_{from} **do**
        Compute corresponding separator index
        Aggregate message value
    end for
    Store message $M(C_{from} \rightarrow C_{to})$
end function
Initialize messages dictionary
Initialize clique potentials
for each clique from deepest to root **do**
    Send messages to parent cliques
end for
for each clique from root to leaves **do**
    Send messages to child cliques
end for
Compute partition function Z using root clique potential and received messages
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

```
Initialize adjacency list for junction tree
Retrieve partition function Z using previously computed values
Initialize marginal probability list M with zeros
for each variable X_i in the graphical model do
    Find a maximal clique C containing X_i
    Extract the potential function for clique C
    for each neighboring clique C' of C do
        Compute separator set S = C ∩ C'
        Retrieve message M(C' → C)
        for each assignment in C do
            Identify corresponding index in S
            Multiply message values with clique potential
        end for
    end for
    Compute marginal probability for X_i
    Normalize values using Z
end for
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

```
procedure COMPUTETOPK( )
    junction_tree ← GetJunctionTree()
    junction_tree_adj_list ← ∅
    for each edge in junction_tree do
        (a, b) ← edge
        Add b to junction_tree_adj_list[a]
        Add a to junction_tree_adj_list[b]
    end for
    root ← tuple of maximal cliques[0]
    depth_map[root] ← 0
    procedure DFS(node, parent, depth)
        for each child in junction_tree_adj_list[node] do
            if child ≠ parent then
                depth_map[child] ← 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 ← Set of variables in from_clique
        for each neighbor in parent_map[from_clique] do
            if neighbor ≠ to_clique and (neighbor, from_clique) ∈ messages then
                variables_seen ← variables_seen ∪ 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 ← 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 ≠ to_clique and (neighbor, from_clique) ∈ messages then
                    Multiply message_to_send[i] by incoming message
                end if
            end for
        end for
        messages[(from_clique, to_clique)] ← message_to_send
    end procedure
    messages ← ∅
    clique_potentials ← self.clique_potentials
    max_depth ← 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 ← messages[(root, None)]
    assignments ← all possible binary assignments of size num_variables
    probabilities ← message_final[1]
    for each i in probabilities do
        probabilities[i] ← probabilities[i]/self.z
    end for
    assignment_prob_pairs ← Zip(assignments, probabilities)
    Sort assignment_prob_pairs in decreasing order of probability
    return first k_value elements of assignment_prob_pairs
end procedure
```