Belief Propagation

In this lab session you will implement the belief propagation (BP) algorithm on a factor graph using the pgmpy library. By the end of this session you will be able to

- **Create a factor graph** respresenting a **hidden Markov model**.
- **Answer inference queries** using the built in methods of **pgmpy**.
- **Experiment with** the fundamental concepts behind **probabilistic inference and message passing**.

The Model

We will consider our previous Markov chain model of wheather change over time. For simplicy, we will assume that the **weather is stationary** over the seven days of a week, **and that it can take three different values:** `sunny`, `cloudy` and `rainy`. The probability of rain or clouds after a rainy day is 0.25 and 0.5, respectively. A sunny day is also followed by a sunny day or a cloudy day with probabilities 0.7 and 0.25, respectively. Finally, if a day is cloudy, the next day will also be cloudy with probability 0.35 or it will rain with probability 0.4.

Let's assume that by some unfortunate reason we are not able to see the weather directly, but we can observe if someone makes use of an umbrella. This will be our observed random variable, which directly depends on the actual weather. The probability of carrying an umbrella is 0.95 for a rainy day, 0.6 for a cloudy day, and 0.2 for a sunny day.

Finally, we know that the first day of the week is sunny, cloudy, or rainy with probabilities 0.3, 0.4 and 0.3, respectively.

Creating the Factor graph

The model is composed of 14 random variables, 7 corresponding to the hidden weather state, and 7 corresponding to the binary observed variable. Let's create the factors and variables enconding the transition and observational model.

```
from pgmpy.factors.discrete import DiscreteFactor

weather = ["w1", "w2", "w3", "w4", "w5", "w6", "w7"]
umbrella = ["u1", "u2", "u3", "u4", "u5", "u6", "u7"]
variables = weather + umbrella
state_names = dict([(k, ["sunny", "cloudy", "rainy"]) for k in weather] +
                   [(k, [True, False]) for k in umbrella])

def day_transition(d1, d2): # P(d2|d1)
     return DiscreteFactor(variables=[d2, d1],
```

```
                        cardinality=[len(state_names[d2]), len(state_names[d1])],
                        values=[0.7, 0.25, 0.25, # d2=sunny:  d1=sunny, cloudy, rainy
                                0.25, 0.35, 0.5, # d2=cloudy:  d1=sunny, cloudy, rainy
                                0.05, 0.4, 0.25], # d2=rainy:  d1=sunny, cloudy, rainy
                        state_names=state_names)

def take_umbrella_transition(d, u): # P(u|d)
    return DiscreteFactor(variables=[u, d],
                        cardinality=[len(state_names[u]), len(state_names[d])],
                        values=[0.2, 0.6, 0.95,   # u=True:  sunny, cloudy, rainy
                                0.8, 0.4, 0.05], # u=False: sunny, cloudy, rainy
                        state_names=state_names)

p = dict()
p["w1"] = DiscreteFactor(variables=["w1"],
                        cardinality=[len(state_names["w1"])],
                        values=[0.3, 0.4, 0.4],
                        state_names=state_names)

p["w2|w1"] = day_transition("w1", "w2")
p["w3|w2"] = day_transition("w2", "w3")
p["w4|w3"] = day_transition("w3", "w4")
p["w5|w4"] = day_transition("w4", "w5")
p["w6|w5"] = day_transition("w5", "w6")
p["w7|w6"] = day_transition("w6", "w7")

print("Table for P(w2|w1):")
print(p["w2|w1"])

print("Table for P(w6|w5):")
print(p["w6|w5"])

p["u1|w1"] = take_umbrella_transition("w1", "u1")
p["u2|w2"] = take_umbrella_transition("w2", "u2")
p["u3|w3"] = take_umbrella_transition("w3", "u3")
p["u4|w4"] = take_umbrella_transition("w4", "u4")
```

```
p["u5|w5"] = take_umbrella_transition("w5", "u5")
p["u6|w6"] = take_umbrella_transition("w6", "u6")
p["u7|w7"] = take_umbrella_transition("w7", "u7")

print("\nTable for P(u3|w3):")
print(p["u1|w1"])

print("\nTable for P(u5|w5):")
print(p["u5|w5"])
```

```
Table for P(w2|w1):
+------------+------------+--------------+
| w2         | w1         |   phi(w2,w1) |
+============+============+==============+
| w2(sunny)  | w1(sunny)  |       0.7000 |
+------------+------------+--------------+
| w2(sunny)  | w1(cloudy) |       0.2500 |
+------------+------------+--------------+
| w2(sunny)  | w1(rainy)  |       0.2500 |
+------------+------------+--------------+
| w2(cloudy) | w1(sunny)  |       0.2500 |
+------------+------------+--------------+
| w2(cloudy) | w1(cloudy) |       0.3500 |
+------------+------------+--------------+
| w2(cloudy) | w1(rainy)  |       0.5000 |
+------------+------------+--------------+
| w2(rainy)  | w1(sunny)  |       0.0500 |
+------------+------------+--------------+
| w2(rainy)  | w1(cloudy) |       0.4000 |
+------------+------------+--------------+
| w2(rainy)  | w1(rainy)  |       0.2500 |
+------------+------------+--------------+
Table for P(w6|w5):
+------------+------------+--------------+
| w6         | w5         |   phi(w6,w5) |
+============+============+==============+
```

w6(sunny)	w5(sunny)	0.7000
w6(sunny)	w5(cloudy)	0.2500
w6(sunny)	w5(rainy)	0.2500
w6(cloudy)	w5(sunny)	0.2500
w6(cloudy)	w5(cloudy)	0.3500
w6(cloudy)	w5(rainy)	0.5000
w6(rainy)	w5(sunny)	0.0500
w6(rainy)	w5(cloudy)	0.4000
w6(rainy)	w5(rainy)	0.2500

Table for P(u3|w3):

u1	w1	phi(u1,w1)
u1(True)	w1(sunny)	0.2000
u1(True)	w1(cloudy)	0.6000
u1(True)	w1(rainy)	0.9500
u1(False)	w1(sunny)	0.8000
u1(False)	w1(cloudy)	0.4000
u1(False)	w1(rainy)	0.0500

```
Table for P(u5|w5):
+-----------+------------+--------------+
| u5        | w5         |   phi(u5,w5) |
+===========+============+==============+
| u5(True)  | w5(sunny)  |       0.2000 |
+-----------+------------+--------------+
| u5(True)  | w5(cloudy) |       0.6000 |
+-----------+------------+--------------+
| u5(True)  | w5(rainy)  |       0.9500 |
+-----------+------------+--------------+
| u5(False) | w5(sunny)  |       0.8000 |
+-----------+------------+--------------+
| u5(False) | w5(cloudy) |       0.4000 |
+-----------+------------+--------------+
| u5(False) | w5(rainy)  |       0.0500 |
+-----------+------------+--------------+
```

Now, let's build the model using the `FactorGraph` class. To do so, we first need to add all variable nodes with `add_nodes_from`. Then, we add each factor `f` to the graph with`add_factors(f)` and, for each of its variables `v` (available in `f.variables`), we create an edge between `f` and `v` with `add_edge(f,v)`. Finally, we verify that the factor graph is correct with the `check_model` method.

```
from pgmpy.models import FactorGraph
G = FactorGraph()

assert set(variables) == set([v for f in p.values() for v in f.variables])

### ADD NODES AND EDGES TO THE FACTOR GRAPH
G.add_nodes_from(variables)
for factor in p.values():
    G.add_factors(factor)
    for var in factor.variables:
        G.add_edge(var, factor)


print("Model is ok: ", G.check_model())
```

```
print("\nNumber of nodes: ", G.number_of_nodes())
print("Number of edges: ", G.number_of_edges())
```

```
Model is ok:  True

Number of nodes:  28
Number of edges:  27
```

```
/home/rcrtss/.pyenv/versions/miis2025-spm/lib/python3.10/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please
  from .autonotebook import tqdm as notebook_tqdm
```

Running Inference Queries

We will use now some methods implemented in pgmpy that perform exact inference. Consider the following queries:

1. We observe that on **Thursday and Friday the umbrella is being used**. What is the **weather prediction for Saturday**?
2. What is the **weather prediction for Sunday**, with the **same observations**?
3. What is the **weather prediction for Sunday** if, instead, we observe that the **umbrella is *NOT* being used on Thursday and Friday**?
4. What is the **most likely weather sequence for Monday and Tuesday**, if we observed that the **umbrella has been used both days**?

Answer the three previous queries using the `query` method of the `BeliefPropagation` class:

```
    def query(self, variables, evidence=None, joint=True, show_progress=True):
        """
        Query method using belief propagation.

        Parameters
        ----------
        variables: list
            list of variables for which you want to compute the probability

        evidence: dict
            a dict key, value pair as {var: state_of_var_observed}
            None if no evidence

        joint: boolean
```

```
            If True, returns a Joint Distribution over `variables`.
            If False, returns a dict of distributions over each of the `variables`
from pgmpy.inference import BeliefPropagation
bp = BeliefPropagation(G)

# Weather prediction for saturday, given that on thursday and friday the umbrella was used
query_1 = bp.query(
    variables=["w6"],
    evidence={"u4": True, "u5": True}
)
print("\nP(w6 | u4=True, u5=True):")
print(query_1)

# Weather prediction for sunday, given that on thursday and friday the umbrella was used
query_2 = bp.query(
    variables=["w7"],
    evidence={"u4": True, "u5": True}
)
print("\nP(w7 | u4=True, u5=True):")
print(query_2)

# Weather prediction for sunday, given that on thursday and friday the umbrella was not used
query_3 = bp.query(
    variables=["w7"],
    evidence={"u4": False, "u5": False}
)
print("\nP(w7 | u4=False, u5=False):")
print(query_3)

# Most likely sequence for monday and tuesday given that the umbrella was used both days
# Use joint of w1 and w2 for the query

query_4 = bp.query(
    variables=["w1", "w2"],
    evidence={"u1": True, "u2": True}
```

```
)

print("\nP(w1, w2 | u1=True, u2=True):")
print(query_4)

# Check if it sums to 1
total_prob = sum(query_4.values)
print("\nTotal probability for P(w1, w2 | u1=True, u2=True):", total_prob.sum())


# MAP (Most Probable Assignment) for w1 and w2 given that the umbrella was used both days:
mpe_query = bp.map_query(
    variables=["w1", "w2"],
    evidence={"u1": True, "u2": True}
)
print("\nMPE for w1 and w2 given u1=True and u2=True:")
print(mpe_query)
```

```
P(w6 | u4=True, u5=True):
+------------+-----------+
| w6         |   phi(w6) |
+============+===========+
| w6(sunny)  |    0.3023 |
+------------+-----------+
| w6(cloudy) |    0.4081 |
+------------+-----------+
| w6(rainy)  |    0.2896 |
+------------+-----------+

P(w7 | u4=True, u5=True):
+------------+-----------+
| w7         |   phi(w7) |
+============+===========+
| w7(sunny)  |    0.3860 |
+------------+-----------+
```

```
| w7(cloudy) |    0.3632 |
+------------+-----------+
| w7(rainy)  |    0.2508 |
+------------+-----------+

P(w7 | u4=False, u5=False):
+------------+-----------+
| w7         |   phi(w7) |
+============+===========+
| w7(sunny)  |    0.5224 |
+------------+-----------+
| w7(cloudy) |    0.3077 |
+------------+-----------+
| w7(rainy)  |    0.1699 |
+------------+-----------+

P(w1, w2 | u1=True, u2=True):
+------------+------------+--------------+
| w1         | w2         |   phi(w1,w2) |
+============+============+==============+
| w1(sunny)  | w2(sunny)  |       0.0212 |
+------------+------------+--------------+
| w1(sunny)  | w2(cloudy) |       0.0227 |
+------------+------------+--------------+
| w1(sunny)  | w2(rainy)  |       0.0072 |
+------------+------------+--------------+
| w1(cloudy) | w2(sunny)  |       0.0302 |
+------------+------------+--------------+
| w1(cloudy) | w2(cloudy) |       0.1269 |
+------------+------------+--------------+
| w1(cloudy) | w2(rainy)  |       0.2297 |
+------------+------------+--------------+
| w1(rainy)  | w2(sunny)  |       0.0478 |
+------------+------------+--------------+
| w1(rainy)  | w2(cloudy) |       0.2871 |
+------------+------------+--------------+
```

```
| w1(rainy)  | w2(rainy)  |       0.2273 |
+------------+------------+--------------+

Total probability for P(w1, w2 | u1=True, u2=True): 1.0

MPE for w1 and w2 given u1=True and u2=True:
{'w1': 'rainy', 'w2': 'cloudy'}
```

Implementing Belief Propagation

Implement your own Belief Propagation algorithm by filling in the provided code skeleton. You will need to provide the code to:

- Add the evidence factors
- Initialize a factor-to-variable message `f->v` and a variable-to-factor message `v->f` for each edge in the graph
- Update messages of each type
- Perform message passing using the collect_evidence and distribute_evidence algorithms
- Compute the marginal using the messages

See function comments for more details. Each comment line `#IMPLEMENT` is to be replaced with as many lines of code as you need. Method definitions should remain unchanged.

```python
from functools import reduce
import operator
from collections import defaultdict
from copy import deepcopy

def prod(iterable):
    """
    Helper function to obtain the product of all the items in the iterable
    given as input
    """

    return reduce(operator.mul, iterable, 1)

class MyBeliefPropagation:
    def __init__(self, factor_graph):
```

```
        assert factor_graph.check_model()
        self.original_graph = factor_graph
        self.variables = factor_graph.get_variable_nodes()

        self.state_names = dict()
        for f in self.original_graph.factors:
            self.state_names.update(f.state_names)

        self.set_bp_done(False)

    def get_evidence_factors(self, evidence: dict):
        """
        For each evidence variable v, create a factor with p(v=e)=1. Receives a dict of
        evidences, where keys are variables and values are variable states. Returns a list of
        DiscreteFactor.
        """
        evidence_factors = []
        for var, state in evidence.items():
            # Cardinality is the number of possible states for this variable
            states = self.state_names[var]
            card = len(states)

            # One-hot values: 1 at the observed state, 0 elsewhere
            values = [0] * card
            idx = states.index(state)
            values[idx] = 1

            evidence_factors.append(DiscreteFactor(
                        variables=[var],
                        cardinality=[card],
                        values=values,
                        state_names=self.state_names))

        return evidence_factors

    def set_evidence(self, evidence):
```

```
        """
        Generates a new graph with the evidence factors
        evidence (keys: variables, values: states)
        """
        evidence_factors = self.get_evidence_factors(evidence)
        self.working_graph = self.original_graph.copy()
        for f in evidence_factors:
            self.working_graph.add_factors(f)
            for var in f.variables:
                self.working_graph.add_edge(var, f)

        # Reset BP for new run
        self.set_bp_done(False)

    def factor_ones(self, v):
        """
        Returns a DiscreteFactor for variable v with all ones.
        """
        card = len(self.state_names[v])
        return DiscreteFactor(
            variables=[v],
            cardinality=[card],
            values=[1] * card,
            state_names=self.state_names
        )

    def initialize_messages(self):
        """
        This function creates, for each edge factor-variable, two messages: m(f->v) and
        m(v->f). It initiliazies each message as a DiscreteFactor with all ones. It stores all
        the messages in a dict of dict. Keys of both dicts are either factors or variables.
        Messages are indexed as messages[to][from]. For example, m(x->y) is in messages[y][x].
        It's done this way because it will be useful to get all messages that go to a variable
        or a factor.
        """
        self.messages = defaultdict(dict)
```

```
    for f in self.working_graph.get_factors():
        for v in f.variables:
          self.messages[v][f] = self.factor_ones(v)
          self.messages[f][v] = self.factor_ones(v)


def factor_to_variable(self, f, v):
    """
    Computes message m from factor to variable. It computes it from all messages from all
    other variables to the factor (i.e. all variables connected the factor except v).
    Returns message m.
    """
    assert v in self.variables and f in self.working_graph.factors

    # Collect messages from all neighboring variables except v
    incoming_messages = []
    for neighbor_var in f.variables:
        if neighbor_var != v:
            incoming_messages.append(self.messages[f][neighbor_var])

    # Multiply factor with all incoming messages
    result = deepcopy(f)
    for msg in incoming_messages:
        result = result * msg

    # Marginalize (sum out) all variables except v
    vars_to_marginalize = [var for var in result.variables if var != v]
    if vars_to_marginalize:
        result = result.marginalize(vars_to_marginalize, inplace=False)

    return result

def variable_to_factor(self, v, f):
    """
    Computes message m from variable to factor. It computes it from all messages from all
    other factors to the variable (i.e. all factors connected the variable except f).
```

```
    Returns message m.
    """
    assert v in self.variables and f in self.working_graph.factors

    # Collect messages from all neighboring factors except f
    incoming_messages = []
    for neighbor_factor in self.working_graph.neighbors(v):
        if neighbor_factor != f:
            incoming_messages.append(self.messages[v][neighbor_factor])

    # If no other neighbors, return a factor of ones, else multiply messages
    output_message = None
    if len(incoming_messages) == 0:
        output_message = self.factor_ones(v)
    else:
        output_message = prod(incoming_messages)

    return output_message


def update(self, m_to, m_from):
    """
    Performs an update of a message depending on whether it is variable-to-factor or
    factor-to-variable.
    """
    if m_from in self.working_graph.factors:
        updated_message = self.factor_to_variable(m_from, m_to)
    else:
        updated_message = self.variable_to_factor(m_from, m_to)

    self.messages[m_to][m_from] = updated_message


def collect_evidence(self, node, parent=None):
    """
    Passes messages from the leaves to the root of the tree.
```

```
    The parent argument is used to avoid an infinite recursion.
    """
    for child in self.working_graph.neighbors(node):
        if child != parent:
            # Recursively collect evidence from children first
            self.collect_evidence(child, node)
            # Then send message from child to node
            self.update(node, child)

def distribute_evidence(self, node, parent=None):
    """
    Passes messages from the root to the leaves of the tree.
    The parent argument is used to avoid an infinite recursion.
    """
    for child in self.working_graph.neighbors(node):
        if child != parent:
            # Send message from node to child
            self.update(child, node)
            # Recursively distribute evidence to children
            self.distribute_evidence(child, node)

def run_bp(self, root):
    """
    After initializing the messages, this function performs Belief Propagation
    using collect_evidence and distribute_evidence from the given root node.
    """
    assert root in self.variables, "Variable not in the model"
    # IMPLEMENT
    self.initialize_messages()
    self.collect_evidence(root)
    self.distribute_evidence(root)

    self.set_bp_done(True)

def get_marginal(self, variable):
    """
```

```
    To be used after run_bp. Returns p(variable | evidence) unnormalized.
    """
    assert self.bp_done, "First run BP!"

    # Multiply all incoming messages from neighboring factors
    incoming_messages = [self.messages[variable][f] for f in self.working_graph.neighbors(variable)]

    if len(incoming_messages) == 0:
        return self.factor_ones(variable)

    return prod(incoming_messages)


def get_marginal_subset(self, variables):
    """
    Returns p(variables | evidence) unnormalized.
    The variables must share a common factor.
    """
    assert self.bp_done, "First run BP!"

    # Find a factor that contains all the variables in the subset
    common_factor = None
    for f in self.working_graph.get_factors():
        if all(v in f.variables for v in variables):
            common_factor = f
            break

    if common_factor is None:
        raise ValueError(f"Variables {variables} do not share a common factor")

    # Start with the common factor
    result = deepcopy(common_factor)

    # For each variable in the factor, multiply by incoming messages from
    # all neighboring factors EXCEPT the common factor
    for v in common_factor.variables:
```

```
            for neighbor_f in self.working_graph.neighbors(v):
                if neighbor_f != common_factor:
                    result = result * self.messages[v][neighbor_f]

        # Marginalize out variables not in the query subset
        vars_to_marginalize = [v for v in result.variables if v not in variables]
        if vars_to_marginalize:
            result = result.marginalize(vars_to_marginalize, inplace=False)

        return result


    def set_bp_done(self, flag):
        """
        Resets the Belief Propagation flag to false to start over
        """
        self.bp_done = flag
```

Check that your implementation produces the same results for the first three queries P(w6|u4=t,u5=t), P(w7|u4=t,u5=t), and P(w6|u4=f,u5=f) as the ones given by the `BeliefPropagation` class. Do you need to run BP each time? Justify your answer.

Answer: It is necessary to run BP again only if the evidence changes. If it does not change, you can run BP only once and query as you like.

```
my_bp = MyBeliefPropagation(G)

my_bp.set_evidence({"u4": True, "u5": True})
my_bp.run_bp("w6")
p["w7|u4=t,u5=t"] = my_bp.get_marginal("w7").normalize(inplace=False)

print("\nP(w7 | u4=True, u5=True):")
print(p["w7|u4=t,u5=t"])


my_bp.set_evidence({"u4": False, "u5": False})
my_bp.run_bp("w3")
p["w7|u4=f,u5=f"] = my_bp.get_marginal("w7").normalize(inplace=False)
```

```
print("\nP(w7 | u4=False, u5=False):")
print(p["w7|u4=f,u5=f"])
```

```
P(w7 | u4=True, u5=True):
+------------+-----------+
| w7         |   phi(w7) |
+============+===========+
| w7(sunny)  |    0.3860 |
+------------+-----------+
| w7(cloudy) |    0.3632 |
+------------+-----------+
| w7(rainy)  |    0.2508 |
+------------+-----------+

P(w7 | u4=False, u5=False):
+------------+-----------+
| w7         |   phi(w7) |
+============+===========+
| w7(sunny)  |    0.5224 |
+------------+-----------+
| w7(cloudy) |    0.3077 |
+------------+-----------+
| w7(rainy)  |    0.1699 |
+------------+-----------+
```

Use BP to compute the fourth query P(w1,w2|u1=t,u2=t). Note that we cannot do it with the `get_marginal` method as it is now. Define a method `get_marginal_subset` that receives a list of variables as input and, if they share a common factor, returns the marginal of the subset. Otherwise, throw an error. Check your result with the one produced by the `BeliefPropagation` class.

```
my_bp.set_evidence({"u1": True, "u2": True})
my_bp.run_bp("w7") # Any node works

# Get unnormalized joint marginal P(w1, w2 | u1=True, u2=True)
p["w1,w2|u1=t,u2=t"] = my_bp.get_marginal_subset(["w1", "w2"]).normalize(inplace=False)
```

```python
print("\nP(w1, w2 | u1=True, u2=True) from MyBeliefPropagation:")
print(p["w1,w2|u1=t,u2=t"])

print("\n" + "="*50)
print("Comparison with pgmpy BeliefPropagation:")
print("="*50)
print(query_4)
```

```
P(w1, w2 | u1=True, u2=True) from MyBeliefPropagation:
+------------+------------+--------------+
| w1         | w2         |   phi(w1,w2) |
+============+============+==============+
| w1(sunny)  | w2(sunny)  |       0.0212 |
+------------+------------+--------------+
| w1(sunny)  | w2(cloudy) |       0.0227 |
+------------+------------+--------------+
| w1(sunny)  | w2(rainy)  |       0.0072 |
+------------+------------+--------------+
| w1(cloudy) | w2(sunny)  |       0.0302 |
+------------+------------+--------------+
| w1(cloudy) | w2(cloudy) |       0.1269 |
+------------+------------+--------------+
| w1(cloudy) | w2(rainy)  |       0.2297 |
+------------+------------+--------------+
| w1(rainy)  | w2(sunny)  |       0.0478 |
+------------+------------+--------------+
| w1(rainy)  | w2(cloudy) |       0.2871 |
+------------+------------+--------------+
| w1(rainy)  | w2(rainy)  |       0.2273 |
+------------+------------+--------------+

==================================================
Comparison with pgmpy BeliefPropagation:
==================================================
+------------+------------+--------------+
```

w1	w2	phi(w1,w2)
w1(sunny)	w2(sunny)	0.0212
w1(sunny)	w2(cloudy)	0.0227
w1(sunny)	w2(rainy)	0.0072
w1(cloudy)	w2(sunny)	0.0302
w1(cloudy)	w2(cloudy)	0.1269
w1(cloudy)	w2(rainy)	0.2297
w1(rainy)	w2(sunny)	0.0478
w1(rainy)	w2(cloudy)	0.2871
w1(rainy)	w2(rainy)	0.2273

For loopy graphs, instead of collect and distribute evidence, iterate BP until convergence. You can already do this.