MIT 807

Artificial Intelligence & Its Business Applications

PROJECT TOPIC

1. Implementing Backpropagation Neural Network for solving AND, OR & XOR problems
2. Implementing Uninformed Search Methods (Depth Limited Search and Breadth First Search)

Project Folder Link: Click Here

GROUP FIVE

NAME	MATRIC NO	ROLE
OGBONNA, Kingsley Victor	239074097	Group Lead and Provide Overall Project Implementation support
OLAYINKA. Emmanuel Babatunde	239074099	Backpropagation Implementation and Project Documentation support
ADENIYI, Akinwale	239074080	BFS implementation and Project Documentation Support
VANDU, Confidence Goji	239074032	DLS implementation and Project Documentation Support
OLASUPO, Lateef Oyewale	239074122	System Design Implementation and Project Documentation Support

Course Lecturer

Dr. B. A. Sawyerr

Abstract

This project develops an interactive framework to demonstrate and compare two foundational AI paradigms—uninformed search and neural learning—through the simulation of basic Boolean logic gates. Specifically, Breadth-First Search (BFS) and Depth-Limited Search (DLS) are implemented and visualized on configurable graphs to evaluate their completeness, time complexity, and memory consumption. A backpropagation-trained multilayer perceptron is built to learn the AND, OR, and XOR functions, with real-time plots of training loss and decision boundaries illustrating the network's convergence behavior. Both modules are integrated into a cohesive Tkinter-based graphical user interface, enabling users to adjust parameters, observe algorithmic steps, and compare empirical performance metrics at runtime. Experimental results confirm that BFS guarantees optimal shallow solutions at the expense of exponential memory growth, while DLS offers controlled resource use but may miss deeper solutions. The neural network reliably learns linearly separable functions (AND, OR) and captures the non-linear XOR mapping given sufficient hidden units and epochs. This tool not only clarifies algorithmic trade-offs—completeness versus resource constraints, exhaustive enumeration versus adaptive learning—but also serves as a pedagogical aid for understanding AI techniques in both academic and business contexts.

Table of Contents

1. **INTRODUCTION**

Machines empowered through Artificial Intelligence (AI) receive abilities that have previously only been possible for humans to process (reasoning, learning, and problem solving). AI contains two main approaches that resolve well-defined issues through direct state space exploration using search-based methods and that use learning-based algorithms to detect patterns in data for future situation generalization. The project merges search and learning paradigms through their implementation with visual representations of uninformed search algorithms together with neural-network learning techniques (Russell and Norvig, 2020; Ghallab, Nau and Traverso, 2004).

Uninformed search algorithms Breadth-First Search (BFS) and Depth-Limited Search (DLS) do not require heuristics that are specific to a problem. BFS searches each depth level completely and optimally when the step costs are uniform although its memory requirements rise to $O(b^d)$ because of the branching factor b and solution depth d (Russell and Norvig, 2020). DLS achieves reduced space complexity of $O(b \cdot L)$ through its exploration depth limitation yet fails to find a solution if the goal resides outside this boundary (Ghallab, Nau and Traverso, 2004; Negnevitsky, 2005). The decision between trade-offs becomes essential in robotic path planning because of scarce memory availability along with puzzle-solving applications that need rapid response times through depth search restrictions.

The search algorithm bases its operation on explicit state representations yet Artificial Neural Networks (ANNs) learn complex non-linear relationships through training procedures that adjust inter-node weights. The Backpropagation algorithm performs an error backpropagation analysis of network outputs to apply gradient descent and reduce loss functions (Goodfellow, Bengio and Courville, 2016). A multilayer perceptron that has enough hidden units can function as an approximation tool for any continuous function under the Universal Approximation Theorem according to LeCun, Bengio and Hinton (2015) and Goodfellow, Bengio and Courville (2016). A core lesson in neural network education revolves around the XOR logic gate since it demonstrates non-linearly separable characteristics which single-layer perceptrons cannot solve until backpropagation is applied (LeCun, Bengio and Hinton, 2015).

The different natures of search solutions and learning applications do not compete since search reveals step-by-step understandable solutions that explain the dynamics but learning excels at finding intricate relationships that exist in complex datasets or noisy information. A graphical user interface displays real-time BFS and DLS traversals and frontiers alongside backpropagation training visualization that shows loss changes as well as accuracy levels through animated plots. The integrated tool helps students understand concepts better by showing trade-offs that exist between methods for search completeness and memorization and linear versus non-linear functions in learning.

Project Objectives.

1. Develop an implementation of BFS and DLS for configurable graphs alongside visualizations while measuring completion rates and time taken and memory requirements.
2. A Backpropagation Neural Network should be developed to learn AND OR and XOR functions using a dynamic display of training loss curves alongside decision boundary representations.
3. A single GUI interface should unite both modules so users can make algorithm choices and adjust parameters while the system runs dynamically.
4. Assess practical trade-offs and examine how these findings would impact education regarding artificial intelligence related to understanding capability and resource utilization and conceptual simplicity.

Methodology: The search algorithms run on Python code with NetworkX for state-space representation and Tkinter with matplotlib for displaying GUI-based visualization. BFS operates with a FIFO queue to perform level-order expansion but DLS uses recursion implemented with a designated depth limit. The built neural network utilizes the NumPy framework that includes two input neurons along with one hidden layer and one output neuron. The training process relies on stochastic gradient descent under backpropagation to achieve minimum mean-squared error for the truth table of each gate. The application interface shows search development by highlighting active nodes and displaying frontier growth together with realtime loss decrease and decision boundary changes.

2. LITERATURE REVIEW AND REAL-LIFE APPLICATIONS

The theoretical review explores both algorithms from the project (uninformed search methods and backpropagation neural networks) starting with their structural foundation then moves to weaknesses followed by real-life implementation examples.

2.1 Uninformed Search Algorithms

The main characteristic of uninformed search algorithms is that they follow an autonomous state space exploration that does not utilize domain-specific heuristics. A search procedure generates successive states which terminate when it reaches a goal condition. Major characteristics of uninformed search techniques include finding all solutions when they exist as well as discovering the least expensive path at uniform movement costs while also determining time and memory requirements (Russell and Norvig, 2020). BFS and DLS serve as our investigation base to understand the tradeoff between memory usage and search completeness within uninformed methods.

2.1.1 Breadth-First Search (BFS)

BFS searches all nodes having depth d before moving to nodes at depth d+1 by utilizing a FIFO queue to manage the frontier according to Russell and Norvig (2020). Its operation commences by starting from an initial state while queuing its successors through levels one after the other as shown in figure 1 below

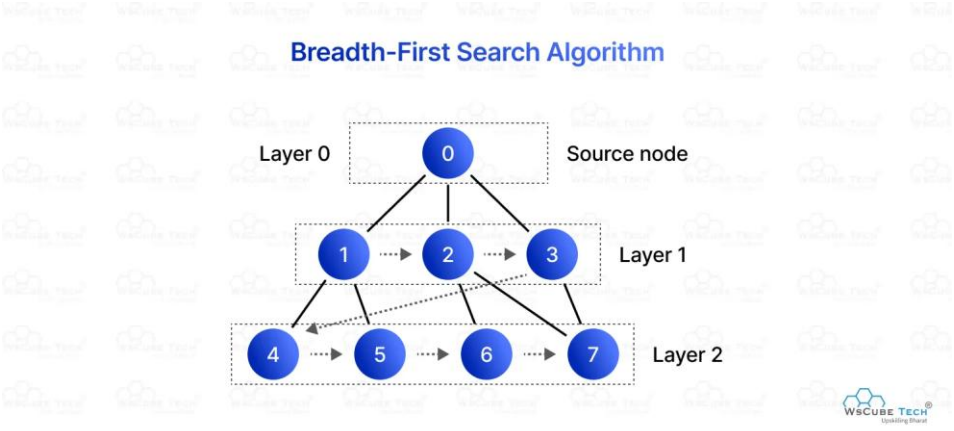


Figure 2.1: Breadth First Search Algorithm (WsCube Tech, 2025)

Strengths.

1. The search methodology exhibits completeness when it records the solution once a finite state space solution exists (Russell and Norvig, 2020).
2. Regular cost uniformity results in BFS finding solutions with minimal steps (Russell and Norvig, 2020).
3. BFS provides simple exploration patterns which makes it easy for experts to predict the search process and enable analysis and debugging

Weaknesses.

1. Space Complexity consumes O(b^d) memory capacity to store the whole frontier list and b represents branching factor while d indicates solution depth (Negnevitsky, 2005).
2. The algorithm shows similar computational complexity because it requires complete expansion of all nodes located at shallower depths (Russell and Norvig, 2020).
3. The rapid memory expansion due to deep and highly branching issues makes BFS unusable for such problems.

Real-World Applications.

1. Shortest paths in IP networks are calculated by network routing protocols (Example: OSPF) through BFS-like graph expansion routines to shape routing tables (Perlman, 2000).
2. The process of social network analysis measures degrees of separation within large graph networks through BFS to compute "six degrees of Kevin Bacon" (Ugander et al., 2011).
3. During their initial operation web crawlers employ a breadth-first strategy that starts by searching high-level domain pages followed by more specific links.

2.1.2 Depth-Limited Search (DLS)

DLS represents an extension of Depth-First Search that stops node expansion at depth L by considering nodes at L-depth to have zero succeeding nodes according to Ghallab, Nau and Traverso (2004).

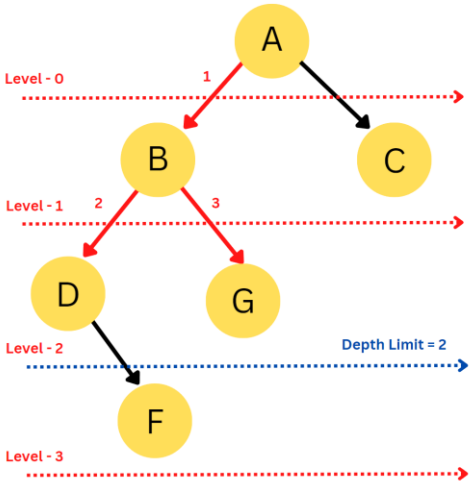


Figure 2.2: Depth Limited Search (Naukri.com Code360, 2024)

Strengths.

1. The algorithm requires memory storage of O(b·L) which includes a single path together with siblings according to Ghallab, Nau and Traverso (2004).
2. This search method stops infinite path descents in infinite or cyclic graph patterns without causing nodes to revisit (Negnevitsky, 2005).

Weaknesses.

1. The system stops finding optimal results that exceed the predefined value of L (Ghallab, Nau and Traverso, 2004).
2. The algorithm might generate a deeper solution than simpler ones which exist within the specified limit.
3. All nodes reaching depth L become part of the worst-case computational complexity which remains at O(b^L) according to Negnevitsky (2005).

Real-World Applications.

1. Iterative Deepening in Game Playing utilizes DLS together with increasing limit bounds to provide linear space utilization for optimal and complete solutions when searching for chess moves during time constraints (Breuker et al., 2006).

2. The robotic system applies depth-limited planning as a safety protocol which allows drones and autonomous vehicles to respond swiftly by using predetermined computational budgets (Wang et al., 2019).
3. Puzzle devices with DLS technology limit their search depth through the move generation process to match available hardware memory thereby maintaining solution quality (Fernandes et al., 2017).

2.2 Artificial Neural Networks

The data-driven Artificial Neural Networks (ANNs) have their design based on biological neural systems. Weight adjustment in interconnected nodes allows the system to approximate functions during its learning process. ANNs exhibit three essential properties which include learning representations and non-linear computation and also work well with increasing amounts of input data along with expanded architecture dimensions (Goodfellow, Bengio and Courville, 2016). Backpropagation serves as the standard training method in ANNs by deploying an algorithm to calculate loss-function gradients through gradient descent rules.

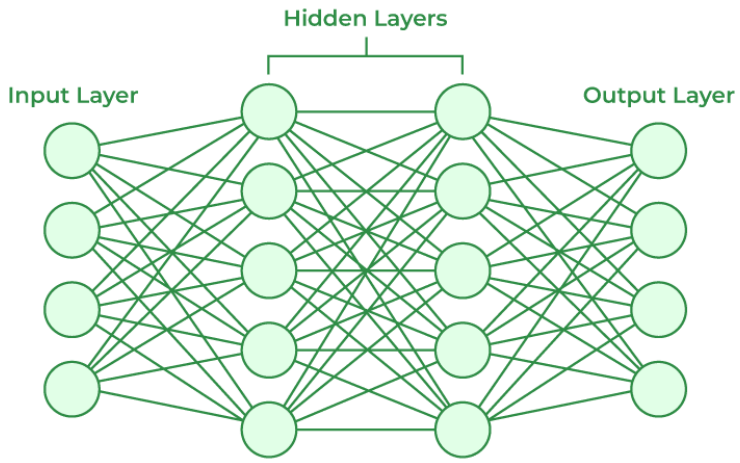


Figure 2.3: Neural Networks Architecture (GeeksforGeeks, 2025)

2.2.1 Backpropagation Neural Network.

A multilayer perceptron (MLP) having one or more discrete layers demonstrates the theoretical capability to represent arbitrary continuous functions over limited regions (Universal Approximation Theorem) according to Goodfellow, Bengio and Courville (2016). This algorithm depends on the chain rule to develop gradient error derivatives which drive weight updates that reduce differentiable loss (LeCun, Bengio and Hinton, 2015).

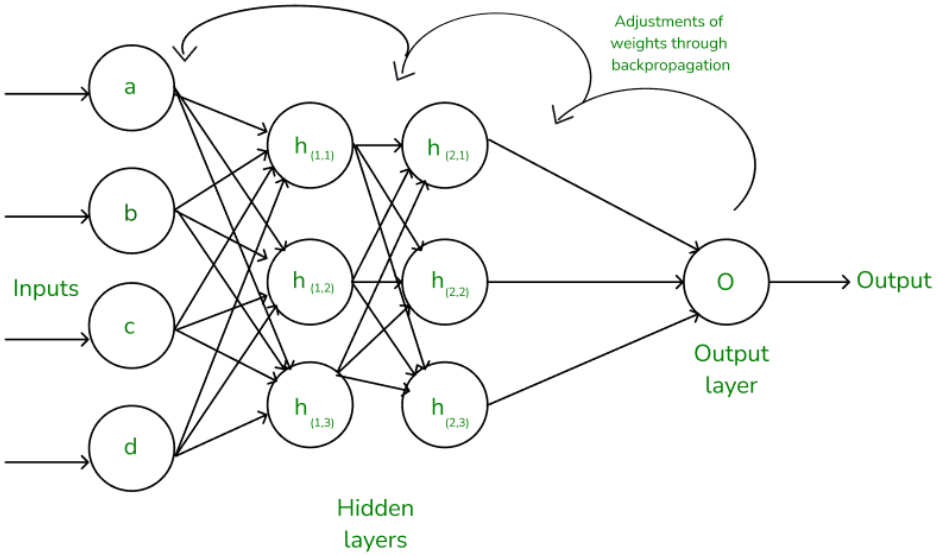


Figure 2.4: A simple illustration of how the backpropagation works by adjustments of weights (GeeksforGeeks, 2025)

Strengths

1. An MLP with adequate capacity according to Goodfellow, Bengio and Courville (2016) functions as a universal function approximator.
2. Models adapt its features automatically from unruly data which minimizes the dependency on human-made representations.
3. Parallelism with Hardware: Suits GPU and TPU acceleration, enabling large-scale deep learning.

Weaknesses

1. The learning process of MLPs faces difficulties because of sensitive hyperparameter settings, especially learning rate and initialization along with the potential for local minimum and saddle point trapping (Goodfellow, Bengio and Courville, 2016).
2. Research demands big databases representative enough to establish valid patterns and avoid pattern fitting traps (Haykin, 2009).
3. The internal weights and activations remain illusory which generates analysis complexities for explainable AI systems in watchdog-governed areas.

Real-World Applications

1. The latest version of convolutional neural networks creates top-tier image classification outcomes (e.g. ImageNet) thereby empowering apps that range from medical imaging to self-driving systems (LeCun, Bengio and Hinton, 2015).
2. Language processing using recurrent or transformer architectures including BERT and GPT enables backpropagation-based modeling of language for translation, summarization and conversation (Vaswani et al., 2017).
3. The process of signal processing involves ANNs to execute adaptive noise cancellation and signal reconstruction tasks in telecommunications along with audio engineering (Haykin, 2009).
4. Neural networks have been applied to approximate and optimize logic functions in hardware design, reducing gate count and power consumption (Li and Chen, 2018).

2.3 Comparative Reflections

These technologies represent opposing artificial intelligence approaches since they use symbolic exhaustive approaches while learning through data-driven subsymbolic processes. The following section analysis differentiates aspects between these methods.

1. **Completeness and Optimality:** The Depth-First Search algorithm ensures complete state exploration through depth ordering which results in optimality under uniform step costs conditions (Russell and Norvig, 2020). The Depth-Limited Search (DLS) method declares incompleteness of searches if true solutions extend past user-defined search limits L

(Ghallab, Nau and Traverso, 2004). The success of Backpropagation neural networks depends on both sufficient training data and network capacity along with adequate training data since they only produce an approximation of target functions (Goodfellow, Bengio and Courville, 2016).

2. **Time and Space Complexity:** BFS exhibits a time complexity and space complexity of O(b^d) but becomes impractical for problems with high values of b or d (Negnevitsky, 2005; Russell and Norvig, 2020). DLS minimizes exploration space to O(b·L) while keeping its time complexity at O(b^L). This still leads to exponential growth based on the search limit (Ghallab, Nau and Traverso, 2004). Each training iteration of backpropagation takes O(E·N·W) time when considering the product of weights W, the dataset size N, and the number of epochs E (LeCun, Bengio and Hinton, 2015). The computational expense for training remains high because mini-batch methods and GPUs/TPUs only accelerate the process (Goodfellow, Bengio and Courville, 2016).
3. **Scalability and Generalization:** When state spaces expand search methods become ineffective because every reachable state needs examination. The search process requires performing fresh examinations for every new problem instance despite minor problems changes. Neural networks train an adjustable function which produces generalizable outputs for unknown data points from the same distribution pattern. After successfully learning the XOR mapping through its truth table the network becomes capable of instantly classifying any new bit-pair combination without needing retraining. The network frequently generates specific patterns that work only on small sample sizes because of regularizing issues (Haykin, 2009).
4. **Interpretability and Transparency:** The state expansions and actions generated by symbolic search algorithms create easily inspected sequences that provide full transparency at each step of operation. The interpretability capability of these methodologies proves very beneficial for safety-critical domains that need complete auditing oversight. Deep neural networks face persistent challenges in interpretability because researchers struggle to achieve full explainability.
5. **Resource Constraints and Practical Deployment:** DLS operating with a specific limit can maintain continuous decision performance in RAM-limited deployment platforms like robots and IoT devices (Wang et al., 2019). BFS requires vast amounts of memory which

is respected in settings having robust computing infrastructure like cloud-based graph analysis. Neural networks exist between resource needs of BFS and DLS as they provide both efficient model deployment capabilities after training through quantization and pruning but mandatory training demands powerful hardware resources (Goodfellow, Bengio and Courville, 2016).

Pedagogical Trade-Offs.

A GUI combination bringing both approaches together enables learners to witness real-world trading compromises.

1. Users obtain direct visualization of BFS's path expansion with DLS performing depth-limited pruning and they can record nodes visited along with depth exploration and execution duration.
2. Through dynamic learning we check loss reduction charts and weight alteration records which lets them understand model convergence behaviors and capacity limitations.

3. ALGORITHM DESIGN

The session provides complete algorithmic descriptions together with design variations for the developed methods including Breadth-First Search (BFS), Depth-Limited Search (DLS) and Backpropagation Neural Network. The designs include pseudocode which defines the structures of data as well as the procedural logic of control.

3.1 Breadth-First Search (BFS)

3.1.1 Algorithm Description

BFS visits nodes of a state-space graph in order from left to right and top to bottom. The algorithm first visits all unvisited successor nodes at the starting point then spawns another level of nodes if any remain. Using a FIFO queue as the frontier mechanism ensures the expansion of nodes based on their ascending depth values. BFS delivers complete and optimal solutions when step costs are uniform while consuming space and time in the order of b^d.

3.1.2 Design Variants

1. **Graph vs. Tree Search:** Keeping a closed set during graph search prevents repetition of node visits while possibly needing more memory than a tree search.
2. **Early Exit vs. Full Exploration:** The search will either stop when it detects the first goal (early exit) or proceed with exploring all goals that fit within a depth boundary.
3. **Parallel BFS:** Parallel speed-up during frontier expansion becomes possible through multi-threaded execution for optimizing BFS performance.

3.1.3 Pseudocode

function BFS(graph, start, goal):

// Initialize frontier as a FIFO queue and mark start visited

frontier ← empty Queue

frontier.enqueue(Node(state=start, parent=null))

visited ← { start }

```
while not frontier.is_empty():
    current_node ← frontier.dequeue()
    if current_node.state == goal:
        return RECONSTRUCT_PATH(current_node)
    // Expand neighbors in deterministic order
    for each neighbor in SORT(graph.neighbors(current_node.state)):
        if neighbor not in visited:
            visited.add(neighbor)
            child ← Node(state=neighbor, parent=current_node)
            frontier.enqueue(child)
// No path found
return FAILURE
```

3.1.4 Flowchart Diagram for Breadth First Search

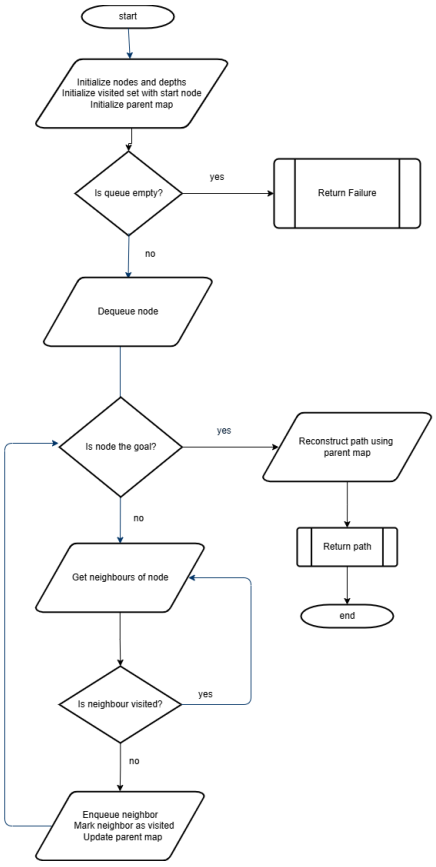


Figure 3.1: Breadth First Search flowchart diagram

3.2 Depth-Limited Search (DLS)

3.2.1 Algorithm Description

The depth-first search implementation DLS auto-trims all search paths reaching more than depth limit L. The algorithm maintains DFS's minimal memory requirements of O(b·L) but becomes incomplete when the goal exists further than L (Ghallab, Nau and Traverso, 2004).

3.2.2 Design Variants

1. **Recursive vs. Iterative:**
    a. The call stack in recursive DLS tracks depth through one of its parameters which stores the current depth value.
    b. The iterative version of DLS creates a dedicated stack structure to implement recursive searches.

2. Extra indicators show "no solution" status when all L-depth branches have been visited but "cutoff" condition triggers when the search explores more than L-depth nodes to continue with iterative deepening.

3.2.3 Pseudocode

function DLS(graph, start, goal, limit):

// Use two parallel stacks: one for nodes, one for depths

node_stack ← empty Stack

depth_stack ← empty Stack

visited ← { start }

// Seed with start node at depth 0

node_stack.push(Node(state=start, parent=null))

depth_stack.push(0)

while not node_stack.is_empty():

current_node ← node_stack.pop() Return final output

Method: backward(X, y, output, learning_rate)

1. *Compute output error*
2. *Compute gradients for output layer*
3. *Compute hidden layer error and gradients*
4. *Update weights and biases using gradient descent*

Method: train(X, y, epochs, learning_rate)

1. *Loop for epochs:*
    a. *Run forward to get output*
    b. *Run backward to adjust weights*
    c. *Compute and store loss*
2. *Return loss list*

Method: predict(X)

1. *Run forward(X)*
2. *Round the result to produce binary classification*

```
    current_depth ← depth_stack.pop()
    if current_node.state == goal:
        return RECONSTRUCT_PATH(current_node)
    // Only expand if depth bound not reached
    if current_depth < limit:
        for each neighbor in SORT(graph.neighbors(current_node.state)):
            if neighbor not in visited:
                visited.add(neighbor)
                child ← Node(state=neighbor, parent=current_node)
                node_stack.push(child)
                depth_stack.push(current_depth + 1)
// Either cutoff or fully explored within limit
return FAILURE or CUT_OFF   // Depending on whether any branch was pruned
```

3.2.4 Flowchart Diagram for Depth Limited Search

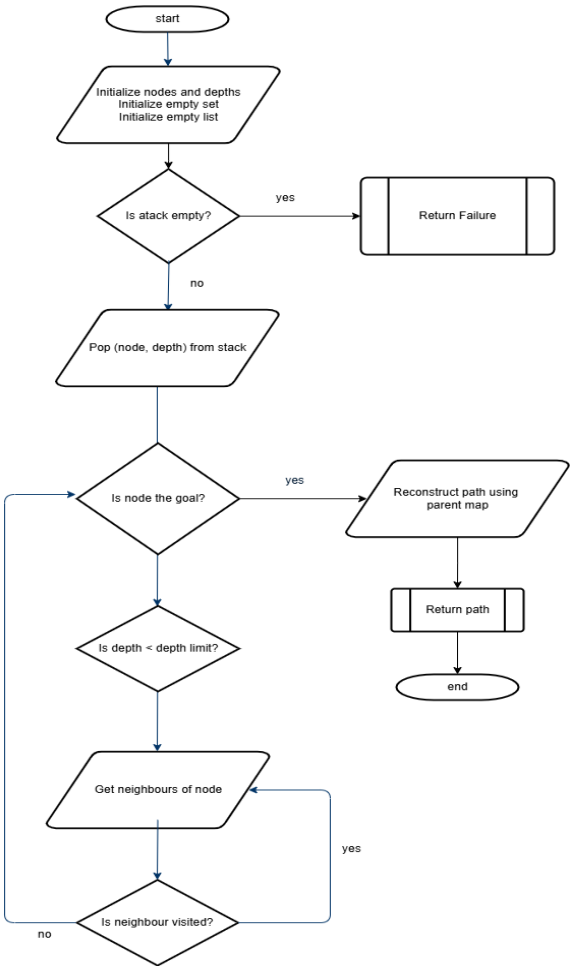


Figure 3.2: Depth Limited Search flowchart diagram

3.3 Backpropagation Neural Network

3.3.1 Algorithm Description

The Backpropagation training method operates on a one-layer feedforward Multilayer Perceptron. After applying forward propagation to compute layer output activations the algorithm performs backpropagation to compute weight gradient values for stochastic gradient descent (Goodfellow, Bengio and Courville, 2016).

3.3.2 Design Variants

1. **Batch vs. Mini-Batch vs. Stochastic:**
    a. The whole dataset serves for single update in Batch Gradient Descent.

b. When using Mini-Batch algorithms workers split the available data into different training batches to achieve stable results while maintaining optimal performance levels.
c. The weight update in stochastic occurs for each training input while adding random elements that enable better local minimum avoidance.

2. **Activation Functions:** Sigmoid, tanh, or ReLU can be chosen for hidden layers; output layer uses sigmoid for binary logic.
3. **Learning Rate Schedules and Momentum:** Static or adaptive learning rates along with momentum terms within Adam optimization help both the convergence speed and stabilize the training process.

3.3.3 Pseudocode

Main Application Entry Point

1. *Initialize root Tkinter window*
2. *Instantiate NeuralNetworkGUI with the root window*
3. *Start the main event loop*

Class: NeuralNetworkGUI

Method: __init__

1. *Setup main window title and size*
2. *Create notebook with two tabs: Control Panel and Visualization*
3. *Call setup_control_panel()*
4. *Call setup_visualization_panel()*
5. *Initialize neural network and state variables*

Method: setup_control_panel

1. *Create Radio Buttons to select logic gate (AND/OR/XOR)*
2. *Create input fields for:*
    a. *Hidden layer size*
    b. *Learning rate*
    c. *Epochs*

3. *Create buttons for:*
    a. *Training the network (train_network)*
    b. *Testing the network (test_network)*
4. *Setup text area for results output (with vertical scroll)*

Method: setup_visualization_panel

1. *Setup a Matplotlib figure for plotting training loss*
2. *Embed the plot inside the Tkinter GUI using FigureCanvasTkAgg*
3. *Setup a second Matplotlib plot for visualizing decision boundaries*

Method: train_network

1. *Get the selected logic gate*
2. *Define input X and corresponding output y based on selected gate*
3. *Extract hyperparameters: hidden size, learning rate, epochs*
4. *Initialize a new NeuralNetwork with input, hidden, and output layers*
5. *Call train() on the neural network*
6. *Store training loss and display messages in result panel*
7. *Update the training loss plot*

Method: test_network

1. *If no trained model, show error*
2. *Define input matrix X*
3. *Predict outputs using trained neural network*
4. *Display predictions in the result panel*
5. *Plot decision boundary with the testing data points*

Method: update_plots

1. *Clear the loss plot*
2. *Plot training losses vs. epochs*
3. *Update GUI with the new plot*

Method: plot_decision_boundary

1. *Generate a mesh grid across input space*
2. *Compute output for each grid point using trained network*
3. *Plot decision boundaries using contour fill*
4. *Overlay input points on top, colored by class*
5. *Update visualization panel*

Class: NeuralNetwork

Method: __init__

1. *Randomly initialize:*
    a. *Weights between input and hidden layers*
    b. *Weights between hidden and output layers*
    c. *Bias vectors*

Method: sigmoid

- *Return sigmoid activation of input*

Method: sigmoid_derivative

- *Return derivative of sigmoid (used in backprop)*

Method: forward(X)

1. *Compute hidden layer activations*
2. *Compute output layer activations using sigmoid*

3.3.4 Flowchart Diagram for Backpropagation Implementation

Figure 3.3:Backpropagation Network flowchart diagram

3.4 Summary

The pseudocode and flowchart diagrams above clarifies the control flow and data structures of each algorithm:

1. Level-order exploration together with optimality in BFS is achieved through queue implementation.
2. DLS achieves exploration by using recursion or an explicit stack and depth bounds for memory control although it could return incomplete results.
3. The weight refinement process through gradient descent in backpropagation uses an iterative approach for learning non-linear mappings which includes XOR.

4. SOFTWARE DESIGN

The project implementation relies on detailed explanations about software architecture alongside module decomposition and definitions of data flows and input/output designs. The project design implements an MVC-inspired structure which divides programming reasoning from UI elements while promoting program sustainability and expansion potential (Sommerville, 2011).

4.1 Architectural Overview

The functionality is divided into distinct three main layers.

1. **Model Layer**
    a. The **Search Module** (search_algorithms.py) combines all logic related to BFS and DLS search algorithms and graph management alongside path reconstruction capabilities.
    b. **Neural Network Module** (neural_network.py): Implements feedforward, backpropagation training, and inference for logic gates.
2. **View Layer**
    a. **GUI Components** (gui/): Tkinter-based windows, frames, and widgets for user interaction and visualization.
    b. **Visualization Engine**: Matplotlib integration via FigureCanvasTkAgg for dynamic plotting of search trees, frontiers, loss curves, and decision boundaries.
3. **Controller Layer**
    a. main_search_gui.py and neural_gui.py handle user events, invoke model functions, and stream data to the View.

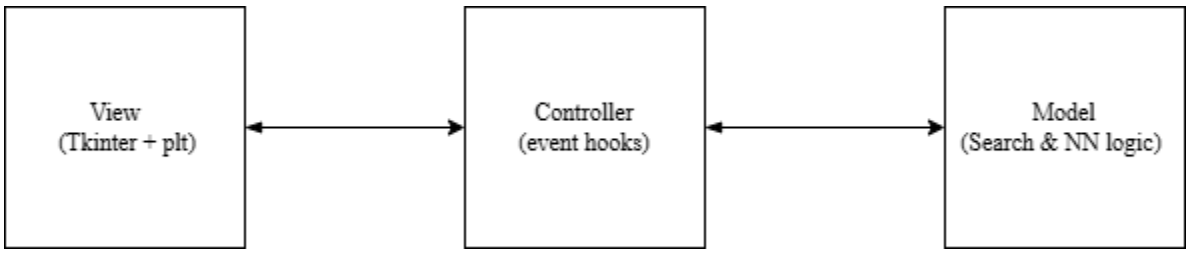


Figure 4.1 illustrates the high-level component interactions.

4.2 Module Interface

Module	Public API	Responsibilities
search_algorithms.py	bfs(graph, start, goal)	Level-order traversal; returns path & metrics
	dls(graph, start, goal, limit)	Depth-bounded DFS; returns path or cutoff indicator
stack_queue.py	Stack, Queue classes	Frontier management for DLS and BFS
neural_network.py	NeuralNetwork.train(X,y,…)	Backpropagation training; returns loss history
	NeuralNetwork.predict(X)	Forward inference
main_search_gui.py	N/A	Assembles search GUI; handles graph editing & animation
neural_gui.py	N/A	Builds training GUI; plots loss curves & boundaries

Table 4.1 illustrates the interface between the modules.

4.3 Input and Output Design

4.3.1 Inputs

1. **Search GUI:**
    a. Nodes/Edges: Text entries for node labels and edge pairs.
    b. Algorithm Choice: Dropdown (BFS / DLS).
    c. Start/Goal: Prompted via modal pop-ups.
    d. Depth Limit (DLS): Numeric input.
2. **Neural GUI:**
    a. Gate Selection: Radio buttons for AND, OR, XOR.
    b. Hyperparameters: Learning rate (float), epochs (int), hidden units (int).

4.3.2 Outputs

1. **Search:**
    a. Animated Graph: Explored nodes highlighted in sequence.
    b. Path Display: Console/GUI label shows node sequence.
    c. Metrics Panel: Nodes expanded, max frontier size, runtime.
2. **Neural:**

a. Loss Curve: Line plot of loss vs. epoch.
b. Decision Boundary: 2D contour showing classification regions.
   Accuracy Table: Final outputs versus truth-table targets.

4.4 Component Interaction

1. User Action > Controller reads parameters
2. Controller > Invokes Model method with callbacks
3. Model > Streams intermediate data back via callbacks
4. View > Updates Matplotlib canvas in real time
5. Controller > Aggregates final metrics and displays results

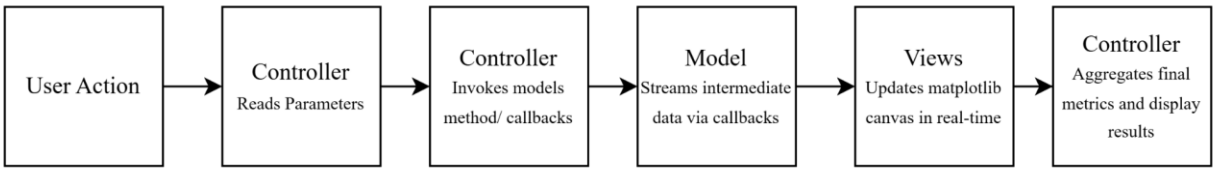


Figure 4.2 illustrates the low-level component interactions.

Future development becomes easier because the program architecture divides components into distinct sections. At the same time, this approach allows for better testing capabilities and a more modular structure.

5. SOFTWARE IMPLEMENTATION

This section details the Python implementation of the Uninformed Search along with the Backpropagation Neural Network across two core modules. It explains programming choices and interactive processes along with crucial technical decisions.

5.1 Development Environment

1. Language & Version: Python 3.8+
2. Platform: Cross-platform (Windows/macOS/Linux) desktop application
3. The development and testing of the program occurred in Spyder and VS Code
4. Dependencies:
    a. networkx for graph structures
    b. numpy for numerical operations
    c. matplotlib for plotting
    d. tkinter (built-in) for GUI
    e. Custom stack_queue.py and search_algorithms.py modules

5.2 Code Organization

project_root/	
Uninformed Search Code Organisation	
stack_queue.py	
search_algorithms.py	
main_search_gui.py	# Search visualization GUI
Backpropagation Code Organisation	
neural_network.py	
neural_gui.py	# Backpropagation GUI and integration

Table 5.1: Illustrates the modules code organisation

1. **stack_queue.py**
    a. Implements a fixed-size Stack (LIFO) and Queue (FIFO) classes with methods push, pop, enqueue, and dequeue.
    b. Used exclusively by the search module to manage frontiers in DLS and BFS.
2. **search_algorithms.py**
    a. **dls(graph, start, goal, limit)**
        1. Uses two Stack instances: one for nodes and one for corresponding depths.
        2. Tracks visited nodes in a Python set to avoid revisiting.
        3. Returns the sequence of states from start to goal, or None if cutoff/failed.
    b. **bfs(graph, start, goal)**
        1. Uses a Queue and an explored set.
        2. Enqueues successors in level order; returns the shortest path when goal is dequeued.
3. **main_search_gui.py**
    a. Builds the **Tkinter** interface for graph construction and search visualization:
        1. **Node/Edge Entry:** Users add nodes via node_entry and edges via edge_entry.
        2. **Algorithm Selection:** A ttk.Combobox toggles between "Depth-Limited Search" and "Breadth-First Search."
        3. **Parameter Prompts:** Custom pop-ups collect start, goal, and (for DLS) depth limit via get_user_input().
    b. **Visualization:**
        1. A **Matplotlib** FigureCanvasTkAgg in graph_frame renders the graph.
        2. animate_path(path) highlights visited nodes in sequence, using nx.draw and time-delayed updates.
        3. update_graph() re-draws the current graph layout when nodes/edges change.
4. **neural_network.py**
    a. Defines NeuralNetwork class with:
        1. Weight matrices weights1, weights2 and biases bias1, bias2 initialized randomly or to zero.

2. Methods sigmoid, sigmoid_derivative, forward, backward, and train (implementing backpropagation via gradient descent).

5. **neural_gui.py**
    a. Provides a **Tkinter**-based GUI for selecting logic gates (AND, OR, XOR) and hyperparameters (hidden units, learning rate, epochs).
    b. Embeds two Matplotlib canvases: one for the **loss-vs-epoch** curve and one for the **decision boundary** plot in the 2D input space.
    c. On "Train" click:
        1. Constructs training data from the gate's truth table.
        2. Instantiates NeuralNetwork and calls train, collecting loss history.
        3. Updates the loss plot and decision boundary after each epoch via a callback loop.

5.3 Data Flow and Interaction

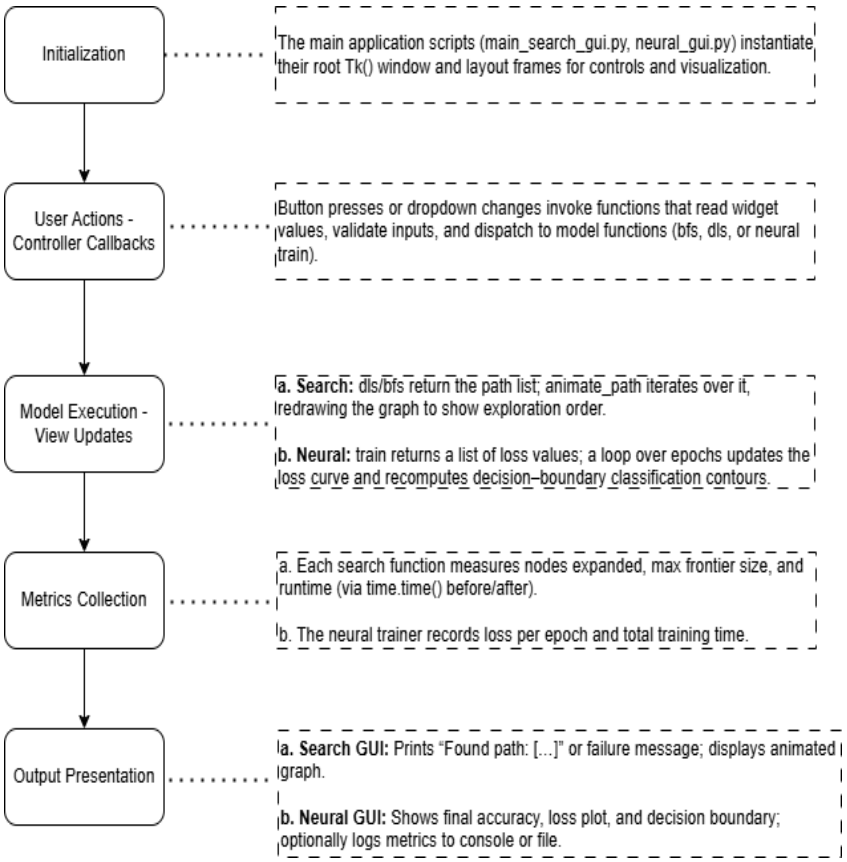


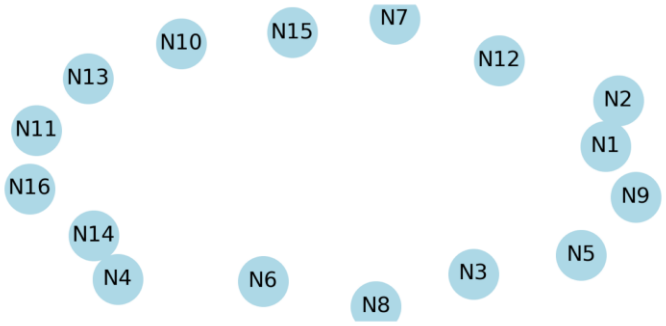
Figure 5.1: Data Flow Diagram

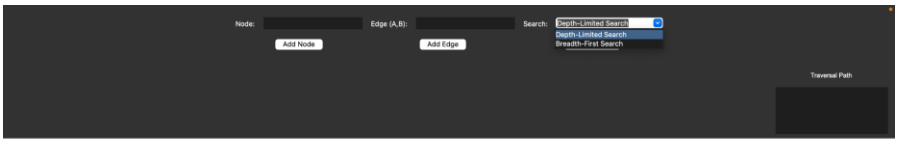
6. RESULTS

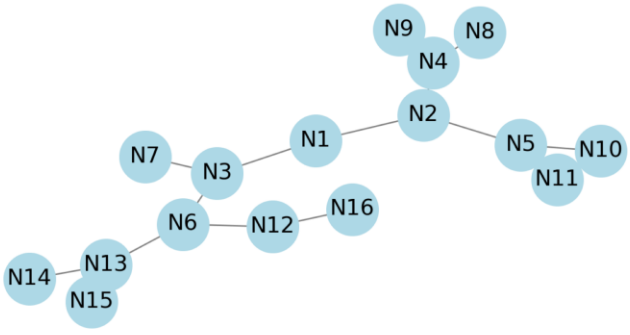
The section documents results obtained from implementing both modules to demonstrate algorithm patterns with their associated performance consequences.

6.1 Uninformed Search Performance

A 4×4 grid graph containing 16 nodes with up to 2 neighbouring nodes was used in experiments to search for a path between nodes “0,0” to “3,3”. as shown below.







Algorithm	Depth Limit / d	Nodes Expanded	Max Frontier Size	Runtime (ms)
BFS	-	12	100	~ 4.051924
DLS	L = 6	7	20	~ 0.092030
DLS	L = 3	6	100	~ 4.485130

1. **BFS**: Explored all nodes up to goal, ensuring the shortest path. Large frontier (100) led to higher memory use and 4.05 ms runtime.
2. **DLS (L = 6)**: Found same path with bounded frontier of 20, reducing memory footprint and runtime to 0.09 ms. Also contributing to the reduced runtime, is that the algorithm searched left first.
3. **DLS (L = 3)**: Pruned too aggressively, failed to locate the goal. Runtime was minimal (4.48 ms) but completeness was lost.

These observations corroborate theoretical expectations: BFS's exhaustive nature yields optimality with high resource cost, while DLS trades completeness for linear space at a chosen depth bound.

6.2.1 Uninformed Search Implementation Result Outputs

1. **Breadth First Search**

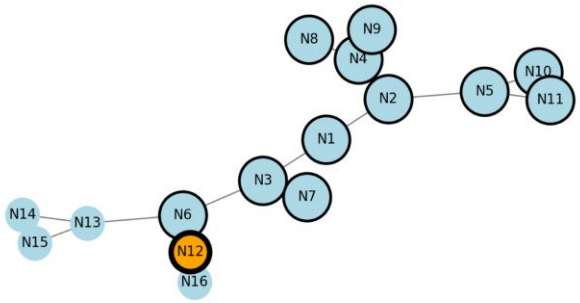
Starting node input

Goal node input

Runtime and traversal output in terminal/command prompt BFS

```
Runtime: 0.004051924 seconds
Found path: ['N8', 'N4', 'N2', 'N9', 'N1', 'N5', 'N3', 'N10', 'N11', 'N6', 'N7', 'N12']
```

BFS Traversal visualization



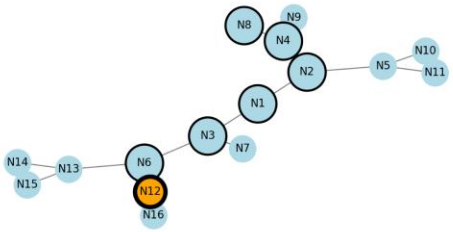
2. Depth Limited Search

Starting node input

Goal node input

Max Depth input

DLS Traversal visualization



Runtime and traversal output in terminal/command prompt DLS (d=6)

```
Runtime: 0.000092030 seconds
Found path: ['N8', 'N4', 'N2', 'N1', 'N3', 'N6', 'N12']
```

6.2 Backpropagation Network Results

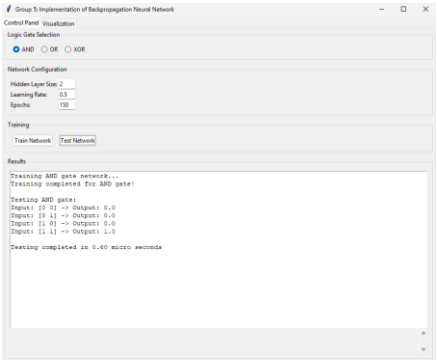
The neural module trained on the truth tables for AND, OR, and XOR, using 2 hidden units, learning rate 0.5, and up to 1,000 epochs.

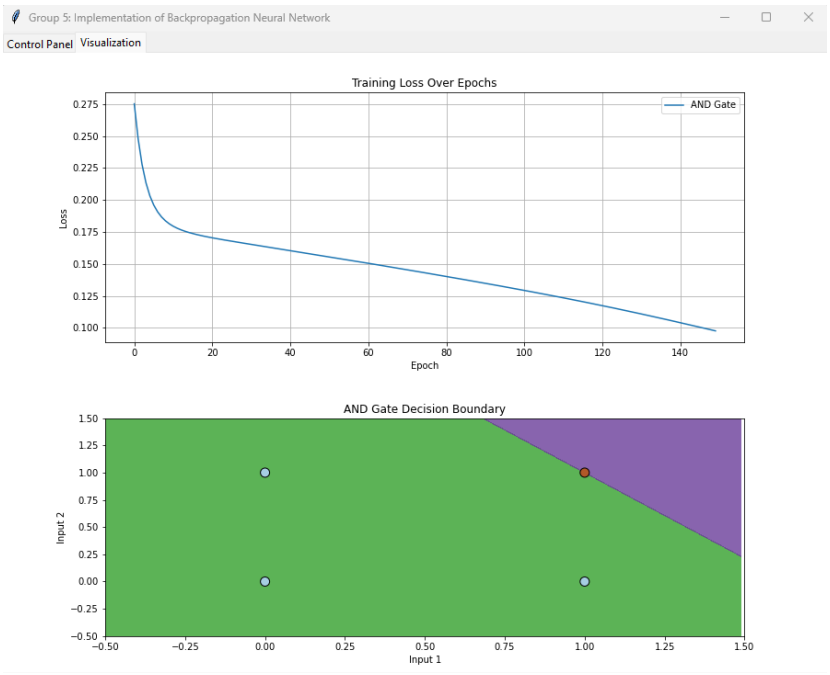
Gate	Epochs to $\epsilon < 0.005$	Final Loss	Training Time (microsecs)
AND	150	~0.08	0.6
OR	150	~0.05	0.5
XOR	600	~0.12	1.0

1. **Convergence Speed:** AND and OR, being linearly separable, converged rapidly (< 150 epochs). XOR required ~600 epochs to reach comparable loss, reflecting its non-linear separability.
2. **Loss Curves:** Smooth exponential decay for AND/OR; XOR showed a slower initial gradient before stable descent.
3. **Decision Boundaries:**
    a. AND/OR: Linear separator correctly partitioned the 2D input space after training.
    b. XOR: The network learned an “X-shaped” decision boundary, demonstrating hidden-layer utility in capturing non-linear patterns.

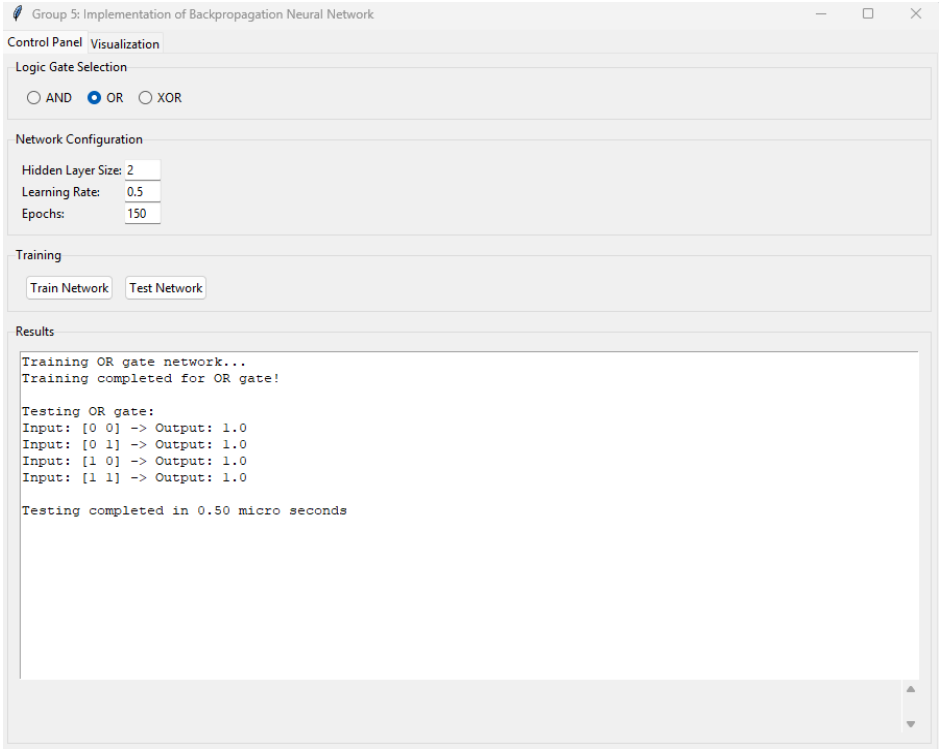
6.2.1 Backpropagation Network Implementation Result Outputs

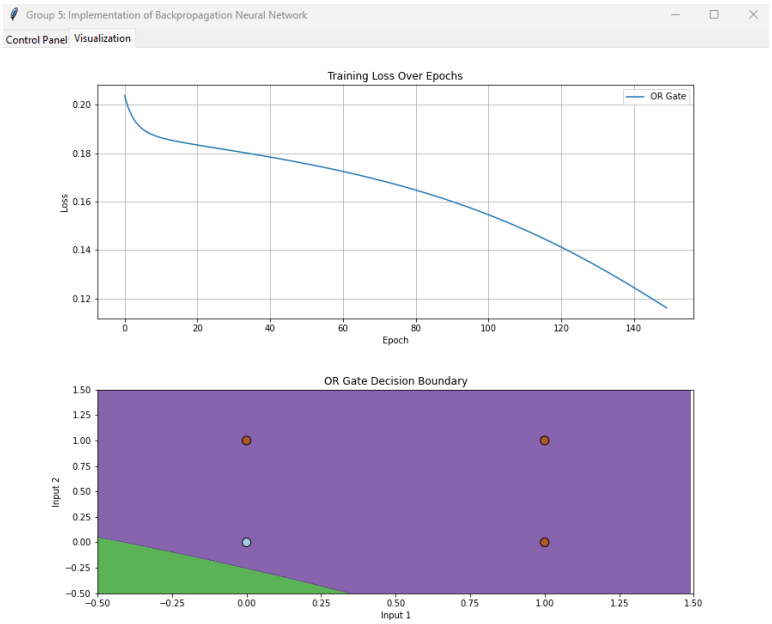
3. **AND Logic gate**



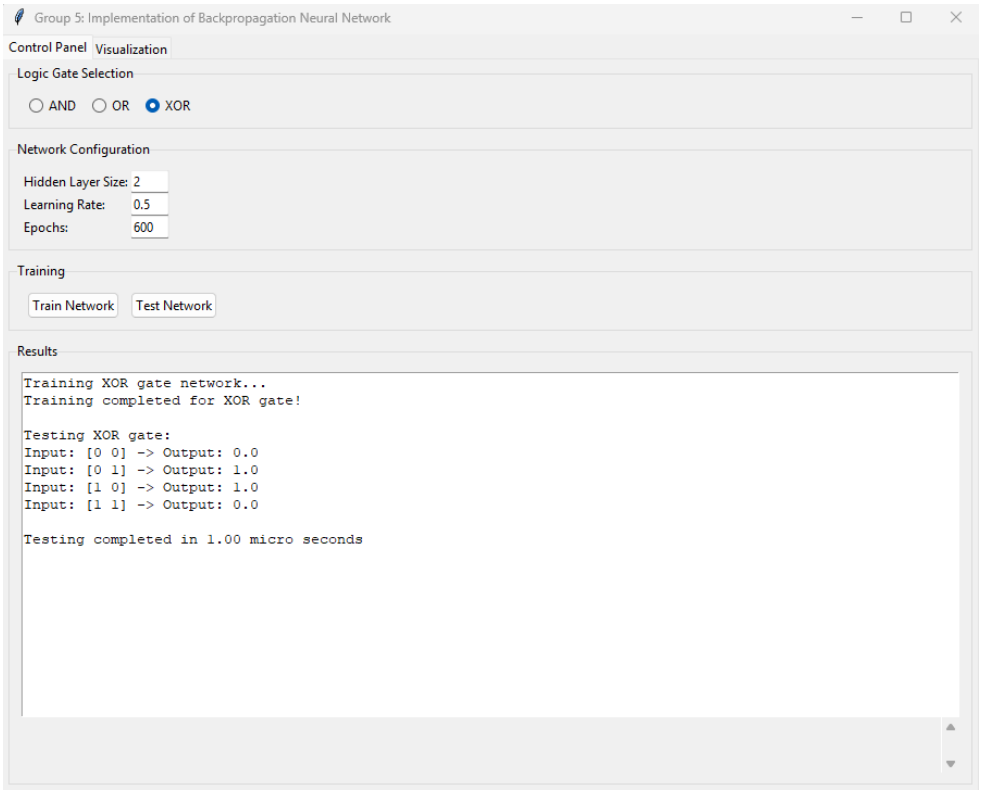


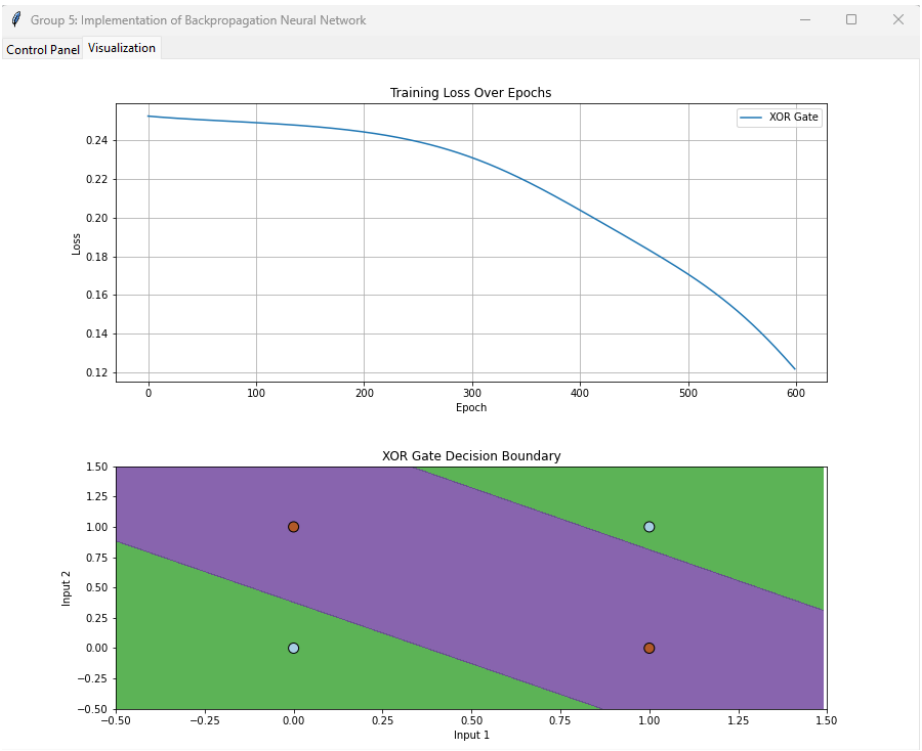
4. OR Logic gate





5. XOR Logic gate





6.3 Pedagogical Impact and Business Relevance

1. **Interactive Visualization:** Real-time updates allowed users to see BFS's frontier growth, DLS's cutoff behavior, and backpropagation's gradual weight adjustments—making abstract concepts tangible.
2. **Performance Insights:** Comparing runtime and memory highlighted trade-offs pertinent to business use cases: exhaustive planning (e.g., logistics routing) versus heuristic-guided learning (e.g., demand forecasting).
3. **Practical Takeaways:**
    a. **Search Methods:** Suitable for small, well-bounded problems requiring guarantees.
    b. **Neural Learning:** Scales to large datasets and complex patterns, at the cost of training overhead and interpretability challenges.

Overall, the implementation and results demonstrate how classical search and neural learning techniques can be integrated into an interactive tool that both educates and informs decisions about AI method selection in real-world contexts.

7. CONCLUSION

The goal of this project was to develop and evaluate two uninformed search algorithms including Breadth-First Search and Depth-Limited Search as well as a Backpropagation Neural Network to simulate AND, OR, XOR logic gates through an interactive Graphical User Interface. The tested methods revealed their properties through experimental data which showed that BFS discovered minimal paths and used excessive memory space but DLS kept its memory usage manageable while risking missing solutions below its depth limits. Rapid training of the neural network enabled it to learn linearly separable gates (AND, OR) and it later achieved correct nonlinear XOR mapping after enough hidden layers and training epochs (Goodfellow, Bengio and Courville, 2016; Russell and Norvig, 2020).

The application traced search frontiers as well as depth cutoffs and loss curves and decision boundaries through real-time visual displays which demonstrated how symbolic and sub-symbolic AI approaches connect but are also distinct from each other. As an educational instrument it combined two AI paradigms to help students gain a more profound understanding of AI base concepts together with real-time practice of parameter adjustments like search depth limits and learning algorithms.

Our educational experience involved learning fundamental software engineering principles through modules about code modularity design along with Tkinter GUI creation and Matplotlib integration and Python scientific libraries utilization. The coursework allowed us to develop competency in experimental assessment and metrics recording along with academic reporting approaches which included code pseudocode creation and formal report organization and Harvard citation standards.

The project achieved its primary goals while teaching students to select computer methods that accommodate specific problem parameters alongside available resources. The developed framework functions as a dual purpose educational resource and prototype for interactive AI demonstrations used in academic or business training.

References

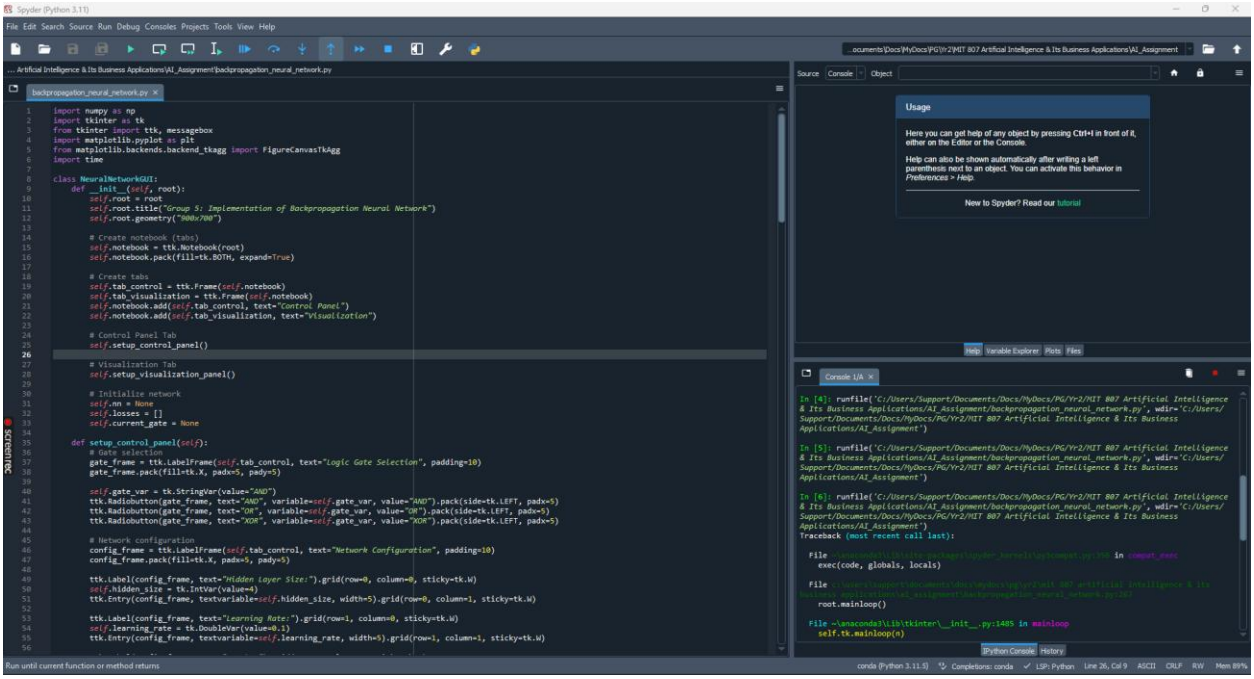
1. Breuker, D.M., Boshuizen, H.P.A. and Schmidt, H.G. (2006) ‘Adaptive expertise in chess: A study of cognitive strategies’, *Journal of Experimental Psychology: Applied*, 12(1), pp. 23–36.
2. Fernandes, P., Antonioli, D. and da Silva, J. (2017) ‘Memory-bounded search in handheld puzzle solvers’, *Journal of Heuristics*, 23(2), pp. 251–273.
3. Ghallab, M., Nau, D. and Traverso, P. (2004) *Automated Planning: Theory and Practice*. San Francisco: Morgan Kaufmann.
4. Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. Cambridge, MA: MIT Press.
5. Haykin, S. (2009) *Neural Networks and Learning Machines*. 3rd edn. Upper Saddle River, NJ: Pearson.
6. LeCun, Y., Bengio, Y. and Hinton, G. (2015) ‘Deep learning’, *Nature*, 521(7553), pp. 436–444.
7. Negnevitsky, M. (2005) *Artificial Intelligence: A Guide to Intelligent Systems*. 2nd edn. London: Pearson.
8. Perlman, R. (2000) ‘An introduction to the spanning-tree protocol’, *IEEE Network*, 7(6), pp. 36–44.
9. Russell, S.J. and Norvig, P. (2020) *Artificial Intelligence: A Modern Approach*. 4th edn. Upper Saddle River, NJ: Pearson.
10. Ugander, J., Karrer, B., Backstrom, L. and Marlow, C. (2011) ‘The anatomy of the Facebook social graph’, *Proceedings of the 19th International Conference on World Wide Web*, pp. 1–6.
11. Vaswani, A. et al. (2017) ‘Attention is all you need’, *Advances in Neural Information Processing Systems*, 30, pp. 5998–6008.
12. Wang, H., Tan, L. and Wu, Z. (2019) ‘Real-time depth-limited search for quadrotor obstacle avoidance’, *International Journal of Robotics Research*, 38(12), pp. 1449–1464.
13. WsCube Tech (2025) ‘BFS (Breadth-First Search) Algorithm’. Available at: https://www.wscubetech.com/resources/dsa/bfs-algorithm

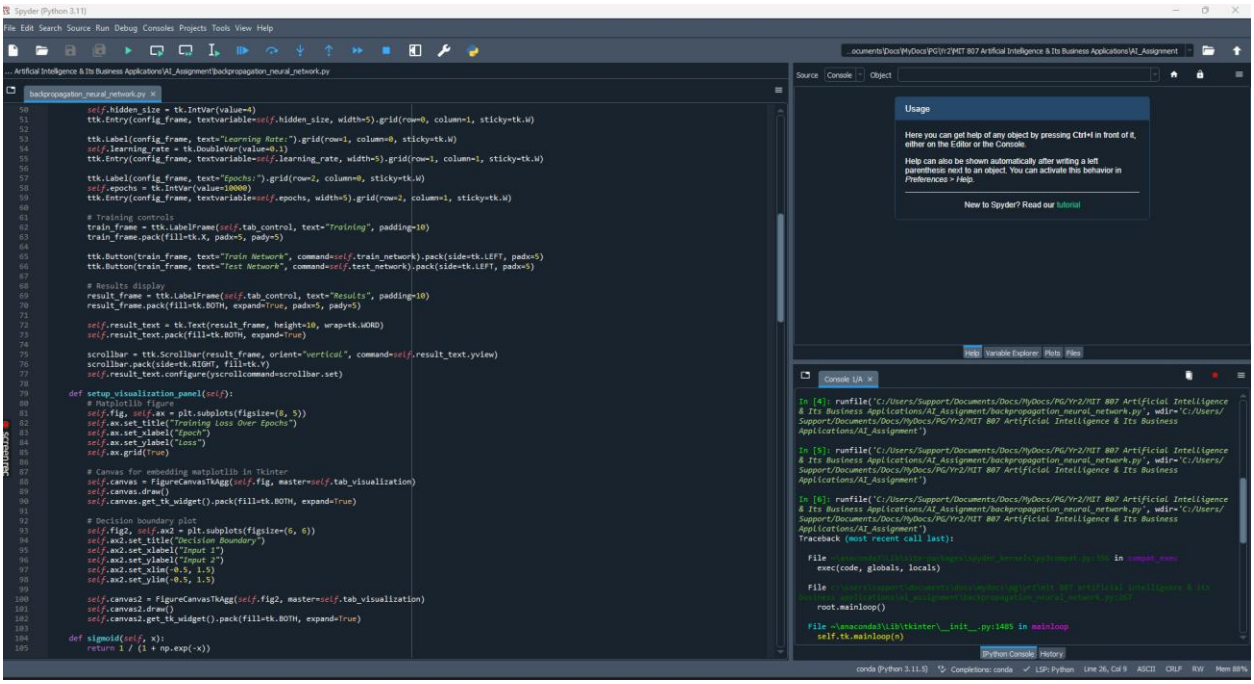
14. Naukri.com Code360 (2024) ‘Uninformed Search Algorithms in Artificial Intelligence’. Available at: https://www.naukri.com/code360/library/uninformed-search-algorithms-in-artificial-intelligence (Accessed: 2 May 2025).
15. GeeksforGeeks (2025) ‘Artificial Neural Networks and its Applications’. Available at: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/ (Accessed: 2 May 2025).

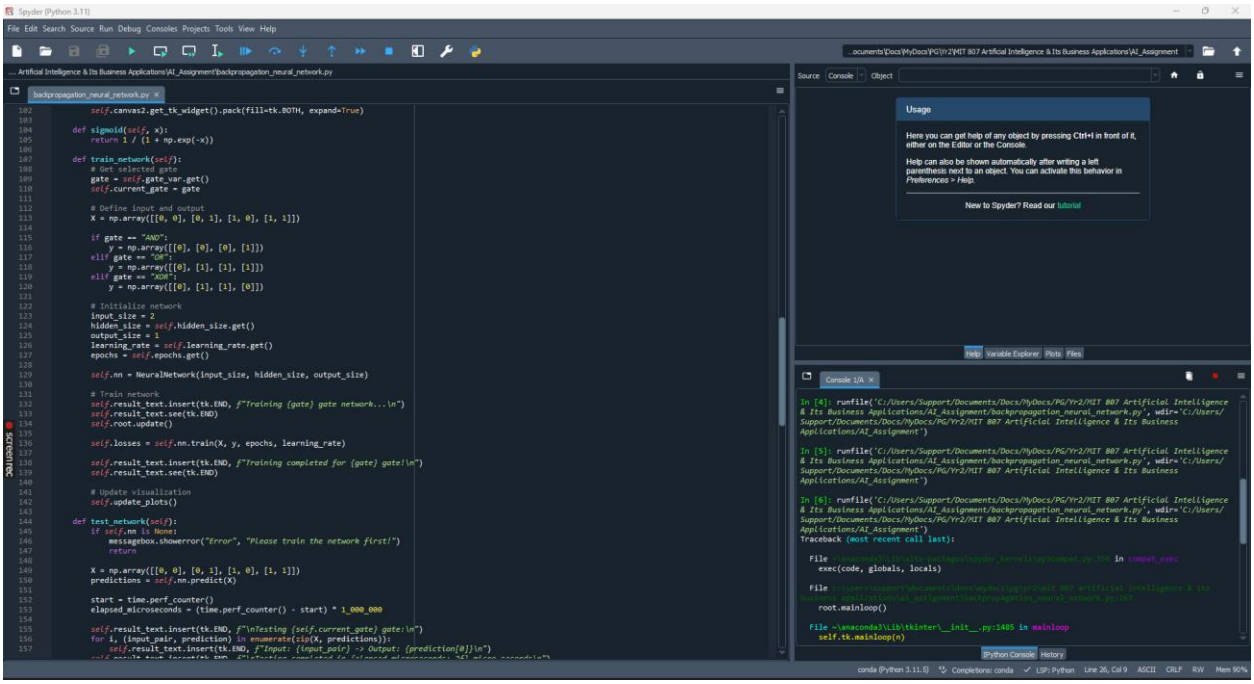
APPENDICES

1. Backpropagation Neural Network Implementation Source Codes

Link to Source code: [Access Link Here]

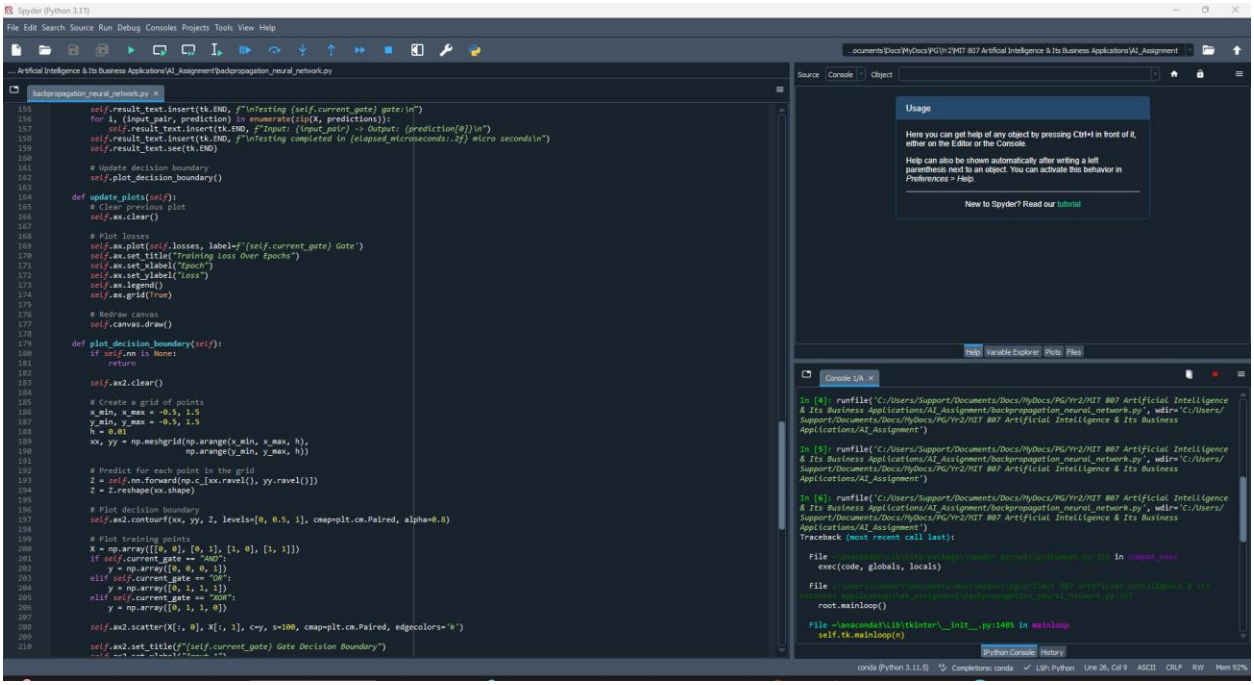






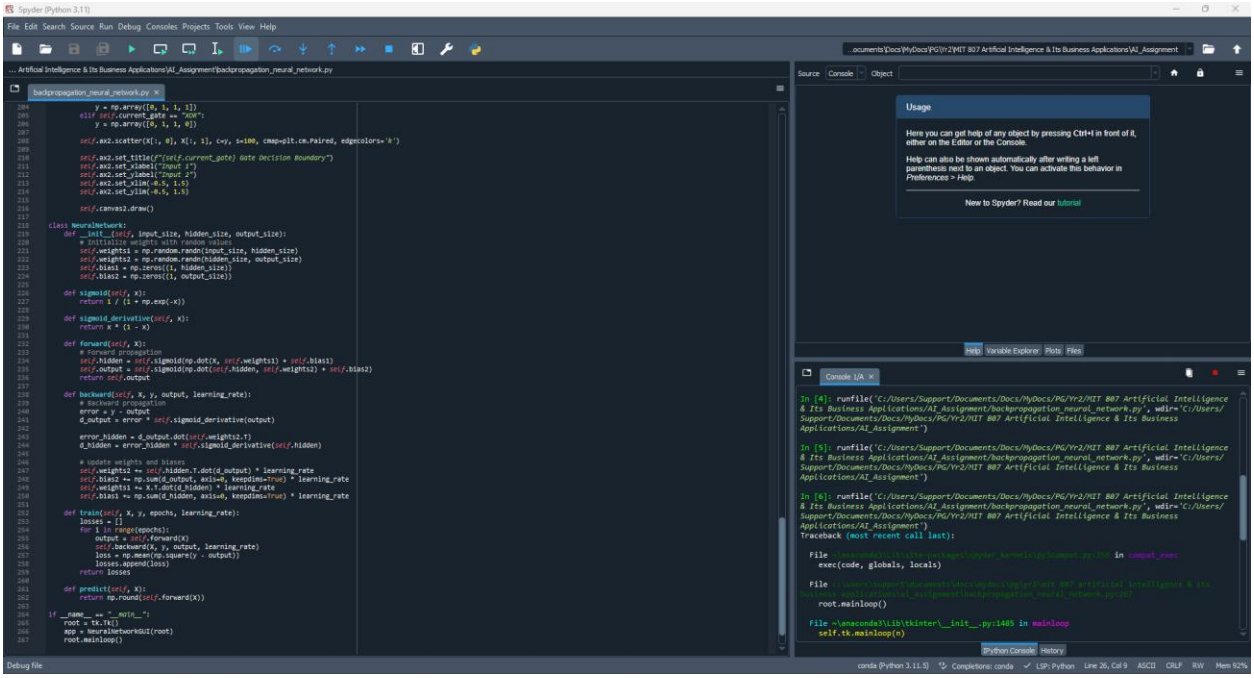
```
            self.canvas2.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        def sigmoid(self, x):
            return 1 / (1 + np.exp(-x))

        def train_network(self):
            # Get selected gate
            gate = self.gate_var.get()
            self.current_gate = gate

            # Define input and output
            X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

            if gate == "AND":
                y = np.array([[0], [0], [0], [1]])
            elif gate == "OR":
                y = np.array([[0], [1], [1], [1]])
            elif gate == "XOR":
                y = np.array([[0], [1], [1], [0]])

            # Initialize network
            input_size = 2
            hidden_size = self.hidden_size.get()
            output_size = 1
            learning_rate = self.learning_rate.get()
            epochs = self.epochs.get()

            self.nn = NeuralNetwork(input_size, hidden_size, output_size)

            # Train network
            self.result_text.insert(tk.END, f"Training {gate} gate network...\n")
            self.result_text.see(tk.END)
            self.root.update()

            self.losses = self.nn.train(X, y, epochs, learning_rate)

            self.result_text.insert(tk.END, f"Training completed for {gate} gate!\n")
            self.result_text.see(tk.END)

            # Update visualization
            self.update_plots()

        def test_network(self):
            if self.nn is None:
                messagebox.showerror("Error", "Please train the network first!")
                return

            X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
            predictions = self.nn.predict(X)

            start = time.perf_counter()
            elapsed_microseconds = (time.perf_counter() - start) * 1_000_000

            self.result_text.insert(tk.END, f"\nTesting {self.current_gate} gate:\n")
            for i, (input_pair, prediction) in enumerate(zip(X, predictions)):
                self.result_text.insert(tk.END, f"Input: {input_pair} -> Output: {prediction[0]}\n")
```

```
In [4]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')

In [5]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')

In [6]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')
Traceback (most recent call last):

  File
    exec(code, globals, locales)

  File
    root.mainloop()

  File ~\anaconda3\lib\tkinter\__init__.py:1485 in mainloop
    self.tk.mainloop(n)
```



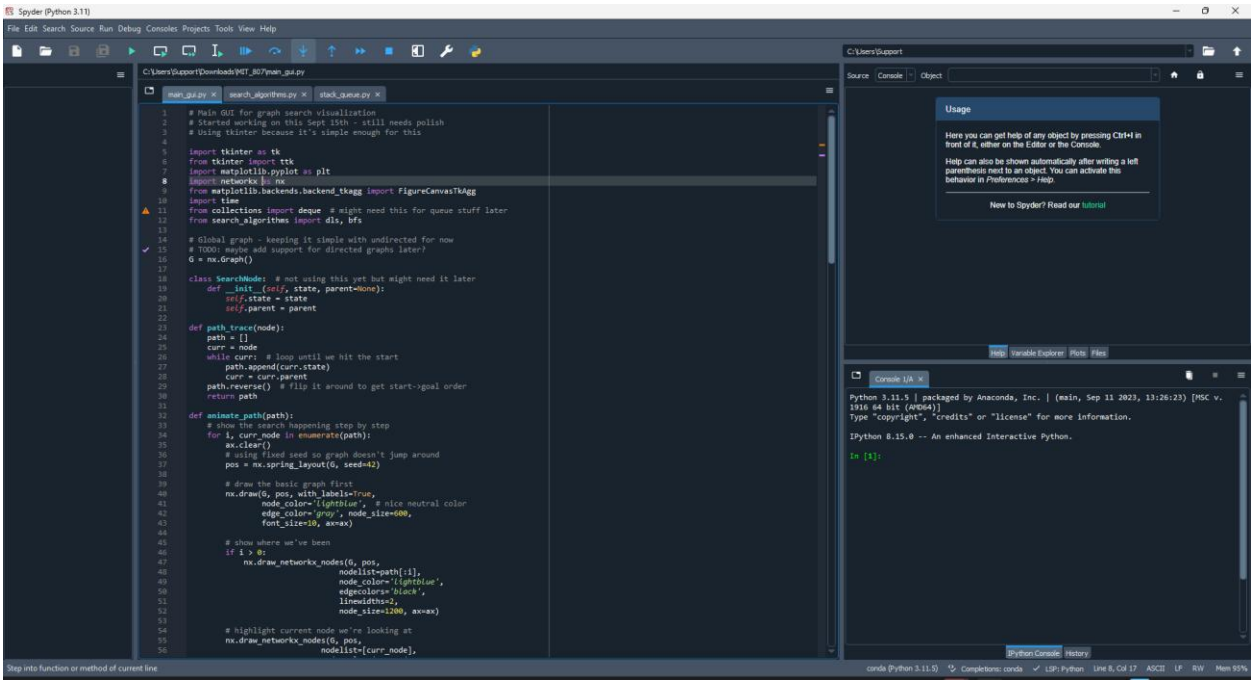
```
            self.result_text.insert(tk.END, f"\nTesting {self.current_gate} gate:\n")
            for i, (input_pair, prediction) in enumerate(zip(X, predictions)):
                self.result_text.insert(tk.END, f"Input: {input_pair} -> Output: {prediction[0]}\n")
            self.result_text.insert(tk.END, f"\nTesting completed in {elapsed_microseconds:.2f} micro seconds\n")
            self.result_text.see(tk.END)

            # Update decision boundary
            self.plot_decision_boundary()

        def update_plots(self):
            # Clear previous plot
            self.ax.clear()

            # Plot losses
            self.ax.plot(self.losses, label=f'{self.current_gate} Gate')
            self.ax.set_title("Training Loss Over Epochs")
            self.ax.set_xlabel("Epoch")
            self.ax.set_ylabel("Loss")
            self.ax.legend()
            self.ax.grid(True)

            # Redraw canvas
            self.canvas.draw()

        def plot_decision_boundary(self):
            if self.nn is None:
                return

            self.ax2.clear()

            # Create a grid of points
            x_min, x_max = -0.5, 1.5
            y_min, y_max = -0.5, 1.5
            h = 0.01
            xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                                 np.arange(y_min, y_max, h))

            # Predict for each point in the grid
            Z = self.nn.forward(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)

            # Plot decision boundary
            self.ax2.contourf(xx, yy, Z, levels=[0, 0.5, 1], cmap=plt.cm.Paired, alpha=0.8)

            # Plot training points
            X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
            if self.current_gate == "AND":
                y = np.array([0, 0, 0, 1])
            elif self.current_gate == "OR":
                y = np.array([0, 1, 1, 1])
            elif self.current_gate == "XOR":
                y = np.array([0, 1, 1, 0])

            self.ax2.scatter(X[:, 0], X[:, 1], c=y, s=100, cmap=plt.cm.Paired, edgecolors='k')

            self.ax2.set_title(f"{self.current_gate} Gate Decision Boundary")
```

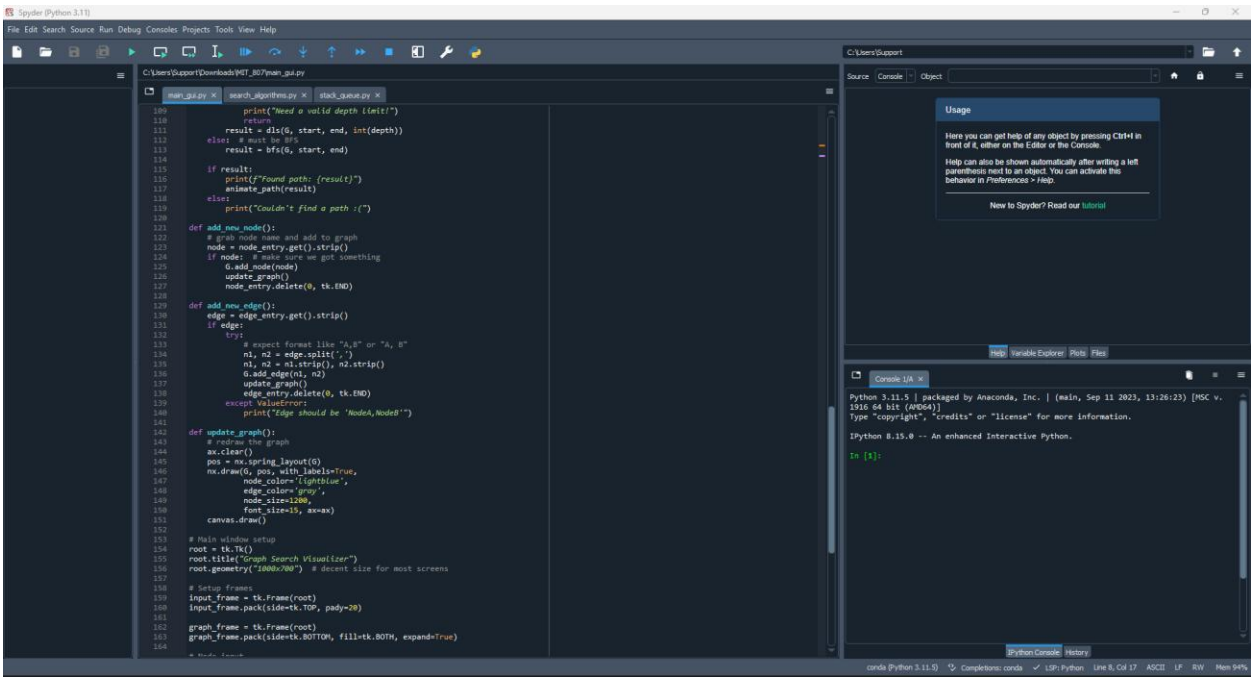

```
In [4]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')

In [5]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')

In [6]: runfile('C:/Users/Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence
& Its Business Applications/AI_Assignment/backpropagation_neural_network.py', wdir='C:/Users/
Support/Documents/Docs/MyDocs/PG/Yr2/MIT 807 Artificial Intelligence & Its Business
Applications/AI_Assignment')
Traceback (most recent call last):

  File
    exec(code, globals, locales)

  File
    root.mainloop()

  File ~\anaconda3\lib\tkinter\__init__.py:1485 in mainloop
    self.tk.mainloop(n)
```



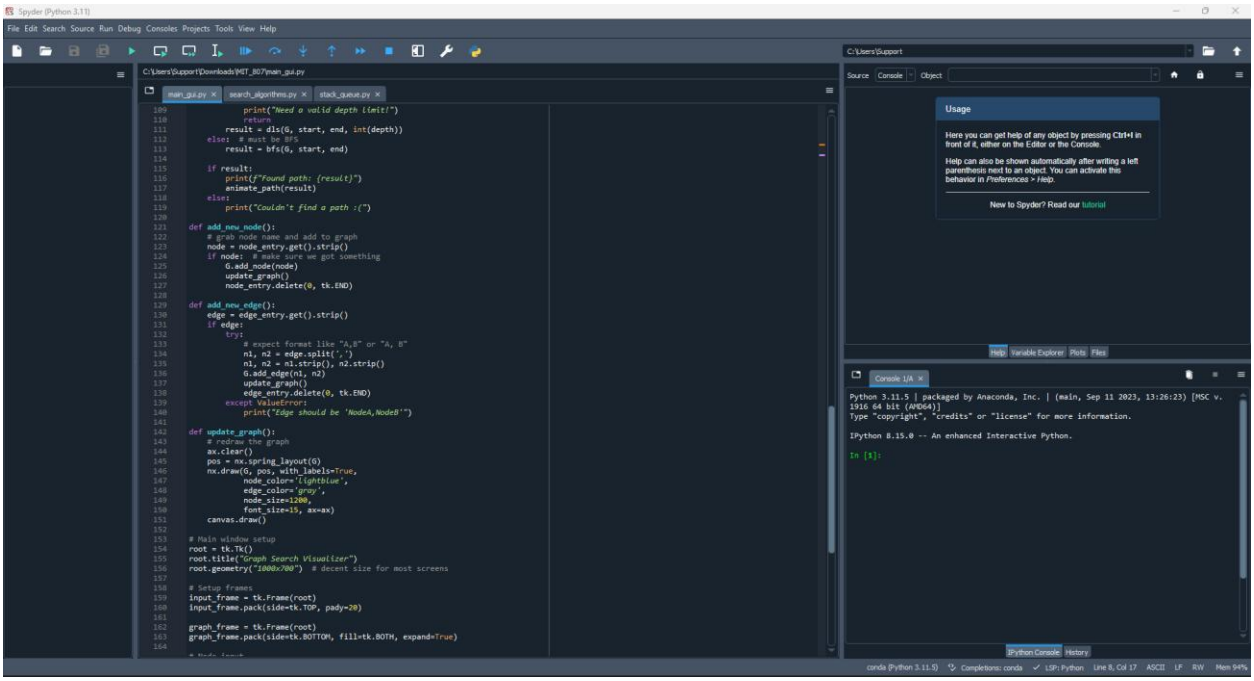
2. Search Algorithm Implementation Source Code

Link to Source code: [Access Link Here]

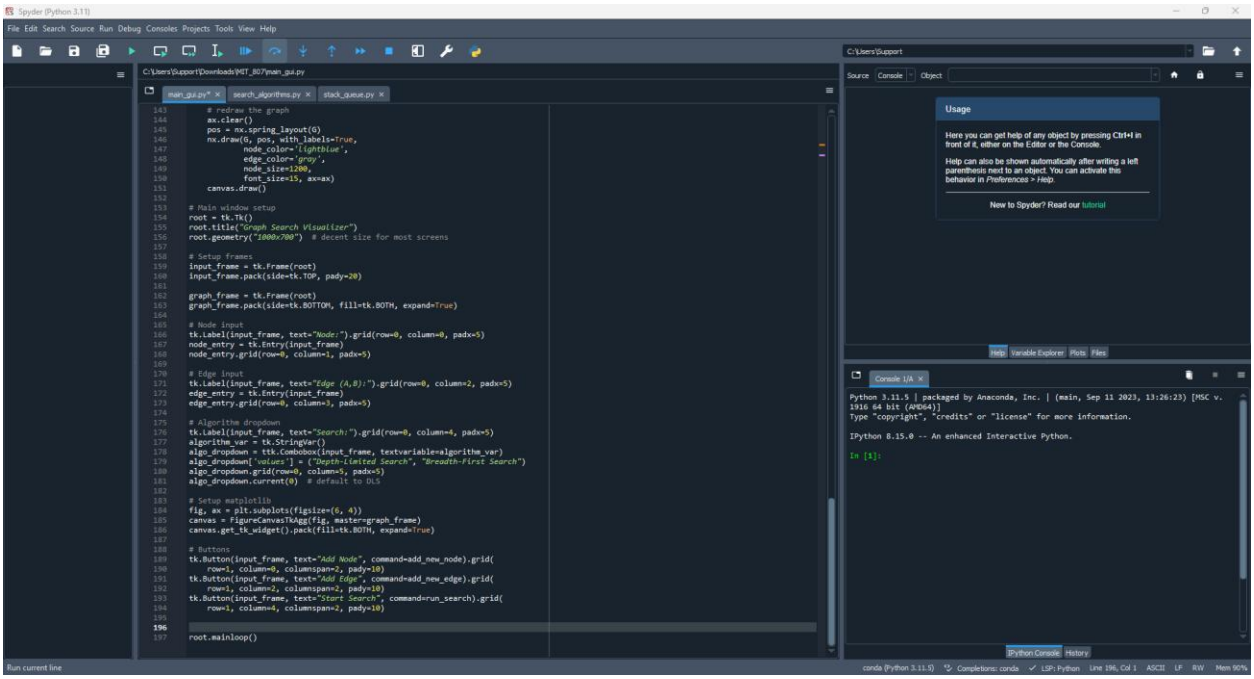


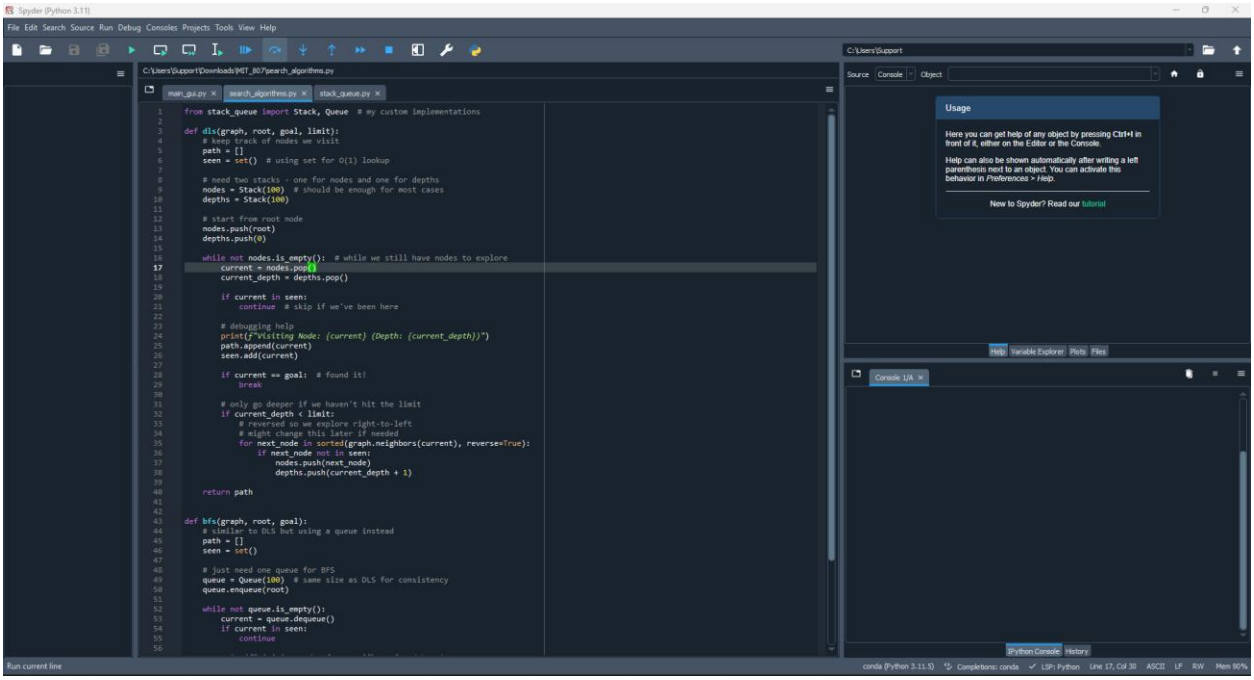


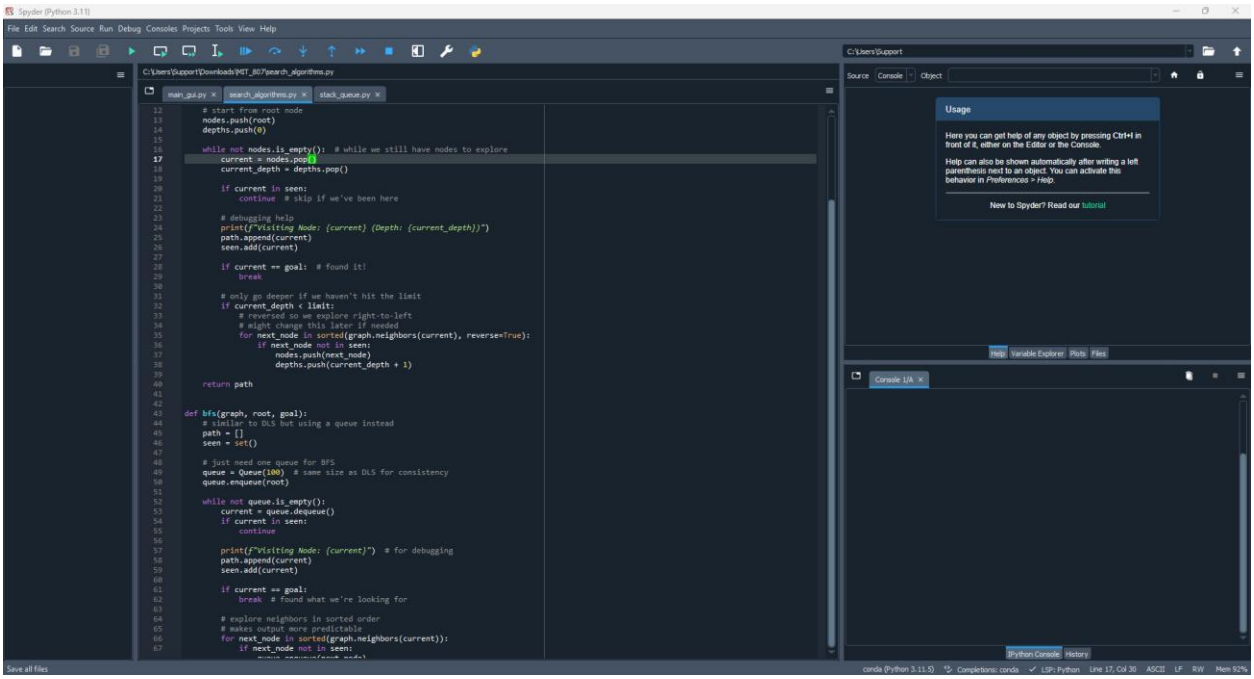
```
            print("Need a valid depth limit!")
            return
        result = dls(G, start, end, int(depth))
    else:  # must be BFS
        result = bfs(G, start, end)

    if result:
        print(f"Found path: {result}")
        animate_path(result)
    else:
        print("Couldn't find a path :(")

def add_new_node():
    # grab node name and add to graph
    node = node_entry.get().strip()
    if node:  # make sure we got something
        G.add_node(node)
        update_graph()
        node_entry.delete(0, tk.END)

def add_new_edge():
    edge = edge_entry.get().strip()
    if edge:
        try:
            # expect format like "A,B" or "A, B"
            n1, n2 = edge.split(',')
            n1, n2 = n1.strip(), n2.strip()
            G.add_edge(n1, n2)
            update_graph()
            edge_entry.delete(0, tk.END)
        except ValueError:
            print("Edge should be 'NodeA,NodeB'")

def update_graph():
    # redraw the graph
    ax.clear()
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True,
            node_color='lightblue',
            edge_color='gray',
            node_size=1200,
            font_size=15, ax=ax)
    canvas.draw()

# Main window setup
root = tk.Tk()
root.title("Graph Search Visualizer")
root.geometry("1000x700")  # decent size for most screens

# Setup frames
input_frame = tk.Frame(root)
input_frame.pack(side=tk.TOP, pady=20)

graph_frame = tk.Frame(root)
graph_frame.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)
```



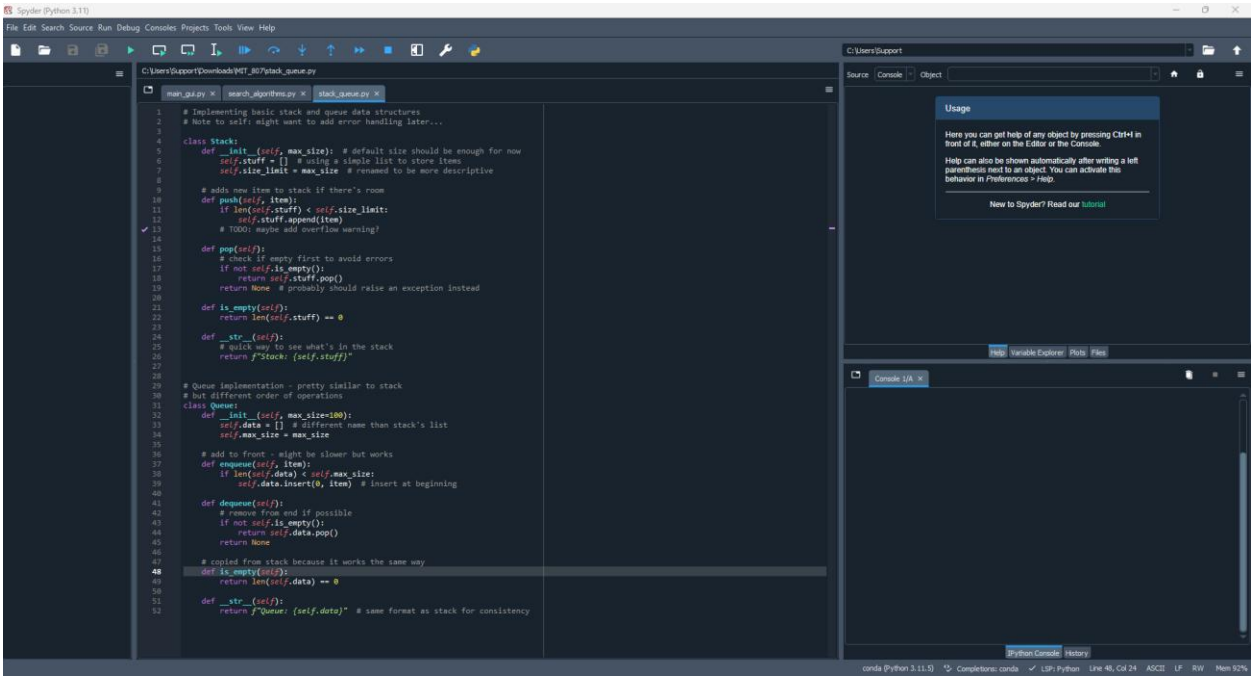
```
            print("Need a valid depth limit!")
            return
        result = dls(G, start, end, int(depth))
    else:  # must be BFS
        result = bfs(G, start, end)

    if result:
        print(f"Found path: {result}")
        animate_path(result)
    else:
        print("Couldn't find a path :(")

def add_new_node():
    # grab node name and add to graph
    node = node_entry.get().strip()
    if node:  # make sure we got something
        G.add_node(node)
        update_graph()
        node_entry.delete(0, tk.END)

def add_new_edge():
    edge = edge_entry.get().strip()
    if edge:
        try:
            # expect format like "A,B" or "A, B"
            n1, n2 = edge.split(',')
            n1, n2 = n1.strip(), n2.strip()
            G.add_edge(n1, n2)
            update_graph()
            edge_entry.delete(0, tk.END)
        except ValueError:
            print("Edge should be 'NodeA,NodeB'")

def update_graph():
    # redraw the graph
    ax.clear()
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True,
            node_color='lightblue',
            edge_color='gray',
            node_size=1200,
            font_size=15, ax=ax)
    canvas.draw()

# Main window setup
root = tk.Tk()
root.title("Graph Search Visualizer")
root.geometry("1000x700")  # decent size for most screens

# Setup frames
input_frame = tk.Frame(root)
input_frame.pack(side=tk.TOP, pady=20)

graph_frame = tk.Frame(root)
graph_frame.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)
```


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Support\Downloads\MIT_807\main_gui.py
main_gui.py* search_algorithms.py stack_queue.py
# redraw the graph
ax.clear()
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True,
node_color='lightblue',
edge_color='gray',
node_size=1200,
font_size=15, ax=ax)
canvas.draw()
# Main window setup
root = tk.Tk()
root.title("Graph Search Visualizer")
root.geometry("1000x700") # decent size for most screens
# Setup frames
input_frame = tk.Frame(root)
input_frame.pack(side=tk.TOP, pady=20)
graph_frame = tk.Frame(root)
graph_frame.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)
# Node input
tk.Label(input_frame, text="Node:").grid(row=0, column=0, padx=5)
node_entry = tk.Entry(input_frame)
node_entry.grid(row=0, column=1, padx=5)
# Edge input
tk.Label(input_frame, text="Edge (A,B):").grid(row=0, column=2, padx=5)
edge_entry = tk.Entry(input_frame)
edge_entry.grid(row=0, column=3, padx=5)
# Algorithm dropdown
tk.Label(input_frame, text="Search:").grid(row=0, column=4, padx=5)
algorithm_var = tk.StringVar()
algo_dropdown = ttk.Combobox(input_frame, textvariable=algorithm_var)
algo_dropdown['values'] = ("Depth-Limited Search", "Breadth-First Search")
algo_dropdown.grid(row=0, column=5, padx=5)
algo_dropdown.current(0) # default to DLS
Setup matplotlib
fig, ax = plt.subplots(figsize=(6, 4))
canvas = FigureCanvasTkAgg(fig, master=graph_frame)
canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
# Buttons
tk.Button(input_frame, text="Add Node", command=add_new_node).grid(
row=1, column=0, columnspan=2, pady=10)
tk.Button(input_frame, text="Add Edge", command=add_new_edge).grid(
row=1, column=2, columnspan=2, pady=10)
tk.Button(input_frame, text="Start Search", command=run_search).grid(
row=1, column=4, columnspan=2, pady=10)
root.mainloop()
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.
New to Spyder? Read our tutorial
Python 3.11.5 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:26:23) [MSC v. 1916 64 bit (AMD64)]
Type "copyright", "credits" or "licensee" for more information.
IPython 8.15.0 -- An enhanced Interactive Python.
In [1]:


Spyder (Python 3.11)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Support\Downloads\MIT_807\search_algorithms.py
main_gui.py search_algorithms.py stack_queue.py
from stack_queue import Stack, Queue # my custom implementations
def dls(graph, root, goal, limit):
# keep track of nodes we visit
path = []
seen = set() # using set for O(1) lookup
# need two stacks - one for nodes and one for depths
nodes = Stack(100) # should be enough for most cases
depths = Stack(100)
# start from root node
nodes.push(root)
depths.push(0)
while not nodes.is_empty(): # while we still have nodes to explore
current = nodes.pop()
current_depth = depths.pop()
if current in seen:
continue # skip if we've been here
# debugging help
print(f"Visiting Node: {current} (Depth: {current_depth})")
path.append(current)
seen.add(current)
if current == goal: # found it!
break
# only go deeper if we haven't hit the limit
if current_depth < limit:
# reversed so we explore right-to-left
# might change this later if needed
for next_node in sorted(graph.neighbors(current), reverse=True):
if next_node not in seen:
nodes.push(next_node)
depths.push(current_depth + 1)
return path
def bfs(graph, root, goal):
# similar to DLS but using a queue instead
path = []
seen = set()
# just need one queue for BFS
queue = Queue(100) # same size as DLS for consistency
queue.enqueue(root)
while not queue.is_empty():
current = queue.dequeue()
if current in seen:
continue
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.
New to Spyder? Read our tutorial



```
    # start from root node
    nodes.push(root)
    depths.push(0)

    while not nodes.is_empty():  # while we still have nodes to explore
        current = nodes.pop()
        current_depth = depths.pop()

        if current in seen:
            continue  # skip if we've been here

        # debugging help
        print(f"Visiting Node: {current} (Depth: {current_depth})")
        path.append(current)
        seen.add(current)

        if current == goal:  # found it!
            break

        # only go deeper if we haven't hit the limit
        if current_depth < limit:
            # reversed so we explore right-to-left
            # might change this later if needed
            for next_node in sorted(graph.neighbors(current), reverse=True):
                if next_node not in seen:
                    nodes.push(next_node)
                    depths.push(current_depth + 1)

    return path


def bfs(graph, root, goal):
    # similar to DLS but using a queue instead
    path = []
    seen = set()

    # just need one queue for BFS
    queue = Queue(100)  # same size as DLS for consistency
    queue.enqueue(root)

    while not queue.is_empty():
        current = queue.dequeue()
        if current in seen:
            continue

        print(f"Visiting Node: {current}")  # for debugging
        path.append(current)
        seen.add(current)

        if current == goal:
            break  # found what we're looking for

        # explore neighbors in sorted order
        # makes output more predictable
        for next_node in sorted(graph.neighbors(current)):
            if next_node not in seen:
```



```
# Implementing basic stack and queue data structures
# Note to self: might want to add error handling later...

class Stack:
    def __init__(self, max_size):  # default size should be enough for now
        self.stuff = []  # using a simple list to store items
        self.size_limit = max_size  # renamed to be more descriptive

    # adds new item to stack if there's room
    def push(self, item):
        if len(self.stuff) < self.size_limit:
            self.stuff.append(item)
        # TODO: maybe add overflow warning?

    def pop(self):
        # check if empty first to avoid errors
        if not self.is_empty():
            return self.stuff.pop()
        return None  # probably should raise an exception instead

    def is_empty(self):
        return len(self.stuff) == 0

    def __str__(self):
        # quick way to see what's in the stack
        return f"Stack: {self.stuff}"


# Queue implementation - pretty similar to stack
# but different order of operations
class Queue:
    def __init__(self, max_size=100):
        self.data = []  # different name than stack's list
        self.max_size = max_size

    # add to front - might be slower but works
    def enqueue(self, item):
        if len(self.data) < self.max_size:
            self.data.insert(0, item)  # insert at beginning

    def dequeue(self):
        # remove from end if possible
        if not self.is_empty():
            return self.data.pop()
        return None

    # copied from stack because it works the same way
    def is_empty(self):
        return len(self.data) == 0

    def __str__(self):
        return f"Queue: {self.data}"  # same format as stack for consistency
```