A Report On
Intelligent Tutor System (ITS)

Govind Tuli (2021A3PS0130P)

Piyush Chanduka (2021A4PS1554P)

Vatsal Agarwal (2021A3PS1074P)

Overview

Introduction:

In this study, we aim to analyze and visualize the effects of varying weights and initial probabilities on a Bayesian network designed for assessing programming knowledge, while also considering the integration of ChatGPT for code evaluation. The network consists of nodes representing different programming concepts, with dependencies modeled by edges connecting parent and child nodes. Our goal is to understand how different parameters affect the network's behavior and use this information to improve its performance in evaluating the user's knowledge.

Methods:

Network Setup: We first set up the Bayesian network with nodes representing programming concepts, assigned content to each node, and initialized the probability_known attribute, which represents the likelihood of the user knowing the concept.

Conditional Probability Distributions (CPDs): We defined functions to update child and parent CPDs based on the user's interaction with the system (i.e., whether they provided a correct answer).

Access Conditions: We implemented a function to determine if the user can access a node based on the satisfaction of its parent nodes.

ChatGPT Integration: We integrated ChatGPT's code evaluation capabilities to assess user performance by detecting syntax and logical errors in their code. We then updated the probability of understanding for specific nodes based on the number of errors, applying penalties to refine the model.

Simulation: We simulated user interaction with the system, updating the CPDs according to their responses, and generated 3D scatter plots to visualize the impact of varying weights and initial probabilities on parent-child node relationships.

Exhaustive Simulation and Heatmap: We performed an exhaustive simulation to calculate the average change in probability across all nodes in the network for various combinations of initial probabilities and weights. We visualized the results using a heatmap, which provided an aggregated view of the network's behavior under different conditions.

Overview of Results:

3D Scatter Plots: The 3D scatter plots provided insights into how changes in child node probabilities and parent node weights affected the resulting probability of the parent nodes. We observed that higher parent node weights resulted in a stronger influence on the parent node's probability as the child node's probability increased.

ChatGPT Integration: With an efficiency of over 98% in detecting syntax and logical errors, the ChatGPT integration offers a robust code evaluation mechanism. However, the study's validity is contingent upon addressing the potential discrepancies between ChatGPT's code evaluation and the Bayesian Network model's probability estimations.

Heatmap: The heatmap revealed patterns in the average change in probability across all nodes in the network as a function of initial probabilities and weights. Higher weights resulted in more significant average probability changes, implying that they have a stronger influence on the network's behavior.

Conclusion:

The study showcases the potential for integrating code evaluation tools like ChatGPT with adaptive learning models like Bayesian Networks. By visualizing the impact of varying weights and initial probabilities on the network and considering the integration of ChatGPT for code evaluation, the study offers valuable insights into improving the network's performance in assessing programming knowledge. However, it is crucial to consider the limitations and potential biases introduced by combining these two technologies and address the potential discrepancies between ChatGPT's code evaluation and the Bayesian Network model's probability estimations. Further validation and testing are needed to ensure that the integration provides meaningful results.

Background

Bayesian networks

Let *U = {a1, a2,...,an}* denote a finite set of discrete random variables. Each variable a_i is associated with a finite domain dom(ai).

Let V be the Cartesian product of the variable domains, namely,

V = dom(a1) × dom(a2) × ... × dom(an). A joint probability distribution is a function p on V such that the following two conditions hold: (i) $0 \leq p(v) \leq 1.0$, for each configuration $v \in V$, (ii) $\sum_{v \in V} p(v)=1.0$.

Clearly, it may be impractical to obtain the joint distribution on U directly: for example, one would have to specify 2n entries for a distribution over n binary variables.

A Bayesian network [11] is a pair B = (D, C). In this pair, D is a directed acyclic graph (DAG) on a set U of variables, and $C = \{p(a_i|P_i) \mid a_i \in D\}$ is the corresponding set of conditional probability distributions (CPDs), where P_i denotes the parent set of variable a_i in the DAG D. A CPD $p(a_i|P_i)$ has the property that for each configuration (instantiation) of the variables in P_i, the sum of the probabilities of a_i is 1.0. Based on the probabilistic conditional independencies [17, 18] encoded in the DAG, the product of the CPDs is a unique joint probability distribution on U, namely, $p(a_1, a_2 \cdots , a_n) = n\ i=1\ p(a_i|P_i)$.

Thus, Bayesian networks provide a semantic modeling tool that facilitates the acquisition of probabilistic knowledge.

In addition, we have assigned weights to each node to indicate its importance. As a path is defined after obtaining the main topic, increasing the probability of a child node causes a slight change in the probability of its parent node. We accomplish this by updating the probabilities using weights and the weighted mean average.

This implementation assists us in producing the graphs we require. To see the user's progress, we created three-dimensional graphs with axes for weights, correct probabilities, and final probabilities.

Also, to calculate the ChatGPT efficiency, we have passed the correct and error code to the model. We then categorize it into 4 parts –

1. True Positive (TP): An error is correctly classified as present (positive).

2. True Negative (TN): An error is correctly classified as not present (negative).

3. False Positive (FP): An error is incorrectly classified as present (positive) when it is actually not present (negative).

4. False Negative (FN): An error is incorrectly classified as not present (negative) when it is actually present (positive).

In other words, true positive and true negative represent correct classifications, while false positive and false negative represent incorrect classifications. This categorization helps us to calculate the accuracy of the model by the formula

Accuracy = (TP + TN)/(TP + TN + FP + FN)

Literature Review

The Bayesian Inference Tutoring System (BITS) is an intelligent tutoring system that aims to assist students in learning C++ programming by providing adaptive guidance based on their current knowledge. BITS leverages Bayesian networks to model the student's knowledge and adapt the learning process accordingly. The study discusses three main adaptive guidance features in BITS: navigation support, prerequisite recommendations, and generating learning sequences.

Navigation Support: BITS uses a navigation menu with traffic lights to indicate the student's knowledge level for different concepts. Concepts are classified as already known (yellow), ready to learn (green), or not ready to learn (red). The system determines these categories by calculating the probability of a concept being known based on the student's knowledge of previous concepts and quiz results.

Prerequisite Recommendations: If a student struggles to understand a "ready to learn" concept, BITS identifies the prerequisite concepts (parent concepts in the Bayesian network) that the student may need to review. This approach helps students reinforce their understanding of foundational topics before moving on to more advanced material.

Generating Learning Sequences: BITS can create customized learning sequences for students who wish to learn a specific concept without covering all prior topics. When a student selects a "not ready to learn" concept, BITS

generates a learning sequence by displaying all unknown ancestral concepts in the Bayesian network, guiding the student in the optimal order for learning.

BITS is a web-based system that stores course materials, quizzes, and lecture notes in hypermedia formats such as HTML, Flash multimedia files, and XML documents. The quizzes consist of interactive Flash files and XML documents to enable dynamic validation of student input and adaptive actions based on their performance.

The system also includes an animated study agent called "genie" that provides emotional support and encouragement to students through voice animation and dialog boxes. This feature aims to create a positive learning environment and keep students engaged in the learning process.

In summary, BITS is an intelligent tutoring system designed to help students learn C++ programming by providing personalized, adaptive guidance based on their current knowledge and understanding. It employs Bayesian networks for modeling student knowledge and incorporates various adaptive features to facilitate effective learning while offering a user-friendly, web-based interface for accessing course materials and quizzes.

Our Assumptions and Limitations:

We have implemented a penalty system to get a probability from the API to be fed into the Bayesian network based on the response of the user.

adjusted_probability = probability_known - (syntax_error_penalty * num_syntax_errors + logical_error_penalty * num_logical_errors) * complexity_factor

In this formula:

probability_known: The initial probability of understanding a concept without considering any errors.
syntax_error_penalty: A penalty factor (between 0 and 1) for syntax errors. You could assign a lower value for syntax errors, as they typically have less impact on understanding programming concepts compared to logical errors.
num_syntax_errors: The number of syntax errors detected by ChatGPT.

logical_error_penalty: A penalty factor (between 0 and 1) for logical errors. You could assign a higher value for logical errors, as they usually have a more significant impact on understanding programming concepts compared to syntax errors.

num_logical_errors: The number of logical errors detected by ChatGPT.

complexity_factor: A factor (between 0 and 1) that represents the complexity of the code. A higher value indicates more complex code, which might be more likely to have errors. (will be based on the length of the code)

This assumes a linear relationship between users' knowledge and the number of errors that might be wrong.

Also, due to the crunch in time, we could not make a full-fledged web application like BITS and hence we have done the implementation for testing all probabilities ranging from 0 to 1, which might lead to some oversight.

Implementation

This section gives a breakdown of our approach to improving the system implemented for the Bayesian network in the research paper we mentioned.

```
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objs as go
import plotly.express as px
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
```

First, libraries like numpy, pandas, pgmpy, and matplotlib are imported to handle data manipulation, create and work with Bayesian models, and visualize the data. These libraries provide essential tools to build a personalized learning path system.

```
# Create the Bayesian Model with the defined edges
edges = [
    ('Introduction to Programming', 'Basic Syntax'),
    ('Basic Syntax', 'Data Types'),
    ('Data Types', 'Variables and Constants'),
    ('Variables and Constants', 'Operators'),
    ('Operators', 'Control Structures'),
    ('Control Structures', 'If-else statements'),
    ('Control Structures', 'Switch statements'),
    ('Operators', 'Loops'),
    ('Loops', 'For loop'),
```

```
    ('Loops', 'While loop'),
    ('Loops', 'Do-while loop'),
    ('Data Types', 'Functions'),
    ('Data Types', 'Arrays and Vectors'),
    ('Data Types', 'Pointers'),
    ('Data Types', 'Strings'),
    ('Data Types', 'File I/O'),
    ('Data Types', 'Object-Oriented Programming'),
    ('Object-Oriented Programming', 'Classes and Objects'),
    ('Object-Oriented Programming', 'Inheritance and Polymorphism'),
]
model = BayesianNetwork(edges)
```

Then the edges of a Bayesian Network model are defined, which represent the dependencies between various programming concepts. The network is designed to guide the learner through the content sequentially, ensuring that they have a sufficient understanding before proceeding to the next topic.

The ‘edges’ list contains tuples, each representing a directed edge (dependency) between two programming concepts(nodes). The first element in each tuple is the parent node (prerequisite topic), while the second element is the child node (dependent topic). The edges outline the flow of learning, with child nodes requiring knowledge of parent nodes.

This code snippet defines the edges of a Bayesian Network model, which represents the dependencies between various programming concepts. The network is designed to guide the learner through the content sequentially, ensuring they have a solid understanding of prerequisite topics before advancing to more complex material.

The edges list contains tuples, each representing a directed edge (dependency) between two programming concepts (nodes). The first element in each tuple is the parent node (prerequisite topic), while the second element is the child node (dependent topic). The edges outline the flow of learning, with child nodes requiring knowledge of their parent nodes.


Introduction to Programming
Basic Syntax
Data Types
Variable And Constants
Functions
Arrays and Vectors
Pointers
Strings
File I/O
Object Oriented Programming
Operators
Classes and Objects
Inheritance and Polymorphism
Control Structures
If - else
Switch
Loops
For Loop
While Loop
Do-while Loop

After defining the edges, the BayesianNetwork model is created using the BayesianNetwork() constructor, with the edges passed as an argument. This initializes the model with the specified dependencies, setting the stage for further customization, such as adding content, probabilities, and weights to the nodes.

```
cpd_intro_prog = TabularCPD(variable='Introduction to
Programming',variable_card=2,
                        values=[[0.5], [0.5]])
cpd_basic_syntax = TabularCPD(variable='Basic Syntax', variable_card=2,
                          values=[[0.8, 0.2], [0.2, 0.8]],
                          evidence=['Introduction to Programming'],
                          evidence_card=[2])
```

This code snippet defines the Conditional Probability Distributions (CPDs) for the node ‘Basic Syntax’ in the Bayesian Network model (all the nodes follow the parameterization. PFA code). CPDs quantify the probability of a node's state, given the states of its parent nodes. Each CPD is represented as a TabularCPD object from the pgmpy library.
For each node, the CPD is created with the following parameters:
variable: The name of the node.
variable_card: The number of possible states the variable can take. In this case, it's 2 for every node, representing "known" and "unknown" states.
values: A list of lists representing the probability distribution for the node, given its parents' states. The outer list corresponds to the node's states, and the inner lists contain probabilities for each combination of parent states.
evidence (optional): A list of the parent node(s) for the current node. This parameter is not needed for nodes without parents, like "Introduction to Programming".
evidence_card (optional): A list of the number of possible states for each parent node. This parameter is not needed for nodes without parents.
cpd_basic_syntax: The CPD for the "Basic Syntax" node depends on the "Introduction to Programming" node. It is represented by a 2x2 table, where each row corresponds to the "known" and "unknown" states of "Basic Syntax," and each column corresponds to the "known" and "unknown" states of "Introduction to Programming".
The remaining CPDs follow a similar structure, with each node's CPD depending on the states of its parent(s).

After defining the CPDs, they are added to the Bayesian Network model, allowing the model to compute joint probabilities and infer the states of nodes given observed evidence.
As the full implementation of the University of Regina's BITS was not available, we improvised where we could.
After a few prints written for debugging, we named the nodes and assigned them weights.

```
nodes_name = ['Introduction to Programming','Basic Syntax','Data
Types','Variables and Constants','Operators',
   'Control Structures','If-else statements','Switch statements','Loops','For
loop','While loop',
   'Do-while loop','Functions','Arrays and Vectors','Pointers','Strings','File
I/O','Object-Oriented Programming',
   'Classes and Objects','Inheritance and Polymorphism']
weight = [1,1,1.5,1.5,1,1.5,1.5,1.5,2,2,2,2,2,2.5,2.5,2.5,2,3,3,3]
```

weight: This is a list of numerical values that represent the weights associated with each node in the Bayesian Network. The weights are used to adjust the influence of the parent nodes on the child nodes when calculating the probability distribution. In the context of this problem, weights can be interpreted as the difficulty of each topic or the strength of the prerequisite relationship between topics. A higher weight means that the prerequisite topic has a stronger influence on the understanding of the current topic. Weights have been implemented in our BITS to better integrate domain knowledge as not topics have the same weightage.
These two lists have the same length, and each element in the weight list corresponds to the node with the same index in the nodes_name list. These weights will be used later in the code to update the CPDs of the Bayesian Network based on the learner's interactions with the content.

```
# Adding content to the nodes
for x in nodes_name:
   model._node[x]['content'] = f'Content for {x}'
# Adding initial probability to the nodes
for x in nodes_name:
   model._node[x]['probability_known'] = 0.3
```

```
# Adding weights to the nodes
for x,i in zip(nodes_name,weight):
    model._node[x]['weight'] = i
# Accessing and printing content of the nodes
for node in model.nodes():
    print(f"Node: {node}")
    print(f"Content: {model._node[node]['content']}\n")
```

This adds custom attributes to the nodes of the Bayesian Network, specifically 'content', 'probability_known', and 'weight'. It then prints the content of each node.
Adding content to the nodes: A for loop iterates through the nodes_name list. For each node, a 'content' attribute is added, containing a string that says "Content for {x}", where {x} is the name of the node. This content represents the learning materials associated with each programming concept.
Adding initial probability to the nodes: Another for loop iterates through the nodes_name list. For each node, a 'probability_known' attribute is added and initialized to 0.3. This attribute represents the learner's initial probability of knowing the corresponding topic. As the learner interacts with the content, these probabilities will be updated to better reflect their understanding.
Adding weights to the nodes: A for loop iterates through both the nodes_name and weight lists simultaneously using the zip() function. For each node, a 'weight' attribute is added and assigned the corresponding value from the weight list. These weights will be used later to adjust the influence of parent nodes on child nodes in the Bayesian Network.
Accessing and printing the content of the nodes: A for loop iterates through the nodes in the Bayesian Network using the model.nodes() method. For each node, it prints the name of the node and the content associated with it. This demonstrates how to access and display the custom attributes that we have added to the nodes.

```
def update_child_cpd(node, correct_answer):
    update_factor = model._node[node]['weight']
    if correct_answer:
        model._node[node]['probability_known'] += update_factor * 0.1
    else:
        model._node[node]['probability_known'] -= update_factor * 0.1
```

```
    model._node[node]['probability_known'] = max(0, min(1,
model._node[node]['probability_known']))
    cpd = TabularCPD(node, 2, [[1 - model._node[node]['probability_known']],
[model._node[node]['probability_known']]])
    model.add_cpds(cpd)
```

This function, update_child_cpd(node, correct_answer), is used to update the Conditional Probability Distribution (CPD) of a given child node based on the learner's performance (whether they answered a question correctly or not).
First, the function extracts the 'weight' attribute of the node using model._node[node]['weight'] and stores it in the variable update_factor. This weight is used to adjust how much the probability_known value should be updated.
The function checks whether the learner's answer (correct_answer) is correct or not. If it's correct, the probability_known attribute for the node is increased by update_factor * 0.1. If it's incorrect, the probability_known attribute is decreased by update_factor * 0.1. This updating mechanism adjusts the learner's probability of knowing a topic based on their performance.
The updated probability_known value is then clamped between 0 and 1 using max(0, min(1, model._node[node]['probability_known'])). This ensures that the probability stays within a valid range.
A new TabularCPD object is created for the child node with the updated probability_known value. The new CPD is defined with 2 states (unknown/known), and their respective probabilities are set as [1 - probability_known] and [probability_known].
Finally, the new CPD is added to the Bayesian Network model by calling model.add_cpds(cpd). This replaces the existing CPD for the node with the updated one, reflecting the learner's updated understanding of the topic.

```
def update_parent_cpd(node):
    child_nodes = model.get_children(node)
    total_weight = sum([model._node[child]['weight'] for child in child_nodes])

    updated_probability_known = 0
    for child in child_nodes:
        weight = model._node[child]['weight']
        updated_probability_known += (weight / total_weight) *
model._node[child]['probability_known']
```

```
    parent_weight = model._node[node]['weight']
    model._node[node]['probability_known'] = (parent_weight *
updated_probability_known + model._node[node]['probability_known']) /
(parent_weight + 1)
    cpd = TabularCPD(node, 2, [[1 - model._node[node]['probability_known']],
[model._node[node]['probability_known']]])
    model.add_cpds(cpd)
```

The update_parent_cpd(node) function is responsible for updating the Conditional Probability Distribution (CPD) of a given parent node based on the updated probability_known values of its child nodes.
The function first finds all the child nodes of the given parent node by calling model.get_children(node) and stores them in the variable child_nodes.
It calculates the total weight of all child nodes by summing up their individual weights using list comprehension and stores the result in the variable total_weight.
The function initializes a variable updated_probability_known with a value of 0. It will be used to store the weighted average probability of the child nodes.
Next, the function iterates through each child node, calculates the weighted probability of that child node (by dividing the child's weight by the total weight and multiplying it by the child's probability_known), and adds this weighted probability to the updated_probability_known variable.
The parent node's weight is then extracted using model._node[node]['weight'], and the parent node's probability_known value is updated. It is updated by taking a weighted average of the updated_probability_known (based on child nodes) and the current probability_known value, using the parent node's weight.
A new TabularCPD object is created for the parent node with the updated probability_known value. The new CPD is defined with 2 states (unknown/known), and their respective probabilities are set as [1 - probability_known] and [probability_known].
Finally, the new CPD is added to the Bayesian Network model by calling model.add_cpds(cpd). This replaces the existing CPD for the parent node with the updated one, reflecting the updated understanding of the parent topic based on the child nodes.

```
def update_cpd(node, correct_answer):
```

```
    update_child_cpd(node, correct_answer)
    parent_nodes = model.get_parents(node)
    for parent_node in parent_nodes:
        update_parent_cpd(parent_node)
```

The update_cpd(node, correct_answer) function is responsible for updating the Conditional Probability Distribution (CPD) of both a given node and its parent nodes, based on whether the user provided a correct answer for the current topic.
The function first calls update_child_cpd(node, correct_answer) to update the given node's CPD based on the user's correct answer. This function adjusts the probability_known of the node based on the user's performance and updates the node's CPD accordingly.
The function then retrieves the parent nodes of the given node by calling model.get_parents(node) and stores the result in the variable parent_nodes.
The function iterates through each parent node in parent_nodes and calls the update_parent_cpd(parent_node) function. This updates the parent node's CPD based on the updated probability_known values of their child nodes (including the current node that was just updated). It calculates the weighted average probability of the child nodes and updates the parent node's CPD accordingly.
By calling this function and providing a node and a correct_answer flag, the Bayesian Network model's understanding of both the specific node and its parent nodes is updated based on the user's performance on the current topic.

```
def can_access_node(node):
    """
    Checks if the user can access a node based on the satisfaction of its parent nodes.
    """
    parents = model.get_parents(node)
    if not parents:
        return True
    inference = VariableElimination(model)
    for parent in parents:
```

```
        probability_known = inference.query([parent]).values[1]
        if probability_known < 0.7:
            return False
    return True
# Simulate user's interaction with the system
nodes = model.nodes()
```

This code defines a function can_access_node(node) and simulates user interaction with the system.
can_access_node(node) function checks if the user can access a specific node (topic) in the Bayesian Network based on whether the prerequisites (parent nodes) have been satisfied or not.
The function first retrieves the parent nodes of the given node using model.get_parents(node). If there are no parent nodes, the function returns True, indicating that the node can be accessed.
An inference object is created using the VariableElimination(model) method from the pgmpy library. This object will be used to query the probabilities of the parent nodes.
The function iterates through each parent node in parents. For each parent, it queries the probability that the user knows the topic using inference.query([parent]).values[1]. The result is stored in the variable probability_known.
If the probability_known value of any parent is less than 0.7, the function returns False, indicating that the user cannot access the current node since the prerequisites are not satisfied.
If all parent nodes have a probability_known value greater than or equal to 0.7, the function returns True, indicating that the user can access the current node.
After defining the can_access_node(node) function, the code then simulates user interaction with the system by getting a list of all the nodes in the model using the model. nodes() and storing it in the variable nodes. This list of nodes will be used in the subsequent simulation of user interactions.

```
def simulate_user_interaction(node, prob_correct):
    if can_access_node(node):
        print(f"Node: {node}")
        print(f"Content: {model._node[node]['content']}\n")
```

```
        # Update the CPD of the node based on the probability of the user's answer being correct
        update_cpd(node, prob_correct)
    else:
        print(f"Cannot access node: {node} due to unsatisfied parent requirements.\n")
# Test with an example
simulate_user_interaction('Data Types', 0.785)
```

This code defines a function simulate_user_interaction(node, prob_correct) that simulates user interaction with the system for a given node (topic) and the probability of the user's answer being correct (prob_correct).
The function checks if the user can access the given node by calling the can_access_node(node) function. If the user can access the node, the code proceeds with the simulation.
The function prints the node name and its content using the print() statements with node and model._node[node]['content']. This simulates the user accessing the content of the topic.
The function then updates the Conditional Probability Distribution (CPD) of the node based on the user's interaction. This is done by calling the update_cpd(node, prob_correct) function, which takes the node name and the probability of the user's answer being correct as input arguments. This simulates the user's response to a question on the topic and updates the model accordingly.
If the user cannot access the given node (i.e., the can_access_node(node) function returns False), the function prints a message stating that the node cannot be accessed due to unsatisfied parent requirements. This simulates the scenario where the user cannot access a topic because they have not yet satisfied its prerequisites.
The code then tests the simulate_user_interaction() function with an example by simulating user interaction with the 'Data Types' node and providing a probability of the user's answer being correct as 0.785.

```
def generate_plot_data(child_node, parent_node):
    weights = np.linspace(0.5, 3, 10)
    probabilities = np.linspace(0, 1, 10)
    original_weight = model._node[parent_node]['weight']
```

```
original_probability_known = model._node[child_node]['probability_known']
final_probabilities = np.zeros((len(weights), len(probabilities)))
for i, weight in enumerate(weights):
    model._node[parent_node]['weight'] = weight
    for j, prob_correct in enumerate(probabilities):
        model._node[child_node]['probability_known'] = prob_correct
        update_cpd(child_node, prob_correct)
        final_probabilities[i, j] = model._node[parent_node]['probability_known']
model._node[parent_node]['weight'] = original_weight
model._node[child_node]['probability_known'] = original_probability_known
    return final_probabilities
```

This code defines a function generate_plot_data(child_node, parent_node) that generates data for plotting the relationship between the weights of parent and child nodes, and the probabilities of the child node being known.
weights and probabilities: These are two numpy arrays, created using numpy.linspace(), which generate evenly spaced values within the given range. weights contains 10 values in the range from 0.5 to 3, and probabilities contains 10 values in the range from 0 to 1.
original_weight and original_probability_known: These variables store the original weight and probability known values of the parent and child nodes, respectively. These values will be used later to restore the original state of the model.
final_probabilities: This is a 2D numpy array of zeros, with the same dimensions as weights and probabilities. It will be used to store the resulting probabilities of the parent node being known after updating the child node's probability known values.
Nested for loops: The outer loop iterates over the weights array, and the inner loop iterates over the probabilities array. For each combination of weight and probability, the code performs the following actions:
a. Update the parent node's weight with the current weight value from the outer loop.
b. Update the child node's probability known value with the current probability value from the inner loop.
c. Call the update_cpd(child_node, prob_correct) function to update the CPDs for the child node and its parent nodes based on the new probability value.
d. Store the updated probability known value of the parent node in the final_probabilities array.

After iterating through all combinations of weights and probabilities, the code restores the original weight and probability known values of the parent and child nodes using the stored original_weight and original_probability_known values. The function generate_plot_data() returns the final_probabilities 2D numpy array, which can be used to create a plot to visualize the relationship between the parent node's weight, the child node's probability known values, and the resulting probability known values of the parent node.

```
def plot_3d_scatter(child_node, parent_node):
    weights = np.linspace(0.5, 3, 10)
    probabilities = np.linspace(0, 1, 10)
    final_probabilities = generate_plot_data(child_node, parent_node)
    x_data = []
    y_data = []
    z_data = []
    for i, weight in enumerate(weights):
        for j, prob_correct in enumerate(probabilities):
            if weight==0.5:
                continue
            x_data.append(weight)
            y_data.append(prob_correct)
            z_data.append(final_probabilities[i, j])
    fig = px.scatter_3d(x=x_data, y=y_data, z=z_data, color=z_data, labels={'x':
'Weight of Parent Node', 'y': 'Probability of Correct Answer', 'z': 'Final Probability'},
title=f'3D Scatter Plot for {parent_node} (Parent of {child_node})')
    fig.show()
```

The plot_3d_scatter(child_node, parent_node) function creates a 3D scatter plot to visualize the relationship between the parent node's weight, the probability of the child node being known, and the final probability known of the parent node.
weights and probabilities: These are two numpy arrays, created using numpy.linspace(), which generate evenly spaced values within the given range. weights contain 10 values in the range from 0.5 to 3, and probabilities contain 10 values in the range from 0 to 1.
final_probabilities: This variable stores the 2D numpy array returned by the generate_plot_data(child_node, parent_node) function. It contains the final

probability known values of the parent node for each combination of weights and probabilities.
x_data, y_data, and z_data: These are lists that store the x, y, and z coordinates for the 3D scatter plot. x_data represents the parent node's weights, y_data represents the probability of the child node being known, and z_data represents the final probability of known values of the parent node.
Nested for loops: The outer loop iterates over the weights array, and the inner loop iterates over the probabilities array. The code skips the first weight value (0.5) to avoid a singular matrix in the plot. For each combination of weight and probability, the code appends the corresponding values to the x_data, y_data, and z_data lists.
fig: The plotly.express.scatter_3d() function is used to create a 3D scatter plot using the x_data, y_data, and z_data lists. The color parameter is set to z_data, which colors the points based on the final probability values. The labels parameter sets the axis labels, and the title parameter sets the plot title.
fig.show(): This line displays the 3D scatter plot in the output.
The plot_3d_scatter() function provides a visual representation of how the weight of a parent node and the probability of a child node being known affect the final probability known value of the parent node.

```
def process_node(node):
    children_nodes = list(model.successors(node))
    if not children_nodes:
        return

    parent_nodes = model.predecessors(node)
    for parent_node in parent_nodes:
        plot_3d_scatter(node, parent_node)

    for child_node in children_nodes:
        process_node(child_node)
root_node = 'Introduction to Programming'
process_node(root_node)
```

The process_node(node) function is a recursive function that processes each node in the Bayesian Network, creating 3D scatter plots for all parent-child node pairs.

children_nodes: This list contains the children of the input node. The model.successors(node) function returns an iterator over the children, and the list() function converts the iterator to a list.
if not children_nodes:: If the node has no children (i.e., it is a leaf node), the function returns and the recursion stops for this branch.
parent_nodes: This list contains the parents of the input node. The model.predecessors(node) function returns an iterator over the parents.
for parent_node in parent_nodes:: This loop iterates over the parent nodes of the input node. For each parent, the plot_3d_scatter(node, parent_node) function is called to generate and display a 3D scatter plot for the parent-child node pair.
for child_node in children_nodes:: This loop iterates over the children nodes of the input node. For each child, the process_node(child_node) function is called recursively. This allows the function to process the entire Bayesian Network tree structure, generating 3D scatter plots for all parent-child node pairs.
root_node = 'Introduction to Programming': This line defines the root node of the Bayesian Network.
process_node(root_node): This line initiates the recursive processing of nodes in the Bayesian Network by calling the process_node() function with the root node as the input.
By calling the process_node(root_node), the function processes the entire Bayesian Network tree and generates 3D scatter plots for all parent-child node pairs. This helps in visualizing the relationships between parent and child nodes in the network.

Exhaustive Simulation using the Simulating_user_interaction Function and plotting Heatmap

```
def exhaustive_simulation(weight_range, probability_range,
num_simulations):
    weights = np.linspace(*weight_range, num_simulations)
    probabilities = np.linspace(*probability_range, num_simulations)
    average_probability_changes = np.zeros((len(weights),
len(probabilities)))

    for node in model.nodes():
        model._node[node]['original_probability_known'] =
model._node[node]['probability_known']

    for i, w in enumerate(weights):
        for j, p in enumerate(probabilities):
            for node in model.nodes():
                model._node[node]['probability_known'] = p
                model._node[node]['weight'] = w

            total_change = 0

            for _ in range(num_simulations):
                selected_node = np.random.choice(model.nodes())
                prob_correct = np.random.rand()
                initial_probability_known =
model._node[selected_node]['probability_known']

                # Temporarily store the original CPDs
                original_cpds = model.get_cpds()

                # Update the CPDs without modifying the network structure
                update_cpd(selected_node, prob_correct)

                # Revert the CPDs to their original state
                model.remove_cpds(*model.cpds)
                model.add_cpds(*original_cpds)

                final_probability_known =
model._node[selected_node]['probability_known']
```

```
                total_change += abs(final_probability_known -
initial_probability_known)

            average_probability_changes[i, j] = total_change /
(num_simulations * len(model.nodes()))

            # Reset the probabilities for the next iteration
            for node in model.nodes():
                model._node[node]['probability_known'] =
model._node[node]['original_probability_known']

    return average_probability_changes, weights, probabilities

weight_range = (0.5, 3)
probability_range = (0, 1)
num_simulations = 10

average_probability_changes, weights, probabilities =
exhaustive_simulation(weight_range, probability_range, num_simulations)
```

The exhaustive_simulation function conducts an exhaustive simulation of the Bayesian network model by iterating over different combinations of initial probabilities and weights. **The goal is to analyze the average change in probability across all nodes under varying conditions**.

Explanation of the function:
The function takes three arguments: weight_range, probability_range, and num_simulations. The first two arguments define the range of weights and initial probabilities to test, and the third argument defines the number of simulation iterations.
The weights and probabilities are generated using the numpy function linspace, which creates evenly spaced values between the given ranges.
The function initializes a 2D array called average_probability_changes to store the average change in probability for each combination of weight and initial probability.

The function iterates through the Bayesian network's nodes and saves the current probability_known values as original_probability_known so they can be restored later.
The function then iterates over each combination of weights and probabilities. For each combination, it sets the probability_known and weight attributes for all nodes in the model.
For each weight-probability combination, the function runs a specified number of simulations (num_simulations). In each simulation, it randomly selects a node, generates a random probability for the user's correct answer, and stores the current probability_known value as initial_probability_known.
The function temporarily saves the current CPDs, updates the CPDs based on the simulated user's response (without modifying the network structure), and then reverts the CPDs to their original state.
The difference between the final and initial probabilities is added to the total_change variable. After all simulations for a specific combination are complete, the average change in probability for this combination is calculated and stored in the average_probability_changes array.
The probability_known values are reset to their original values (original_probability_known) for the next iteration.
The function returns the average_probability_changes array along with the generated weights and probabilities.

This function helps analyze the impact of different initial probabilities and weights on the average change in probability across all nodes in the Bayesian network.

Plotting the values obtained from this function

```
# Create the heatmap
fig, ax = plt.subplots(figsize=(10, 7))
sns.heatmap(average_probability_changes, annot=True, cmap="YlGnBu",
xticklabels=np.round(probabilities, 2), yticklabels=np.round(weights, 2),
ax=ax)
ax.set_xlabel("Initial Probability")
ax.set_ylabel("Weight")
ax.set_title("Average Change in Probability Across All Nodes")
plt.show()
```

This code snippet creates a heatmap using the average_probability_changes array obtained from the exhaustive_simulation function. A heatmap is a data visualization technique that uses colors to represent numerical values in a grid. In this case, the heatmap displays the average change in probability across all nodes in the Bayesian network for different combinations of initial probabilities and weights.

Here's a step-by-step explanation of the code:

The plt.subplots function is called to create a new figure and a single subplot (ax) with a specified size of 10x7 inches.

The sns.heatmap function from the seaborn library is used to create the heatmap. It takes the following arguments:

average_probability_changes: The 2D array containing the average change in probability for each combination of weight and initial probability.
annot=True: This option enables the display of the numerical values inside each cell of the heatmap.
cmap="YlGnBu": This sets the color map to use for the heatmap. In this case, it's a yellow-to-green-to-blue color map.
xticklabels and yticklabels: These arguments set the labels for the x-axis and y-axis ticks. They are set to the rounded values of probabilities and weights, respectively.

ax=ax: This argument specifies the subplot (ax) on which the heatmap will be drawn.

The set_xlabel, set_ylabel, and set_title methods of the ax object are used to set the x-axis label, y-axis label, and title of the heatmap, respectively.

Finally, the plt.show() function is called to display the heatmap.

The resulting heatmap visualizes the average change in probability across all nodes in the Bayesian network under various combinations of initial probabilities and weights. This helps to understand the impact of different initial conditions on the network's behavior.

Need for plotting the Heatmap and why the 3D plots plotted earlier are not sufficient to visualize the effect of weights on the whole network:

While both the 3D scatter plots and the heatmap provide insights into the behavior of the Bayesian network under varying conditions, they serve different purposes and offer unique perspectives on the data. Here's how both the visualizations are justified :

3D scatter plots: These plots focus on the relationship between a child node's probability, the weight of its parent node, and the resulting probability of the parent node after updating the child node's probability. They help visualize the impact of these factors on individual parent-child node pairs in the Bayesian network.

Heatmap: The heatmap offers a broader perspective by aggregating the average change in probability across all nodes in the network for different combinations of initial probabilities and weights. This visualization will enable us to identify patterns and trends that may not be apparent from looking at individual parent-child relationships. The heatmap provides an overall view of how the network behaves and responds to varying conditions, which can help you identify areas of sensitivity or stability.

In summary, the 3D scatter plots and heatmap complement each other by providing different levels of detail and perspectives on the network's behavior.

The 3D scatter plots focus on individual node relationships, while the heatmap gives a comprehensive view of the network's behavior under various conditions. Using both visualizations together helps to gain a deeper understanding of the network's dynamics and can guide further improvements or adjustments to the network structure or parameters.

Results and Analysis

We are using **Chatgpt as our error detection tool**. It will function as a bridge between the user's code and the result shown to the user corresponding to their input. ChatGpt will have the correct code for the problem and the code inputted by the user and then it will tell the error, and error type in the user's code.

As we did not have any dataset with enforced errors and the correct code, we made a small dataset independently. We classified the coding problems as **easy, medium, and hard** depending on the complexity of the code. The dataset consists of the correct code, code with enforced errors, errors, and error types listed by ChatGpt. The error types are classified only as logical and syntax errors. Further, to calculate the efficiency of ChatGpt for error detection we use the concept of Accuracy.

Accuracy: Accuracy is the ratio of the number of correctly classified errors to the total number of errors. The formula for accuracy is: Accuracy = (True Positive + True Negative) / (True Positive + False Positive + True Negative + False Negative)

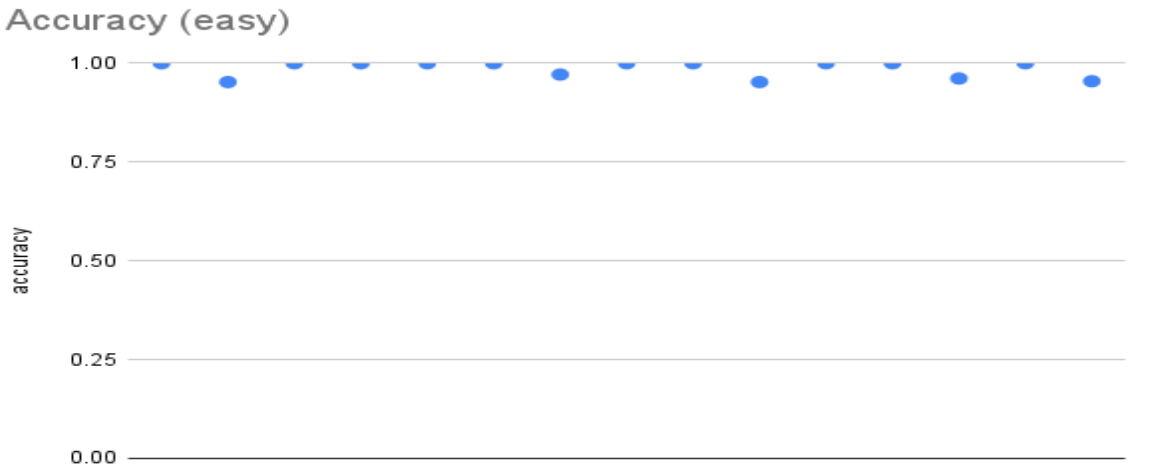
Where, True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) are terms used to describe the results of a binary classification problem. In the context of error detection, a binary classification problem refers to the task of classifying whether an error is present or not.

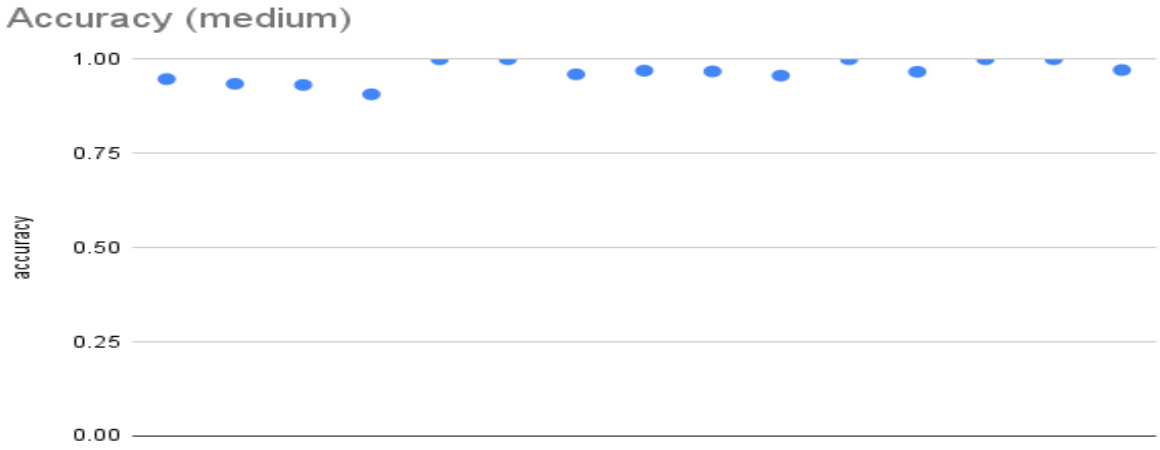
1. True Positive (TP): An error is correctly classified as present (positive).
2. True Negative (TN): An error is correctly classified as not present (negative).
3. False Positive (FP): An error is incorrectly classified as present (positive) when it is actually not present (negative).
4. False Negative (FN): An error is incorrectly classified as not present (negative) when it is actually present (positive).

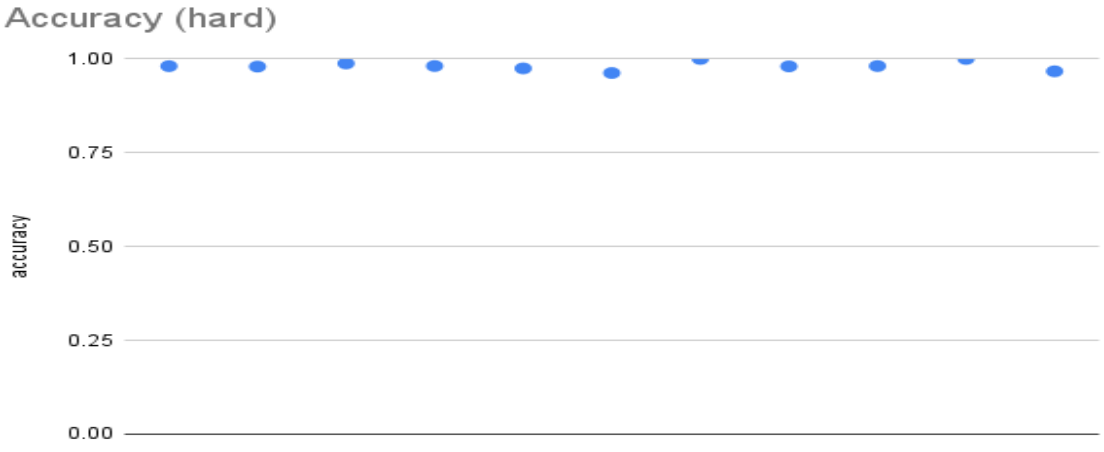
In other words, true positive and true negative represent correct classifications, while false positive and false negative represent incorrect classifications.

The scatter plot shown below shows that our Chatgpt model has an **accuracy of greater than 0.9** for the testing of the dataset for all 3 classifications (easy, medium, hard) and it is recommendable for integration in our model for Intelligent Tutoring System(ITS) as an error detecting tool.

Link to our dataset: dataset







Result for the Probability curves and their analysis:

The significance of the 3D scatter plots generated by the plot_3d_scatter() function lies in the visualization of the **relationship between the weight of a parent node, the probability of a correct answer for the child node, and the final probability of knowing the parent node** in the Bayesian Network model. These plots provide insights into how the personalized learning path system adapts to the learner's understanding and performance on related topics.

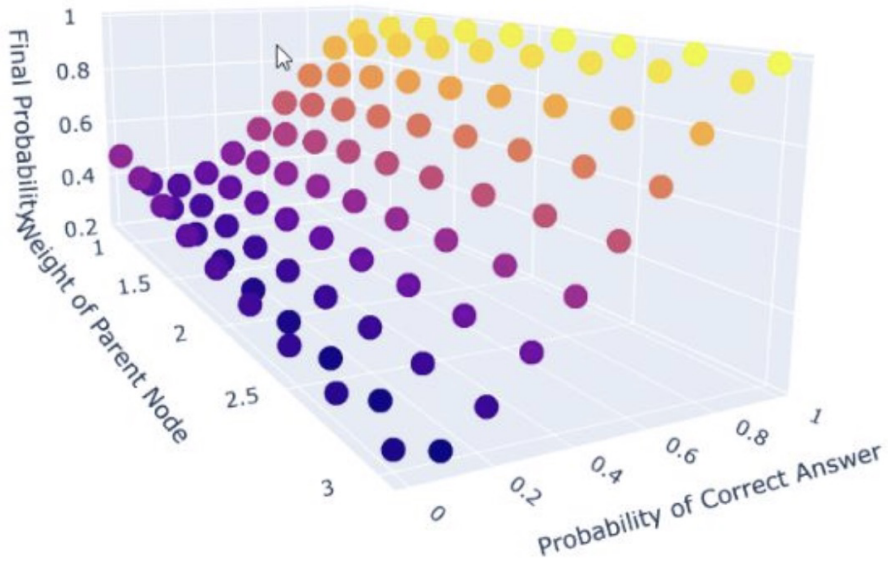
The shape of the surface obtained from the generate_plot_data() function, which computes the final probability of knowing the parent node for different combinations of weights and probabilities, is based on the principles of Bayesian Networks.

As the weight of the parent node increases, its influence on the child node becomes stronger, meaning that the learner's understanding of the parent node has a more significant effect on their understanding of the child node. This causes the surface to rise in the direction of increasing parent node weight.

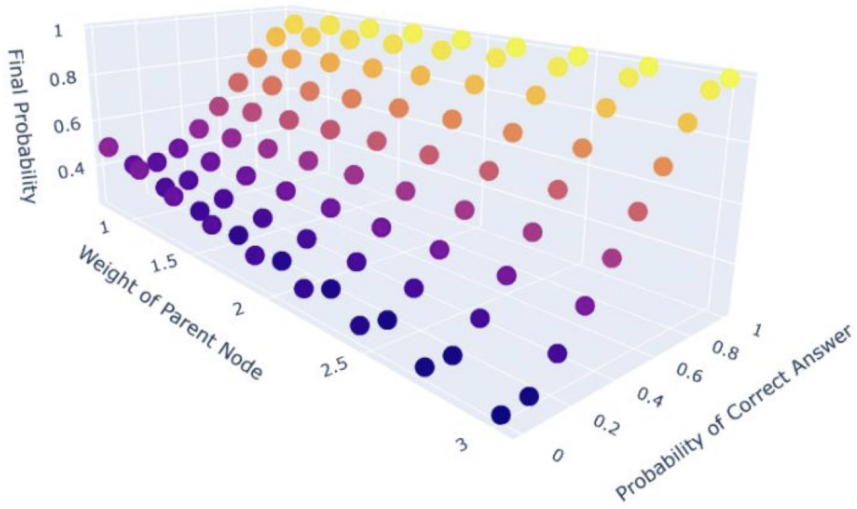
Similarly, as the probability of a correct answer for the child node increases, the likelihood of the learner having a good understanding of the child node also increases. This causes the surface to rise in the direction of increasing the probability of a correct answer for the child node.

Taking these two factors into account, we can predict that the surface will have a sloping shape, with the lowest values in the corner representing low parent node weights and low probabilities of correct answers, and the highest values in the corner representing high parent node weights and high probabilities of correct answers. This surface shape visually demonstrates the adaptive nature of the Bayesian Network model in the personalized learning path system, as it shows how the final probability of knowing a parent node (prerequisite topic) changes based on the learner's understanding and performance on related topics.

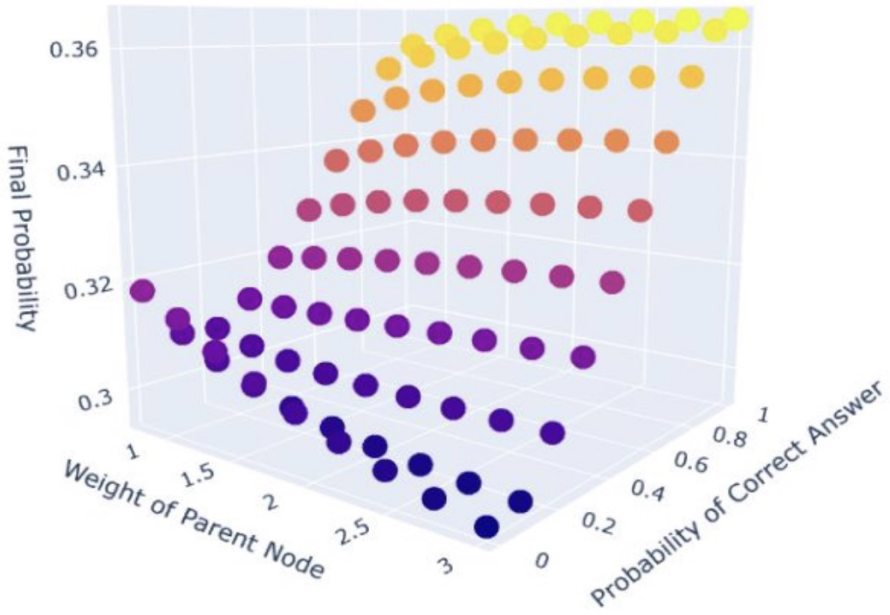
3D Scatter Plot for Introduction to Programming (Parent of Basic Syntax)



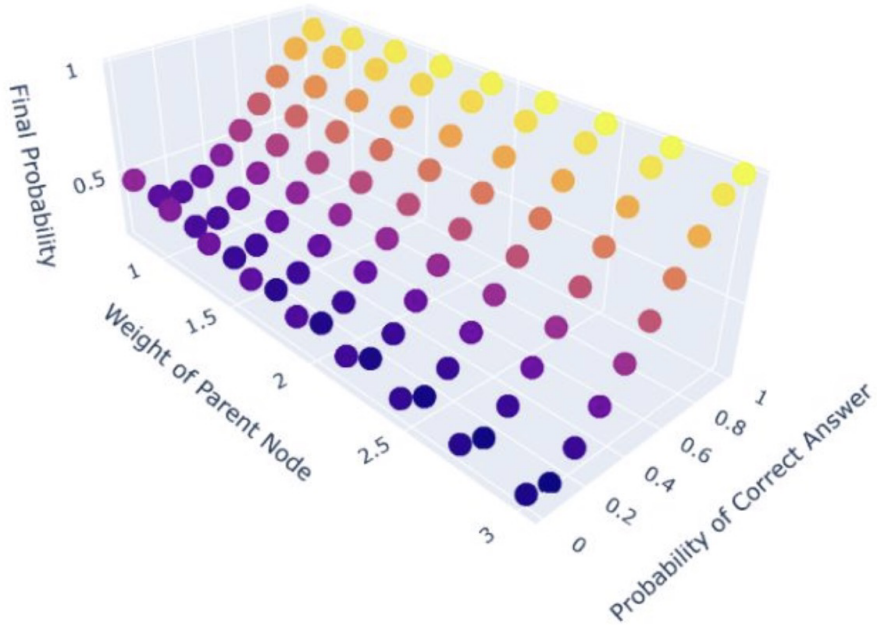
A 3D Scatter Plot for Basic Syntax (Parent of Data Types)



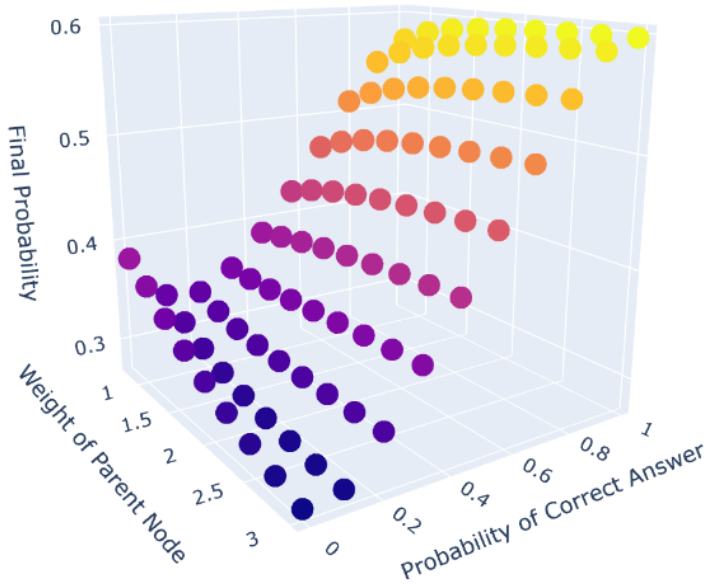
3D Scatter Plot for Data Types (Parent of Variables and Constants)



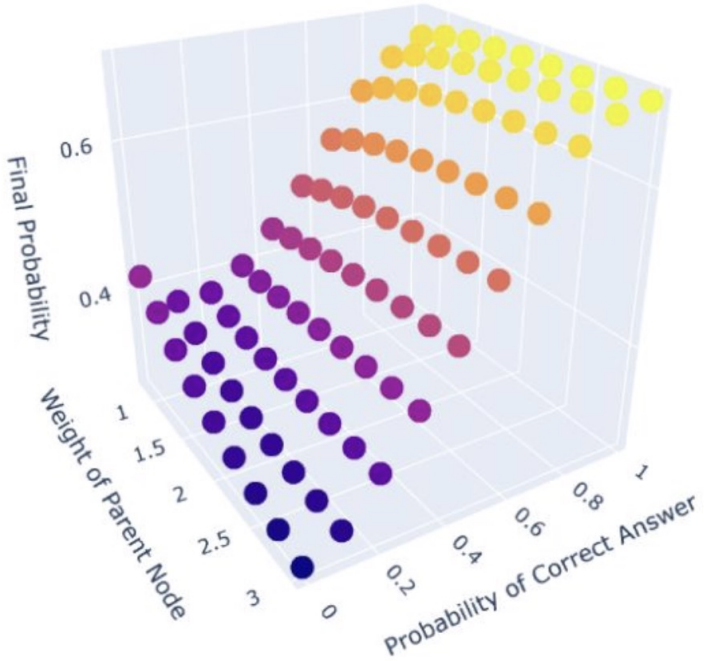
3D Scatter Plot for Variables and Constants (Parent of Operators)



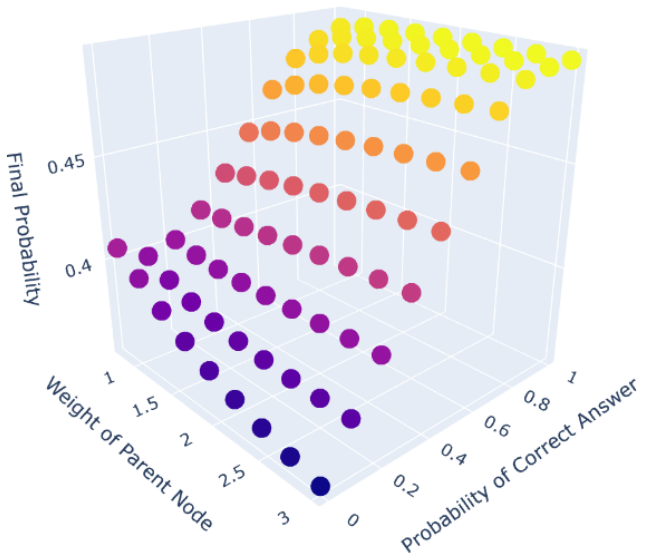
3D Scatter Plot for Operators (Parent of Control Structures)



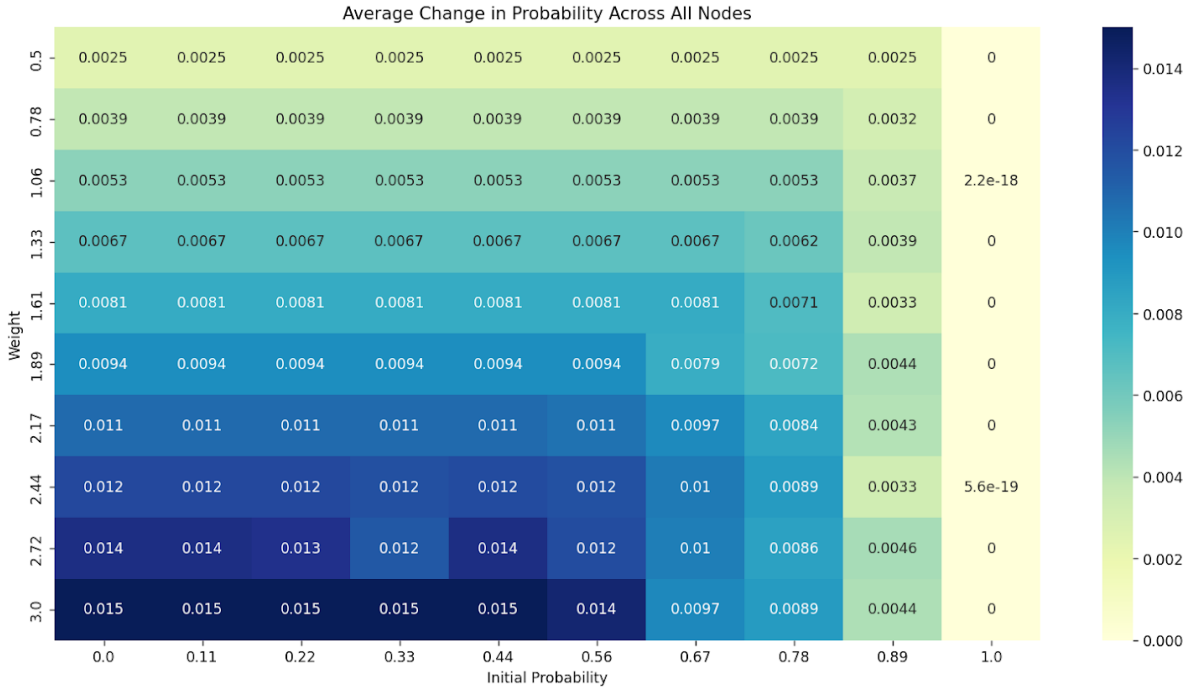
3D Scatter Plot for Operators (Parent of Loops)



3D Scatter Plot for Data Types (Parent of Object-Oriented Programming)



Heatmap and its explanation



Nature of the Heatmap

The heatmap displays the average change in probability across all nodes with respect to the initial probability and weight settings. The x-axis corresponds to the initial probability, and the y-axis corresponds to the weight.

High initial probability: The heatmap should show lower average changes in probability towards the right side (higher initial probabilities). With a higher initial probability, users are more likely to answer questions correctly, resulting in smaller changes in their probability of understanding the concept.

Low initial probability: Conversely, the heatmap should show higher average changes in probability towards the left side (lower initial probabilities). With a lower initial probability, users are more likely to answer questions incorrectly, leading to larger changes in their probability of understanding the concept.

Low weight: The heatmap should show lower average changes in probability towards the top side (lower weights). Lower weight implies that a node has less impact on the overall understanding of the parent concepts. As a result, the changes in the probability of understanding the concept might be less pronounced.

High weight: The heatmap should show higher average changes in probability towards the bottom side (higher weights). Higher weight means a node has a more significant impact on the overall understanding of the parent concepts, resulting in more substantial changes in the probability of understanding the concept.

Conclusion

From this whole study, we can conclude the following:
Network Structure: The Bayesian Network model can effectively represent the knowledge structure of a tutorial or learning system, with nodes representing topics and edges representing dependencies between them. The model can also incorporate additional information like weight and probability of understanding a topic, which can help in simulating user interactions and analyzing the network's behavior.

Content Access: The study demonstrates the ability to control access to certain nodes based on the user's understanding of the parent nodes. This ensures that the user has a solid understanding of prerequisite topics before moving on to more advanced ones.

Adaptive Learning: The model can be adapted based on user interactions, updating the probabilities of understanding for each node as the user interacts with the system. This allows the learning system to adjust its content and recommendations to better suit the individual needs of each user.

Parameter Sensitivity: The study investigates the sensitivity of the network to changes in the initial probability and weight values. By simulating user interactions and visualizing the average change in probability across different combinations of initial probabilities and weights, we can better understand how these parameters affect the network's behavior.

Visualization & Analysis: The study showcases various methods for visualizing and analyzing the network, such as 3D scatter plots and a heat map. These visualizations help in understanding the relationships between different parameters in the network and can provide insights for improving the network's performance.

In summary, this study demonstrates the potential of using Bayesian Networks for modeling and simulating adaptive learning systems. The analysis of the network's behavior and sensitivity to different parameters can provide valuable

insights for designing more effective learning systems tailored to individual users' needs.

References

-A Web-based Intelligent Tutoring System for Computer Programming C.J. Butz, S. Hua, R.B. Maguire Department of Computer Science

-J. Beck, M. Stern, and E. Haugsjaa, “Applications of AI in education,” ACM Crossroads, pp. 11-15, 1996.

-B. Bloom, “The 2 sigma problem: The search for methods of group instruction as effective as one on one tutoring,” Educational Researcher, 13(6): pp. 4- 16, 1984.

-https://www.codezclub.com/cpp-solved-programs-problems-solutions/ for datasets

- https://openai.com/blog/chatgpt

- https://www.programiz.com/cpp-programming/examples