Notebook

April 22, 2025

\# AMTAIR Prototype Demonstration (Public Colab Notebook)

\# \\\# Instructions — How to use this notebook:

1. **Import Libraries \& Install Packages**: Run Section 0.1 to set up the necessary dependencies for data processing and visualization.
2. **Connect to GitHub Repository \& Load Data files**: Run Section 0.2 to establish connections to the data repository and load example datasets. This step retrieves sample ArgDown files and extracted data for demonstration.
3. **Process Source Documents to ArgDown**: Sections 1.0-1.8 demonstrate the extraction of argument structures from source documents (such as PDFs) into ArgDown format, a markdown-like notation for structured arguments.
4. **Convert ArgDown to BayesDown**: Sections 2.0-2.3 handle the transformation of ArgDown files into BayesDown format, which incorporates probabilistic information into the argument structure.
5. **Extract Data into Structured Format**: Section 3.0 processes BayesDown format into structured database entries (CSV) that can be used for analysis.
6. **Create and Analyze Bayesian Networks**: Section 4.0 demonstrates how to build Bayesian networks from the extracted data and provides tools for analyzing risk pathways.
7. **Save and Export Results**: Sections 5.0-6.0 provide methods for archiving results and exporting visualizations.

\# \\\# Key Concepts:

- **ArgDown**: A structured format for representing arguments, with hierarchical relationships between statements.
- **BayesDown**: An extension of ArgDown that incorporates probabilistic information, allowing for Bayesian network construction.
- **Extraction Pipeline**: The process of converting unstructured text to structured argument representations.
- **Bayesian Networks**: Probabilistic graphical models that represent variables and their conditional dependencies.

\# \\\# Example Workflow:

1. Load a sample ArgDown file from the repository
2. Extract the hierarchical structure and relationships
3. Add probabilistic information to create a BayesDown representation
4. Generate a Bayesian network visualization
5. Analyze conditional probabilities and risk pathways

\# \\\# Troubleshooting:

- If connectivity issues occur, ensure you have access to the GitHub repository
- For visualization errors, check that all required libraries are properly installed
- When processing custom files, ensure they follow the expected format conventions

\# 0.1 Prepare Colab/Python Environment — Import Libraries \\\& Packages

[27]:
```
# @title 0.1 --- Install & Import Libraries & Packages (One-Time Setup) ---

#  Stores Boolean Flag in Environment runs only when flag is absent
#  Check if the setup flag variable exists in the global scope

try:
    # If this variable exists, setup was already done successfully in this␣
 ↪session.
    _setup_imports_done
```

```
    print("  Libraries already installed and imported in this session. Skipping␣
 ↪setup.")

except NameError:
    print("  Performing one-time library installation and imports...")

    # 1. Install Packages (Quietly using -q) Requiring Installation (not␣
 ↪avialable in Colab by default)
    !pip install -q pyvis
    !apt-get install pandoc -y
    # Combine Google-related packages for slightly cleaner install
    !pip install -q --upgrade gspread pandas google-auth google-colab
    !pip install -q pgmpy
    !pip install -q nbconvert  # Often pre-installed, but good to ensure

    print("   --> Installations complete.")

    # 2. Import Libraries
    try:
        import requests      # For making HTTP requests
        import io            # For working with in-memory file-like objects
        import pandas as pd  # For data manipulation
        import numpy as np
        import json
        import re
        import matplotlib.pyplot as plt
        from IPython.display import HTML, display, Markdown # Combined imports

        # Packages not avialable in Colab by default and require above␣
 ↪installations below:
        import networkx as nx

        from pgmpy.models import BayesianNetwork
        from pgmpy.factors.discrete import TabularCPD
        from pgmpy.inference import VariableElimination

        from pyvis.network import Network

        # Also good practice to print key library versions after import
        print(f"      pandas version: {pd.__version__}")
        print(f"      networkx version: {nx.__version__}")
        # Add others if specific versions are critical

        print("   --> Imports complete.")

        # 3. Set the flag ONLY if all installs and imports were successful
        _setup_imports_done = True
```

```
        print("  One-time setup finished successfully.")

    except ImportError as e:
        print(f"  ERROR during import: {e}")
        print("   --> Setup did not complete successfully. Please check␣
 ↪installations.")
    except Exception as e:
        print(f"  UNEXPECTED ERROR during setup: {e}")
        print("   --> Setup did not complete successfully.")

# --- End of One-Time Setup Cell ---

# Now you can proceed with the rest of your code, knowing the imports exist
# if the setup cell didn't raise a critical error.
```

\# \\\# 0.2 Connect to GitHub Repository

The Public GitHub Repo Url in use:

https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/main/

Note: When encountering errors, accessing the data, try using “RAW” Urls.

[4]:
```
# @title 0.2 --- Connect to GitHub Repository --- Load Files


# Specify the base repository URL
repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
 ↪main/data/example_1/"

def load_file_from_repo(relative_path):
  """Loads a file from the specified GitHub repository using a relative path."""
  file_url = repo_url + relative_path
  response = requests.get(file_url)

  # Check for bad status codes and print more helpful error messages
  if response.status_code == 404:
    raise HTTPError(f"File not found at URL: {file_url}. Check the file path/
 ↪name and ensure the file is publicly accessible.", response=response)
  else:
    response.raise_for_status() # Raise for other error codes

  file_object = io.StringIO(response.text)

  if relative_path.endswith(".csv"):
    return pd.read_csv(file_object)
  elif relative_path.endswith(".json"):
    return pd.read_json(file_object)
  elif relative_path.endswith(".md"):
```

```
    return file_object.read()  # Return the raw content for .md files
  else:
    raise ValueError("Unsupported file type. Add Support in GitHub Connection␣
 ↪in the Second Section of this Python Notebook")

# Load files using relative paths

df = load_file_from_repo("extracted_data.csv") # Update if the file path is␣
 ↪incorrect

md_content = load_file_from_repo("ArgDown_TestText.md")

# print(df.head()) # To see the output, run the code.

print(md_content) # To see the output, run the code.
```

[8]:
```
print(df.head()) # To see the output, run the code.
```

\# \\\# 0.3 File Import

[9]:
```
md_content
```

\# 1.0 Sources (PDF's of Papers) to ArgDown (.md file)

\# \\\# 1.1 Specify Source Document (e.g. PDF)

Review the source document, ensure it is suitable for API call and upload to / store it in the correct location.

[4]:
```
# @title 1.1.a) --- MTAIR Online Model (Analytica) ---

from IPython.display import IFrame

IFrame(src="https://acp.analytica.com/view0?
 ↪invite=4560&code=3000289064591444815", width="100%", height="900px")
```

\# \\\# 1.2 Generate ArgDown Extraction Prompt

Generate Extraction Prompt

[ ]:
```
# @title 1.2.0 --- Prompt Template Function Definitions ---

from string import Template
from typing import Dict, Optional, Union, List

class PromptTemplate:
    """Template system for LLM prompts with variable substitution"""

    def __init__(self, template: str):
```

```
        """Initialize with template string using $variable format"""
        self.template = Template(template)

    def format(self, **kwargs) -> str:
        """Substitute variables in the template"""
        return self.template.safe_substitute(**kwargs)

    @classmethod
    def from_file(cls, filepath: str) -> 'PromptTemplate':
        """Load template from a file"""
        with open(filepath, 'r') as f:
            template = f.read()
        return cls(template)

class PromptLibrary:
    """Collection of prompt templates for different extraction tasks"""

    # ArgDown extraction prompt
    ARGDOWN_EXTRACTION = PromptTemplate("""
You are an expert in creating structured argument maps in ArgDown format. Your␣
 ↪task is to extract the key arguments, premises, and conclusions from the␣
 ↪provided text, and represent them in a hierarchical ArgDown format.

Follow these guidelines:
1. Use the format [Statement]: Description for main claims
2. Use the + symbol and indentation to indicate supporting statements
3. Capture the core argumentative structure, focusing on causal relationships␣
 ↪and key claims
4. Ensure each statement has a clear, concise title followed by a fuller␣
 ↪description
5. Add the "instantiations" field to indicate possible states of each variable

Here is the metadata format to include for each node:
{"instantiations": ["node_TRUE", "node_FALSE"]}

Example:
[Thesis]: Main claim of the text. {"instantiations": ["thesis_TRUE",␣
 ↪"thesis_FALSE"]}
 + [Support1]: First supporting argument. {"instantiations": ["support1_TRUE",␣
 ↪"support1_FALSE"]}
   + [Evidence1]: Evidence for Support1. {"instantiations": ["evidence1_TRUE",␣
 ↪"evidence1_FALSE"]}
 + [Support2]: Second supporting argument. {"instantiations": ["support2_TRUE",␣
 ↪"support2_FALSE"]}

Text to analyze:
```

```
$text

Create an ArgDown representation that captures the key arguments, their␣
 ↪relationships, and possible states:
""")

    # BayesDown probability extraction prompt
    BAYESDOWN_EXTRACTION = PromptTemplate("""
You are an expert in probabilistic reasoning and Bayesian networks. Your task␣
 ↪is to extend the provided ArgDown structure with probability information,␣
 ↪creating a BayesDown representation.

For each statement in the ArgDown structure, you need to:
1. Estimate prior probabilities for each possible state
2. Estimate conditional probabilities given parent states
3. Maintain the original structure and relationships

Here is the format to follow:
[Node]: Description. { "instantiations": ["node_TRUE", "node_FALSE"], "priors":␣
 ↪{ "p(node_TRUE)": "0.7", "p(node_FALSE)": "0.3" }, "posteriors": {␣
 ↪"p(node_TRUE|parent_TRUE)": "0.9", "p(node_TRUE|parent_FALSE)": "0.4",␣
 ↪"p(node_FALSE|parent_TRUE)": "0.1", "p(node_FALSE|parent_FALSE)": "0.6" } }
 [Parent]: Parent description. {...}


Here are the specific probability questions to answer:
$questions

ArgDown structure to enhance:
$argdown

Provide the complete BayesDown representation with probabilities:
""")

    @classmethod
    def get_template(cls, template_name: str) -> PromptTemplate:
        """Get a prompt template by name"""
        if hasattr(cls, template_name):
            return getattr(cls, template_name)
        else:
            raise ValueError(f"Template not found: {template_name}")
```

\# \\\# 1.3 Prepare LLM API Call

Combine Systemprompt + API Specifications + ArgDown Instructions + Prompt + Source PDF for API Call

[ ]:
```python
# @title 1.3.0 --- Provider-Agnostic LLM API Interface ---

import os
import json
import time
import requests
from abc import ABC, abstractmethod
from typing import Dict, List, Optional, Union, Any
from dataclasses import dataclass

@dataclass
class LLMResponse:
    """Standard response object for LLM completions"""
    content: str
    model: str
    usage: Dict[str, int]
    raw_response: Dict[str, Any]
    created_at: float = time.time()

class LLMProvider(ABC):
    """Abstract base class for LLM providers"""

    @abstractmethod
    def complete(self,
                 prompt: str,
                 system_prompt: Optional[str] = None,
                 temperature: float = 0.7,
                 max_tokens: int = 4000) -> LLMResponse:
        """Generate a completion from the LLM"""
        pass

    @abstractmethod
    def get_available_models(self) -> List[str]:
        """Return a list of available models from this provider"""
        pass

class OpenAIProvider(LLMProvider):
    """OpenAI API implementation"""

    def __init__(self, api_key: Optional[str] = None, organization:␣
↪Optional[str] = None):
        """Initialize with API key from args or environment"""
        self.api_key = api_key or os.environ.get("OPENAI_API_KEY")
        if not self.api_key:
            raise ValueError("OpenAI API key is required. Provide as argument␣
↪or set OPENAI_API_KEY environment variable.")
```

```
        self.organization = organization or os.environ.
↪get("OPENAI_ORGANIZATION")
        self.api_base = "https://api.openai.com/v1"

    def complete(self,
                 prompt: str,
                 system_prompt: Optional[str] = None,
                 model: str = "gpt-4-turbo",
                 temperature: float = 0.7,
                 max_tokens: int = 4000) -> LLMResponse:
        """Generate a completion using OpenAI's API"""

        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.api_key}"
        }

        if self.organization:
            headers["OpenAI-Organization"] = self.organization

        messages = []
        if system_prompt:
            messages.append({"role": "system", "content": system_prompt})

        messages.append({"role": "user", "content": prompt})

        data = {
            "model": model,
            "messages": messages,
            "temperature": temperature,
            "max_tokens": max_tokens
        }

        response = requests.post(
            f"{self.api_base}/chat/completions",
            headers=headers,
            json=data
        )

        response.raise_for_status()
        result = response.json()

        return LLMResponse(
            content=result["choices"][0]["message"]["content"],
            model=result["model"],
            usage=result["usage"],
            raw_response=result
```

```
        )

    def get_available_models(self) -> List[str]:
        """Return a list of available OpenAI models"""
        headers = {
            "Authorization": f"Bearer {self.api_key}"
        }

        if self.organization:
            headers["OpenAI-Organization"] = self.organization

        response = requests.get(
            f"{self.api_base}/models",
            headers=headers
        )

        response.raise_for_status()
        models = response.json()["data"]
        return [model["id"] for model in models]

class AnthropicProvider(LLMProvider):
    """Anthropic Claude API implementation"""

    def __init__(self, api_key: Optional[str] = None):
        """Initialize with API key from args or environment"""
        self.api_key = api_key or os.environ.get("ANTHROPIC_API_KEY")
        if not self.api_key:
            raise ValueError("Anthropic API key is required. Provide as␣
 ↪argument or set ANTHROPIC_API_KEY environment variable.")

        self.api_base = "https://api.anthropic.com/v1"

    def complete(self,
                 prompt: str,
                 system_prompt: Optional[str] = None,
                 model: str = "claude-3-opus-20240229",
                 temperature: float = 0.7,
                 max_tokens: int = 4000) -> LLMResponse:
        """Generate a completion using Anthropic's API"""

        headers = {
            "Content-Type": "application/json",
            "X-API-Key": self.api_key,
            "anthropic-version": "2023-06-01"
        }

        data = {
```

```
            "model": model,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": temperature,
            "max_tokens": max_tokens
        }

        if system_prompt:
            data["system"] = system_prompt

        response = requests.post(
            f"{self.api_base}/messages",
            headers=headers,
            json=data
        )

        response.raise_for_status()
        result = response.json()

        return LLMResponse(
            content=result["content"][0]["text"],
            model=result["model"],
            usage={"prompt_tokens": result.get("usage", {}).get("input_tokens",␣
↪0),
                   "completion_tokens": result.get("usage", {}).
↪get("output_tokens", 0)},
            raw_response=result
        )

    def get_available_models(self) -> List[str]:
        """Return a list of available Anthropic models"""
        # Anthropic doesn't have a models endpoint, so we return a static list
        return [
            "claude-3-opus-20240229",
            "claude-3-sonnet-20240229",
            "claude-3-haiku-20240307"
        ]

class LLMFactory:
    """Factory for creating LLM providers"""

    @staticmethod
    def create_provider(provider_name: str, **kwargs) -> LLMProvider:
        """Create and return an LLM provider instance"""
        if provider_name.lower() == "openai":
            return OpenAIProvider(**kwargs)
        elif provider_name.lower() == "anthropic":
            return AnthropicProvider(**kwargs)
```

```
        else:
            raise ValueError(f"Unsupported provider: {provider_name}")
```

[ ]:
```
# @title 1.3.0 --- API Call Function Definitions ---

def extract_argdown_from_text(text: str, provider_name: str = "openai", model:␣
 ↪str = None) -> str:
    """
    Extract ArgDown representation from text using LLM

    Args:
        text: The source text to extract arguments from
        provider_name: The LLM provider to use (openai or anthropic)
        model: Specific model to use, or None for default

    Returns:
        Extracted ArgDown representation
    """
    # Create LLM provider
    provider = LLMFactory.create_provider(provider_name)

    # Get extraction prompt
    prompt_template = PromptLibrary.get_template("ARGDOWN_EXTRACTION")
    prompt = prompt_template.format(text=text)

    # Set model-specific parameters
    if provider_name.lower() == "openai":
        model = model or "gpt-4-turbo"
        temperature = 0.3  # Lower temperature for more deterministic extraction
        max_tokens = 4000
    elif provider_name.lower() == "anthropic":
        model = model or "claude-3-opus-20240229"
        temperature = 0.2
        max_tokens = 4000

    # Call the LLM
    system_prompt = "You are an expert in argument mapping and causal reasoning.
 ↪"
    response = provider.complete(
        prompt=prompt,
        system_prompt=system_prompt,
        model=model,
        temperature=temperature,
        max_tokens=max_tokens
    )

    # Extract the ArgDown content (remove any markdown code blocks if present)
```

```
    argdown_content = response.content
    if "```" in argdown_content:
        # Extract content between code blocks if present
        import re
        matches = re.findall(r"```(?:argdown)?\n([\s\S]*?)\n```",␣
↪argdown_content)
        if matches:
            argdown_content = matches[0]

    return argdown_content

def validate_argdown(argdown_text: str) -> Dict[str, Any]:
    """
    Validate ArgDown representation to ensure it's well-formed

    Args:
        argdown_text: ArgDown representation to validate

    Returns:
        Dictionary with validation results
    """
    # Initialize validation results
    results = {
        "is_valid": True,
        "errors": [],
        "warnings": [],
        "stats": {
            "node_count": 0,
            "relationship_count": 0,
            "max_depth": 0
        }
    }

    # Basic syntax checks
    lines = argdown_text.split("\n")
    node_pattern = r'\[(.*?)\]:'
    instantiation_pattern = r'{"instantiations":'

    # Track nodes and relationships
    nodes = set()
    relationships = []
    current_depth = 0
    max_depth = 0

    for i, line in enumerate(lines):
        # Skip empty lines
        if not line.strip():
```

```
            continue

        # Calculate indentation depth
        indent = 0
        if '+' in line:
            indent = line.find('+') // 2

        current_depth = indent
        max_depth = max(max_depth, current_depth)

        # Check for node definitions
        import re
        node_matches = re.findall(node_pattern, line)
        if node_matches:
            node = node_matches[0]
            nodes.add(node)
            results["stats"]["node_count"] += 1

            # Check for instantiations
            if instantiation_pattern not in line:
                results["warnings"].append(f"Line {i+1}: Node '{node}' is␣
↪missing instantiations metadata")

        # Check parent-child relationships
        if indent > 0 and '+' in line and node_matches:
            # This is a child node; find its parent
            parent_indent = indent - 1
            j = i - 1
            while j >= 0:
                if '+' in lines[j] and lines[j].find('+') // 2 == parent_indent:
                    parent_matches = re.findall(node_pattern, lines[j])
                    if parent_matches:
                        parent = parent_matches[0]
                        relationships.append((parent, node))
                        results["stats"]["relationship_count"] += 1
                        break
                j -= 1

    results["stats"]["max_depth"] = max_depth

    # If we didn't find any nodes, that's a problem
    if results["stats"]["node_count"] == 0:
        results["is_valid"] = False
        results["errors"].append("No valid nodes found in ArgDown␣
↪representation")

    return results
```

```python
def process_source_document(file_path: str, provider_name: str = "openai") ->␣
↪Dict[str, Any]:
    """
    Process a source document to extract ArgDown representation

    Args:
        file_path: Path to the source document
        provider_name: The LLM provider to use

    Returns:
        Dictionary with extraction results
    """
    # Load the source document
    text = ""
    if file_path.endswith(".pdf"):
        # PDF handling requires additional libraries
        try:
            import PyPDF2
            with open(file_path, 'rb') as file:
                reader = PyPDF2.PdfReader(file)
                text = ""
                for page in reader.pages:
                    text += page.extract_text() + "\n"
        except ImportError:
            raise ImportError("PyPDF2 is required for PDF processing. Install␣
↪it with: pip install PyPDF2")
    elif file_path.endswith(".txt"):
        with open(file_path, 'r') as file:
            text = file.read()
    elif file_path.endswith(".md"):
        with open(file_path, 'r') as file:
            text = file.read()
    else:
        raise ValueError(f"Unsupported file format: {file_path}")

    # Extract ArgDown
    argdown_content = extract_argdown_from_text(text, provider_name)

    # Validate the extraction
    validation_results = validate_argdown(argdown_content)

    # Prepare results
    results = {
        "source_path": file_path,
        "extraction_timestamp": time.time(),
        "argdown_content": argdown_content,
```

```
        "validation": validation_results,
        "provider": provider_name
    }

    return results

def save_argdown_extraction(results: Dict[str, Any], output_path: str) -> None:
    """
    Save ArgDown extraction results

    Args:
        results: Extraction results dictionary
        output_path: Path to save the results
    """
    # Save the ArgDown content
    with open(output_path, 'w') as file:
        file.write(results["argdown_content"])

    # Save metadata alongside
    metadata_path = output_path.replace('.md', '_metadata.json')
    metadata = {
        "source_path": results["source_path"],
        "extraction_timestamp": results["extraction_timestamp"],
        "validation": results["validation"],
        "provider": results["provider"]
    }

    with open(metadata_path, 'w') as file:
        json.dump(metadata, file, indent=2)
```

[ ]:
```
# @title 1.3 --- Prepare LLM API Call ---
def prepare_extraction_call(source_path, provider_name="openai", model=None):
    """Prepare the LLM API call for ArgDown extraction"""

    # Load the source document
    print(f"Processing source document: {source_path}")

    # Determine provider and model
    provider = provider_name.lower()
    if provider not in ["openai", "anthropic"]:
        raise ValueError(f"Unsupported provider: {provider}. Use 'openai' or␣
 ↪'anthropic'.")

    # Set default model if none provided
    if model is None:
        if provider == "openai":
            model = "gpt-4-turbo"
```

```
        elif provider == "anthropic":
            model = "claude-3-opus-20240229"

    # Print configuration
    print(f"Using provider: {provider}")
    print(f"Selected model: {model}")

    return {
        "source_path": source_path,
        "provider": provider,
        "model": model
    }

# Usage example:
source_path = "example_document.pdf"  # Replace with actual document path
extraction_config = prepare_extraction_call(source_path, provider_name="openai")
```

\# \\\# 1.4 Make ArgDown Extraction LLM API Call

[6]:
```
# @title 1.4 --- Make ArgDown Extraction LLM API Call ---
def execute_extraction(extraction_config):
    """Execute the ArgDown extraction using the LLM API"""

    print(f"Starting extraction from {extraction_config['source_path']}")
    start_time = time.time()

    try:
        # Process the document
        results = process_source_document(
            extraction_config["source_path"],
            provider_name=extraction_config["provider"]
        )

        # Print success message
        elapsed_time = time.time() - start_time
        print(f"Extraction completed in {elapsed_time:.2f} seconds")
        print(f"Extracted {results['validation']['stats']['node_count']} nodes␣
↪with "
              f"{results['validation']['stats']['relationship_count']}␣
↪relationships")

        # Print any warnings
        if results['validation']['warnings']:
            print("\nWarnings:")
            for warning in results['validation']['warnings']:
                print(f"- {warning}")
```

```
        return results

    except Exception as e:
        print(f"Error during extraction: {str(e)}")
        raise

# Usage example:
extraction_results = execute_extraction(extraction_config)
```

\# \\\# 1.5 Save ArgDown Extraction Response

1. Save and log API return
2. Save ArgDown.md file for further Proecessing

[6]:
```
# @title 1.5 --- Save ArgDown Extraction Response ---

def save_extraction_results(results, output_directory="./outputs"):
    """Save the extraction results to file"""

    # Ensure output directory exists
    import os
    os.makedirs(output_directory, exist_ok=True)

    # Create base filename from source
    import os.path
    base_name = os.path.basename(results["source_path"]).split('.')[0]
    timestamp = time.strftime("%Y%m%d-%H%M%S")
    output_filename = f"{base_name}_argdown_{timestamp}.md"
    output_path = os.path.join(output_directory, output_filename)

    # Save the results
    save_argdown_extraction(results, output_path)

    print(f"Saved ArgDown extraction to: {output_path}")
    print(f"Metadata saved to: {output_path.replace('.md', '_metadata.json')}")

    # Also save to standard location for further processing
    standard_path = os.path.join(output_directory, "ArgDown.md")
    with open(standard_path, 'w') as f:
        f.write(results["argdown_content"])
    print(f"Also saved to standard location: {standard_path}")

    return output_path

# Usage example:
output_path = save_extraction_results(extraction_results)
```

```
# Preview the extracted ArgDown
from IPython.display import Markdown, display

# Display the first 500 characters of the extracted ArgDown
preview = extraction_results["argdown_content"][:500] + "..." if␣
 ↪len(extraction_results["argdown_content"]) > 500 else␣
 ↪extraction_results["argdown_content"]
display(Markdown(f"## Extracted ArgDown Preview\n\n```\n{preview}\n```"))
```

\# \\\# 1.6 Review and Check ArgDown.md File

[10]:
```
display(Markdown(md_content))
```

\# \\\# 1.6.2 Check the Graph Structure with the ArgDown Sandbox Online Copy and paste the BayesDown formatted … in the ArgDown Sandbox below to quickly verify that the network renders correctly.

[12]:
```
# @title 1.6.2 --- ArgDown Online Sandbox ---

from IPython.display import IFrame

IFrame(src="https://argdown.org/sandbox/map/", width="100%", height="600px")
```

\# \\\# 1.7 Extract ArgDown Graph Information as DataFrame

Extract:

- Nodes (Variable_Title)
- Edges (Parents)
- Instantiations
- Description

Implementation nodes: - One function for ArgDown and BayesDown extraction, but: - IF YOU ONLY WANT ARGDOWN EXTRACTION: USE ARGUMENT IN FUNCTION CALL "parse_markdown_hierarchy(markdown_text, ArgDown = True)" - so if you set ArgDown = True, it gives you only instantiations, no probabilities.

[8]:
```
# @title 1.7 --- Parsing ArgDown & BayesDown (.md to .csv) ---

def parse_markdown_hierarchy_fixed(markdown_text, ArgDown = False):
    """Main function to parse markdown hierarchy into a DataFrame with correct␣
 ↪parent-child relationships"""

    # Remove comments
    clean_text = remove_comments(markdown_text)

    # Extract all titles with their descriptions and indentation levels
    titles_info = extract_titles_info(clean_text)
```

```
    # Establish parent-child relationships - Use fixed function here
    titles_with_relations = establish_relationships_fixed(titles_info,␣
 ↪clean_text)

    # Convert to DataFrame
    df = convert_to_dataframe(titles_with_relations, ArgDown)

    # Add No_Parent and No_Children columns
    df = add_no_parent_no_child_columns_to_df(df)

    # Add Parents instantiation columns
    df = add_parents_instantiation_columns_to_df(df)

    return df

def remove_comments(markdown_text):
    """Remove comment blocks from markdown text"""
    return re.sub(r'/\*.*?\*/', '', markdown_text, flags=re.DOTALL)

def extract_titles_info(text):
    """Extract titles with their descriptions and indentation levels"""
    lines = text.split('\n')
    titles_info = {}

    for line in lines:
        if not line.strip():
            continue

        title_match = re.search(r'[<\[](.+?)[>\]]', line)
        if not title_match:
            continue

        title = title_match.group(1)

        # Extract description and metadata
        title_pattern_in_line = r'[<\[]' + re.escape(title) + r'[>\]]:'
        description_match = re.search(title_pattern_in_line + r'\s*(.*)', line)

        if description_match:
            full_text = description_match.group(1).strip()

            # Check if description contains a "{" to not include metadata in␣
 ↪description
            if "{" in full_text:
                # Split at the first "{"
                split_index = full_text.find("{")
                description = full_text[:split_index].strip()
```

```
                metadata = full_text[split_index:].strip()
            else:
                # Keep the entire description and no metadata
                description = full_text
                metadata = ''
        else:
            description = ''
            metadata = ''  # Ensure metadata is initialized as empty string

        indentation = 0
        if '+' in line:
            symbol_index = line.find('+')
            # Count spaces before the '+' symbol
            i = symbol_index - 1
            while i >= 0 and line[i] == ' ':
                indentation += 1
                i -= 1
        elif '-' in line:
            symbol_index = line.find('-')
            # Count spaces before the '-' symbol
            i = symbol_index - 1
            while i >= 0 and line[i] == ' ':
                indentation += 1
                i -= 1

        # If neither symbol exists, indentation remains 0

        if title in titles_info:
            # Only update description if it's currently empty and we found a␣
↪new one
            if not titles_info[title]['description'] and description:
                titles_info[title]['description'] = description

            # Store all indentation levels for this title
            titles_info[title]['indentation_levels'].append(indentation)

            # Keep max indentation for backward compatibility
            if indentation > titles_info[title]['indentation']:
                titles_info[title]['indentation'] = indentation

            # Do NOT update metadata here - keep the original metadata
        else:
            # First time seeing this title, create a new entry
            titles_info[title] = {
                'description': description,
                'indentation': indentation,
```

```
                'indentation_levels': [indentation],  # Initialize with first␣
↪indentation level
                'parents': [],
                'children': [],
                'line': None,
                'line_numbers': [],  # Initialize an empty list for all␣
↪occurrences
                'metadata': metadata  # Set metadata explicitly from what we␣
↪found
            }

    return titles_info

def establish_relationships_fixed(titles_info, text):
    """
    Establish parent-child relationships between titles using BayesDown␣
↪indentation rules.

    In BayesDown syntax:
    - More indented nodes (with + symbol) are PARENTS of less indented nodes
    - The relationship reads as "Effect is caused by Cause" (Effect + Cause)
    - This aligns with how Bayesian networks represent causality
    """
    lines = text.split('\n')

    # Dictionary to store line numbers for each title occurrence
    title_occurrences = {}

    # Record line number for each title (including multiple occurrences)
    line_number = 0
    for line in lines:
        if not line.strip():
            line_number += 1
            continue

        title_match = re.search(r'[<\[](.+?)[>\]]', line)
        if not title_match:
            line_number += 1
            continue

        title = title_match.group(1)

        # Store all occurrences of each title with their line numbers
        if title not in title_occurrences:
            title_occurrences[title] = []
        title_occurrences[title].append(line_number)
```

```
        # Store all line numbers where this title appears
        if 'line_numbers' not in titles_info[title]:
            titles_info[title]['line_numbers'] = []
        titles_info[title]['line_numbers'].append(line_number)

        # For backward compatibility, keep the first occurrence in 'line'
        if titles_info[title]['line'] is None:
            titles_info[title]['line'] = line_number

        line_number += 1

    # Create an ordered list of all title occurrences with their line numbers
    all_occurrences = []
    for title, occurrences in title_occurrences.items():
        for line_num in occurrences:
            all_occurrences.append((title, line_num))

    # Sort occurrences by line number
    all_occurrences.sort(key=lambda x: x[1])

    # Get indentation for each occurrence
    occurrence_indents = {}
    for title, line_num in all_occurrences:
        for line in lines[line_num:line_num+1]:  # Only check the current line
            indent = 0
            if '+' in line:
                symbol_index = line.find('+')
                # Count spaces before the '+' symbol
                j = symbol_index - 1
                while j >= 0 and line[j] == ' ':
                    indent += 1
                    j -= 1
            elif '-' in line:
                symbol_index = line.find('-')
                # Count spaces before the '-' symbol
                j = symbol_index - 1
                while j >= 0 and line[j] == ' ':
                    indent += 1
                    j -= 1
            occurrence_indents[(title, line_num)] = indent

    # REMOVED: The problematic forward pass that was reversing relationships

    # Enhanced backward pass for correct parent-child relationships
    for i, (title, line_num) in enumerate(all_occurrences):
        current_indent = occurrence_indents[(title, line_num)]
```

```
        # Skip root nodes (indentation 0) for processing
        if current_indent == 0:
            continue

        # Look for the immediately preceding node with lower indentation
        j = i - 1
        while j >= 0:
            prev_title, prev_line = all_occurrences[j]
            prev_indent = occurrence_indents[(prev_title, prev_line)]

            # If we find a node with less indentation, it's a child of current␣
 ↪node
            if prev_indent < current_indent:
                # In BayesDown: More indented node is a parent (cause) of less␣
 ↪indented node (effect)
                if title not in titles_info[prev_title]['parents']:
                    titles_info[prev_title]['parents'].append(title)
                if prev_title not in titles_info[title]['children']:
                    titles_info[title]['children'].append(prev_title)

                # Only need to find the immediate child (closest preceding node␣
 ↪with lower indentation)
                break

            j -= 1

    return titles_info

def convert_to_dataframe(titles_info, ArgDown):
    """Convert the titles information dictionary to a pandas DataFrame"""
    if ArgDown == True:
        df = pd.DataFrame(columns=['Title', 'Description', 'line',␣
 ↪'line_numbers', 'indentation',
                                   'indentation_levels', 'Parents', 'Children',␣
 ↪'instantiations'])
    else:
        df = pd.DataFrame(columns=['Title', 'Description', 'line',␣
 ↪'line_numbers', 'indentation',
                                   'indentation_levels', 'Parents', 'Children',␣
 ↪'instantiations',
                                   'priors', 'posteriors'])

    for title, info in titles_info.items():
        # Parse the metadata JSON string into a Python dictionary
        if 'metadata' in info and info['metadata']:
            try:
```

```
                # Only try to parse if metadata is not empty
                if info['metadata'].strip():
                    jsonMetadata = json.loads(info['metadata'])
                    if ArgDown == True:
                        # Create the row dictionary with instantitions as␣
↪metadata only, no probabilites yet
                        row = {
                            'Title': title,
                            'Description': info.get('description', ''),
                            'line': info.get('line',''),
                            'line_numbers': info.get('line_numbers', []),
                            'indentation': info.get('indentation',''),
                            'indentation_levels': info.
↪get('indentation_levels', []),
                            'Parents': info.get('parents', []),
                            'Children': info.get('children', []),
                            # Extract specific metadata fields, defaulting to␣
↪empty if not present
                            'instantiations': jsonMetadata.
↪get('instantiations', []),
                        }


                    else:
                        # create dict with probabilites
                        row = {
                            'Title': title,
                            'Description': info.get('description', ''),
                            'line': info.get('line',''),
                            'line_numbers': info.get('line_numbers', []),
                            'indentation': info.get('indentation',''),
                            'indentation_levels': info.
↪get('indentation_levels', []),
                            'Parents': info.get('parents', []),
                            'Children': info.get('children', []),
                            # Extract specific metadata fields, defaulting to␣
↪empty if not present
                            'instantiations': jsonMetadata.
↪get('instantiations', []),
                            'priors': jsonMetadata.get('priors', {}),
                            'posteriors': jsonMetadata.get('posteriors', {})
                        }
                else:
                    # Empty metadata case
                    row = {
                        'Title': title,
```

```
                        'Description': info.get('description', ''),
                        'line': info.get('line',''),
                        'line_numbers': info.get('line_numbers', []),
                        'indentation': info.get('indentation',''),
                        'indentation_levels': info.get('indentation_levels',␣
↪[]),
                        'Parents': info.get('parents', []),
                        'Children': info.get('children', []),
                        'instantiations': [],
                        'priors': {},
                        'posteriors': {}
                    }
            except json.JSONDecodeError:
                # Handle case where metadata isn't valid JSON
                row = {
                    'Title': title,
                    'Description': info.get('description', ''),
                    'line': info.get('line',''),
                    'line_numbers': info.get('line_numbers', []),
                    'indentation': info.get('indentation',''),
                    'indentation_levels': info.get('indentation_levels', []),
                    'Parents': info.get('parents', []),
                    'Children': info.get('children', []),
                    'instantiations': [],
                    'priors': {},
                    'posteriors': {}
                }
        else:
            # Handle case where metadata field doesn't exist or is empty
            row = {
                'Title': title,
                'Description': info.get('description', ''),
                'line': info.get('line',''),
                'line_numbers': info.get('line_numbers', []),
                'indentation': info.get('indentation',''),
                'indentation_levels': info.get('indentation_levels', []),
                'Parents': info.get('parents', []),
                'Children': info.get('children', []),
                'instantiations': [],
                'priors': {},
                'posteriors': {}
            }

        # Add the row to the DataFrame
        df.loc[len(df)] = row

    return df
```

```
def add_no_parent_no_child_columns_to_df(dataframe):
    """Add No_Parent and No_Children boolean columns to the DataFrame"""
    no_parent = []
    no_children = []

    for _, row in dataframe.iterrows():
        no_parent.append(not row['Parents'])
        no_children.append(not row['Children'])

    dataframe['No_Parent'] = no_parent
    dataframe['No_Children'] = no_children

    return dataframe

def add_parents_instantiation_columns_to_df(dataframe):
    """Add all possible instantiations of all parents as list with lists column␣
 ↪to the DataFrame"""
    # Create a new column to store parent instantiations
    parent_instantiations = []

    # Iterate through each row in the dataframe
    for _, row in dataframe.iterrows():
        parents = row['Parents']
        parent_insts = []

        # For each parent, find its instantiations and add to the list
        for parent in parents:
            # Find the row where Title matches the parent
            parent_row = dataframe[dataframe['Title'] == parent]

            # If parent found in the dataframe
            if not parent_row.empty:
                # Get the instantiations of this parent
                parent_instantiation = parent_row['instantiations'].iloc[0]
                parent_insts.append(parent_instantiation)

        # Add the list of parent instantiations to our new column
        parent_instantiations.append(parent_insts)

    # Add the new column to the dataframe
    dataframe['parent_instantiations'] = parent_instantiations

    return dataframe
```

[9]:
```
# example use case:
ex_csv = parse_markdown_hierarchy_fixed(md_content, ArgDown = True)
```

```
ex_csv
```

\# \\\# 1.8 Store ArgDown Information as 'ArgDown.csv' file

[10]:
```
# Assuming 'md_content' holds the markdown text
# Store the results of running the function␣
 ↪parse_markdown_hierarchy(md_content, ArgDown = True) as the file 'ArgDown.
 ↪csv'
result_df = parse_markdown_hierarchy_fixed(md_content, ArgDown = True)

# Save to CSV
result_df.to_csv('ArgDown.csv', index=False)
```

[11]:
```
# Test if 'ArgDown.csv' has been saved correctly with the correct information
# Load the data from the CSV file
argdown_df = pd.read_csv('ArgDown.csv')

# Display the DataFrame
print(argdown_df)
```

\# 2.0 Probability Extractions: ArgDown (.csv) to BayesDown (.md + plugin JSON syntax)

\# \\\# 2.1 Probability Extraction Questions — 'ArgDown.csv' to 'ArgDown_WithQuestions.csv'

[121]:
```
import pandas as pd
import re
import json
import itertools
from IPython.display import Markdown, display



def parse_instantiations(instantiations_str):
    """
    Parse instantiations from string or list format.
    Handles various input formats flexibly.
    """
    if pd.isna(instantiations_str) or instantiations_str == '':
        return []

    if isinstance(instantiations_str, list):
        return instantiations_str

    try:
        # Try to parse as JSON
        return json.loads(instantiations_str)
    except:
```

```
        # Try to parse as string list
        if isinstance(instantiations_str, str):
            # Remove brackets and split by comma
            clean_str = instantiations_str.strip('[]"\'')
            if not clean_str:
                return []
            return [s.strip(' "\'') for s in clean_str.split(',') if s.strip()]

    return []

def parse_parents(parents_str):
    """
    Parse parents from string or list format.
    Handles various input formats flexibly.
    """
    if pd.isna(parents_str) or parents_str == '':
        return []

    if isinstance(parents_str, list):
        return parents_str

    try:
        # Try to parse as JSON
        return json.loads(parents_str)
    except:
        # Try to parse as string list
        if isinstance(parents_str, str):
            # Remove brackets and split by comma
            clean_str = parents_str.strip('[]"\'')
            if not clean_str:
                return []
            return [s.strip(' "\'') for s in clean_str.split(',') if s.strip()]

    return []

def get_parent_instantiations(parent, df):
    """
    Get the instantiations for a parent node from the DataFrame.
    Returns default instantiations if not found.
    """
    parent_row = df[df['Title'] == parent]
    if parent_row.empty:
        return [f"{parent}_TRUE", f"{parent}_FALSE"]

    instantiations = parse_instantiations(parent_row.iloc[0]['instantiations'])
    if not instantiations:
        return [f"{parent}_TRUE", f"{parent}_FALSE"]
```

```
    return instantiations

def generate_instantiation_questions(title, instantiation, parents, df):
    """
    Generate questions for a specific instantiation of a node.

    Args:
        title (str): The title of the node
        instantiation (str): The specific instantiation (e.g., "title_TRUE")
        parents (list): List of parent nodes
        df (DataFrame): The full DataFrame for looking up parent instantiations

    Returns:
        dict: Dictionary mapping questions to estimate keys
    """
    questions = {}

    # Always generate a prior probability question, regardless of parents
    prior_question = f"What is the probability for {title}={instantiation}?"
    questions[prior_question] = 'prior'  # Change here: question is the key,␣
↪'prior' is the value

    # If no parents, return only the prior question
    if not parents:
        return questions

    # For nodes with parents, generate conditional probability questions
    # Get all combinations of parent instantiations
    parent_instantiations = []
    for parent in parents:
        parent_insts = get_parent_instantiations(parent, df)
        parent_instantiations.append([(parent, inst) for inst in parent_insts])

    # Generate all combinations
    all_combinations = list(itertools.product(*parent_instantiations))

    # Create conditional probability questions for each combination
    # and use questions as keys, estimate_i as values
    for i, combination in enumerate(all_combinations):
        condition_str = ", ".join([f"{parent}={inst}" for parent, inst in␣
↪combination])
        question = f"What is the probability for {title}={instantiation} if␣
↪{condition_str}?"
        questions[question] = f'estimate_{i + 1}'  # Change here: question is␣
↪the key, estimate_i is the value
```

```
    return questions


def generate_argdown_with_questions(argdown_csv_path, output_csv_path):
    """
    Generate probability questions based on the ArgDown CSV file and save to a␣
 ↪new CSV file.

    Args:
        argdown_csv_path (str): Path to the input ArgDown CSV file
        output_csv_path (str): Path to save the output CSV file with questions
    """
    print(f"Loading ArgDown CSV from {argdown_csv_path}...")

    # Load the ArgDown CSV file
    try:
        df = pd.read_csv(argdown_csv_path)
        print(f"Successfully loaded CSV with {len(df)} rows.")
    except Exception as e:
        raise Exception(f"Error loading ArgDown CSV: {e}")

    # Validate required columns
    required_columns = ['Title', 'Parents', 'instantiations']
    missing_columns = [col for col in required_columns if col not in df.columns]
    if missing_columns:
        raise Exception(f"Missing required columns: {', '.
 ↪join(missing_columns)}")

    # Initialize columns for questions
    df['Generate_Positive_Instantiation_Questions'] = None
    df['Generate_Negative_Instantiation_Questions'] = None

    print("Generating probability questions for each node...")

    # Process each row to generate questions
    for idx, row in df.iterrows():
        title = row['Title']
        instantiations = parse_instantiations(row['instantiations'])
        parents = parse_parents(row['Parents'])

        if len(instantiations) < 2:
            # Default instantiations if not provided
            instantiations = [f"{title}_TRUE", f"{title}_FALSE"]

        # Generate positive instantiation questions
        positive_questions = generate_instantiation_questions(title,␣
 ↪instantiations[0], parents, df)
```

```
        # Generate negative instantiation questions
        negative_questions = generate_instantiation_questions(title,␣
 ↪instantiations[1], parents, df)

        # Update the DataFrame
        df.at[idx, 'Generate_Positive_Instantiation_Questions'] = json.
 ↪dumps(positive_questions)
        df.at[idx, 'Generate_Negative_Instantiation_Questions'] = json.
 ↪dumps(negative_questions)

    # Save the enhanced DataFrame
    df.to_csv(output_csv_path, index=False)
    print(f"Generated questions saved to {output_csv_path}")

    return df

# Example usage:
df_with_questions = generate_argdown_with_questions("ArgDown.csv",␣
 ↪"ArgDown_WithQuestions.csv")
df_with_questions
```

[122]:
```
# Load the data from the ArgDown_WithQuestions CSV file
argdown_with_questions_df = pd.read_csv('ArgDown_WithQuestions.csv')

# Display the DataFrame
print(argdown_with_questions_df)
argdown_with_questions_df
```

\# \\\# 2.2 'ArgDown_WithQuestions.csv' to 'BayesDownQuestions.md'

2.2 Save BayesDown Extraction Questions as 'BayesDownQuestions.md'

[123]:
```
def extract_bayesdown_questions_fixed(argdown_with_questions_path,␣
 ↪output_md_path, include_questions_as_comments=True):
    """
    Generate BayesDown syntax from the ArgDown_WithQuestions CSV file with␣
 ↪correct parent-child relationships.

    Args:
        argdown_with_questions_path (str): Path to the CSV file with probability␣
 ↪questions
        output_md_path (str): Path to save the output BayesDown file
        include_questions_as_comments (bool, optional): Whether to include the␣
 ↪original
                                                        questions as comments.␣
 ↪Defaults to True.
```

```
"""
print(f"Loading CSV from {argdown_with_questions_path}...")

# Load the CSV file
try:
    df = pd.read_csv(argdown_with_questions_path)
    print(f"Successfully loaded CSV with {len(df)} rows.")
except Exception as e:
    raise Exception(f"Error loading CSV: {e}")

# Validate required columns
required_columns = ['Title', 'Description', 'Parents', 'Children',␣
↪'instantiations']
missing_columns = [col for col in required_columns if col not in df.columns]
if missing_columns:
    raise Exception(f"Missing required columns: {', '.join(missing_columns)}")

print("Generating BayesDown syntax with placeholder probabilities...")

# Build a directed graph of nodes
G = nx.DiGraph()

# Add nodes to the graph
for idx, row in df.iterrows():
    G.add_node(row['Title'], data=row)

# Add edges to the graph based on parent-child relationships - CORRECTLY
for idx, row in df.iterrows():
    child = row['Title']

    # Parse parents and add edges
    parents = row['Parents']
    if isinstance(parents, str):
        # Handle string representation of list
        if parents.startswith('[') and parents.endswith(']'):
            parents = parents.strip('[]')
            if parents:  # Check if not empty
                parent_list = [p.strip().strip('\'"') for p in parents.
↪split(',')]
                for parent in parent_list:
                    if parent in G.nodes():
                        # In BayesDown: Parent (cause) -> Child (effect)
                        G.add_edge(parent, child)
    elif isinstance(parents, list):
        # Handle actual list
        for parent in parents:
            if parent in G.nodes():
```

```
                G.add_edge(parent, child)

# Function to safely parse JSON strings
def safe_parse_json(json_str):
    if pd.isna(json_str):
        return {}

    if isinstance(json_str, dict):
        return json_str

    try:
        return json.loads(json_str)
    except:
        return {}

# Start building the BayesDown content
bayesdown_content = ""  # Initialize as empty

if include_questions_as_comments:
  bayesdown_content = "# BayesDown Representation with Placeholder␣
↪Probabilities\n\n"
  bayesdown_content += "/* This file contains BayesDown syntax with␣
↪placeholder probabilities.\n"
  bayesdown_content += "   Replace the placeholders with actual probability␣
↪values based on the \n"
  bayesdown_content += "   questions in the comments. */\n\n"

# Get leaf nodes (nodes with no outgoing edges) - these are effects without␣
↪children
leaf_nodes = [n for n in G.nodes() if G.out_degree(n) == 0]

# Helper function to process a node and its parents recursively
def process_node(node, indent_level=0, processed_nodes=None):
    if processed_nodes is None:
        processed_nodes = set()

    # Create the indentation string
    indent = ' ' * (indent_level * 2)
    prefix = f"{indent}+ " if indent_level > 0 else ""

    # Get node data
    node_data = G.nodes[node]['data']
    title = node_data['Title']
    description = node_data['Description'] if not pd.
↪isna(node_data['Description']) else ""

    # Parse instantiations from the row data
```

```
    instantiations = parse_instantiations_safely(node_data['instantiations'])

    # Build the node string
    node_output = ""

    # Add comments with questions if requested
    if include_questions_as_comments:
        # Add positive questions as comments
        if 'Generate_Positive_Instantiation_Questions' in node_data:
            positive_questions =␣
↪safe_parse_json(node_data['Generate_Positive_Instantiation_Questions'])
            for question in positive_questions.keys():
                node_output += f"{indent}/* {question} */\n"

        # Add negative questions as comments
        if 'Generate_Negative_Instantiation_Questions' in node_data:
            negative_questions =␣
↪safe_parse_json(node_data['Generate_Negative_Instantiation_Questions'])
            for question in negative_questions.keys():
                node_output += f"{indent}/* {question} */\n"

    # Check if this node was already fully defined elsewhere
    if node in processed_nodes:
        # Just add a reference to the node
        node_output += f"{prefix}[{title}]\n"
        return node_output

    # Mark this node as processed
    processed_nodes.add(node)

    # Prepare the metadata JSON
    metadata = {
        "instantiations": instantiations
    }

    # Add priors with full questions as keys
    priors = {}
    if 'Generate_Positive_Instantiation_Questions' in node_data:
        positive_questions =␣
↪safe_parse_json(node_data['Generate_Positive_Instantiation_Questions'])
        for question, estimate_key in positive_questions.items():
            if estimate_key == 'prior':
                priors[question] = "%?"  # Default placeholder

    if 'Generate_Negative_Instantiation_Questions' in node_data:
        negative_questions =␣
↪safe_parse_json(node_data['Generate_Negative_Instantiation_Questions'])
```

```
        for question, estimate_key in negative_questions.items():
            if estimate_key == 'prior':
                priors[question] = "%?"  # Default placeholder

    metadata["priors"] = priors

    # Add posteriors with full questions as keys
    parents = list(G.predecessors(node))
    if parents:
        posteriors = {}
        if 'Generate_Positive_Instantiation_Questions' in node_data:
            positive_questions =␣
↪safe_parse_json(node_data['Generate_Positive_Instantiation_Questions'])
            for question, estimate_key in positive_questions.items():
                if estimate_key.startswith('estimate_'):
                    posteriors[question] = "?%"  # Default placeholder

        if 'Generate_Negative_Instantiation_Questions' in node_data:
            negative_questions =␣
↪safe_parse_json(node_data['Generate_Negative_Instantiation_Questions'])
            for question, estimate_key in negative_questions.items():
                if estimate_key.startswith('estimate_'):
                    posteriors[question] = "?%"  # Default placeholder

        metadata["posteriors"] = posteriors

    # Format the node with metadata
    node_output += f"{prefix}[{title}]: {description} {json.
↪dumps(metadata)}\n"

    # Process parent nodes
    for parent in parents:
        if parent != node:  # Avoid self-references
            parent_output = process_node(parent, indent_level + 1,␣
↪processed_nodes)
            node_output += parent_output

    return node_output

# Helper function to parse instantiations safely
def parse_instantiations_safely(instantiations_data):
    if isinstance(instantiations_data, list):
        return instantiations_data if instantiations_data else [f"TRUE",␣
↪f"FALSE"]

    if isinstance(instantiations_data, str):
        try:
```

```
                parsed = json.loads(instantiations_data)
                if isinstance(parsed, list):
                    return parsed if parsed else [f"TRUE", f"FALSE"]
            except:
                if instantiations_data.startswith('[') and instantiations_data.
 ↪endswith(']'):
                    items = instantiations_data.strip('[]').split(',')
                    result = [item.strip(' "\'') for item in items if item.
 ↪strip()]
                    return result if result else [f"TRUE", f"FALSE"]

        return [f"TRUE", f"FALSE"]  # Default

    # Process each leaf node and its ancestors
    for leaf in leaf_nodes:
        bayesdown_content += process_node(leaf)

    # Save the BayesDown content
    with open(output_md_path, 'w') as f:
        f.write(bayesdown_content)

    print(f"BayesDown Questions saved to {output_md_path}")
    return bayesdown_content
```

[124]:
```
# Explicitly set the value of include_questions_as_comments
include_questions_as_comments=False  # or False, depending on your needs

# Get the markdown content
bayesdown_questions = extract_bayesdown_questions_fixed(
  "ArgDown_WithQuestions.csv",
  "BayesDownQuestions.md",␣
 ↪include_questions_as_comments=include_questions_as_comments
)

# Determine the output file path based on include_questions_as_comments
if include_questions_as_comments: # Assuming include_questions_as_comments is␣
 ↪defined somewhere in previous cells
    output_file_path = "FULL_BayesDownQuestions.md"
else:
    output_file_path = "BayesDownQuestions.md"

# Save the markdown content to the appropriate file
with open(output_file_path, 'w') as f:
    f.write(md_content)

print(f"Markdown content saved to {output_file_path}")
```

[125]:
```
# Generate BayesDown format
bayesdown_questions = extract_bayesdown_questions_fixed(
    "ArgDown_WithQuestions.csv",
    "FULL_BayesDownQuestions.md",
    include_questions_as_comments=True
)

# Display a preview of the format
print("\nBayesDown Format Preview:")
print(bayesdown_questions[:5000] + "...\n")
```

[126]:
```
# Load and print the content of the 'FULL_BayesDownQuestions.md' file
with open("FULL_BayesDownQuestions.md", "r") as f:
    file_content = f.read()
    print(file_content)
```

[127]:
```
# Generate BayesDown format
bayesdown_questions = extract_bayesdown_questions_fixed(
    "ArgDown_WithQuestions.csv",
    "BayesDownQuestions.md",
    include_questions_as_comments=False
)

# Display a preview of the format
print(

)
print(bayesdown_questions[:5000] + "...\n")
```

[128]:
```
# Load and print the content of the 'BayesDownQuestions.md' file
with open("BayesDownQuestions.md", "r") as f:
    file_content = f.read()
    print(file_content)
```

\# \\\# 2.3 Generate BayesDown Probability Extraction Prompt

Generate 2nd Extraction Prompt for Probabilities based on the questions generated from the 'ArgDown.csv' extraction

\# \\\# 2.4 Prepare 2nd API call

\# \\\# 2.5 Make BayesDown Probability Extraction API Call

\# \\\# 2.6 Save BayesDown with Probability Estimates (.csv)

\# \\\# 2.7 Review \\\& Verify BayesDown Probability Estimates

\# \\\# 2.7.2 Check the Graph Structure with the ArgDown Sandbox Online Copy and paste the BayesDown formatted … in the ArgDown Sandbox below to quickly verify that the network renders correctly.

\# \\\#\\\# 2.3.1 BayesDown Format Specification

BayesDown augments ArgDown with probability data in a structured JSON format:

“‘json \{ “instantiations”: [”state_TRUE“,”state_FALSE”], “priors”: \{ ”p(state_TRUE)“:”0.7“,”p(state_FALSE)“:”0.3” \}, “posteriors”: \{ ”p(state_TRUE|condition1_TRUE,condition2_FALSE)“:”0.9“,”p(state_TRUE|condition1_FALSE,conditi \} \}

2.3.2 Probability Extraction Process The probability extraction pipeline follows these steps:

Identify variables and their possible states Extract prior probability statements Identify conditional relationships Extract conditional probability statements Format the data in BayesDown syntax

2.3.3 Implementation Steps To extract probabilities and create BayesDown format:

Run the extract_probabilities function on ArgDown text Process the results into a structured format Validate the probability distributions (ensure they sum to 1) Generate the enhanced BayesDown representation

2.3.4 Validation and Quality Control The probability extraction process includes validation steps:

Ensuring coherent probability distributions Checking for logical consistency in conditional relationships Verifying that all required probability statements are present Handling missing data with appropriate default values

\# \\\# 2.8 Extract BayesDown with Probability Estimates as Dataframe

\# 3.0 Data Extraction: BayesDown (.md) to Database (.csv)

\# \\\#\\\# 3.1 ExtractBayesDown-Data_v1 Build data frame with extractable information from BayesDown

[53]:
```
# read sprinkler example -- Occam Colab Online
file_path_ex_rain = "https://raw.githubusercontent.com/SingularitySmith/
 ↪AMTAIR_Prototype/main/data/example_1/BayesDown_Example.md"

# Use requests.get to fetch content from URL
response = requests.get(file_path_ex_rain)
response.raise_for_status()  # Raise HTTPError for bad responses (4xx or 5xx)

# Read content from the response
md_content_ex_rain = response.text

md_content_ex_rain
```

\# \\\# 3.1.2 Test BayesDown Extraction

[54]:
```
display(Markdown(md_content_ex_rain)) # view BayesDown file formatted as␣
 ↪MarkDown
```

\# \\\# 3.1.2.2 Check the Graph Structure with the ArgDown Sandbox Online Copy and paste the BayesDown formatted … in the ArgDown Sandbox below to quickly verify that the network renders correctly.

\# \\\#\\\# 3.1.2.B Test with 'Example_file_combined_withBayesDown_Crossgenerational.md'

[55]:
```
# read basic ArgDown example With BayesDown syntax added and corss generational␣
 ↪added
import requests  # Import the requests library

# **Corrected URL with /main/**
file_path_easy_ex_B_CG = "https://raw.githubusercontent.com/SingularitySmith/
 ↪AMTAIR_Prototype/main/Example_file_combined_withBayesDown_Crossgenerational.
 ↪md"

# Use requests.get to fetch content from URL
response = requests.get(file_path_easy_ex_B_CG)
response.raise_for_status()  # Raise HTTPError for bad responses (4xx or 5xx)

# Read content from the response
md_content_easy_ex_B_CG = response.text

md_content_easy_ex_B_CG
```

\# \\\# 3.3 Extraction BayesDown Extraction Code already part of ArgDown extraction code, therefore just use same function "parse_markdown_hierarchy(markdown_data)" and ignore the extra argument ("ArgDown") because it is automatically set to false amd will by default extract BayesDown.

[56]:
```
result_df = parse_markdown_hierarchy(md_content_ex_rain)
result_df
```

\# \\\#\\\# 3.3 Data-Post-Processing Add rows to data frame that can be calculated from the extracted rows

[14]:
```
# @title 3.3.1 Data Post-Processing Functions ---

# --- 3.3 Data-Post-Processing ---

def enhance_extracted_data(df):
    """
    Enhance the extracted data with calculated columns

    Args:
        df: DataFrame with extracted BayesDown data

    Returns:
        Enhanced DataFrame with additional columns
    """
    # Create a copy to avoid modifying the original
    enhanced_df = df.copy()
```

```
    # 1. Calculate joint probabilities
    enhanced_df['joint_probabilities'] = None

    for idx, row in enhanced_df.iterrows():
        title = row['Title']
        priors = row['priors'] if isinstance(row['priors'], dict) else {}
        posteriors = row['posteriors'] if isinstance(row['posteriors'], dict)␣
↪else {}
        parents = row['Parents'] if isinstance(row['Parents'], list) else []

        # Skip if no parents or no priors
        if not parents or not priors:
            continue

        # Initialize joint probabilities dictionary
        joint_probs = {}

        # Get instantiations
        instantiations = row['instantiations']
        if not isinstance(instantiations, list) or not instantiations:
            continue

        # For each parent and child instantiation combination, calculate joint␣
↪probability
        for inst in instantiations:
            # Get this instantiation's prior probability
            inst_prior_key = f"p({inst})"
            if inst_prior_key not in priors:
                continue

            try:
                inst_prior = float(priors[inst_prior_key])
            except (ValueError, TypeError):
                continue

            # For each parent
            for parent in parents:
                parent_row = enhanced_df[enhanced_df['Title'] == parent]
                if parent_row.empty:
                    continue

                parent_insts = parent_row.iloc[0]['instantiations']
                if not isinstance(parent_insts, list) or not parent_insts:
                    continue

                for parent_inst in parent_insts:
                    # Get conditional probability
```

```
                    cond_key = f"p({inst}|{parent}={parent_inst})"
                    if cond_key in posteriors:
                        try:
                            cond_prob = float(posteriors[cond_key])

                            # Get parent's prior
                            parent_priors = parent_row.iloc[0]['priors']
                            if not isinstance(parent_priors, dict):
                                continue

                            parent_prior_key = f"p({parent_inst})"
                            if parent_prior_key not in parent_priors:
                                continue

                            try:
                                parent_prior =␣
↪float(parent_priors[parent_prior_key])

                                # Calculate joint probability: P(A,B) = P(A|B)␣
↪* P(B)
                                joint_prob = cond_prob * parent_prior
                                joint_key = f"p({inst},{parent}={parent_inst})"
                                joint_probs[joint_key] = str(round(joint_prob,␣
↪4))
                            except (ValueError, TypeError):
                                continue
                        except (ValueError, TypeError):
                            continue

        # Store joint probabilities in dataframe
        enhanced_df.at[idx, 'joint_probabilities'] = joint_probs

    # 2. Calculate network metrics
    # Create a directed graph
    import networkx as nx
    G = nx.DiGraph()

    # Add nodes
    for idx, row in enhanced_df.iterrows():
        G.add_node(row['Title'])

    # Add edges
    for idx, row in enhanced_df.iterrows():
        child = row['Title']
        parents = row['Parents'] if isinstance(row['Parents'], list) else []

        for parent in parents:
```

```
            if parent in G.nodes():
                G.add_edge(parent, child)

    # Calculate centrality measures
    degree_centrality = nx.degree_centrality(G)
    in_degree_centrality = nx.in_degree_centrality(G)
    out_degree_centrality = nx.out_degree_centrality(G)

    try:
        betweenness_centrality = nx.betweenness_centrality(G)
    except:
        betweenness_centrality = {node: 0 for node in G.nodes()}

    # Add metrics to dataframe
    enhanced_df['degree_centrality'] = None
    enhanced_df['in_degree_centrality'] = None
    enhanced_df['out_degree_centrality'] = None
    enhanced_df['betweenness_centrality'] = None

    for idx, row in enhanced_df.iterrows():
        title = row['Title']
        enhanced_df.at[idx, 'degree_centrality'] = degree_centrality.get(title,␣
↪0)
        enhanced_df.at[idx, 'in_degree_centrality'] = in_degree_centrality.
↪get(title, 0)
        enhanced_df.at[idx, 'out_degree_centrality'] = out_degree_centrality.
↪get(title, 0)
        enhanced_df.at[idx, 'betweenness_centrality'] = betweenness_centrality.
↪get(title, 0)

    # 3. Add Markov blanket information (parents, children, and children's␣
↪parents)
    enhanced_df['markov_blanket'] = None

    for idx, row in enhanced_df.iterrows():
        title = row['Title']
        parents = row['Parents'] if isinstance(row['Parents'], list) else []
        children = row['Children'] if isinstance(row['Children'], list) else []

        # Get children's parents (excluding this node)
        childrens_parents = []
        for child in children:
            child_row = enhanced_df[enhanced_df['Title'] == child]
            if not child_row.empty:
                child_parents = child_row.iloc[0]['Parents']
                if isinstance(child_parents, list):
```

```
                    childrens_parents.extend([p for p in child_parents if p !=␣
 ↪title])

        # Remove duplicates
        childrens_parents = list(set(childrens_parents))

        # Combine to get Markov blanket
        markov_blanket = list(set(parents + children + childrens_parents))
        enhanced_df.at[idx, 'markov_blanket'] = markov_blanket

    return enhanced_df
```

[ ]:
```
# here we add all the rows that we have to calculate (joint probability...,␣
 ↪maybe in several rounds (e.g. first add conditional proability, then use␣
 ↪this column to calc joint probability...)

# 3.3 Data Post-Processing

# Enhance the extracted dataframe with calculated columns
enhanced_df = enhance_extracted_data(result_df)

# Display the enhanced dataframe
print("Enhanced DataFrame with additional calculated columns:")
enhanced_df.head()

# Check some calculated metrics
print("\nJoint Probabilities Example:")
example_node = enhanced_df.loc[0, 'Title']
joint_probs = enhanced_df.loc[0, 'joint_probabilities']
print(f"Joint probabilities for {example_node}:")
print(joint_probs)

print("\nNetwork Metrics:")
for idx, row in enhanced_df.iterrows():
    print(f"{row['Title']}:")
    print(f"  Degree Centrality: {row['degree_centrality']:.3f}")
    print(f"  Betweenness Centrality: {row['betweenness_centrality']:.3f}")

# Save the enhanced dataframe
enhanced_df.to_csv('enhanced_extracted_data.csv', index=False)
print("\nEnhanced data saved to 'enhanced_extracted_data.csv'")
```

\# \\\#\\\# 3.4 Download and save finished data frame as .csv file

[ ]:
```
result_df.to_csv('extracted_data.csv', index=False) # save dataframe in␣
 ↪environment as .csv file
```

```
# Attention: if the new or updated .csv file is required later, it needs to be␣
 ↪pushed to the GitRepository!
```

\# 4.0 Analysis \\\& Inference: Practical Software Tools ()

\# \\\# Phase 1: Dependencies/Functions

[ ]:
```
from pyvis.network import Network
import networkx as nx
from IPython.display import HTML
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import io
import base64
import colorsys
import json

def create_bayesian_network_with_probabilities(df):
    """
    Create an interactive Bayesian network visualization with enhanced␣
 ↪probability visualization
    and node classification based on network structure.
    """
    # Create a directed graph
    G = nx.DiGraph()

    # Add nodes with proper attributes
    for idx, row in df.iterrows():
        title = row['Title']
        description = row['Description']

        # Process probability information
        priors = get_priors(row)
        instantiations = get_instantiations(row)

        # Add node with base information
        G.add_node(
            title,
            description=description,
            priors=priors,
            instantiations=instantiations,
            posteriors=get_posteriors(row)
        )

    # Add edges
    for idx, row in df.iterrows():
        child = row['Title']
```

```
        parents = get_parents(row)

        # Add edges from each parent to this child
        for parent in parents:
            if parent in G.nodes():
                G.add_edge(parent, child)

    # Classify nodes based on network structure
    classify_nodes(G)

    # Create network visualization
    net = Network(notebook=True, directed=True, cdn_resources="in_line",␣
↪height="600px", width="100%")

    # Configure physics for better layout
    net.force_atlas_2based(gravity=-50, spring_length=100, spring_strength=0.02)
    net.show_buttons(filter_=['physics'])

    # Add the graph to the network
    net.from_nx(G)

    # Enhance node appearance with probability information and classification
    for node in net.nodes:
        node_id = node['id']
        node_data = G.nodes[node_id]

        # Get node type and set border color
        node_type = node_data.get('node_type', 'unknown')
        border_color = get_border_color(node_type)

        # Get probability information
        priors = node_data.get('priors', {})
        true_prob = priors.get('true_prob', 0.5) if priors else 0.5

        # Get proper state names
        instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])
        true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
        false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

        # Create background color based on probability
        background_color = get_probability_color(priors)

        # Create tooltip with probability information
        tooltip = create_tooltip(node_id, node_data)

        # Create a simpler node label with probability
        simple_label = f"{node_id}\np={true_prob:.2f}"
```

```
        # Store expanded content as a node attribute for use in click handler
        node_data['expanded_content'] = create_expanded_content(node_id,␣
↪node_data)

        # Set node attributes
        node['title'] = tooltip  # Tooltip HTML
        node['label'] = simple_label  # Simple text label
        node['shape'] = 'box'
        node['color'] = {
            'background': background_color,
            'border': border_color,
            'highlight': {
                'background': background_color,
                'border': border_color
            }
        }

    # Set up the click handler with proper data
    setup_data = {
        'nodes_data': {node_id: {
            'expanded_content': json.dumps(G.nodes[node_id].
↪get('expanded_content', '')),
            'description': G.nodes[node_id].get('description', ''),
            'priors': G.nodes[node_id].get('priors', {}),
            'posteriors': G.nodes[node_id].get('posteriors', {})
        } for node_id in G.nodes()}
    }

    # Add custom click handling JavaScript
    click_js = """
    // Store node data for click handling
    var nodesData = %s;

    // Add event listener for node clicks
    network.on("click", function(params) {
        if (params.nodes.length > 0) {
            var nodeId = params.nodes[0];
            var nodeInfo = nodesData[nodeId];

            if (nodeInfo) {
                // Create a modal popup for expanded content
                var modal = document.createElement('div');
                modal.style.position = 'fixed';
                modal.style.left = '50%%';
                modal.style.top = '50%%';
                modal.style.transform = 'translate(-50%%, -50%%)';
```

```
                modal.style.backgroundColor = 'white';
                modal.style.padding = '20px';
                modal.style.borderRadius = '5px';
                modal.style.boxShadow = '0 0 10px rgba(0,0,0,0.5)';
                modal.style.zIndex = '1000';
                modal.style.maxWidth = '80%%';
                modal.style.maxHeight = '80%%';
                modal.style.overflow = 'auto';

                // Add expanded content
                modal.innerHTML = nodeInfo.expanded_content || 'No detailed␣
↪information available';

                // Add close button
                var closeBtn = document.createElement('button');
                closeBtn.innerHTML = 'Close';
                closeBtn.style.marginTop = '10px';
                closeBtn.style.padding = '5px 10px';
                closeBtn.style.cursor = 'pointer';
                closeBtn.onclick = function() {
                    document.body.removeChild(modal);
                };
                modal.appendChild(closeBtn);

                // Add modal to body
                document.body.appendChild(modal);
            }
        }
    });
    """ % json.dumps(setup_data['nodes_data'])

    # Save the graph to HTML
    html_file = "bayesian_network.html"
    net.save_graph(html_file)

    # Inject custom click handling into HTML
    try:
        with open(html_file, "r") as f:
            html_content = f.read()

        # Insert click handling script before the closing body tag
        html_content = html_content.replace('</body>', f'<script>{click_js}</
↪script></body>')

        # Write back the modified HTML
        with open(html_file, "w") as f:
            f.write(html_content)
```

```
        return HTML(html_content)
    except Exception as e:
        return HTML(f"<p>Error rendering HTML: {str(e)}</p><p>The network␣
↪visualization has been saved to '{html_file}'</p>")

def classify_nodes(G):
    """
    Classify nodes as parent, child, or leaf based on network structure
    """
    for node in G.nodes():
        predecessors = list(G.predecessors(node))
        successors = list(G.successors(node))

        if not predecessors:  # No parents
            if successors:  # Has children
                G.nodes[node]['node_type'] = 'parent'
            else:  # No children either
                G.nodes[node]['node_type'] = 'isolated'
        else:  # Has parents
            if not successors:  # No children
                G.nodes[node]['node_type'] = 'leaf'
            else:  # Has both parents and children
                G.nodes[node]['node_type'] = 'child'

def get_border_color(node_type):
    """
    Return border color based on node type
    """
    if node_type == 'parent':
        return '#0000FF'  # Blue
    elif node_type == 'child':
        return '#800080'  # Purple
    elif node_type == 'leaf':
        return '#FF00FF'  # Magenta
    else:
        return '#000000'  # Default black

def get_probability_color(priors):
    """
    Create background color based on probability (red to green gradient)
    """
    # Default to neutral color if no probability
    if not priors or 'true_prob' not in priors:
        return '#F8F8F8'  # Light grey

    # Get probability value
```

```
    prob = priors['true_prob']

    # Create color gradient from red (0.0) to green (1.0)
    hue = 120 * prob  # 0 = red, 120 = green (in HSL color space)
    saturation = 0.75
    lightness = 0.8  # Lighter color for better text visibility

    # Convert HSL to RGB
    r, g, b = colorsys.hls_to_rgb(hue/360, lightness, saturation)

    # Convert to hex format
    hex_color = "#{:02x}{:02x}{:02x}".format(int(r*255), int(g*255), int(b*255))

    return hex_color

def create_tooltip(node_id, node_data):
    """
    Create rich HTML tooltip with probability information
    Uses simplified HTML that works well in tooltips
    """
    description = node_data.get('description', '')
    priors = node_data.get('priors', {})
    instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])

    # Start building the HTML tooltip
    html = f"""
    <div style='max-width:350px; padding:10px; background-color:#f8f9fa;␣
 ↪border-radius:5px; font-family:Arial, sans-serif;'>
        <h3 style='margin-top:0; color:#202124;'>{node_id}</h3>
        <p style='font-style:italic;'>{description}</p>
    """

    # Add probability information if available
    if priors and 'true_prob' in priors:
        true_prob = priors['true_prob']
        false_prob = 1.0 - true_prob

        # Get proper state names
        true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
        false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

        html += f"""
        <div style='margin-top:10px; background-color:#fff; padding:8px;␣
 ↪border-radius:4px; border:1px solid #ddd;'>
            <h4 style='margin-top:0; font-size:14px;'>Probabilities:</h4>
            <div>{true_state}: <b>{true_prob:.3f}</b></div>
            <div>{false_state}: <b>{false_prob:.3f}</b></div>
```

```
            <div style='width:100%; height:20px; margin-top:5px; border:1px␣
↪solid #ccc;'>
                <div style='float:left; width:{true_prob*100}%; height:100%;␣
↪background-color:rgba(0,200,0,0.5); border-right:2px solid green;'></div>
                <div style='float:left; width:{false_prob*100}%; height:100%;␣
↪background-color:rgba(255,0,0,0.5);'></div>
            </div>
        </div>
        """

    # Add click instruction
    html += """
    <div style='margin-top:10px; font-size:12px; text-align:center; color:#666;
↪'>
        Click for detailed information
    </div>
    """

    # Close the main div
    html += "</div>"

    return html

def create_expanded_content(node_id, node_data):
    """
    Create expanded content shown when a node is clicked
    This is stored as a string and converted to HTML in the click handler
    """
    description = node_data.get('description', '')
    priors = node_data.get('priors', {})
    posteriors = node_data.get('posteriors', {})
    instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])

    # Get probability values
    true_prob = priors.get('true_prob', 0.5) if priors else 0.5
    false_prob = 1.0 - true_prob

    # Get proper state names
    true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
    false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

    # Start building HTML content
    html = f"""
    <div style="max-width:600px; padding:20px;">
        <h2 style="margin-top:0;">{node_id}</h2>
        <p style="font-style:italic;">{description}</p>
```

```
        <div style="margin-top:20px;">
            <h3>Prior Probabilities</h3>
            <table style="width:100%; border-collapse:collapse;">
                <tr style="background-color:#f0f0f0;">
                    <th style="padding:8px; border:1px solid #ddd; text-align:
↪left;">State</th>
                    <th style="padding:8px; border:1px solid #ddd; text-align:
↪right;">Probability</th>
                    <th style="padding:8px; border:1px solid #ddd;
↪">Visualization</th>
                </tr>
                <tr>
                    <td style="padding:8px; border:1px solid #ddd;
↪">{true_state}</td>
                    <td style="padding:8px; border:1px solid #ddd; text-align:
↪right;">{true_prob:.3f}</td>
                    <td style="padding:8px; border:1px solid #ddd;">
                        <div style="width:100%; height:20px; background-color:
↪#f0f0f0;">
                            <div style="width:{true_prob*100}%; height:100%;␣
↪background-color:rgba(0,200,0,0.5);"></div>
                        </div>
                    </td>
                </tr>
                <tr>
                    <td style="padding:8px; border:1px solid #ddd;
↪">{false_state}</td>
                    <td style="padding:8px; border:1px solid #ddd; text-align:
↪right;">{false_prob:.3f}</td>
                    <td style="padding:8px; border:1px solid #ddd;">
                        <div style="width:100%; height:20px; background-color:
↪#f0f0f0;">
                            <div style="width:{false_prob*100}%; height:100%;␣
↪background-color:rgba(255,0,0,0.5);"></div>
                        </div>
                    </td>
                </tr>
            </table>
        </div>
    """

    # Add conditional probabilities if available
    if posteriors and len(posteriors) > 0:
        html += """
        <div style="margin-top:20px;">
            <h3>Conditional Probabilities</h3>
```

```
            <table style="width:100%; border-collapse:collapse;">
                <tr style="background-color:#f0f0f0;">
                    <th style="padding:8px; border:1px solid #ddd; text-align:
↪left;">Condition</th>
                    <th style="padding:8px; border:1px solid #ddd; text-align:
↪right;">Value</th>
                </tr>
        """

        # Add each conditional probability
        for key, value in posteriors.items():
            html += f"""
            <tr>
                <td style="padding:8px; border:1px solid #ddd;">{key}</td>
                <td style="padding:8px; border:1px solid #ddd; text-align:right;
↪">{value}</td>
            </tr>
            """

        html += """
            </table>
        </div>
        """

    # Close the main container
    html += """
    </div>
    """

    return html
```

\# \\\# Phase 2: Node Classification and Styling Module

[ ]:
```
def classify_nodes(G):
    """
    Classify nodes as parent, child, or leaf based on network structure
    """
    for node in G.nodes():
        predecessors = list(G.predecessors(node))
        successors = list(G.successors(node))

        if not predecessors:  # No parents
            if successors:  # Has children
                G.nodes[node]['node_type'] = 'parent'
            else:  # No children either
                G.nodes[node]['node_type'] = 'isolated'
        else:  # Has parents
```

```
            if not successors:  # No children
                G.nodes[node]['node_type'] = 'leaf'
            else:  # Has both parents and children
                G.nodes[node]['node_type'] = 'child'

def get_border_color(node_type):
    """
    Return border color based on node type
    """
    if node_type == 'parent':
        return '#0000FF'  # Blue
    elif node_type == 'child':
        return '#800080'  # Purple
    elif node_type == 'leaf':
        return '#FF00FF'  # Magenta
    else:
        return '#000000'  # Default black

def get_probability_color(priors):
    """
    Create background color based on probability (red to green gradient)
    """
    # Default to neutral color if no probability
    if not priors or 'true_prob' not in priors:
        return '#F8F8F8'  # Light grey

    # Get probability value
    prob = priors['true_prob']

    # Create color gradient from red (0.0) to green (1.0)
    hue = 120 * prob  # 0 = red, 120 = green (in HSL color space)
    saturation = 0.75
    lightness = 0.8  # Lighter color for better text visibility

    # Convert HSL to RGB
    r, g, b = colorsys.hls_to_rgb(hue/360, lightness, saturation)

    # Convert to hex format
    hex_color = "#{:02x}{:02x}{:02x}".format(int(r*255), int(g*255), int(b*255))

    return hex_color

def get_parents(row):
    """
    Extract parent nodes from row data, with safe handling for different data␣
 ↪types
    """
```

```
    if 'Parents' not in row:
        return []

    parents_data = row['Parents']

    # Handle NaN, None, or empty list
    if isinstance(parents_data, float) and pd.isna(parents_data):
        return []

    if parents_data is None:
        return []

    # Handle different data types
    if isinstance(parents_data, list):
        # Return a list with NaN and empty strings removed
        return [p for p in parents_data if not (isinstance(p, float) and pd.
 ↪isna(p)) and p != '']

    if isinstance(parents_data, str):
        if not parents_data.strip():
            return []

        # Remove brackets and split by comma, removing empty strings and NaN
        cleaned = parents_data.strip('[]"\'')
        if not cleaned:
            return []

        return [p.strip(' "\'') for p in cleaned.split(',') if p.strip()]

    # Default: empty list
    return []

def get_instantiations(row):
    """
    Extract instantiations with safe handling for different data types
    """
    if 'instantiations' not in row:
        return ["TRUE", "FALSE"]

    inst_data = row['instantiations']

    # Handle NaN or None
    if isinstance(inst_data, float) and pd.isna(inst_data):
        return ["TRUE", "FALSE"]

    if inst_data is None:
        return ["TRUE", "FALSE"]
```

```
    # Handle different data types
    if isinstance(inst_data, list):
        return inst_data if inst_data else ["TRUE", "FALSE"]

    if isinstance(inst_data, str):
        if not inst_data.strip():
            return ["TRUE", "FALSE"]

        # Remove brackets and split by comma
        cleaned = inst_data.strip('[]"\'')
        if not cleaned:
            return ["TRUE", "FALSE"]

        return [i.strip(' "\'') for i in cleaned.split(',') if i.strip()]

    # Default
    return ["TRUE", "FALSE"]

def get_priors(row):
    """
    Extract prior probabilities with safe handling for different data types
    """
    if 'priors' not in row:
        return {}

    priors_data = row['priors']

    # Handle NaN or None
    if isinstance(priors_data, float) and pd.isna(priors_data):
        return {}

    if priors_data is None:
        return {}

    result = {}

    # Handle dictionary
    if isinstance(priors_data, dict):
        result = priors_data
    # Handle string representation of dictionary
    elif isinstance(priors_data, str):
        if not priors_data.strip() or priors_data == '{}':
            return {}

        try:
            # Try to evaluate as Python literal
```

```
            import ast
            result = ast.literal_eval(priors_data)
        except:
            # Simple parsing for items like {'p(TRUE)': '0.2', 'p(FALSE)': '0.
↪8'}
            if '{' in priors_data and '}' in priors_data:
                content = priors_data[priors_data.find('{')+1:priors_data.
↪rfind('}')]
                items = [item.strip() for item in content.split(',')]

                for item in items:
                    if ':' in item:
                        key, value = item.split(':', 1)
                        key = key.strip(' \'\"')
                        value = value.strip(' \'\"')
                        result[key] = value

    # Extract main probability for TRUE state
    instantiations = get_instantiations(row)
    true_state = instantiations[0] if instantiations else "TRUE"
    true_key = f"p({true_state})"

    if true_key in result:
        try:
            result['true_prob'] = float(result[true_key])
        except:
            pass

    return result

def get_posteriors(row):
    """
    Extract posterior probabilities with safe handling for different data types
    """
    if 'posteriors' not in row:
        return {}

    posteriors_data = row['posteriors']

    # Handle NaN or None
    if isinstance(posteriors_data, float) and pd.isna(posteriors_data):
        return {}

    if posteriors_data is None:
        return {}

    result = {}
```

```
    # Handle dictionary
    if isinstance(posteriors_data, dict):
        result = posteriors_data
    # Handle string representation of dictionary
    elif isinstance(posteriors_data, str):
        if not posteriors_data.strip() or posteriors_data == '{}':
            return {}

        try:
            # Try to evaluate as Python literal
            import ast
            result = ast.literal_eval(posteriors_data)
        except:
            # Simple parsing
            if '{' in posteriors_data and '}' in posteriors_data:
                content = posteriors_data[posteriors_data.find('{')+1:
↪posteriors_data.rfind('}')]
                items = [item.strip() for item in content.split(',')]

                for item in items:
                    if ':' in item:
                        key, value = item.split(':', 1)
                        key = key.strip(' \'\"')
                        value = value.strip(' \'\"')
                        result[key] = value

    return result
```

\# \\\# Phase 3: HTML Content Generation Module

[ ]:
```
def create_probability_bar(true_prob, false_prob, height="15px",␣
↪show_values=True, value_prefix=""):
    """
    Creates a reusable HTML component to visualize probability distribution
    """
    true_label = f"{value_prefix}{true_prob:.3f}" if show_values else ""
    false_label = f"{value_prefix}{false_prob:.3f}" if show_values else ""

    html = f"""
    <div style="width:100%; height:{height}; display:flex; border:1px solid␣
↪#ccc; overflow:hidden; border-radius:3px; margin-top:3px; margin-bottom:3px;
↪">
        <div style="flex-basis:{true_prob*100}%; background:linear-gradient(to␣
↪bottom, rgba(0,180,0,0.9), rgba(0,140,0,0.7)); border-right:2px solid␣
↪#008800; display:flex; align-items:center; justify-content:center; overflow:
↪hidden; min-width:{2 if true_prob > 0 else 0}px;">
```

```
            <span style="font-size:10px; color:white; text-shadow:0px 0px 2px␣
↪#000;">{true_label}</span>
        </div>
        <div style="flex-basis:{false_prob*100}%; background:linear-gradient(to␣
↪bottom, rgba(220,0,0,0.9), rgba(180,0,0,0.7)); border-left:2px solid #880000;
↪ display:flex; align-items:center; justify-content:center; overflow:hidden;␣
↪min-width:{2 if false_prob > 0 else 0}px;">
            <span style="font-size:10px; color:white; text-shadow:0px 0px 2px␣
↪#000;">{false_label}</span>
        </div>
    </div>
    """
    return html

def create_tooltip(node_id, node_data):
    """
    Create rich HTML tooltip with probability information
    """
    description = node_data.get('description', '')
    priors = node_data.get('priors', {})
    instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])

    # Start building the HTML tooltip
    html = f"""
    <div style="max-width:350px; padding:10px; background-color:#f8f9fa;␣
↪border-radius:5px; font-family:Arial, sans-serif;">
        <h3 style="margin-top:0; color:#202124;">{node_id}</h3>
        <p style="font-style:italic;">{description}</p>
    """

    # Add prior probabilities section
    if priors and 'true_prob' in priors:
        true_prob = priors['true_prob']
        false_prob = 1.0 - true_prob

        # Get proper state names
        true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
        false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

        html += f"""
        <div style="margin-top:10px; background-color:#fff; padding:8px;␣
↪border-radius:4px; border:1px solid #ddd;">
            <h4 style="margin-top:0; font-size:14px;">Prior Probabilities:</h4>
            <div style="display:flex; justify-content:space-between;␣
↪margin-bottom:4px;">
                <div style="font-size:12px;">{true_state}: {true_prob:.3f}</div>
```

```
                <div style="font-size:12px;">{false_state}: {false_prob:.3f}</
↪div>
            </div>
            {create_probability_bar(true_prob, false_prob, "20px", True)}
        </div>
        """

    # Add click instruction
    html += """
    <div style="margin-top:8px; font-size:12px; color:#666; text-align:center;">
        Click node to see full probability details
    </div>
    </div>
    """

    return html

def create_expanded_content(node_id, node_data):
    """
    Create expanded content shown when a node is clicked
    """
    description = node_data.get('description', '')
    priors = node_data.get('priors', {})
    posteriors = node_data.get('posteriors', {})
    instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])

    # Get proper state names
    true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
    false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

    # Extract probabilities
    true_prob = priors.get('true_prob', 0.5)
    false_prob = 1.0 - true_prob

    # Start building the expanded content
    html = f"""
    <div style="max-width:500px; padding:15px; font-family:Arial, sans-serif;">
        <h2 style="margin-top:0; color:#333;">{node_id}</h2>
        <p style="font-style:italic; margin-bottom:15px;">{description}</p>

        <div style="margin-bottom:20px; padding:12px; border:1px solid #ddd;␣
↪background-color:#f9f9f9; border-radius:5px;">
            <h3 style="margin-top:0; color:#333;">Prior Probabilities</h3>
            <div style="display:flex; justify-content:space-between;␣
↪margin-bottom:5px;">
                <div><strong>{true_state}:</strong> {true_prob:.3f}</div>
                <div><strong>{false_state}:</strong> {false_prob:.3f}</div>
```

```
            </div>
            {create_probability_bar(true_prob, false_prob, "25px", True)}
        </div>
    """

    # Add conditional probability table if available
    if posteriors:
        html += """
        <div style="padding:12px; border:1px solid #ddd; background-color:
↪#f9f9f9; border-radius:5px;">
            <h3 style="margin-top:0; color:#333;">Conditional Probabilities</h3>
            <table style="width:100%; border-collapse:collapse; font-size:13px;
↪">
                <tr style="background-color:#eee;">
                    <th style="padding:8px; text-align:left; border:1px solid␣
↪#ddd;">Condition</th>
                    <th style="padding:8px; text-align:center; border:1px solid␣
↪#ddd; width:80px;">Value</th>
                    <th style="padding:8px; text-align:center; border:1px solid␣
↪#ddd;">Visualization</th>
                </tr>
        """

        # Sort posteriors to group by similar conditions
        posterior_items = list(posteriors.items())
        posterior_items.sort(key=lambda x: x[0])

        # Add rows for conditional probabilities
        for key, value in posterior_items:
            try:
                # Try to parse probability value
                prob_value = float(value)
                inv_prob = 1.0 - prob_value

                # Add row with probability visualization
                html += f"""
                <tr>
                    <td style="padding:8px; border:1px solid #ddd;">{key}</td>
                    <td style="padding:8px; text-align:center; border:1px solid␣
↪#ddd;">{prob_value:.3f}</td>
                    <td style="padding:8px; border:1px solid #ddd;">
                        {create_probability_bar(prob_value, inv_prob, "20px",␣
↪False)}
                    </td>
                </tr>
                """
```

```
            except:
                # Fallback for non-numeric values
                html += f"""
                <tr>
                    <td style="padding:8px; border:1px solid #ddd;">{key}</td>
                    <td style="padding:8px; text-align:center; border:1px solid␣
↪#ddd;" colspan="2">{value}</td>
                </tr>
                """

        html += """
            </table>
        </div>
        """

    html += "</div>"

    return html
```

\# \\\# Phase 4: Main Visualization Function

[ ]:
```
def create_bayesian_network_with_probabilities(df):
    """
    Create an interactive Bayesian network visualization with enhanced␣
↪probability visualization
    and node classification based on network structure.
    """
    # Create a directed graph
    G = nx.DiGraph()

    # Add nodes with proper attributes
    for idx, row in df.iterrows():
        title = row['Title']
        description = row['Description']

        # Process probability information
        priors = get_priors(row)
        instantiations = get_instantiations(row)

        # Add node with base information
        G.add_node(
            title,
            description=description,
            priors=priors,
            instantiations=instantiations,
            posteriors=get_posteriors(row)
        )
```

```
    # Add edges
    for idx, row in df.iterrows():
        child = row['Title']
        parents = get_parents(row)

        # Add edges from each parent to this child
        for parent in parents:
            if parent in G.nodes():
                G.add_edge(parent, child)

    # Classify nodes based on network structure
    classify_nodes(G)

    # Create network visualization
    net = Network(notebook=True, directed=True, cdn_resources="in_line",␣
↪height="600px", width="100%")

    # Configure physics for better layout
    net.force_atlas_2based(gravity=-50, spring_length=100, spring_strength=0.02)
    net.show_buttons(filter_=['physics'])

    # Add the graph to the network
    net.from_nx(G)

    # Enhance node appearance with probability information and classification
    for node in net.nodes:
        node_id = node['id']
        node_data = G.nodes[node_id]

        # Get node type and set border color
        node_type = node_data.get('node_type', 'unknown')
        border_color = get_border_color(node_type)

        # Get probability information
        priors = node_data.get('priors', {})
        true_prob = priors.get('true_prob', 0.5) if priors else 0.5

        # Get proper state names
        instantiations = node_data.get('instantiations', ["TRUE", "FALSE"])
        true_state = instantiations[0] if len(instantiations) > 0 else "TRUE"
        false_state = instantiations[1] if len(instantiations) > 1 else "FALSE"

        # Create background color based on probability
        background_color = get_probability_color(priors)

        # Create tooltip with probability information
```

```
        tooltip = create_tooltip(node_id, node_data)

        # Create a simpler node label with probability
        simple_label = f"{node_id}\np={true_prob:.2f}"

        # Store expanded content as a node attribute for use in click handler
        node_data['expanded_content'] = create_expanded_content(node_id,␣
↪node_data)

        # Set node attributes
        node['title'] = tooltip  # Tooltip HTML
        node['label'] = simple_label  # Simple text label
        node['shape'] = 'box'
        node['color'] = {
            'background': background_color,
            'border': border_color,
            'highlight': {
                'background': background_color,
                'border': border_color
            }
        }

    # Set up the click handler with proper data
    setup_data = {
        'nodes_data': {node_id: {
            'expanded_content': json.dumps(G.nodes[node_id].
↪get('expanded_content', '')),
            'description': G.nodes[node_id].get('description', ''),
            'priors': G.nodes[node_id].get('priors', {}),
            'posteriors': G.nodes[node_id].get('posteriors', {})
        } for node_id in G.nodes()}
    }

    # Add custom click handling JavaScript
    click_js = """
    // Store node data for click handling
    var nodesData = %s;

    // Add event listener for node clicks
    network.on("click", function(params) {
        if (params.nodes.length > 0) {
            var nodeId = params.nodes[0];
            var nodeInfo = nodesData[nodeId];

            if (nodeInfo) {
                // Create a modal popup for expanded content
                var modal = document.createElement('div');
```

```
                modal.style.position = 'fixed';
                modal.style.left = '50%%';
                modal.style.top = '50%%';
                modal.style.transform = 'translate(-50%%, -50%%)';
                modal.style.backgroundColor = 'white';
                modal.style.padding = '20px';
                modal.style.borderRadius = '5px';
                modal.style.boxShadow = '0 0 10px rgba(0,0,0,0.5)';
                modal.style.zIndex = '1000';
                modal.style.maxWidth = '80%%';
                modal.style.maxHeight = '80%%';
                modal.style.overflow = 'auto';

                // Parse the JSON string back to HTML content
                try {
                    var expandedContent = JSON.parse(nodeInfo.expanded_content);
                    modal.innerHTML = expandedContent;
                } catch (e) {
                    modal.innerHTML = 'Error displaying content: ' + e.message;
                }

                // Add close button
                var closeBtn = document.createElement('button');
                closeBtn.innerHTML = 'Close';
                closeBtn.style.marginTop = '10px';
                closeBtn.style.padding = '5px 10px';
                closeBtn.style.cursor = 'pointer';
                closeBtn.onclick = function() {
                    document.body.removeChild(modal);
                };
                modal.appendChild(closeBtn);

                // Add modal to body
                document.body.appendChild(modal);
            }
        }
    });
    """ % json.dumps(setup_data['nodes_data'])

    # Save the graph to HTML
    html_file = "bayesian_network.html"
    net.save_graph(html_file)

    # Inject custom click handling into HTML
    try:
        with open(html_file, "r") as f:
            html_content = f.read()
```

```
        # Insert click handling script before the closing body tag
        html_content = html_content.replace('</body>', f'<script>{click_js}</
↪script></body>')

        # Write back the modified HTML
        with open(html_file, "w") as f:
            f.write(html_content)

        return HTML(html_content)
    except Exception as e:
        return HTML(f"<p>Error rendering HTML: {str(e)}</p><p>The network␣
↪visualization has been saved to '{html_file}'</p>")
```

\# Quickly check HTML Outputs

[ ]:
```
create_bayesian_network_with_probabilities(result_df)
```

[ ]:
```
# Use the function to create and display the visualization

print(result_df)
```

\# 5.0 Archive_version_histories

[ ]:
```
import pandas as pd
import json
from IPython.display import Markdown, display


def generate_bayesdown_questions_md(argdown_with_questions_path,␣
↪output_md_path, QuestionsMinimal=False):
    """
    Generate comprehensive BayesDown questions based on the enhanced CSV file.

    Args:
        argdown_with_questions_path (str): Path to the CSV file with␣
↪probability questions
        output_md_path (str): Path to save the output markdown file
        QuestionsMinimal (bool, optional): If True, only return the questions␣
↪generated for each node,
                                           excluding terminology explanations.␣
↪Defaults to False.
    """
    print(f"Loading enhanced CSV from {argdown_with_questions_path}...")

    # Load the enhanced CSV file
    try:
```

```
        df = pd.read_csv(argdown_with_questions_path)
        print(f"Successfully loaded CSV with {len(df)} rows.")
    except Exception as e:
        raise Exception(f"Error loading CSV: {e}")

    # Validate required columns
    required_columns = ['Title', 'Generate_Positive_Instantiation_Questions',␣
↪'Generate_Negative_Instantiation_Questions']
    missing_columns = [col for col in required_columns if col not in df.columns]
    if missing_columns:
        raise Exception(f"Missing required columns: {', '.
↪join(missing_columns)}")

    print("Generating comprehensive BayesDown questions...")

    # Start building the markdown content
    md_content = ""  # Initialize as empty string

    if not QuestionsMinimal:
        md_content += "# BayesDown Probability Questions\n\n"
        md_content += "This document contains questions for extracting␣
↪probability estimates for BayesDown models.\n\n"

        # Add comprehensive terminology explanation
        md_content += "## Probability Terminology\n\n"


        md_content += "### Types of Probabilities\n\n"
        md_content += "- **Prior Probability**: The unconditional probability␣
↪of a variable having a specific value before considering any evidence or␣
↪parent variable states. For example, P(X=TRUE) represents the probability␣
↪that X is TRUE without any additional information.\n\n"
        md_content += "- **Conditional Probability**: The probability of a␣
↪variable having a specific value given the values of its parent variables.␣
↪For example, P(X=TRUE|Y=TRUE, Z=FALSE) represents the probability that X is␣
↪TRUE when we know that Y is TRUE and Z is FALSE.\n\n"
        md_content += "- **Posterior Probability**: The updated probability of␣
↪a hypothesis after considering new evidence, calculated using Bayes' theorem.
↪ This represents a revised belief based on additional information.\n\n"
        md_content += "- **Joint Probability**: The probability of multiple␣
↪events occurring together. For example, P(X=TRUE, Y=FALSE) represents the␣
↪probability that X is TRUE and Y is FALSE simultaneously.\n\n"
        md_content += "- **Marginal Probability**: The probability of an event␣
↪across all possible states of another variable. It can be calculated by␣
↪summing the joint probability over all possible values of the other␣
↪variables.\n\n"
```

```
        md_content += "### Source of Probability Estimates\n\n"
        md_content += "For each probability estimate, please identify the␣
↪source using one of the following categories:\n\n"
        md_content += "- **Direct Statement**: The probability is explicitly␣
↪stated in the text.\n"
        md_content += "- **Derived Estimate**: The probability is calculated or␣
↪inferred from other probabilities mentioned in the text.\n"
        md_content += "- **Context-Based Estimate**: The probability is␣
↪inferred from the general context, tone, or strength of assertions in the␣
↪text.\n"
        md_content += "- **Expert Judgment**: The probability is based on␣
↪domain expertise, not directly stated in the text.\n"
        md_content += "- **Default Assignment**: The probability is assigned a␣
↪reasonable default value due to lack of information.\n\n"

        md_content += "### Certainty of Estimates\n\n"
        md_content += "For each probability estimate, please assess your␣
↪certainty using one of the following approaches:\n\n"
        md_content += "- **Confidence Interval**: Provide a range that likely␣
↪contains the true probability (e.g., \"80% confidence interval: 0.3-0.5\").
↪\n"
        md_content += "- **Confidence Level**: Rate your confidence in the␣
↪estimate on a scale (e.g., \"High confidence: 85%\").\n"
        md_content += "- **Error Margin**: Specify how much the estimate might␣
↪vary (e.g., \"0.7 ± 0.1\").\n"
        md_content += "- **Qualitative Assessment**: Describe your certainty␣
↪qualitatively (e.g., \"Very certain\", \"Moderately certain\", \"Highly␣
↪uncertain\").\n\n"


    # Generate questions for each node
    for idx, row in df.iterrows():
        title = row['Title']
        description = row['Description'] if 'Description' in df.columns and not␣
↪pd.isna(row['Description']) else ""

        md_content += f"## {title}\n\n"  # Still include title even in minimal␣
↪mode

        if description:
            md_content += f"{description}\n\n"

        # Process positive instantiation questions
        try:
```

```
            positive_questions = json.
↪loads(row['Generate_Positive_Instantiation_Questions'])

            md_content += "### Positive Instantiation Questions\n\n"

            for q_type, question in positive_questions.items():
                md_content += f"1. **{question}**\n"

                # Add source question with appropriate terminology based on␣
↪question type
                if q_type == 'prior':
                    md_content += f"   - **Source**: What is the source for␣
↪this prior probability estimate? (Direct statement, derived estimate,␣
↪context-based, expert judgment, default)\n"
                else:
                    md_content += f"   - **Source**: What is the source for␣
↪this conditional probability estimate? (Direct statement, derived estimate,␣
↪context-based, expert judgment, default)\n"

                # Add certainty question
                md_content += f"   - **Certainty**: How certain are you about␣
↪this probability estimate? (Provide a confidence interval, confidence level,␣
↪error margin, or qualitative assessment)\n\n"
        except Exception as e:
            md_content += f"No positive instantiation questions available.␣
↪Error: {e}\n\n"

        # Process negative instantiation questions
        try:
            negative_questions = json.
↪loads(row['Generate_Negative_Instantiation_Questions'])

            md_content += "### Negative Instantiation Questions\n\n"

            for q_type, question in negative_questions.items():
                md_content += f"1. **{question}**\n"

                # Add source question with appropriate terminology based on␣
↪question type
                if q_type == 'prior':
                    md_content += f"   - **Source**: What is the source for␣
↪this prior probability estimate? (Direct statement, derived estimate,␣
↪context-based, expert judgment, default)\n"
                else:
```

```
                    md_content += f"   - **Source**: What is the source for␣
 ↪this conditional probability estimate? (Direct statement, derived estimate,␣
 ↪context-based, expert judgment, default)\n"

                # Add certainty question
                md_content += f"   - **Certainty**: How certain are you about␣
 ↪this probability estimate? (Provide a confidence interval, confidence level,␣
 ↪error margin, or qualitative assessment)\n\n"
        except Exception as e:
            md_content += f"No negative instantiation questions available.␣
 ↪Error: {e}\n\n"

    # Save the markdown content
    with open(output_md_path, 'w') as f:
        f.write(md_content)

    print(f"BayesDown questions saved to {output_md_path}")
    return md_content

# Example usage:
md_content = generate_bayesdown_questions_md("ArgDown_WithQuestions.csv",␣
 ↪"BayesDownQuestions.md", QuestionsMinimal=True)
# To get only the questions, set QuestionsMinimal=True

print(md_content)  # Print the returned content to see the questions
```

[ ]:
```
# @title 0.0.0 --- Install & Import Libraries & Packages (One-Time Colab Setup)␣
 ↪---

# implement boolean flags to indicate parts of the code that has to be skipped
# best practice in Colab?

# !pip install ...    vs.  %pip install ...   vs.  from  ...  import ...  as ...
 ↪    vs.   !apt-get -qq install -y
# "!" preceding a code block line in tells Colab/Jupyter to execute as command␣
 ↪line

# Check if the setup flag variable exists in the global scope
if 'setup_complete' not in globals():
    print("Performing one-time setup...")
    # --- Your one-time code goes here ---
    # Example: Install packages, download small files, initialize complex␣
 ↪objects
    # !pip install -q some_package
    # import some_package
    # data = download_small_dataset()
```

```
    my_initialized_object = "This is initialized"
    # --- End of one-time code ---

    # Set the flag to indicate setup is done for this session
    setup_complete = True
    print("One-time setup finished.")
else:
    print("Setup already completed in this session. Skipping.")

# You can now safely use variables/objects created during setup
# print(my_initialized_object)

try:
    # If this variable exists, the block was already run successfully.
    _setup_marker
    print("Setup already completed in this session. Skipping.")
except NameError:
    print("Performing one-time setup...")
    # --- Your one-time code goes here ---
    # !pip install -q another_package
    # configuration = load_config()
    # --- End of one-time code ---

    # Create the marker variable ONLY after successful execution
    _setup_marker = True
    print("One-time setup finished.")

# Use things created in setup
# print(configuration)


library_name = "your_library_name"
try:
    __import__(library_name)
    print(f"The library '{library_name}' is available in Colab.")
except ImportError:
    print(f"The library '{library_name}' is not available in Colab.")
```

1. Import Libraries \& Install Packages: Run Section 0.1
2. Connect to GitHub Repository \& Load Data files: Run Section 0.2
3. ...
4. Link Text Requires:
5. Test

\# \\\#\\\#\\\#\\\# Heading This is the cell I'm linking to

[ ]:
```
# prompt: how to title code blocks

import requests      # For making HTTP requests
import io           # For working with in-memory file-like objects
import pandas as pd   # For data manipulation
import numpy as np
import json
import re
import matplotlib.pyplot as plt
from IPython.display import HTML, display
from IPython.display import Markdown, display
import networkx as nx
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
from pyvis.network import Network
import os
import os.path
from IPython.display import IFrame
import itertools


# --- 2. Data Processing: ArgDown to BayesDown ---
# Load the data from the ArgDown_WithQuestions CSV file
argdown_with_questions_df = pd.read_csv('ArgDown_WithQuestions.csv')

# Display the DataFrame
print(argdown_with_questions_df)
argdown_with_questions_df


# --- 2.2 ArgDown_WithQuestions.csv to BayesDownQuestions.md ---
def extract_bayesdown_questions_fixed(argdown_with_questions_path,␣
 ↪output_md_path, include_questions_as_comments=True):
    # ... (function code as before) ...


# --- 2.3 Generate BayesDown Probability Extraction Prompt ---
# ... (code for generating the prompt) ...


# --- 2.4 Prepare 2nd API call ---
# ... (code for preparing the API call) ...


# --- 2.5 Make BayesDown Probability Extraction API Call ---
# ... (code for making the API call) ...
```

```
# --- 2.6 Save BayesDown with Probability Estimates (.csv) ---
# ... (code for saving the data) ...


# --- 2.7 Review & Verify BayesDown Probability Estimates ---
# ... (code for review and verification) ...


# --- 2.8 Extract BayesDown with Probability Estimates as Dataframe ---
# ... (code for extraction) ...


# --- 3. Data Extraction: BayesDown (.md) to Database (.csv) ---
# --- 3.1 ExtractBayesDown-Data_v1 ---
# ... (code for BayesDown data extraction) ...


# --- 3.1.2 Test BayesDown Extraction ---
# ... (code for testing BayesDown extraction) ...


# --- 3.3 Extraction ---
# ... (code for extraction) ...


# --- 3.3 Data-Post-Processing ---
# ... (code for data post-processing) ...
```

\# \\\# COMBINED: 2.1 Generate and Extract "Prior-, Conditional- and Posterior Probability Questions" from 'ArgDown.csv' to 'ArgDown_WithQuestions.csv'

[49]:
```
# Main function to fix the ArgDown → BayesDown generation pipeline
def fix_bayesdown_generation():
    # Step 1: Fix the relationship establishment function
    def establish_relationships_fixed(titles_info, text):
        """
        Establish parent-child relationships between titles using BayesDown␣
 ↪indentation rules.

        In BayesDown syntax:
        - More indented nodes (with + symbol) are PARENTS of less indented nodes
        - The relationship reads as "Effect is caused by Cause" (Effect + Cause)
        - This aligns with how Bayesian networks represent causality
        """
        lines = text.split('\n')
```

```
        # Dictionary to store line numbers for each title occurrence
        title_occurrences = {}

        # Record line number for each title (including multiple occurrences)
        line_number = 0
        for line in lines:
            if not line.strip():
                line_number += 1
                continue

            title_match = re.search(r'[<\[](.+?)[>\]]', line)
            if not title_match:
                line_number += 1
                continue

            title = title_match.group(1)

            # Store all occurrences of each title with their line numbers
            if title not in title_occurrences:
                title_occurrences[title] = []
            title_occurrences[title].append(line_number)

            # Store all line numbers where this title appears
            if 'line_numbers' not in titles_info[title]:
                titles_info[title]['line_numbers'] = []
            titles_info[title]['line_numbers'].append(line_number)

            # For backward compatibility, keep the first occurrence in 'line'
            if titles_info[title]['line'] is None:
                titles_info[title]['line'] = line_number

            line_number += 1

        # Create an ordered list of all title occurrences with their line␣
↪numbers
        all_occurrences = []
        for title, occurrences in title_occurrences.items():
            for line_num in occurrences:
                all_occurrences.append((title, line_num))

        # Sort occurrences by line number
        all_occurrences.sort(key=lambda x: x[1])

        # Get indentation for each occurrence
        occurrence_indents = {}
        for title, line_num in all_occurrences:
```

```
            for line in lines[line_num:line_num+1]:  # Only check the current line
                indent = 0
                if '+' in line:
                    symbol_index = line.find('+')
                    # Count spaces before the '+' symbol
                    j = symbol_index - 1
                    while j >= 0 and line[j] == ' ':
                        indent += 1
                        j -= 1
                elif '-' in line:
                    symbol_index = line.find('-')
                    # Count spaces before the '-' symbol
                    j = symbol_index - 1
                    while j >= 0 and line[j] == ' ':
                        indent += 1
                        j -= 1
                occurrence_indents[(title, line_num)] = indent

        # For each line, find the proper parent-child relationships
        # In BayesDown, a more indented node is a parent of the less indented node above it
        for i, (title, line_num) in enumerate(all_occurrences):
            current_indent = occurrence_indents[(title, line_num)]

            # Find the closest previous node with less indentation
            # This will be the child of the current node
            j = i - 1
            while j >= 0:
                prev_title, prev_line = all_occurrences[j]
                prev_indent = occurrence_indents[(prev_title, prev_line)]

                # If we found a node with less indentation, it's a child of current node
                if prev_indent < current_indent:
                    # This is the key relationship: more indented node (current) is parent of less indented node (previous)
                    if title not in titles_info[prev_title]['parents']:
                        titles_info[prev_title]['parents'].append(title)
                    if prev_title not in titles_info[title]['children']:
                        titles_info[title]['children'].append(prev_title)
                    break  # Only need the immediate child

                j -= 1

        return titles_info
```

```
    # Step 2: Updated main parsing function
    def parse_markdown_hierarchy_fixed(markdown_text, ArgDown=False):
        """Main function to parse markdown hierarchy into a DataFrame with␣
↪correct parent-child relationships"""

        # Remove comments
        clean_text = remove_comments(markdown_text)

        # Extract all titles with their descriptions and indentation levels
        titles_info = extract_titles_info(clean_text)

        # Establish parent-child relationships - Use fixed function here
        titles_with_relations = establish_relationships_fixed(titles_info,␣
↪clean_text)

        # Convert to DataFrame
        df = convert_to_dataframe(titles_with_relations, ArgDown)

        # Add No_Parent and No_Children columns
        df = add_no_parent_no_child_columns_to_df(df)

        # Add Parents instantiation columns
        df = add_parents_instantiation_columns_to_df(df)

        return df

    # Helper function to safely parse lists
    def parse_list_safely(list_data):
        if isinstance(list_data, list):
            return list_data

        if isinstance(list_data, str):
            try:
                # Try to parse as JSON
                parsed = json.loads(list_data.replace("'", "\""))
                if isinstance(parsed, list):
                    return parsed
            except:
                # Try to parse as string list
                if list_data.startswith('[') and list_data.endswith(']'):
                    items = list_data.strip('[]').split(',')
                    return [item.strip(' "\'') for item in items if item.
↪strip()]
                elif list_data.strip():
                    # Handle single item
                    return [list_data.strip()]
```

```
        # Default case
        return []

    # Step 3: Updated BayesDown generation function
    def generate_bayesdown_format_md_fixed(argdown_with_questions_path,␣
↪output_md_path, QuestionsMinimal=False):
        """
        Generate BayesDown format file with correct parent-child relationships.
        """
        print(f"Loading enhanced CSV from {argdown_with_questions_path}...")

        # Load the enhanced CSV file
        try:
            df = pd.read_csv(argdown_with_questions_path)
            print(f"Successfully loaded CSV with {len(df)} rows.")
        except Exception as e:
            raise Exception(f"Error loading CSV: {e}")

        # Validate required columns
        required_columns = ['Title', 'Description', 'Parents', 'instantiations']
        missing_columns = [col for col in required_columns if col not in df.
↪columns]
        if missing_columns:
            raise Exception(f"Missing required columns: {', '.
↪join(missing_columns)}")

        print(f"Generating BayesDown format file...")

        # Start building the markdown content
        md_content = ""  # Initialize as empty string

        # Add explanations if QuestionsMinimal is False
        if not QuestionsMinimal:
            md_content += "# BayesDown Format\n\n"
            md_content += "This document contains the BayesDown representation␣
↪for the Bayesian network.\n\n"
            md_content += "## Format Description\n\n"
            md_content += "BayesDown is a format that extends ArgDown with␣
↪probabilistic information. It uses:\n\n"
            md_content += "- **Node definitions**: `[Node_Name]: Description␣
↪{\"metadata\": ...}`\n"
            md_content += "- **Hierarchical relationships**: Parent nodes␣
↪(causes) are indented and prefixed with `+` below their effects\n"
            md_content += "- **Metadata**: JSON structure containing␣
↪instantiations, priors, and posteriors\n\n"
            md_content += "## Network Structure\n\n"
```

```
        # Create a dictionary for easy lookup of node information
        nodes_dict = {}
        for _, row in df.iterrows():
            # Parse instantiations
            instantiations = parse_list_safely(row['instantiations'])

            # Parse parents
            parents = parse_list_safely(row['Parents'])

            # Create node entry
            nodes_dict[row['Title']] = {
                'description': row['Description'] if not pd.
↪isna(row['Description']) else "",
                'instantiations': instantiations if instantiations else␣
↪["TRUE", "FALSE"],
                'parents': parents,
                'children': []  # Will be filled in based on parent␣
↪relationships
            }

        # Set up children based on parent relationships
        for node_name, node_info in nodes_dict.items():
            for parent in node_info['parents']:
                if parent in nodes_dict:
                    if node_name not in nodes_dict[parent]['children']:
                        nodes_dict[parent]['children'].append(node_name)

        # Identify leaf nodes (effects without causes)
        leaf_nodes = []
        for node_name, node_info in nodes_dict.items():
            if not node_info['children']:
                leaf_nodes.append(node_name)

        # If no leaf nodes found, use nodes without parents
        if not leaf_nodes:
            leaf_nodes = [node for node, info in nodes_dict.items() if not␣
↪info['parents']]

        # If still no nodes found, use the first node as a starting point
        if not leaf_nodes and nodes_dict:
            leaf_nodes = [next(iter(nodes_dict))]

        # Function to generate BayesDown syntax for a node and its parents
        def generate_node_syntax(node_name, indent_level=0,␣
↪processed_nodes=None):
            if processed_nodes is None:
```

```
            processed_nodes = set()

        if node_name not in nodes_dict:
            return ""

        # If we've already fully processed this node, just add a reference
        if node_name in processed_nodes:
            indent = ' ' * indent_level
            return f"{indent}+ [{node_name}]\n"

        node_info = nodes_dict[node_name]
        indent = " " * indent_level
        prefix = f"{indent}+ " if indent_level > 0 else ""

        # Mark this node as processed
        processed_nodes.add(node_name)

        # Create metadata with instantiations
        metadata = {
            "instantiations": node_info['instantiations'],
            "priors": {},
            "posteriors": {}
        }

        # Add placeholder priors based on instantiations
        for instantiation in node_info['instantiations']:
            metadata["priors"][f"p({instantiation})"] = "0.6"  # Default␣
↪placeholder

        # Add placeholder posteriors if node has parents
        if node_info['parents']:
            posteriors = {}
            for parent in node_info['parents']:
                if parent in nodes_dict:
                    for parent_inst in nodes_dict[parent]['instantiations']:
                        for inst in node_info['instantiations']:
                            # Create placeholder conditional probabilities
                            posteriors[f"p({inst}|{parent}={parent_inst})"]␣
↪= "0.59"
            metadata["posteriors"] = posteriors

        # Format the node definition with metadata
        metadata_json = json.dumps(metadata, indent=None).replace('\n', ' ')
        node_syntax = f"{prefix}[{node_name}]: {node_info['description']}␣
↪{metadata_json}\n"

        # Add parents with proper indentation
```

```
            for parent in node_info['parents']:
                if parent in nodes_dict and parent != node_name:  # Avoid␣
↪self-references
                    parent_syntax = generate_node_syntax(parent, indent_level +␣
↪2, processed_nodes)
                    node_syntax += parent_syntax

            return node_syntax

        # Generate BayesDown syntax for each leaf node (effect)
        for leaf in leaf_nodes:
            md_content += generate_node_syntax(leaf) + "\n"

        # Save the markdown content
        with open(output_md_path, 'w') as f:
            f.write(md_content)

        print(f"BayesDown format file saved to {output_md_path}")
        return md_content

    # Return the fixed functions
    return {
        'establish_relationships': establish_relationships_fixed,
        'parse_markdown_hierarchy': parse_markdown_hierarchy_fixed,
        'generate_bayesdown_format_md': generate_bayesdown_format_md_fixed
    }

# Usage:
fixed_functions = fix_bayesdown_generation()

# Replace the original functions with fixed versions
establish_relationships = fixed_functions['establish_relationships']
parse_markdown_hierarchy = fixed_functions['parse_markdown_hierarchy']
generate_bayesdown_format_md = fixed_functions['generate_bayesdown_format_md']

# Now use the updated function to process ArgDown content
result_df = parse_markdown_hierarchy(md_content)

# Generate BayesDown format
bayesdown_format = generate_bayesdown_format_md(
    "ArgDown_WithQuestions.csv",
    "BayesDownFormat.md",
    QuestionsMinimal=True
)
```

[ ]:
```
# notebook_name = "NoHTML_AMTAIR_Prototype"
```

```
# repo_url = "https://raw.githubusercontent.com/SingularitySmith/
 ↪AMTAIR_Prototype/main/data/example_1/"


# !wget {repo_url}{notebook_name}.ipynb
# !jupyter nbconvert --to markdown {notebook_name}.ipynb --output␣
 ↪{notebook_name}.md --no-input
```

[ ]:
```
# Convert ipynb to HTML in Colab
# Upload ipynb
# from google.colab import files
# f = files.upload()

# Convert ipynb to html
# import subprocess
# file0 = list(f.keys())[0]
# _ = subprocess.run(["pip", "install", "nbconvert"])
# _ = subprocess.run(["jupyter", "nbconvert", file0, "--to", "html"])

# download the html
# files.download(file0[:-5]+"html")
```

[120]:
```
import pandas as pd
import json
from IPython.display import Markdown, display

# Helper function to parse lists safely
def parse_list_safely(list_data):
    if isinstance(list_data, list):
        return list_data

    if isinstance(list_data, str):
        try:
            # Try to parse as JSON
            parsed = json.loads(list_data)
            if isinstance(parsed, list):
                return parsed
        except:
            # Try to parse as string list
            if list_data.startswith('[') and list_data.endswith(']'):
                items = list_data.strip('[]').split(',')
                return [item.strip(' "\'') for item in items if item.strip()]
            elif list_data.strip():
                # Handle single item
                return [list_data.strip()]

    # Default case
```

```
    return []

def generate_bayesdown_format_md_fixed(argdown_with_questions_path,␣
 ↪output_md_path, QuestionsMinimal=False):
    """
    Generate BayesDown format file based on the enhanced CSV file,
    with correct parent-child relationships.

    Args:
        argdown_with_questions_path (str): Path to the CSV file with␣
 ↪probability questions
        output_md_path (str): Path to save the output markdown file
        QuestionsMinimal (bool, optional): If True, only generate the BayesDown␣
 ↪format without explanations.
                                           Defaults to False.
    """
    print(f"Loading enhanced CSV from {argdown_with_questions_path}...")

    # Load the enhanced CSV file
    try:
        df = pd.read_csv(argdown_with_questions_path)
        print(f"Successfully loaded CSV with {len(df)} rows.")
    except Exception as e:
        raise Exception(f"Error loading CSV: {e}")

    # Validate required columns
    required_columns = ['Title', 'Description', 'Parents', 'instantiations']
    missing_columns = [col for col in required_columns if col not in df.columns]
    if missing_columns:
        raise Exception(f"Missing required columns: {', '.
 ↪join(missing_columns)}")

    print(f"Generating BayesDown format file...")

    # Start building the markdown content
    md_content = ""  # Initialize as empty string

    # Add explanations if QuestionsMinimal is False
    if not QuestionsMinimal:
        md_content += "# BayesDown Format\n\n"
        md_content += "This document contains the BayesDown representation for␣
 ↪the Bayesian network.\n\n"
        md_content += "## Format Description\n\n"
        md_content += "BayesDown is a format that extends ArgDown with␣
 ↪probabilistic information. It uses:\n\n"
```

```
        md_content += "- **Node definitions**: `[Node_Name]: Description␣
↪{\"metadata\": ...}`\n"
        md_content += "- **Hierarchical relationships**: Parent nodes are␣
↪indented and prefixed with `+`\n"
        md_content += "- **Metadata**: JSON structure containing␣
↪instantiations, priors, and posteriors\n\n"
        md_content += "## Network Structure\n\n"

    # Create a dictionary for easy lookup of node information
    nodes_dict = {}
    for _, row in df.iterrows():
        # Parse instantiations
        instantiations = parse_list_safely(row['instantiations'])

        # Parse parents
        parents = parse_list_safely(row['Parents'])

        # Create node entry
        nodes_dict[row['Title']] = {
            'description': row['Description'],
            'instantiations': instantiations,
            'parents': parents,
            'children': [],  # Will be filled in based on parent relationships
            'questions_positive': row.
↪get('Generate_Positive_Instantiation_Questions', '{}'),
            'questions_negative': row.
↪get('Generate_Negative_Instantiation_Questions', '{}')
        }

    # Set up children based on parent relationships
    for node_name, node_info in nodes_dict.items():
        for parent in node_info['parents']:
            if parent in nodes_dict:
                nodes_dict[parent]['children'].append(node_name)

    # Identify root nodes (effects that aren't causes for anything else)
    root_nodes = []
    for node_name, node_info in nodes_dict.items():
        if not node_info['children']:
            root_nodes.append(node_name)

    # If no root nodes found, use the first node as root
    if not root_nodes and nodes_dict:
        root_nodes = [next(iter(nodes_dict))]

    # Function to recursively generate BayesDown syntax
    def generate_node_syntax(node_name, indent_level=0, processed_nodes=None):
```

```
        if processed_nodes is None:
            processed_nodes = set()

        if node_name not in nodes_dict:
            return ""

        # If we've already fully processed this node, just add a reference
        if node_name in processed_nodes:
            return f"{' ' * indent_level}+ [{node_name}]\n"

        processed_nodes.add(node_name)

        node_info = nodes_dict[node_name]
        indent = " " * indent_level

        # Create metadata with instantiations
        metadata = {
            "instantiations": node_info['instantiations'],
            "priors": {},
            "posteriors": {}
        }

        # Add placeholder priors based on instantiations
        for instantiation in node_info['instantiations']:
            metadata["priors"][f"p({instantiation})"] = "? %"  # Default␣
↪placeholder

        # Add placeholder posteriors if node has parents
        if node_info['parents']:
            for parent in node_info['parents']:
                if parent in nodes_dict:
                    for parent_inst in nodes_dict[parent]['instantiations']:
                        for inst in node_info['instantiations']:
                            # Create placeholder conditional probabilities
                            ␣
↪metadata["posteriors"][f"p({inst}|{parent}={parent_inst})"] = "?? %"

        # Format the node definition with metadata
        metadata_json = json.dumps(metadata, indent=None).replace('\n', ' ')
        node_syntax = f"{indent}[{node_name}]: {node_info['description']}␣
↪{metadata_json}\n"

        # Add parents with proper indentation
        for parent in node_info['parents']:
            if parent in nodes_dict and parent != node_name:  # Avoid␣
↪self-references
```

```
                parent_syntax = generate_node_syntax(parent, indent_level + 2,␣
↪processed_nodes)
                node_syntax += parent_syntax

        return node_syntax


    # Generate BayesDown syntax for each root node
    for root in root_nodes:
        md_content += generate_node_syntax(root) + "\n"

    # Save the markdown content
    with open(output_md_path, 'w') as f:
        f.write(md_content)

    print(f"BayesDown format file saved to {output_md_path}")
    return md_content



# Example of how to use the new functionality:

# Generate BayesDown format
bayesdown_format_fixed = generate_bayesdown_format_md_fixed(
    "ArgDown_WithQuestions.csv",
    "BayesDownFormat.md",
    QuestionsMinimal=True
)

# Display a preview of the format
print("\nBayesDown Format Preview:")
print(bayesdown_format_fixed[:5000] + "...\n")
```

\# 6.0 Save Outputs

\# \\\# Convert ipynb to HTML in Colab

Instruction:

Download the ipynb, which you want to convert, on your local computer. Run the code below to upload the ipynb.

The html version will be downloaded automatically on your local machine. Enjoy it!

[11]:
```
#@title Convert ipynb to HTML in Colab
import nbformat
from nbconvert import HTMLExporter
import os
```

```
repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
 ↪main/data/example_1/"
notebook_name = "AMTAIR_Prototype_example1"  #Change Notebook name and path␣
 ↪when working on different examples

# Download the notebook file
!wget {repo_url}{notebook_name}.ipynb -O {notebook_name}.ipynb  # Corrected line

# Load the notebook
# add error handling for file not found
try:
  with open(f"{notebook_name}.ipynb") as f:
    nb = nbformat.read(f, as_version=4)
except FileNotFoundError:
  print(f"Error: File '{notebook_name}.ipynb' not found. Please check if it was␣
 ↪downloaded correctly.")

# Initialize the HTML exporter
exporter = HTMLExporter()

# Convert the notebook to HTML
(body, resources) = exporter.from_notebook_node(nb)

# Save the HTML to a file
with open(f"{notebook_name}IPYNB.html", "w") as f:
    f.write(body)
```

\# \\\# Convert .ipynb Notebook to MarkDown

[10]:
```
# @title --- Convert .ipynb Notebook to MarkDown ---

import nbformat
from nbconvert import MarkdownExporter
import os

repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
 ↪main/data/example_1/"
notebook_name = "AMTAIR_Prototype_example1"  #Change Notebook name and path␣
 ↪when working on different examples

# Download the notebook file
!wget {repo_url}{notebook_name}.ipynb -O {notebook_name}.ipynb  # Corrected line

# Load the notebook
# add error handling for file not found
try:
  with open(f"{notebook_name}.ipynb") as f:
```

```
    nb = nbformat.read(f, as_version=4)
except FileNotFoundError:
  print(f"Error: File '{notebook_name}.ipynb' not found. Please check if it was␣
 ↪downloaded correctly.")


# Initialize the Markdown exporter
exporter = MarkdownExporter(exclude_output=True)  # Correct initialization

# Convert the notebook to Markdown
(body, resources) = exporter.from_notebook_node(nb)

# Save the Markdown to a file
with open(f"{notebook_name}IPYNB.md", "w") as f:
    f.write(body)
```

[26]:
```
import nbformat
from nbconvert import PDFExporter
import os
import subprocess
import re

def escape_latex_special_chars(text):
  """Escapes special LaTeX characters in a string."""
  latex_special_chars = ['&', '%', '#', '_', '{', '}', '~', '^', '\\']
  replacement_patterns = [
      (char, '\\' + char) for char in latex_special_chars
  ]

  # Escape reserved characters
  for original, replacement in replacement_patterns:
    text = text.replace(original, replacement) # This is the fix
  return text

# Function to check if a command is available
def is_command_available(command):
    try:
        subprocess.run([command], capture_output=True, check=True)
        return True
    except (subprocess.CalledProcessError, FileNotFoundError):
        return False

# Check if xelatex is installed, and install if necessary
if not is_command_available("xelatex"):
    print("Installing necessary TeX packages...")
    !apt-get install -y texlive-xetex texlive-fonts-recommended␣
 ↪texlive-plain-generic
```

```
    print("TeX packages installed successfully.")
else:
    print("xelatex is already installed. Skipping installation.")

repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
 ↪main/data/example_1/"
notebook_name = "AMTAIR_Prototype_example1"  #Change Notebook name and path␣
 ↪when working on different examples

# Download the notebook file
!wget {repo_url}{notebook_name}.ipynb -O {notebook_name}.ipynb  # Corrected line

# Load the notebook
# add error handling for file not found
try:
  with open(f"{notebook_name}.ipynb") as f:
    nb = nbformat.read(f, as_version=4)
except FileNotFoundError:
  print(f"Error: File '{notebook_name}.ipynb' not found. Please check if it was␣
 ↪downloaded correctly.")


# Initialize the PDF exporter
exporter = PDFExporter(exclude_output=True)  # Changed to PDFExporter

# Sanitize notebook cell titles to escape special LaTeX characters like '&'
for cell in nb.cells:
    if 'cell_type' in cell and cell['cell_type'] == 'markdown':
        if 'source' in cell and isinstance(cell['source'], str):
            # Replace '&' with '\protect&' in markdown cell titles AND CONTENT
            # Updated to use escape_latex_special_chars function
            cell['source'] = escape_latex_special_chars(cell['source'])
            # Additionally, escape special characters in headings
            cell['source'] = re.sub(r'(#+)\s*(.*)', lambda m: m.group(1) + ' '␣
 ↪+ escape_latex_special_chars(m.group(2)), cell['source'])



# Convert the notebook to PDF
(body, resources) = exporter.from_notebook_node(nb)


# Save the PDF to a file
with open(f"{notebook_name}IPYNB.pdf", "wb") as f:  # Changed to 'wb' for␣
 ↪binary writing
    f.write(body)
```