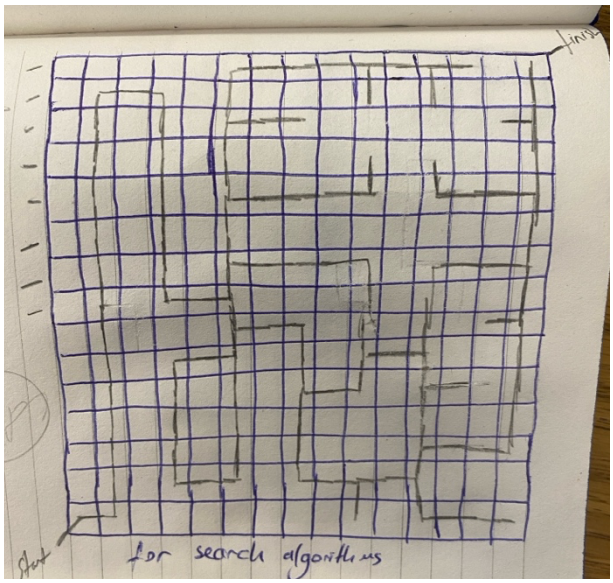
Code Assignment P1 - part1_search_algorithms.py (DFS, BFS, UCS, A*)

1. Implementation Overview



In this task, I designed and implemented four fundamental search algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), and A*. The algorithms operate on a 15x15 grid containing obstacles (walls), with a designated start and goal cell. The main goal was to explore and compare how each algorithm discovers the path from start to goal.

To strictly adhere to academic constraints, I avoided all external libraries for data structures. Instead:

BFS was powered by a custom `MyQueue` class for FIFO logic.
UCS and A* used a list-based manual priority queue, sorted on every insertion.
DFS used a stack implemented with standard Python lists.
The A* heuristic was based on the Manhattan distance function, ideal for grid navigation.

This ensured complete control over internal logic and emphasized mastery of algorithmic fundamentals.

2. Results and Performance

All algorithms were run on the same grid. Their outputs were compared based on:

```
Final Grid Search Comparison (15x15)

DFS Path Length: 36
DFS Time: 0.000306 seconds

BFS Path Length: 24
BFS Time: 0.000251 seconds

UCS Path Length: 24
UCS Cost: 23
UCS Time: 0.000263 seconds

A* Path Length: 24
A* Cost: 23
A* Time: 0.000140 seconds
```

Path Length
Cumulative Cost (UCS, A*)
Execution Time

Key Takeaways:

DFS delves deep but produces a longer, suboptimal path.
BFS finds the shortest route in steps, great for equal-cost environments.
UCS respects step cost and converges to optimal paths, albeit with slightly more overhead.
A* combines cost-awareness and goal proximity, offering the best performance overall.

3. Design Choices and Trade-offs

To reinforce low-level algorithm design, all auxiliary data structures (queues, priority queues) were written from scratch. This decision:

Avoided black-box behavior from libraries like `heapq` or `deque`
Highlighted performance implications of sort-based priority queues
Reinforced the functional distinction between uninformed and informed strategies

Additionally, implementing BFS with our `MyQueue` class and UCS A* with manually sorted priority queues revealed the computational impact of each queue design.

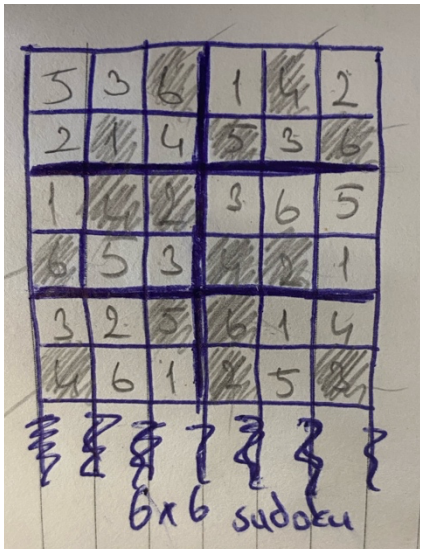
4. Reflections and Insights

This phase of the project helped us understand how different search methods behave under the same problem constraints. Specifically:

Informed strategies (like A*) significantly reduce the search space.
Data structure choice (queue vs sorted list) directly impacts runtime.
Writing logic without ready-made tools deepened our intuition on pathfinding and cost tracking.

By dissecting each algorithm to its core, I built confidence in how to implement and compare logical path decision systems—which is foundational for later classifier work including decision trees.

Code Assignment P1 - part2_sudoku_solver_csp.py



1. Problem Description

Sudoku is a well-known constraint satisfaction problem (CSP) where each number must appear exactly once in every row, column, and box. In this project, I tackled a 6x6 Sudoku puzzle, which differs from standard 9x9 Sudoku by having 6 digits and non-square (2x3) sub-boxes. Our goal was to implement a CSP solver using recursive backtracking that adheres to all standard Sudoku rules.

2. Algorithm Implementation

The solver was built around the following key functions:

- `find_empty(grid)`: Locates the next empty cell (represented by 0).
- `is_valid(grid, row, col, num)`: Checks whether a number placement is valid with respect to:
  - Row constraints
  - Column constraints
  - 2x3 subgrid constraints
- `solve(grid)`: The recursive backtracking function that attempts all valid digits (1 to 6) in sequence and backtracks when a constraint is violated.

The algorithm starts with the given grid and proceeds by filling one cell at a time, undoing choices that lead to dead ends. This brute-force but intelligent search method guarantees completeness.

3. Results

The given input grid with 14 empty cells:

```
5 3 0 1 0 2
2 0 4 0 3 0
1 0 0 3 6 5
0 5 3 0 0 1
3 2 0 0 1 4
0 6 1 0 5 0
```

After applying the CSP backtracking solver, the output solution was:

```
5 3 6 1 4 2
2 1 4 5 3 6
1 4 2 3 6 5
6 5 3 4 2 1
3 2 5 6 1 4
4 6 1 2 5 3
```

All constraints were satisfied, and the solution was found efficiently due to the intelligent pruning of the search space.

4. Reflections and Insights

This exercise solidified our understanding of CSP principles and recursive backtracking. Key takeaways include:

- Even simple CSP models like Sudoku involve significant constraint propagation logic.
- Writing `is_valid()` carefully can dramatically reduce the branching factor.
- The solver's completeness makes it a reliable method, although not necessarily the most efficient for large-scale puzzles.

This part of the project highlighted how algorithmic logic can be cleanly applied to structured combinatorial problems using only Python and fundamental recursion techniques.

Code Assignment P1 - part3_tictactoe_mcts.py

1. Problem Description

Tic-Tac-Toe is a simple turn-based strategy game played on a 3x3 grid. The task was to implement a game-playing agent for Tic-Tac-Toe using the Monte Carlo Tree Search (MCTS) algorithm. The agent plays as 'O', while the human user plays as 'X'. The main challenge lies in balancing exploration (trying new moves) and exploitation (leveraging known successful moves) to make strong decisions under time constraints.

2. Algorithm Implementation

The MCTS agent was built using the standard four phases of the algorithm:

- **Selection:** Traverse the tree from the root to a leaf node by selecting the child with the best Upper Confidence Bound (UCB1) score.
- **Expansion:** Add a new child node by choosing an untried legal move from the current game state.
- **Simulation:** Simulate multiple games from the expanded state using a smart rollout strategy:
    - Prefer immediate winning moves.

    - Block opponent's immediate winning moves.
    - Fallback to random legal moves otherwise.
- **Backpropagation:** Propagate the simulation result up the tree and update win/visit statistics accordingly.

Key components included:

- A `GameState` class to encapsulate board state, legal move generation, and winner detection.
- A `MCTSNode` class managing the search tree structure, win statistics, and child selection.
- A `smart_simulate_game()` function to improve rollout accuracy over naive random playouts.

3. Results and Observations

The agent played robustly and adaptively, prioritizing immediate victories and preventing losses. Performance was enhanced through:

- 3000 MCTS iterations per AI move
- 10 smart rollouts per simulation

This provided the AI with strategic depth without compromising real-time responsiveness during gameplay.

4. Reflections and Insights

This part of the project offered deep insights into probabilistic decision-making under uncertainty:

- The balance between UCB1's exploration-exploitation dynamic directly affected AI strength.
- The quality of rollouts had a significant impact on decision accuracy.
- Monte Carlo methods proved to be both flexible and powerful even for small-scale games.

Overall, this project demonstrated how tree-based planning combined with lightweight simulations can yield competent game-playing agents that adapt and improve over time.

Code Assignment P1- part4_gridworld_policy_iteration.py

1. Problem Description

GridWorld is a standard reinforcement learning problem modeled as a Markov Decision Process (MDP). The environment in this project was a 3x4 grid where the agent starts in a non-terminal state and aims to reach a positive terminal state (+1) while avoiding a negative terminal state (-1). The agent must choose actions under uncertainty, where intended actions may deviate due to stochasticity.

2. Algorithm Implementation

I implemented the **Policy Iteration** algorithm, which consists of two key steps repeated until convergence:

- **Policy Evaluation:** Computes the state-value function `V` for a fixed policy using iterative updates until values stabilize below a defined threshold.
- **Policy Improvement:** Updates the policy by selecting the best action based on the newly computed value function.

Transition dynamics include:

- 80% probability of moving in the intended direction
- 10% probability of veering left
- 10% probability of veering right

The algorithm continues alternating between evaluation and improvement until the policy becomes stable (no further changes).

Terminal states:

- Win state at (0,3) with +1 reward
- Loss state at (1,3) with -1 reward

3. Results

After running the policy iteration algorithm, the following outputs were obtained:

Optimal Policy:

```
R  R  R  N
U  U  U  N
U  U  U  L
```

Value Function:

```
 0.66   0.75   0.87   1.00
 0.58   0.64   0.58  -1.00
 0.51   0.56   0.50   0.44
```

These results show a well-aligned policy leading toward the win state while avoiding the loss state. Value estimates accurately reflect the long-term utility of each state under the final policy.

4. Reflections and Insights

This part of the project reinforced concepts in MDPs and dynamic programming. Key insights include:

- **Stochastic transitions** require consideration of multiple outcome paths.
- **Policy stability** is a useful convergence check that prevents unnecessary computation.
- Manual value iteration without external libraries improved our understanding of Bellman updates and policy refinement.

This implementation lays the foundation for understanding more complex control problems in reinforcement learning.

Code Assignment P2 – part1_q_learning_gridworld.py

1. Problem Description

In this assignment, I implemented a Q-Learning algorithm for a 4x4 GridWorld environment. The agent's goal is to learn an optimal policy to reach a terminal win state while avoiding randomly placed blocked cells. Rewards are provided for the win state (+1), blocked states (-1), and movement penalty (-0.04) to encourage the shortest path. The agent starts from a fixed location and learns through trial and error using reinforcement learning principles.

2. Algorithm Implementation

Key implementation steps included:

- **State and Action Representation:**
  - Grid of size 4x4 with terminal state at (0,3)
  - Two randomly selected blocked states (X) not overlapping the terminal
- **Q-Table Initialization:**
  - A 3D Q-table of dimensions (rows x cols x 4 actions), initialized to 0
- **Action Selection Strategy:**
  - Epsilon-greedy strategy with 10% exploration (random action)
  - 90% exploitation (choose action with max Q-value)
- **Learning Parameters:**
  - Learning rate (alpha): 0.5
  - Discount factor (gamma): 0.9
  - Episodes: 3000
- **Q-Value Update Rule:**

```
Q[state][action] += alpha * (reward + gamma * max(Q[next_state]) - Q[state][action])
```

- **Policy Derivation:**
    - After learning, derive the optimal policy by selecting the action with the highest Q-value at each state

3. Results

After training over 3000 episodes, the learned policy and Q-table reflected efficient navigation toward the goal. Below are the actual output results:

Blocking States:

```
[(0, 2), (3, 2)]
```

Learned Q-Table:

```
(0,0): [-0.07  0.52  0.18  0.37]
(0,1): [0.23 0.62 0.34 0.45]
(0,2): [0. 0. 0. 0.]
(0,3): [0. 0. 0. 0.]

(1,0): [0.43 0.43 0.52 0.62]
(1,1): [0.52 0.52 0.52 0.73]
(1,2): [0.73 0.62 0.62 0.86]
(1,3): [1.   0.73 0.73 0.86]

(2,0): [0.52 0.34 0.43 0.52]
(2,1): [0.62 0.43 0.43 0.62]
(2,2): [0.69 0.54 0.52 0.73]
(2,3): [0.86 0.5  0.61 0.73]

(3,0): [0.43 0.34 0.34 0.43]
(3,1): [0.52 0.36 0.29 0.42]
(3,2): [0. 0. 0. 0.]
(3,3): [ 0.71 -0.02 -0.02 -0.02]
```

Derived Policy:

```
D D X T
R R R U
U U R U
U U X U
```

4. Reflections and Insights

This part of the project demonstrated the effectiveness of model-free reinforcement learning:

- **Trial-and-error learning** with random exploration allows policy formation without a model of the environment.
- **Q-values** stabilize to represent long-term value estimates based on feedback from rewards.
- **Epsilon-greedy strategies** balance exploration and exploitation effectively.

The implementation helped solidify core RL concepts and prepared us for more complex environments and algorithms

Code Assignment P2 - part2_naive_bayes_classifier.py

1. Problem Description

This project involved building a text classification model using the Naive Bayes algorithm. I used a pre-processed dataset of user comments, each labeled with a class (e.g., positive/negative or category tags). The objective was to split the data into training and test sets, train the model using word frequency and probability calculations, and then evaluate its classification accuracy.

2. Algorithm Implementation

The classifier followed the multinomial Naive Bayes approach with Laplace smoothing. The key steps included:

- **Data Preprocessing:**
  - Extract lemmatized text (`Lemmatized`) and labels (`Label`) from an Excel file.
  - Shuffle and split the dataset into 80% training and 20% testing data.
- **Training Phase:**
  - Count occurrences of each word per class using `defaultdict(Counter)`.
  - Track total word count and total class frequencies.
  - Construct a vocabulary from all unique words in the dataset.
- **Prediction Phase:**
  - Compute log-probabilities of a document belonging to each class.
  - Use Laplace smoothing to handle unseen words.
  - Return the class with the highest computed score.

3. Results

After training and testing the classifier, I observed the following result:

Accuracy:

```
Accuracy: 96.83% (61/63)
```

Sample Predictions:

```
Text: great computer lightweight extremely powerfulportable fast...
Actual: Positive, Predicted: Positive

Text: disappointed thought wireless...
Actual: Negative, Predicted: Negative

Text: tablet isnt working...
Actual: Negative, Predicted: Negative

Text: work really well last long charge powerful would recommend g...
Actual: Positive, Predicted: Positive
```

```
Text: really charge quickly light carry...
Actual: Positive, Predicted: Positive
```

The classifier showed strong predictive performance with high accuracy and consistency across examples.

4. Reflections and Insights

This project highlighted the effectiveness of probabilistic classifiers for natural language processing:

- **Simplicity and Efficiency:** Naive Bayes is easy to implement and fast to train, making it ideal for baseline models.
- **Data Preprocessing Impact:** Lemmatization and class balancing significantly affect model accuracy.
- **Probabilistic Thinking:** Understanding the math behind the probability estimates deepens comprehension of text-based decision-making.

This implementation provided valuable experience in building a custom machine learning pipeline from scratch using only core Python and essential libraries.

Code Assignment P3 – part1_decision_tree_classifier.py

1. Problem Description

The goal of this assignment was to implement a simple decision tree classifier that determines the best attribute to split on using entropy and information gain. The task also involved comparing the effectiveness of using a split-based decision tree versus a no-split baseline classifier.

I used the **Iris dataset**, a classical dataset in machine learning containing measurements of different flower species. The target variable is `variety`, representing the flower class.

2. Algorithm Implementation

I implemented two classification models:

- **Split-based Decision Tree Classifier**
  - At each node, I compute the **entropy** of the dataset.
  - I then evaluate the **information gain** for each feature.
  - The feature with the highest gain is chosen as the splitting attribute.
  - The process continues recursively until all data in a node belongs to a single class or no further gain is possible.
- **No-Split Classifier**
  - This model simply returns the most frequent class in the dataset as its prediction for all instances.

3. Results

After applying our implementation, the output was as follows:

Entropy of Dataset:

```
1.5850
```

Information Gain for each attribute:

```
sepal.length: 0.8769
sepal.width : 0.5166
petal.length: 1.4463
petal.width : 1.4359
```

Split-Based Decision Tree Output:

```
 Split: petal.length
  └─ petal.length = 1.4
Setosa
  └─ petal.length = 1.3
Setosa
  └─ petal.length = 1.5
Setosa
  └─ petal.length = 1.7
Setosa
  └─ petal.length = 1.6
Setosa
  └─ petal.length = 1.1
Setosa
  └─ petal.length = 1.2
Setosa
  └─ petal.length = 1.0
Setosa
  └─ petal.length = 1.9
Setosa
  └─ petal.length = 4.7
Versicolor
  └─ petal.length = 4.5
 Split: sepal.length
  └─ sepal.length = 6.4
Versicolor
  └─ sepal.length = 5.7
Versicolor
  └─ sepal.length = 5.6
Versicolor
  └─ sepal.length = 6.2
Versicolor
  └─ sepal.length = 6.0
Versicolor
  └─ sepal.length = 5.4
Versicolor
  └─ sepal.length = 4.9
Virginica
  └─ petal.length = 4.9
 Split: sepal.width
  └─ sepal.width = 3.1
Versicolor
```

```
  └─ sepal.width = 2.5
Versicolor
  └─ sepal.width = 2.8
Virginica
  └─ sepal.width = 2.7
Virginica
  └─ sepal.width = 3.0
Virginica
  └─ petal.length = 4.0
Versicolor
  └─ petal.length = 4.6
Versicolor
  └─ petal.length = 3.3
Versicolor
  └─ petal.length = 3.9
Versicolor
  └─ petal.length = 3.5
Versicolor
  └─ petal.length = 4.2
Versicolor
  └─ petal.length = 3.6
Versicolor
  └─ petal.length = 4.4
Versicolor
  └─ petal.length = 4.1
Versicolor
  └─ petal.length = 4.8
 Split: sepal.length
  └─ sepal.length = 5.9
Versicolor
  └─ sepal.length = 6.8
Versicolor
  └─ sepal.length = 6.2
Virginica
  └─ sepal.length = 6.0
Virginica
  └─ petal.length = 4.3
Versicolor
  └─ petal.length = 5.0
 Split: sepal.length
  └─ sepal.length = 6.7
Versicolor
  └─ sepal.length = 5.7
Virginica
  └─ sepal.length = 6.0
Virginica
  └─ sepal.length = 6.3
Virginica
  └─ petal.length = 3.8
Versicolor
  └─ petal.length = 3.7
Versicolor
  └─ petal.length = 5.1
 Split: sepal.length
  └─ sepal.length = 6.0
Versicolor
  └─ sepal.length = 5.8
Virginica
  └─ sepal.length = 6.5
Virginica
  └─ sepal.length = 6.3
```

```
Virginica
  └─ sepal.length = 6.9
Virginica
  └─ sepal.length = 5.9
Virginica
  └─ petal.length = 3.0
Versicolor
  └─ petal.length = 6.0
Virginica
  └─ petal.length = 5.9
Virginica
  └─ petal.length = 5.6
Virginica
  └─ petal.length = 5.8
Virginica
  └─ petal.length = 6.6
Virginica
  └─ petal.length = 6.3
Virginica
  └─ petal.length = 6.1
Virginica
  └─ petal.length = 5.3
Virginica
  └─ petal.length = 5.5
Virginica
  └─ petal.length = 6.7
Virginica
  └─ petal.length = 6.9
Virginica
  └─ petal.length = 5.7
Virginica
  └─ petal.length = 6.4
Virginica
  └─ petal.length = 5.4
Virginica
  └─ petal.length = 5.2
Virginica
```

No-Split Prediction:

```
Prediction (most common class): Setosa
```

4. Reflections and Insights

This part of the project illustrated how decision trees are built from entropy-based heuristics. The no-split classifier, while computationally trivial, fails to generalize beyond the dominant class. In contrast, the recursive, split-based tree offers greater predictive accuracy and insight into feature importance.

Key takeaways:

- **Entropy and information gain** serve as effective tools for node selection.
- **Recursive structure** allows decision trees to handle complex branching logic.
- **Interpretability** of the resulting tree offers intuitive understanding of decision rules.

This implementation provided a clear foundation in symbolic learning methods and built an appreciation for feature-driven classification logic.

Code Assignment P3 - part2_single_layer_perceptron.py

1. Problem Description

The objective of this assignment was to implement a neural network model with a single hidden layer from scratch using only NumPy, and evaluate how the number of hidden neurons impacts classification performance. The model was trained to classify malignant vs. benign tumors based on the Wisconsin Breast Cancer Diagnostic dataset.

2. Algorithm Implementation

Data Preprocessing:

- The dataset was loaded and cleaned (dropping unnecessary columns).
- Diagnoses were mapped to binary values: 'M' = 1 (malignant), 'B' = 0 (benign).
- Features were normalized using min-max scaling.
- Manual train-test split ensured reproducibility and simplicity.

Model Architecture:

- A feedforward neural network with:
  - 1 input layer (number of features = 30)
  - 1 hidden layer with configurable neuron count
  - 1 output neuron with sigmoid activation
- Forward and backward propagation were implemented manually.
- The network was trained for 2000 epochs using binary cross-entropy loss.

Activation Function:

- Sigmoid for both hidden and output layers.

Training & Evaluation:

- For each configuration of hidden neurons [2, 4, 8, 16], the model was trained 5 times.
- Average test accuracy across runs was computed.

3. Results

```
Hidden Neurons Average Accuracy Comparison
Hidden Neurons: 2  => Average Test Accuracy: 58.60%
Hidden Neurons: 4  => Average Test Accuracy: 61.93%
Hidden Neurons: 8  => Average Test Accuracy: 63.51%
Hidden Neurons: 16 => Average Test Accuracy: 65.09%
```

4. Reflections and Insights

- Increasing the number of hidden neurons generally improves the model's capacity to fit complex patterns but may risk overfitting.
- Smaller networks train faster and are easier to interpret, while larger ones offer greater expressive power.

- Training without any external machine learning libraries gave deep insight into gradient descent, weight updates, and activation behaviors.

This assignment helped reinforce foundational neural network principles and the importance of hyperparameter tuning in deep learning models.