CS 726: Programming Assignment 1

Deeptanshu Malu   Deevyanshu Malu   Neel Rambhia

1 Preprocessing

The preprocessing step involves converting the given data into a suitable format for further processing. We have done the following preprocessing steps:

1. **Clique Potentials:** Converted all the cliques and potentials data into a dictionary format. The keys of the dictionary are the cliques and the values are the potentials. If a clique already exists in the dictionary, we multiply the potential with the existing potential.

2. **Edges:** Converted all the cliques into a set of tuples, where each tuple consists of two nodes that are connected by an edge.

3. **Nodes:** Converted the edge set into a set of all the nodes in the graph.

2 Triangulation

2.1 triangulate_and_get_cliques

Here are some of the functions we have defined for the triangulation and maximum clique finding process:

1. **is_simplicial:** This function checks if a given node is simplicial in the graph. A node is simplicial if the neighbors of the node form a clique.

Algorithm 1 Check if a vertex is simplicial

```
function IsSimplicial(adjlist, node)
    neighbors ← adjlist[node]
    for i ← 1 to |neighbors| do
        for j ← i + 1 to |neighbors| do
            if neighbors[j] ∉ adjlist[neighbors[i]] then
                return False
            end if
        end for
    end for
    return True
end function
```

2. **find_simplicial_vertex:** This function finds a simplicial vertex in the graph. If no simplicial vertex is found, it returns None.

Algorithm 2 Find a simplicial vertex

```
function FindSimplicialVertex(adjlist)
    for node in adjlist do
        if IsSimplicial(adjlist, node) then
            return node
        end if
    end for
    return None
end function
```

3. **make_vertex_simplicial:** This function makes a given node simplicial by adding edges between all the neighbors of the node.

Algorithm 3 Make a vertex simplicial

```
function MakeVertexSimplicial(adjlist, node)
    neighbors ← adjlist[node]
    for i ← 1 to |neighbors| do
        for j ← i + 1 to |neighbors| do
            if neighbors[j] ∉ adjlist[neighbors[i]] then
                adjlist[neighbors[i]].append(neighbors[j])
                adjlist[neighbors[j]].append(neighbors[i])
            end if
        end for
    end for
    return adjlist
end function
```

4. **chordal_graph_with_heuristic:** In this function we do the following:

   (a) Find a simplicial vertex in the graph.

   (b) If a simplicial vertex is found, take its neighbors and store it in the maximum clique list.

   (c) If no simplicial vertex is found, take a node based on the heuristic that the node has the minimum degree. Make this node simplicial and take its neighbors and store it in the maximum clique list.

   (d) Repeat the process until all the nodes are covered.

   (e) If a clique found is already a subset of a previously found clique, discard it.

   (f) Return the maximal cliques found.

Algorithm 4 Compute chordal graph with heuristic

```
function ChordalGraphWithHeuristic(adjlist)
    chordal_adjlist ← deepcopy(adjlist)
    elimination_order, cliques ← [], []
    while |elimination_order| < |adjlist| do
        node ← FindSimplicialVertex(chordal_adjlist)
        if node = None then
            node ← arg min{|neighbors| : node ∈ chordal_adjlist}
            chordal_adjlist ← MakeVertexSimplicial(chordal_adjlist, node)
        end if
        elimination_order.append(node)
        clique ← {node} ∪ {n | n ∈ chordal_adjlist[node], ∀i ∈ clique, n ∈ chordal_adjlist[i]}
        cliques.append(clique)
        for neighbor ∈ chordal_adjlist[node] do
            chordal_adjlist[neighbor].remove(node)
        end for
        delete chordal_adjlist[node]
    end while
    return FilterMaximalCliques(cliques)
end function
```

3 Junction Tree Construction

3.1 get_junction_tree

In this section, we have defined the following functions for constructing the junction tree:

1. **create_junction_graph**: This function creates a junction graph from the found maximal cliques. We iterate over all the maximal cliques and add an edge between two maximal cliques if they have a common node. We also assign a weight to each edge which is the size of the intersection of the two maximal cliques.

Algorithm 5 Create Junction Graph

```
function CreateJunctionGraph(cliques)
    J ← {}
    for C1, C2 ∈ cliques, C1 ≠ C2 do
        I ← C1 ∩ C2
        if I ≠ ∅ then
            w ← |I|
            J[C1] ← J[C1] ∪ {(C2, w)}
            J[C2] ← J[C2] ∪ {(C1, w)}
        end if
    end for
    return J
end function
```

2. **make_junction_tree**: This function creates a junction tree from the junction

graph. We first find the maximum spanning tree of the junction graph using Kruskal's algorithm. This tree is the junction tree.

Algorithm 6 Make Junction Tree

function MAKEJUNCTIONTREE(J)
  $E \leftarrow$ sorted edges of J by weight (descending)
  Initialize $parent, rank$ for union-find
  $T \leftarrow \{\}$
  for $(C_1, C_2, w) \in E$ **do**
    if FIND(C_1) $\neq$ FIND(C_2) **then**
      UNION(C_1, C_2)
      $T[C_1] \leftarrow T[C_1] \cup \{C_2\}$
      $T[C_2] \leftarrow T[C_2] \cup \{C_1\}$
    end if
  end for
  return T
end function

3. **get_message_passing_order**: This function gets the order in which the messages should be passed in the junction tree. We do a depth-first search on the junction tree and store the order in which the nodes are visited. Here first a tree is created from the junction tree and then a post-order traversal and a pre-order traversal is done to get the message passing order.

Algorithm 7 Get Message Passing Order

function GETMESSAGEORDER($T, root$)
  Initialize D, P with BFS from $root$
  $C2P, P2C \leftarrow []$
  procedure POSTORDER($node$)
    for $child \in D[node]$ **do**
      POSTORDER($child$)
      $C2P.append((child, node))$
    end for
  end procedure
  procedure PREORDER($node$)
    for $child \in D[node]$ **do**
      $P2C.append((node, child))$
      PREORDER($child$)
    end for
  end procedure
  POSTORDER($root$), PREORDER($root$)
  return $C2P + P2C$
end function

3.2 assign_potentials_to_cliques

In this section, we have defined the following functions for assigning potentials to the cliques:

1. **multiply_potentials**: This function takes two potentials and there corresponding cliques and multiplies them to get a new potential and a new clique.

Algorithm 8 Multiply Potentials

```
function MultiplyPotentials(P_1, P_2, D_1, D_2)
    D ← D_1 ∪ D_2
    P_new ← array of size 2^|D| initialized to None
    for val ∈ {0, 1}^|D| do
        A ← mapping of D to val
        i_1 ← index of A restricted to D_1
        i_2 ← index of A restricted to D_2
        i_new ← index of A over D
        P_new[i_new] ← P_1[i_1] × P_2[i_2]
    end for
    return (P_new, D)
end function
```

After this we iterate over all the maximal cliques and for each clique, we find all the clique potentials that are subsets of the maximal clique. We multiply all these potentials to get the potential for the maximal clique.

Algorithm 9 Compute Maximum Clique Potentials

```
MCP ← {}                                                ▷ Stores max clique potentials
VC ← {}                                                 ▷ Tracks visited cliques
CA ← {}                                                 ▷ Stores assigned cliques
for MC ∈ max_cliques do
    SC ← {}                                             ▷ Subset cliques of MC
    CA[MC] ← []
    for C ∈ clique_potentials do
        if C ⊆ MC and C ∉ VC then
            SC ← SC ∪ {C}
            VC ← VC ∪ {C}
            CA[MC] ← CA[MC] ∪ {C}
        end if
    end for
    (P, D) ← (clique_potentials[first(SC)], first(SC))
    for C ∈ SC[1 :] do
        (P, D) ← MultiplyPotentials(P, clique_potentials[C], D, C)
    end for
    USC ← ⋃ SC                                          ▷ Union of subset cliques
    LN ← MC − USC                                       ▷ Remaining nodes
    for N ∈ LN do
        (P, D) ← MultiplyPotentials(P, [1, 1], D, {N})
    end for
    MCP[MC] ← P
end for
```

4 Marginal Probability

4.1 get_z_value

In this section, we have defined the following functions for finding the Z value:

1. **create_empty_message_dict**: This function creates an empty message dictionary for all the cliques in the junction tree. The dictionary has the sender clique as the key and the value is a dictionary with the receiver clique as the key and its value is the message from the sender clique to the receiver clique.

Algorithm 10 Create Empty Message Dictionary

function CreateEmptyMessageDict(J)
    $M \leftarrow \{\}$
    for $N \in J$ **do**
        $M[N] \leftarrow \{\}$
        for $N' \in J[N]$ **do**
            $M[N][N'] \leftarrow$ None
        end for
    end for
    return M
end function

2. **multiply_messages**: This function takes a potential of the clique or a precomputed value (potential is already multiplied with some messages) and a message to be multiplied as input and it multiplies them to get a new potential.

Algorithm 11 Multiply Messages

function MultiplyMessages(P, M, N, D)
    $P' \leftarrow$ copy of P
    for $val \in \{0,1\}^{|N|}$ **do**
        $A \leftarrow$ mapping of N to val
        $i_m \leftarrow$ index of A restricted to D
        $i_p \leftarrow$ index of A over N
        $P'[i_p] \leftarrow P'[i_p] \times M[i_m]$
    end for
    return P'
end function

3. **condense_message**: This function takes a potential and the nodes to be marginalized on and marginalizes the potential to get a new potential.

4. **message_passing_opt_order**: This function takes the message passing order and the junction tree and passes the messages in the optimal order. It updates the message dictionary with the new messages.

5. **calc_z**: This function calculates the Z value using the fully updated message dictionary. We simply choose any maximal clique and marginalize it on all it component nodes to get the Z value.

Algorithm 12 Condense Message

function CondenseMessage(P, N, C, S)
    $P' \leftarrow []$
    for $val \in \{0,1\}^{|C|}$ **do**
        $sum\_val \leftarrow 0$
        for $val' \in \{0,1\}^{|N|-|C|}$ **do**
            $A \leftarrow$ merge of $C \mapsto val$ and $S \mapsto val'$
            $i \leftarrow$ index of A over N
            $sum\_val \leftarrow sum\_val + P[i]$
        end for
        Append sum_val to P'
    end for
    return P'
end function

Algorithm 13 Calculate Partition Function Z

function CalcZ(M, P)
    $N \leftarrow$ first key in P
    $Z \leftarrow P[N]$
    for $N' \in M[N]$ **do**
        $Z \leftarrow$ MultiplyMessages($Z, M[N'][N], N, N \cap N'$)
    end for
    $Z \leftarrow$ CondenseMessage($Z, N, [], N$)
    return $Z[0]$
end function

Algorithm 14 Message Passing with Optimized Order

function MessagePassingOptOrder(J, P, O)
    $M \leftarrow$ CreateEmptyMessageDict(J)
    for $(C, N) \in O$ **do**
        $P' \leftarrow$ copy of $P[C]$
        for $N' \in J[C] \setminus \{N\}$ **do**
            $P' \leftarrow$ MultiplyMessages($P', M[N'][C], C, C \cap N'$)
        end for
        $D \leftarrow C \setminus N$
        $S \leftarrow C \setminus D$
        $M[C][N] \leftarrow$ CondenseMessage(P', C, S, D)
    end for
    return M
end function

4.2 compute_marginals

In this section, we have defined the following functions for computing the marginals:

1. **multiply_messages**: Same as defined before
2. **condense_message**: Same as defined before

After this we iterate over each node. For each node we find a maximal clique that contains the node. We then multiply all the messages from the maximal clique to the node to get the marginal probability.

Algorithm 15 Compute Marginals

$M \leftarrow$ list of size $|N|$ initialized to $[0, 0]$
for i, n in N **do**
  $C \leftarrow$ first clique in C_{max} containing n
  $P \leftarrow$ copy of $P[C]$
  for $N' \in J[C]$ **do**
    $P \leftarrow \textsc{MultiplyMessages}(P, M_{dict}[N'][C], C, C \cap N')$
  end for
  $P \leftarrow \textsc{CondenseMessage}(P, C, \{n\}, C \setminus \{n\})$
  $M[i][0] \leftarrow P[0]/Z$
  $M[i][1] \leftarrow P[1]/Z$
end for
return M

5 MAP Assignment

In this section we have to find the MAP assignment. This can be done by putting k=1 in the top k assignments function (compute_top_k).

6 Top k Assignments

In this section, we have defined the following functions for computing the top k assignments:

1. **create_empty_message_dict**: Same as defined before
2. **multiply_messages**: This is similar to the previous function but it also keeps track of the assignments and potential values for each assignment.

Algorithm 16 Multiply Messages

```
function MultiplyMessages(potential, message, node, mesg_depends_on)
    potential_new ← empty list
    for val in all combinations of [0, 1] for node do
        assignmt ← map node to val
        targ_idx ← index from assignmt for mesg_depends_on
        node_idx ← index from assignmt for node
        for pot_assgn in potential[node_idx] do
            for mesg_assgn in message[targ_idx] do
                if assignments match on mesg_depends_on then
                    new_assgn ← combine pot_assgn and mesg_assgn
                    Append new_assgn to potential_new[node_idx]
                end if
            end for
        end for
    end for
    return potential_new
end function
```

3. **condense_message**: Similar to the previous function but it also keeps track of the top k assignments and potential values for each assignment.
4. **message_passing_opt_order_topk**: This function is similar to the previous function but it also keeps track of the top k assignments and potential values for each assignment.

Finally, we take a sample maximal clique and multiply all the messages from the maximal clique to the node to get the top k assignments.

Algorithm 17 Condense Message

```
function CondenseMessage(potential, node, compl_to_topk_on, to_topk_on)
    new_potential ← empty list
    for val in all combinations of [0, 1] for compl_to_topk_on do
        topk_things ← empty list
        for val1 in all combinations of [0, 1] for to_topk_on do
            assignmt ← combine val and val1
            idx ← index from assignmt for node
            for pot_assgn in potential[idx] do
                Update pot_assgn with val1
                Append pot_assgn to topk_things
            end for
        end for
        Sort topk_things and keep top k
        Append topk_things to new_potential
    end for
    return new_potential
end function
```

Algorithm 18 Message Passing with Optimal Order and Top-K

```
function MessagePassingOptOrderTopK(junc_tree, potentials, opt_order)
    message_dict ← CreateEmptyMessageDict(junc_tree)
    for (m_clique, neigh) in opt_order do
        potn_message ← initialize with potentials[m_clique]
        for neigh1 in junc_tree[m_clique] do
            if neigh1 ≠ neigh then
                potn_message ← MultiplyMessages(potn_message,
                    message_dict[neigh1][m_clique], m_clique,
                    intersection of m_clique and neigh1)
            end if
        end for
        message_dict[m_clique][neigh] ← CondenseMessage(
          potn_message, m_clique,
          difference of m_clique and neigh,
          intersection)
    end for
    return message_dict
end function
```

Algorithm 19 Sampling and Returning Top-K Assignments

```
message_dict ← MessagePassingOptOrderTopK(
    junction_tree, max_clique_potentials, opt_order)
sample_node ← first node in max_cliques
sample_node_potential ← initialize with max_clique_potentials[sample_node]
for neigh in junction_tree[sample_node] do
    sample_node_potential ← MultiplyMessages(
            sample_node_potential, message_dict[neigh][sample_node],
            sample_node, intersection of sample_node and neigh)
end for
top_k_assignments ← CondenseMessage(
    sample_node_potential, sample_node, [], sample_node)
top_k_dict_list ← empty list
for assgn in top_k_assignments[0] do
    assign_list ← sorted values from assgn[1]
    dict_entry ← {
            "assignment" : assign_list,
            "probability" : assgn[0]/z
        }
    Append dict_entry to top_k_dict_list
end for
return top_k_dict_list
```

Contributions

- Deeptanshu Malu: Primarily responsible for:
    1. Triangulation implementation and testing.
    2. Junction Graph Construction and validation.
    3. Development and optimization of message and potential multiplication/condensation for Marginal Probability, Z Value, and Top k Assignments.
    4. Implementation of the Message Passing Algorithm.
- Deevyanshu Malu: Primarily responsible for:
    1. Triangulation implementation and testing.
    2. Junction Graph Construction and validation.
    3. Implementation of the potential assignment to cliques.
    4. Development of the algorithm for determining the optimal message passing order.
    5. Development and optimization of message and potential multiplication/condensation for Top k Assignments.

- Neel Rambhia: Primarily responsible for:
    1. Triangulation implementation and testing.
    2. Junction Graph Construction and validation.
    3. Implementation of the potential assignment to cliques.
    4. Development and optimization of message and potential multiplication/condensation for Marginal Probability, Z Value, and Top k Assignments.

Acknowledgements

We used Deepseek to create code for commom algorithms like Kruskal's algorithm and union-find data structure. We used ChatGPT to generate the code to get optimal order for message passing, but not for the actual implementation of message passing. We used GitHub Copilot to assist in writing the code.