Notebook

April 27, 2025

\# AMTAIR Prototype Demonstration (Public Colab Notebook)

\# AMTAIR Prototype: Automating Transformative AI Risk Modeling

\# \\\# Executive Summary

This notebook implements a prototype of the AMTAIR (Automating Transformative AI Risk Modeling) project, which addresses the critical coordination failure in AI governance by developing computational tools that automate the extraction of probabilistic world models from AI safety literature.

The prototype demonstrates the transformation pipeline from structured argument representations (ArgDown) to probabilistic Bayesian networks (BayesDown), enabling the visualization and analysis of causal relationships and probability distributions that underlie AI risk assessments and policy evaluations.

\# \\\#\\\# Purpose Within the Master's Thesis

This notebook serves as the technical implementation component of the Master's thesis "Automating Transformative AI Risk Modeling: A Computational Approach to Policy Impact Evaluation." It demonstrates the feasibility of automating the extraction and formalization of world models, focusing on the core extraction pipeline and visualization capabilities that form the foundation for more sophisticated analysis.

\# \\\#\\\# Relevance to AI Governance

The coordination crisis in AI governance stems from different stakeholders working with incompatible assumptions, terminologies, and priorities. By making implicit models explicit through automated extraction and formalization, this work helps bridge communication gaps between technical researchers, policy specialists, and other stakeholders, contributing to more effective coordination in addressing existential risks from advanced AI.

\# \\\# Notebook Structure and Workflow

This notebook implements a multi-stage pipeline for transforming argument structures into interactive Bayesian network visualizations:

1. **Environment Setup** (Sections 0.1-0.3): Establishes the technical environment with necessary libraries and data connections
2. **Argument Extraction** (Sections 1.0-1.8): Processes source documents into structured ArgDown representations
3. **Probability Integration** (Sections 2.0-2.8): Enhances ArgDown with probability information to create BayesDown

4. **Data Transformation** (Section 3.0): Converts BayesDown into structured DataFrame format
5. **Visualization and Analysis** (Section 4.0): Creates interactive Bayesian network visualizations
6. **Archiving and Export** (Sections 5.0-6.0): Provides utilities for saving and sharing results

Throughout this notebook, we use the classic rain-sprinkler-lawn example as a canonical test case, demonstrating how a simple causal scenario (rain and sprinkler use affecting wet grass) can be represented, processed, and visualized using our automated pipeline.

\# \\\# Project Context and Purpose

This notebook implements a prototype of the Automating Transformative AI Risk Modeling (AMTAIR) project, which addresses a critical coordination failure in AI governance by developing computational tools to automate the extraction of probabilistic world models from AI safety literature.

The coordination crisis in AI governance stems from different stakeholders (technical researchers, policy specialists, ethicists) operating with different terminologies, priorities, and implicit theories of change. This fragmentation systematically increases existential risk through safety gaps, resource misallocation, and capability-governance mismatches.

The AMTAIR project aims to bridge these divides by: 1. Making implicit models explicit through automated extraction and formalization 2. Enabling comparison across different worldviews 3. Providing a common language for discussing probabilistic relationships 4. Supporting policy evaluation across diverse scenarios

\# \\\# Notebook Overview and Pipeline

This notebook demonstrates the core extraction pipeline from structured argument representations (ArgDown) to probabilistic Bayesian networks (BayesDown), using the classic rain-sprinkler-lawn example as a canonical test case.

The pipeline consists of five main stages: 1. **Environment Setup**: Libraries, GitHub repository access, and data loading 2. **Argument Extraction**: Processing source documents into structured ArgDown format 3. **Probability Integration**: Enhancing ArgDown with probabilistic information to create BayesDown 4. **Data Transformation**: Converting BayesDown into structured DataFrame format 5. **Visualization \& Analysis**: Creating interactive Bayesian network visualizations

\# \\\# Connection to Master's Thesis

This notebook serves as the technical implementation component of the Master's thesis "Automating Transformative AI Risk Modeling: A Computational Approach to Policy Impact Evaluation" (see PY_Thesis_OutlineNDraft), demonstrating the feasibility of automating the process of extracting and formalizing world models from AI safety literature.

The thesis positions this work as a solution to the coordination crisis in AI governance, where the AMTAIR tools provide a crucial bridge between different stakeholder communities by creating formal representations that can be analyzed, compared, and used for policy evaluation.

For broader context on the project's motivation and placement within AI governance efforts, see PY_Post0.0 ("The Missing Piece: Why We Need a Grand Strategy for AI") and

PY_AMTAIRDescription, which explain how this technical work contributes to the development of a comprehensive AI safety grand strategy.

\# \\\# 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. Environment Setup and Data Access

This section establishes the technical foundation for the AMTAIR prototype by: 1. Installing and importing necessary libraries 2. Setting up access to the GitHub repository 3. Loading example data files

The environment setup is designed to be run once per session, with flags to prevent redundant installations and imports. This section forms the basis for the subsequent extraction and analysis steps in the pipeline.

The key goal is to create a reproducible environment where the Bayesian network extraction and visualization can be performed consistently, with appropriate error handling and resource management.

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

[1]:
```
# @title 0.1 --- Install & Import Libraries & Packages (One-Time Setup) ---

"""
BLOCK PURPOSE: Establishes the core technical environment for the AMTAIR␣
↪prototype.
Sets up all required libraries for Bayesian network processing, visualization,␣
↪and data manipulation.
Uses a flag-based approach to ensure setup only runs once per session,␣
↪enhancing efficiency.

The setup follows a three-stage process:
1. Install required packages not available in Colab by default
2. Import all necessary libraries with error handling
```

```
3. Set a global flag to prevent redundant execution

DEPENDENCIES: Requires internet connection for package installation
OUTPUTS: Global variable _setup_imports_done and loaded Python libraries
"""

#  Check if setup has already been completed in this session using environment␣
↪flag
try:
    # If this variable exists, setup was already done successfully
    _setup_imports_done
    print("  Libraries already installed and imported in this session. Skipping␣
↪setup.")

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

    # --- STAGE 1: Install required packages ---
    # Install visualization and network analysis libraries
    !pip install -q pyvis  # Network visualization library
    !apt-get install pandoc -y  # Document conversion utility

    # Install Google API and data processing packages
    !pip install -q --upgrade gspread pandas google-auth google-colab  # Data␣
↪manipulation and Google integration

    # Install Bayesian network and probabilistic modeling tools
    !pip install -q pgmpy  # Probabilistic graphical models library

    # Install notebook conversion tools
    !pip install -q nbconvert  # Often pre-installed, but ensures availability

    print("   --> Installations complete.")

    # --- STAGE 2: Import libraries with error handling ---
    try:
        # Network and HTTP libraries
        import requests      # For making HTTP requests to APIs and GitHub
        import io            # For handling in-memory file-like objects

        # Data processing libraries
        import pandas as pd  # For structured data manipulation
        import numpy as np   # For numerical operations
        import json          # For JSON parsing and serialization
        import re            # For regular expression pattern matching

        # Visualization libraries
```

```
        import matplotlib.pyplot as plt  # For creating plots and charts
        from IPython.display import HTML, display, Markdown  # For rich output␣
↪in notebook

        # --- Specialized libraries requiring installation ---
        # Network analysis library
        import networkx as nx  # For graph representation and analysis

        # Probabilistic modeling libraries
        from pgmpy.models import BayesianNetwork  # For Bayesian network␣
↪structure
        from pgmpy.factors.discrete import TabularCPD  # For conditional␣
↪probability tables
        from pgmpy.inference import VariableElimination  # For probabilistic␣
↪inference

        # Interactive network visualization
        from pyvis.network import Network  # For interactive network␣
↪visualization

        # Output version information for key libraries
        print(f"      pandas version: {pd.__version__}")
        print(f"      networkx version: {nx.__version__}")
        # Add others if specific versions are critical

        print("   --> Imports complete.")

        # --- STAGE 3: Set flag to indicate successful setup ---
        _setup_imports_done = True
        print("  One-time setup finished successfully.")

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

# Environment is now ready for AMTAIR processing
```

\# \\\# 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.

[9]:
```
# @title 0.2 --- Connect to GitHub Repository --- Load Files

"""
BLOCK PURPOSE: Establishes connection to the AMTAIR GitHub repository and␣
↪provides
functions to load example data files for processing.

This block creates a reusable function for accessing files from the project's
GitHub repository, enabling access to example files like the rain-sprinkler-lawn
Bayesian network that serves as our canonical test case.

DEPENDENCIES: requests library, io library
OUTPUTS: load_file_from_repo function and test file loads
"""

from requests.exceptions import HTTPError

# Specify the base repository URL for the AMTAIR project
repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
↪main/data/example_carlsmith/"
print(f"Connecting to repository: {repo_url}")

def load_file_from_repo(relative_path):
    """
    Loads a file from the specified GitHub repository using a relative path.

    Args:
        relative_path (str): Path to the file relative to the repo_url

    Returns:
        For CSV/JSON: pandas DataFrame
        For MD: string containing file contents

    Raises:
        HTTPError: If file not found or other HTTP error occurs
        ValueError: If unsupported file type is requested
    """
    file_url = repo_url + relative_path
    print(f"Attempting to load: {file_url}")

    # Fetch the file content from GitHub
    response = requests.get(file_url)

    # Check for bad status codes with enhanced 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

    # Convert response to file-like object
    file_object = io.StringIO(response.text)

    # Process different file types appropriately
    if relative_path.endswith(".csv"):
        return pd.read_csv(file_object)  # Return DataFrame for CSV
    elif relative_path.endswith(".json"):
        return pd.read_json(file_object)  # Return DataFrame for JSON
    elif relative_path.endswith(".md"):
        return file_object.read()  # Return raw content for MD files
    else:
        raise ValueError(f"Unsupported file type: {relative_path.split('.
↪')[-1]}. Add support in the GitHub Connection section of this notebook.")

# Load example files to test connection
try:
    # Load the extracted data CSV file
#     df = load_file_from_repo("extracted_data.csv")

    # Load the ArgDown test text
    md_content = load_file_from_repo("ArgDown.md")

    print("  Successfully connected to repository and loaded test files.")
except Exception as e:
    print(f"  Error loading files: {str(e)}")
    print("Please check your internet connection and the repository URL.")

# Display preview of loaded content (commented out to avoid cluttering output)
print(md_content)
```

\# \\\# 0.3 File Import

[8]:
```
# @title
md_content
```

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

\# 1. Sources to ArgDown: Structured Argument Extraction

\# \\\# Process Overview

This section implements the first major stage of the AMTAIR pipeline: transforming source documents (such as research papers, blog posts, or expert analyses) into structured argument representations using the ArgDown format.

ArgDown is a markdown-like notation for representing arguments in a hierarchical structure. In the context of AMTAIR, it serves as the first step toward creating formal Bayesian networks by: 1. Identifying key variables/statements in the text 2. Capturing their hierarchical relationships 3. Preserving their descriptive content 4. Defining their possible states (instantiations)

The extraction process uses Large Language Models (LLMs) to identify the structure and relationships in the text, though in this notebook we focus on processing pre-formatted examples rather than performing the full extraction from raw text.

\# \\\# What is ArgDown?

ArgDown uses a simple syntax where: - Statements are represented as `[Statement]: Description` - Relationships are indicated with `+` symbols and indentation - Metadata is added in JSON format, including possible states of each variable

For example:

```
[MainClaim]: Description of the main claim. \\{"instantiations": ["claim\\_TRUE", "claim\\_FALS

 + [SupportingEvidence]: Description of evidence. \\{"instantiations": ["evidence\\_TRUE", "evi
```

This structure will later be enhanced with probability information to create BayesDown, which can be transformed into a Bayesian network for analysis and visualization.

\# \\\# 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.

[10]:
```python
# @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

[11]:
```python
# @title 1.2.0 --- Prompt Template Function Definitions ---

"""
BLOCK PURPOSE: Defines a flexible template system for LLM prompts used in the␣
↪extraction pipeline.

This block implements two key classes:
1. PromptTemplate: A simple template class supporting variable substitution for␣
↪dynamic prompts
2. PromptLibrary: A collection of pre-defined prompt templates for different␣
↪extraction tasks
```

```
These templates are used in the ArgDown extraction and BayesDown probability␣
↪extraction
stages of the pipeline, providing consistent and well-structured prompts to the␣
↪LLMs.

DEPENDENCIES: string.Template for variable substitution
OUTPUTS: PromptTemplate and PromptLibrary classes
"""

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 - transforms source text into structured␣
↪argument map
    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 - enhances ArgDown with probability information
    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

[12]:
```
# @title 1.3.0 --- Provider-Agnostic LLM API Interface ---

"""
BLOCK PURPOSE: Provides a unified interface for interacting with different LLM␣
↪providers.

This block implements a flexible, provider-agnostic system for making LLM API␣
↪calls:
1. Base abstract class (LLMProvider) defining the common interface
2. Implementation classes for specific providers (OpenAI and Anthropic)
3. Factory class for creating appropriate provider instances

This abstraction allows the extraction pipeline to work with different LLM␣
↪providers
without changing the core code, supporting both current and future LLM backends.

DEPENDENCIES: requests for API calls, os for environment variables, abstract␣
↪base classes
OUTPUTS: LLMProvider abstract class and concrete implementations for OpenAI and␣
↪Anthropic
"""

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
```

```python
@dataclass
class LLMResponse:
    """Standard response object for LLM completions"""
    content: str             # The generated text response
    model: str               # The model used for generation
    usage: Dict[str, int]    # Token usage statistics
    raw_response: Dict[str, Any]  # Complete provider-specific response
    created_at: float = time.time()  # Timestamp of response creation

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:
```

```python
        """Generate a completion using OpenAI's API"""

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

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

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

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

        # Prepare request data
        data = {
            "model": model,
            "messages": messages,
            "temperature": temperature,
            "max_tokens": max_tokens
        }

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

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

        # Transform into standardized response format
        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"""

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

        # Prepare request data in Anthropic-specific format
        data = {
            "model": model,
            "messages": [{"role": "user", "content": prompt}],
            "temperature": temperature,
```

```
            "max_tokens": max_tokens
        }

        # Add system prompt if provided (Anthropic uses a different format)
        if system_prompt:
            data["system"] = system_prompt

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

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

        # Transform into standardized response format
        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}")
```

[13]:

```
# @title 1.3.0 --- API Call Function Definitions ---

"""
BLOCK PURPOSE: Provides core functions for extracting ArgDown representations␣
↪from text using LLMs.

This block implements the main extraction functionality:
1. extract_argdown_from_text: Sends text to LLM to extract structured ArgDown␣
↪representation
2. validate_argdown: Verifies the extracted ArgDown for correctness and␣
↪completeness
3. process_source_document: Handles source files (PDF, TXT, MD) and manages␣
↪extraction
4. save_argdown_extraction: Saves extraction results with metadata for further␣
↪processing

These functions form the first stage of the AMTAIR pipeline, transforming
unstructured text into structured argument representations.

DEPENDENCIES: LLMFactory from previous cell, re for pattern matching
OUTPUTS: Functions for ArgDown extraction, validation, and storage
"""

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)
```

```python
    # 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

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:
```

```python
            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"]
```

```python
    }

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

[14]:

```python
# @title 1.3 --- Prepare LLM API Call ---

"""
BLOCK PURPOSE: Prepares parameters for LLM API calls used in ArgDown extraction.

This function handles the configuration for LLM API calls, including:
1. Source document path validation
2. LLM provider selection and validation
3. Model selection with appropriate defaults

The function returns a configuration dictionary that can be passed to the
extraction function in the next step of the pipeline.

DEPENDENCIES: None (uses standard Python functionality)
OUTPUTS: Dictionary with extraction configuration parameters
"""

def prepare_extraction_call(source_path, provider_name="openai", model=None):
    """
    Prepare the LLM API call for ArgDown extraction

    Args:
        source_path (str): Path to the source document to extract from
        provider_name (str): LLM provider to use ('openai' or 'anthropic')
        model (str, optional): Specific model to use. Defaults to None (uses␣
 ↪provider's default).

    Returns:
        dict: Configuration parameters for extraction

    Raises:
        ValueError: If an unsupported provider is specified
    """
    # 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

[15]:
```
# @title 1.4 --- Make ArgDown Extraction LLM API Call ---

"""
BLOCK PURPOSE: Executes the ArgDown extraction process using the LLM API.

This function performs the actual extraction of ArgDown representations from␣
 ↪source documents:
1. Takes the configuration parameters prepared in the previous step
2. Processes the document using the LLM API
3. Validates the extraction results
4. Provides timing and statistics about the extraction

The extraction process transforms unstructured text into a structured argument
representation following the ArgDown syntax defined in the AMTAIR project.

DEPENDENCIES: process_source_document function from previous cells
OUTPUTS: Dictionary with extraction results including ArgDown content and␣
 ↪validation info
"""

def execute_extraction(extraction_config):
    """
    Execute the ArgDown extraction using the LLM API
```

```
    Args:
        extraction_config (dict): Configuration parameters for extraction

    Returns:
        dict: Extraction results including ArgDown content and validation info

    Raises:
        Exception: For any errors during extraction
    """
    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

[16]:

```python
# @title 1.5 --- Save ArgDown Extraction Response ---

"""
BLOCK PURPOSE: Saves the extracted ArgDown content to files for further␣
↪processing.

This function handles saving the extraction results:
1. Creates an output directory if it doesn't exist
2. Saves the extracted ArgDown content with a timestamp in the filename
3. Saves accompanying metadata in a JSON file
4. Saves a copy at a standard location for the next steps in the pipeline
5. Provides a preview of the extracted content

The saved files serve as inputs for the next stage of the pipeline where
probability information will be added to create BayesDown.

DEPENDENCIES: os module for directory operations
OUTPUTS: Saved ArgDown files and preview of extracted content
"""

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

    Args:
        results (dict): Extraction results from execute_extraction
        output_directory (str): Directory to save results

    Returns:
        str: Path to the saved ArgDown 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

[17]:
```
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.

[18]:
```
# @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.

[19]:
```python
# @title 1.7 --- Parsing ArgDown & BayesDown (.md to .csv) ---

"""
BLOCK PURPOSE: Provides the core parsing functionality for transforming ArgDown␣
↪and BayesDown
text representations into structured DataFrame format for further processing.

This block implements the critical extraction pipeline described in the AMTAIR␣
↪project
(see PY_TechnicalImplementation) that converts argument structures into␣
↪Bayesian networks.
The function can handle both basic ArgDown (structure-only) and BayesDown (with␣
↪probabilities).

Key steps in the parsing process:
1. Remove comments from the markdown text
2. Extract titles, descriptions, and indentation levels
3. Establish parent-child relationships based on indentation
4. Convert the structured information into a DataFrame
5. Add derived columns for network analysis

DEPENDENCIES: pandas, re, json libraries
INPUTS: Markdown text in ArgDown/BayesDown format
OUTPUTS: Structured DataFrame with node information, relationships, and␣
↪properties
"""

def parse_markdown_hierarchy_fixed(markdown_text, ArgDown=False):
    """
    Parse ArgDown or BayesDown format into a structured DataFrame with␣
↪parent-child relationships.

    Args:
        markdown_text (str): Text in ArgDown or BayesDown format
        ArgDown (bool): If True, extracts only structure without probabilities
                        If False, extracts both structure and probability␣
↪information

    Returns:
        pandas.DataFrame: Structured data with node information, relationships,␣
↪and attributes
    """
    # PHASE 1: Clean and prepare the text
    clean_text = remove_comments(markdown_text)

    # PHASE 2: Extract basic information about nodes
```

```
    titles_info = extract_titles_info(clean_text)

    # PHASE 3: Determine the hierarchical relationships
    titles_with_relations = establish_relationships_fixed(titles_info,␣
↪clean_text)

    # PHASE 4: Convert to structured DataFrame format
    df = convert_to_dataframe(titles_with_relations, ArgDown)

    # PHASE 5: Add derived columns for analysis
    df = add_no_parent_no_child_columns_to_df(df)
    df = add_parents_instantiation_columns_to_df(df)

    return df

def remove_comments(markdown_text):
    """
    Remove comment blocks from markdown text using regex pattern matching.

    Args:
        markdown_text (str): Text containing potential comment blocks

    Returns:
        str: Text with comment blocks removed
    """
    # Remove anything between /* and */ using regex
    return re.sub(r'/\*.*?\*/', '', markdown_text, flags=re.DOTALL)

def extract_titles_info(text):
    """
    Extract titles with their descriptions and indentation levels from markdown␣
↪text.

    Args:
        text (str): Cleaned markdown text

    Returns:
        dict: Dictionary with titles as keys and dictionaries of attributes as␣
↪values
    """
    lines = text.split('\n')
    titles_info = {}

    for line in lines:
        # Skip empty lines
        if not line.strip():
            continue
```

```python
    # Extract title within square or angle brackets
    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()

        # Split description and metadata at the first "{"
        if "{" in full_text:
            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 = ''  # Initialize as empty string
    else:
        description = ''
        metadata = ''  # Ensure metadata is initialized

    # Calculate indentation level based on spaces before + or - symbol
    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

    Args:
        titles_info (dict): Dictionary with information about titles
        text (str): Original markdown text (for identifying line numbers)

    Returns:
```

```
        dict: Updated dictionary with parent-child relationships
    """
    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
```

```python
    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

    # 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.

    Args:
        titles_info (dict): Dictionary with information about titles
        ArgDown (bool): If True, extract only structural information without␣
↪probabilities

    Returns:
        pandas.DataFrame: Structured data with node information and␣
↪relationships
    """
    if ArgDown == True:
        # For ArgDown, exclude probability columns
        df = pd.DataFrame(columns=['Title', 'Description', 'line',␣
↪'line_numbers', 'indentation',
                                    'indentation_levels', 'Parents', 'Children',␣
↪'instantiations'])
    else:
        # For BayesDown, include probability columns
        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 instantiations as␣
↪metadata only, no probabilities 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 probabilities for BayesDown
                        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 to identify␣
 ↪root and leaf nodes.

    Args:
        dataframe (pandas.DataFrame): The DataFrame to enhance

    Returns:
        pandas.DataFrame: Enhanced DataFrame with additional boolean columns
    """
```

```
    no_parent = []
    no_children = []

    for _, row in dataframe.iterrows():
        no_parent.append(not row['Parents'])  # True if Parents list is empty
        no_children.append(not row['Children'])  # True if Children list is␣
↪empty

    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 parents as a list of lists column to the␣
↪DataFrame.
    This is crucial for generating conditional probability tables.

    Args:
        dataframe (pandas.DataFrame): The DataFrame to enhance

    Returns:
        pandas.DataFrame: Enhanced DataFrame with parent_instantiations column
    """
    # 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
```

[23]:
```
# example use case:
ex_csv = parse_markdown_hierarchy_fixed(md_content, ArgDown = True)
ex_csv
```

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

[24]:
```
# 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)
```

[25]:
```
# 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. ArgDown to BayesDown: Adding Probability Information

\# \\\# Process Overview

This section implements the second major stage of the AMTAIR pipeline: enhancing the structured argument representation (ArgDown) with probability information to create BayesDown.

BayesDown extends ArgDown by adding: 1. Prior probabilities for each variable (unconditional beliefs) 2. Conditional probabilities representing the relationships between variables 3. The full parameter specification needed for a Bayesian network

The process follows these steps: 1. Generate probability questions for each node and its relationships 2. Create a BayesDown template with placeholders for these probabilities 3. Answer the probability questions (manually or via LLM) 4. Substitute the answers into the BayesDown representation

This enhanced representation contains all the information needed to construct a formal Bayesian network, enabling probabilistic reasoning and policy evaluation.

\# \\\# What is BayesDown?

BayesDown maintains the ArgDown structure but adds probability metadata:

[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"
\\}

The result is a hybrid representation that preserves the narrative structure of arguments while adding the mathematical precision of Bayesian networks.

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

[26]:
```
# @title 2.1 --- Probability Extraction Questions Generation ---

"""
BLOCK PURPOSE: Generates probability questions for ArgDown nodes to prepare for␣
↪BayesDown conversion.

This block implements a key step in the pipeline where structure (from ArgDown)␣
↪is prepared
for probability integration (to create BayesDown). It:

1. Processes a CSV file containing ArgDown structure
2. For each node, generates appropriate probability questions:
   - Prior probability questions for all nodes
   - Conditional probability questions for nodes with parents
3. Creates a new CSV file with these questions ready for the next stage

The generated questions serve as placeholders that will be answered in the
probability extraction phase to complete the Bayesian network.

DEPENDENCIES: pandas, json, itertools libraries
INPUTS: ArgDown CSV file
OUTPUTS: Enhanced CSV with probability questions for each node
"""

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.
```

```
    Args:
        instantiations_str: Instantiations in string or list format

    Returns:
        list: Parsed instantiations as a list
    """
    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.

    Args:
        parents_str: Parents in string or list format

    Returns:
        list: Parsed parents as a list
    """
    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:
```

```python
        # 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.

    Args:
        parent (str): Parent node name
        df (DataFrame): DataFrame containing node information

    Returns:
        list: Instantiations for the parent node
    """
    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'  # 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}'  # 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

    Returns:
        DataFrame: Enhanced DataFrame with probability questions

    Raises:
        Exception: If CSV loading fails or required columns are missing
    """
```

```
    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")
```

[27]:
```
# 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'

[28]:
```
# @title 2.2 --- BayesDown Questions Generation ---

"""
BLOCK PURPOSE: Transforms the ArgDown with questions into a BayesDown template␣
 ↪with placeholders.

This function creates a BayesDown representation with probability placeholders␣
 ↪based on
the questions generated in the previous step. It:

1. Loads the CSV file with probability questions
2. Constructs a directed graph to represent the causal structure
3. Processes each node to create BayesDown syntax with probability placeholders
4. Optionally includes comments with the specific questions to be answered
5. Saves the result as a markdown file for the next stage of the pipeline

The output is a BayesDown template that can be used in the probability␣
 ↪extraction phase,
where the placeholders will be replaced with actual probability values.

DEPENDENCIES: networkx, pandas, json libraries
INPUTS: CSV file with ArgDown structure and probability questions
OUTPUTS: BayesDown markdown file with probability placeholders
"""

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.

Returns:
    str: The generated BayesDown content

Raises:
    Exception: If CSV loading fails or required columns are missing
"""
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']
```

```python
    # 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
```

[29]:
```
# 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}")
```

[31]:
```
# 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[:50000] + "...\n")
```

[32]:
```
# 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)
```

[35]:
```
# 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[:50000] + "...\n")
```

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

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

\# \\\# 2.3.1 BayesDown Format Specification

BayesDown extends ArgDown with probability data in a structured JSON format to represent Bayesian networks. This intermediate representation bridges the gap between natural language arguments and formal probabilistic models, preserving both narrative structure and quantitative relationships.

\# \\\#\\\# Core Structure

A BayesDown representation consists of:

1. **Nodes**: Variables or statements in brackets `[Node\\_Name]` with descriptive text
2. **Relationships**: Hierarchical structure with indentation and `+` symbols
3. **Metadata**: JSON objects containing probability information:

"'json \{ "instantiations": [”state_TRUE“,”state_FALSE”], // Possible states of variable "priors": \{ ”p(state_TRUE)“:”0.7”, // Unconditional probability of state_TRUE ”p(state_FALSE)“:”0.3” // Unconditional probability of state_FALSE \}, "posteriors": \{ ”p(state_TRUE|condition1_TRUE,condition2_FALSE)“:”0.9“, // Conditional on parent states”p(state_TRUE|condition1_FALSE,condition2_TRUE)“:”0.4” // Different parent configuration \} \}

\# \\\#\\\#\\\#\\\# Rain-Sprinkler-Lawn Example [Grass_Wet]: Concentrated moisture on grass. \{"instantiations": [”grass_wet_TRUE“,”grass_wet_FALSE”], "priors": \{”p(grass_wet_TRUE)“:”0.322“,”p(grass_wet_FALSE)“:”0.678“\},”posteriors“: \{”p(grass_wet_TRUE|sprinkler_TRUE,rain_TRUE)“:”0.99“,”p(grass_wet_TRUE|sprinkler_TRUE,ra + [Rain]: Water falling from the sky. \{”instantiations“: [”rain_TRUE”, ”rain_FALSE”],”priors“: \{”p(rain_TRUE)“:”0.2“,”p(rain_FALSE)“:”0.8“\}\} + [Sprinkler]: Artificial watering system. \{”instantiations“: [”sprinkler_TRUE”, ”sprinkler_FALSE”],”priors“: \{”p(sprinkler_TRUE)“:”0.44838“,”p(sprinkler_FALSE)“:”0.55162“\},”posteriors“: \{”p(sprinkler_TRUE|rain_TRUE)“:”0.01“,”p(sprinkler_TRUE|rain_FALSE)“:”0.4”\}\} + [Rain]

In this example:

- Grass_Wet is the effect/outcome node
- Rain and Sprinkler are parent nodes (causes)
- Rain also influences Sprinkler (people tend not to use sprinklers when it's raining)

Role in AMTAIR BayesDown serves as the critical intermediate representation in the AMTAIR extraction pipeline, bridging between qualitative arguments in AI safety literature and formal Bayesian networks that can be used for probabilistic reasoning and policy evaluation. By preserving both narrative explanation and probabilistic information, it enables the automated extraction of world models while maintaining traceability to the original arguments. For full syntax details, see the BayesDownSyntax.md file in the repository.

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

\# \\\# 2.4 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.

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

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

\# 3. BayesDown to Structured Data: Network Construction

\# \\\# Extraction Pipeline Overview

This section implements the core extraction pipeline described in the AMTAIR project documentation (see `PY\\_TechnicalImplementation.md`), which transforms structured argument representations into formal Bayesian networks through a series of processing steps:

1. **Input**: Text in BayesDown format (see Section 2.3.1)
2. **Parsing**: Extract nodes, relationships, and probability information
3. **Structuring**: Organize into a DataFrame with formal relationships
4. **Enhancement**: Add derived properties and network metrics
5. **Output**: Structured data ready for Bayesian network construction

\# \\\#\\\# Theoretical Foundation

This implementation follows the extraction algorithm outlined in the AMTAIR project description:

1. Get nodes: All premises and conclusions from the argument structure
2. Get edges: Parent-child relationships between nodes
3. Extract probability distributions: Prior and conditional probabilities
4. Calculate derived metrics: Network statistics and node classifications

The resulting structured data maintains the complete information needed to reconstruct the Bayesian network while enabling additional analysis and visualization.

\# \\\#\\\# Role in Thesis Research

This extraction pipeline represents a key contribution of the Master's thesis, demonstrating how argument structures from AI safety literature can be automatically transformed into formal probabilistic models. While the current implementation focuses on pre-formatted BayesDown, the architecture is designed to be extended with LLM-powered extraction directly from natural language in future work.

The rain-sprinkler-lawn example serves as a simple but complete test case, demonstrating every step in the pipeline from structured text to interactive Bayesian network visualization.

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

[44]:
```
# read sprinkler example -- Occam Colab Online
file_path_ex_rain = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/main/data/example_carlsmith/BayesDown.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

[45]:
```
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.

\# \\\# 3.3 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.

[46]:
```
result_df = parse_markdown_hierarchy_fixed(md_content_ex_rain)
result_df
```

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

[47]:
```
# @title 3.3.1 Data Post-Processing Functions ---

"""
BLOCK PURPOSE: Enhances the extracted BayesDown data with calculated metrics and network properties.
```

```
This block provides functions to enrich the basic extracted data with additional
calculated columns that are useful for analysis and visualization:

1. Joint probabilities - Calculating P(A,B) from conditional and prior␣
↪probabilities
2. Network metrics - Centrality measures that indicate importance of nodes in␣
↪the network
3. Markov blanket - Identifying the minimal set of nodes that shield a node␣
↪from the rest

These enhancements provide valuable context for understanding the network␣
↪structure
and the relationships between variables, enabling more advanced analysis and
improving visualization.

DEPENDENCIES: networkx for graph calculations
INPUTS: DataFrame with basic extracted BayesDown data
OUTPUTS: Enhanced DataFrame with additional calculated columns
"""

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 - P(A,B) = P(A|B) * P(B)
    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):
                                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)  # Overall connectedness
    in_degree_centrality = nx.in_degree_centrality(G)  # How many nodes affect␣
↪this one
```

```python
    out_degree_centrality = nx.out_degree_centrality(G)  # How many nodes this one affects

    try:
        betweenness_centrality = nx.betweenness_centrality(G)  # Node's role as a connector
    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
```

[48]:
```
# @title 3.3 --- Enhance Extracted Data with Network Metrics ---

"""
BLOCK PURPOSE: Applies the post-processing functions to enhance the extracted␣
 ↪data.

This block takes the basic extracted DataFrame from the BayesDown parsing step
and enriches it with calculated metrics that provide deeper insight into the
network structure and relationships. It:

1. Applies the enhancement functions defined previously
2. Displays summary information about key calculated metrics
3. Saves the enhanced data for further analysis and visualization

The enhanced DataFrame provides a richer representation of the Bayesian network,
including measures of node importance and conditional relationships that are
essential for effective analysis and visualization.

DEPENDENCIES: enhance_extracted_data function
INPUTS: DataFrame with basic extracted BayesDown data
OUTPUTS: Enhanced DataFrame with additional calculated columns, saved to CSV
"""

# 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

[49]:
```
# @title 3.4 --- Save Extracted Data for Further Processing ---

"""
BLOCK PURPOSE: Saves the extracted data to a CSV file for further processing.

This step is essential for:
1. Persisting the structured representation of the Bayesian network
2. Enabling further analysis in other tools or notebook sections
3. Creating a permanent record of the extraction results
4. Making the data available for the visualization pipeline

The CSV format provides a standardized, tabular representation of the network
that can be easily loaded and processed in subsequent analysis steps.

DEPENDENCIES: pandas DataFrame operations
INPUTS: Extracted DataFrame from the parsing step
OUTPUTS: CSV file containing the structured network data
"""

# Save the extracted data as a CSV file
result_df.to_csv('extracted_data.csv', index=False)

print("  Extracted data saved successfully to 'extracted_data.csv'")
print("Note: If using updated data in future steps, the file must be pushed to␣
 ↪the GitHub repository")
```

\# 4. Analysis \\\& Inference: Bayesian Network Visualization

\# \\\# Bayesian Network Visualization Approach

This section implements the visualization component of the AMTAIR project, transforming the structured data extracted from BayesDown into an interactive network visualization that makes complex probabilistic relationships accessible to human understanding.

\# \\\#\\\# Visualization Philosophy

A key challenge in AI governance is making complex probabilistic relationships understandable to diverse stakeholders. This visualization system addresses this challenge through:

1. **Visual Encoding of Probability**: Node colors reflect probability values (green for high

probability, red for low)
2. **Structural Classification**: Border colors indicate node types (blue for root causes, purple for intermediate nodes, magenta for leaf nodes)
3. **Progressive Disclosure**: Basic information in tooltips, detailed probability tables in modal popups
4. **Interactive Exploration**: Draggable nodes, configurable physics, click interactions

\# \\\#\\\# Connection to AMTAIR Goals

This visualization approach directly supports the AMTAIR project's goal of improving coordination in AI governance by:

1. Making implicit models explicit through visual representation
2. Providing a common language for discussing probabilistic relationships
3. Enabling non-technical stakeholders to engage with formal models
4. Creating shareable artifacts that facilitate collaboration

\# \\\#\\\# Implementation Structure

The visualization system is implemented in four phases:

1. **Network Construction**: Creating a directed graph representation using NetworkX
2. **Node Classification**: Identifying node types based on network position
3. **Visual Enhancement**: Adding color coding, tooltips, and interactive elements
4. **Interactive Features**: Implementing click handling for detailed exploration

The resulting visualization serves as both an analytical tool for experts and a communication tool for broader audiences, bridging the gap between technical and policy domains in AI governance discussions.

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

[50]:
```
# @title 4.0 --- Bayesian Network Visualization Functions ---

"""
BLOCK PURPOSE: Provides functions to create interactive Bayesian network␣
 ↪visualizations
from DataFrame representations of ArgDown/BayesDown data.

This block implements the visualization pipeline described in the AMTAIR␣
 ↪project, transforming
the structured DataFrame extracted from ArgDown/BayesDown into an interactive␣
 ↪network graph
that displays nodes, relationships, and probability information. The␣
 ↪visualization leverages
NetworkX for graph representation and PyVis for interactive display.

Key visualization features:
1. Color-coding of nodes based on probability values
2. Border styling to indicate node types (root, intermediate, leaf)
3. Interactive tooltips with probability information
```

```
4. Modal popups with detailed conditional probability tables
5. Physics-based layout for intuitive exploration

DEPENDENCIES: networkx, pyvis, HTML display from IPython
INPUTS: DataFrame with node information, relationships, and probabilities
OUTPUTS: Interactive HTML visualization of the Bayesian network
"""

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.

    Args:
        df (pandas.DataFrame): DataFrame containing node information,␣
↪relationships, and probabilities

    Returns:
        IPython.display.HTML: Interactive HTML visualization of the Bayesian␣
↪network
    """
    # PHASE 1: Create a directed graph representation
    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 based on parent-child relationships
    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)

    # PHASE 2: Classify nodes based on network structure
    classify_nodes(G)

    # PHASE 3: Create interactive 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'])  # Allow user to adjust physics␣
↪settings

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

    # PHASE 4: Enhance node appearance with probability information
    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
            }
        }

    # PHASE 5: Setup interactive click handling
    # Prepare data for click handler
    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()}
    }

    # JavaScript code for handling node clicks
    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'])

# PHASE 6: Save the graph to HTML and inject custom click handling
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>")
```

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

[51]:
```
# @title 4.1 --- Node Classification and Styling Functions ---

"""
BLOCK PURPOSE: Implements the visual classification and styling of nodes in the␣
↪Bayesian network.

This module handles the identification of node types based on their position in␣
↪the network
and provides appropriate visual styling for each type. The functions:

1. Classify nodes as parents (causes), children (intermediate effects), or␣
↪leaves (final effects)
2. Assign appropriate border colors to visually distinguish node types
3. Calculate background colors based on probability values
4. Extract relevant information from DataFrame rows in a robust manner

The visual encoding helps users understand both the structure of the network
and the probability distributions at a glance.

DEPENDENCIES: colorsys for color manipulation
INPUTS: Graph structure and node data
OUTPUTS: Classification and styling information for visualization
"""

def classify_nodes(G):
    """
    Classify nodes as parent, child, or leaf based on network structure
```

```
    Args:
        G (networkx.DiGraph): Directed graph representation of the Bayesian␣
↪network

    Effects:
        Adds 'node_type' attribute to each node in the graph:
        - 'parent': Root node with no parents but has children (causal source)
        - 'child': Node with both parents and children (intermediate)
        - 'leaf': Node with parents but no children (final effect)
        - 'isolated': Node with no connections (rare in Bayesian networks)
    """
    for node in G.nodes():
        predecessors = list(G.predecessors(node))  # Nodes pointing to this one␣
↪(causes)
        successors = list(G.successors(node))      # Nodes this one points to␣
↪(effects)

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

def get_border_color(node_type):
    """
    Return border color based on node type

    Args:
        node_type (str): Type of node ('parent', 'child', 'leaf', or 'isolated')

    Returns:
        str: Hex color code for node border
    """
    if node_type == 'parent':
        return '#0000FF'  # Blue for root causes
    elif node_type == 'child':
        return '#800080'  # Purple for intermediate nodes
    elif node_type == 'leaf':
        return '#FF00FF'  # Magenta for final effects
    else:
        return '#000000'  # Default black for any other type
```

```python
def get_probability_color(priors):
    """
    Create background color based on probability (red to green gradient)

    Args:
        priors (dict): Dictionary containing probability information

    Returns:
        str: Hex color code for node background, ranging from red (low␣
↪probability)
             to green (high probability)
    """
    # 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

    Args:
        row (pandas.Series): Row from DataFrame containing node information

    Returns:
        list: List of parent node names
    """
    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

    Args:
        row (pandas.Series): Row from DataFrame containing node information

    Returns:
        list: List of possible instantiations (states) for the node
    """
    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"]
```

```python
    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

    Args:
        row (pandas.Series): Row from DataFrame containing node information

    Returns:
        dict: Dictionary of prior probabilities with 'true_prob' added for convenience
    """
    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

    Args:
        row (pandas.Series): Row from DataFrame containing node information
```

```
    Returns:
        dict: Dictionary of conditional probabilities
    """
    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

[52]:
```
# @title 4.2 --- HTML Content Generation Functions ---

"""
BLOCK PURPOSE: Creates rich HTML content for the interactive Bayesian network␣
↪visualization.

This module generates the HTML components that enhance the Bayesian network␣
↪visualization:
1. Probability bars - Visual representation of probability distributions
2. Node tooltips - Rich information displayed on hover
3. Expanded content - Detailed probability information shown when clicking nodes

These HTML components make the mathematical concepts of Bayesian networks more
intuitive and accessible to users without requiring deep statistical knowledge.
The visual encoding of probabilities (colors, bars) and the progressive␣
↪disclosure
of information (hover, click) help users build understanding at their own pace.

DEPENDENCIES: HTML generation capabilities
INPUTS: Node data from the Bayesian network
OUTPUTS: HTML content for visualization components
"""

def create_probability_bar(true_prob, false_prob, height="15px",␣
↪show_values=True, value_prefix=""):
    """
    Creates a reusable HTML component to visualize probability distribution

    Args:
        true_prob (float): Probability of the true state (0.0-1.0)
        false_prob (float): Probability of the false state (0.0-1.0)
        height (str): CSS height of the bar
        show_values (bool): Whether to display numerical values
        value_prefix (str): Prefix to add before values (e.g., "p=")

    Returns:
        str: HTML for a horizontal bar showing probabilities
    """
    # Prepare display labels if showing values
    true_label = f"{value_prefix}{true_prob:.3f}" if show_values else ""
    false_label = f"{value_prefix}{false_prob:.3f}" if show_values else ""

    # Create the HTML for a horizontal stacked bar
    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

    Args:
        node_id (str): Identifier of the node
        node_data (dict): Node attributes including probabilities

    Returns:
        str: HTML content for tooltip displayed on hover
    """
    # Extract node 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

    Args:
        node_id (str): Identifier of the node
        node_data (dict): Node attributes including probabilities

    Returns:
        str: HTML content for detailed view displayed on click
    """
    # Extract node information
    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

[53]:
```
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
```

```python
    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

[54]:
```
create_bayesian_network_with_probabilities(result_df)
```

[55]:
```
# Use the function to create and display the visualization

print(result_df)
```

\# Conclusion: From Prototype to Production

\# \\\# Summary of Achievements

This notebook has successfully demonstrated the core AMTAIR extraction pipeline, transforming structured argument representations into interactive Bayesian network visualizations through the following steps:

1. **Environment Setup**: Established a reproducible environment with necessary libraries and data access
2. **Argument Extraction**: Processed structured ArgDown representations preserving the hierarchical relationships
3. **Probability Integration**: Enhanced arguments with probability information to create BayesDown
4. **Data Transformation**: Converted BayesDown into structured DataFrame representation
5. **Visualization \& Analysis**: Created interactive Bayesian network visualizations with probability encoding

The rain-sprinkler-lawn example, though simple, demonstrates all the key components of the extraction pipeline that can be applied to more complex AI safety arguments.

\# \\\# Limitations and Future Work

While this prototype successfully demonstrates the core pipeline, several limitations and opportunities for future work remain:

1. **LLM Extraction**: The current implementation focuses on processing pre-formatted ArgDown rather than performing extraction directly from unstructured text. Future work will integrate LLM-powered extraction.

2. **Scalability**: The system has been tested on small examples; scaling to larger, more complex arguments will require additional optimization and handling of computational complexity.

3. **Policy Evaluation**: The current implementation focuses on representation and visualization; future work will add policy evaluation capabilities by implementing intervention modeling.

4. **Prediction Market Integration**: Future versions will integrate with forecasting platforms to incorporate live data into the models.

\# \\\# Connection to AMTAIR Project

This prototype represents just one component of the broader AMTAIR project described in the project documentation (see PY_AMTAIRDescription and PY_AMTAIR_SoftwareToolsNMilestones). The full project includes:

1. **AI Risk Pathway Analyzer (ARPA)**: The core extraction and visualization system demonstrated in this notebook
2. **Worldview Comparator**: Tools for comparing different perspectives on AI risk
3. **Policy Impact Evaluator**: Systems for evaluating intervention effects across scenarios
4. **Strategic Intervention Generator**: Tools for identifying robust governance strategies

Together, these components aim to address the coordination crisis in AI governance by providing computational tools that make implicit models explicit, identify cruxes of disagreement, and evaluate policy impacts across diverse worldviews.

By transforming unstructured text into formal, analyzable representations, the AMTAIR project helps bridge the gaps between technical researchers, policy specialists, and other stakeholders, enabling more effective coordination in addressing existential risks from advanced AI.

\# 6.0 Save Outputs

\# 6. Saving and Exporting Results

This section provides tools for saving the notebook results and visualizations in various formats:

1. **HTML Export**: Creates a self-contained HTML version of the notebook with all visualizations
2. **Markdown Export**: Generates documentation-friendly Markdown version of the notebook
3. **PDF Export**: Creates a PDF document for formal sharing (requires LaTeX installation)

These exports are essential for: - Sharing analysis results with colleagues and stakeholders - Including visualizations in presentations and reports - Creating documentation for the AMTAIR project - Preserving results for future reference

The different formats serve different purposes, from interactive exploration (HTML) to documentation (Markdown) to formal presentation (PDF).

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!

[ ]:
```
# @title 6.0 --- Save Visualization and Notebook Outputs as .HTML---

"""
BLOCK PURPOSE: Provides tools for saving the notebook results in various␣
 ↪formats.

This block offers functions to:
1. Convert the notebook to HTML for easy sharing and viewing
2. Convert the notebook to Markdown for documentation purposes
3. Save the visualization outputs for external use

These tools are essential for preserving the analysis results and making them
accessible outside the notebook environment, supporting knowledge transfer
and integration with other AMTAIR project components.

DEPENDENCIES: nbformat, nbconvert modules
INPUTS: Current notebook state
OUTPUTS: HTML, Markdown, or other format versions of the notebook
"""

import nbformat
from nbconvert import HTMLExporter
import os

# Repository URL variable for file access
repo_url = "https://raw.githubusercontent.com/SingularitySmith/AMTAIR_Prototype/
 ↪main/data/example_carlsmith/"
notebook_name = "AMTAIR_Prototype_example_carlsmith"  # Change when working␣
 ↪with different examples
```

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

# Load the notebook
try:
  with open(f"{notebook_name}.ipynb") as f:
    nb = nbformat.read(f, as_version=4)
  print(f" Successfully loaded notebook: {notebook_name}.ipynb")
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
try:
    (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)
    print(f" Successfully saved HTML version to: {notebook_name}IPYNB.html")
except Exception as e:
    print(f" Error converting notebook to HTML: {str(e)}")
```

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

[ ]:
```
# @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_example_carlsmith"  #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)
```

[ ]:
```
# @title 6.1 --- Convert Notebook to Markdown Documentation ---

"""
BLOCK PURPOSE: Converts the notebook to Markdown format for documentation␣
↪purposes.

Markdown is a lightweight markup language that is widely used for documentation
and is easily readable in both plain text and rendered formats. This conversion:

1. Preserves the structure and content of the notebook
2. Creates a format suitable for inclusion in documentation systems
3. Excludes code outputs to focus on the process and methodology
4. Supports version control and collaboration on GitHub

The resulting Markdown file can be used in project documentation, GitHub wikis,
or as a standalone reference guide to the AMTAIR extraction pipeline.

DEPENDENCIES: nbformat, nbconvert.MarkdownExporter modules
INPUTS: Current notebook state
OUTPUTS: Markdown version of the notebook
"""

import nbformat
from nbconvert import MarkdownExporter
import os

# Repository URL variable for file access
# repo_url = "https://raw.githubusercontent.com/SingularitySmith/
↪AMTAIR_Prototype/main/data/example_carlsmith/"
notebook_name = "AMTAIR_Prototype_example_carlsmith"  # Change when working␣
↪with different examples
```

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

# Load the notebook
try:
  with open(f"{notebook_name}.ipynb") as f:
    nb = nbformat.read(f, as_version=4)
  print(f" Successfully loaded notebook: {notebook_name}.ipynb")
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)  # Exclude outputs for cleaner␣
↪documentation

# Convert the notebook to Markdown
try:
    (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)
    print(f" Successfully saved Markdown version to: {notebook_name}IPYNB.md")
except Exception as e:
    print(f" Error converting notebook to Markdown: {str(e)}")
```

[ ]:
```
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_example_carlsmith"  #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)
```