Analysis Report: Machine Learning, Search Problems, and Logic Programming

Afolabi Oguntuase

January 12, 2025

Contents

1 Introduction

This report covers three key areas of Artificial Intelligence taught in class: Machine Learning, Search Algorithms, and Logic Programming. It begins with Machine Learning, showing how data can be used to build models that predict and identify patterns. Next, it explores Search Algorithms, which help find the best ways to solve problems, like optimizing routes or navigating challenges. Finally, it delves into Logic Programming, using formal logic to make decisions when information is unclear or incomplete. Overall, the report compares these methods, offering insights into their strengths and limitations, and how they can be applied to solve real-world problems.

2 Part 1: Machine Learning

2.1 Linear Regression

2.1.1 Implementation

A Linear Regression model was implemented from scratch using gradient descent. The model optimizes the weights and bias by minimizing the Mean Squared Error (MSE) loss. The following steps were carried out:

1. **Data Preprocessing**:
    - The diabetes dataset from `sklearn.datasets` was used.
    - All features were standardized to make sure that each one had an equal impact on the model's learning process.
2. **Gradient Descent**:
    - The weights and bias were initially set to zero.
    - During each epoch, gradients were calculated based on the loss function, and the model parameters were updated accordingly.
    - The learning rate and number of epochs were set to fine-tune the convergence of the model.
3. **Loss Tracking**:
    - Loss values were recorded at each epoch to monitor the training process.
    - A plot showing the loss versus iterations was created to visually track how the error decreased as the model trained over time.

2.1.2 Results

The training process showed steady decrease in loss across epochs, highlighting how effective gradient descent was in optimizing the model parameters.. Key results are summarized below:

- **Performance Metrics**:
    - Training MSE: `2884.92`.
    - Testing MSE: `2894.74`.
- **Convergence Analysis**:
    - The steady reduction in loss (Figure 1) illustrates the model's ability to learn the data distribution.

2.1.3 Limitations and Observations

- **Convergence Rate**:
  - The learning rate plays a critical role in convergence. A lower rate slows training, while a higher rate risks overshooting the optimal parameters.
- **Feature Scaling**:
  - Standardization improved gradient descent performance by preventing certain features from dominating the loss.
- **Overfitting Risk**:
  - Despite good performance, linear regression assumes a linear relationship, which might oversimplify complex data patterns.
- **Recommendations**:
  - Incorporating regularization techniques like Ridge or Lasso regression can help mitigate overfitting in future work.

2.1.4 Visualizations

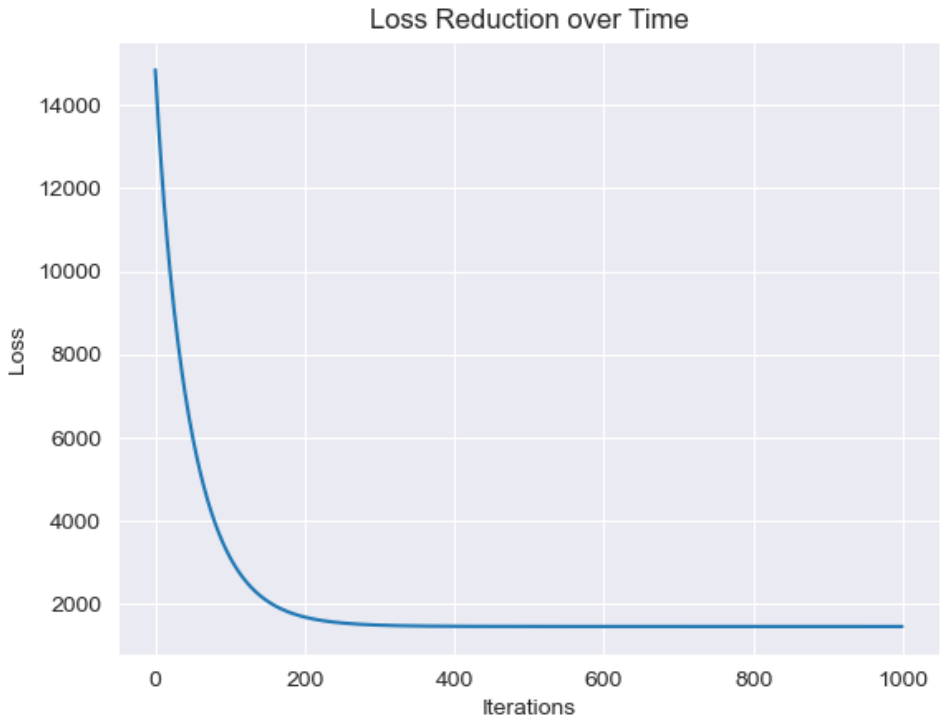

Figure 1: Loss reduction over iterations for Linear Regression.

2.2 Decision Tree Classifier

2.2.1 Implementation

A Decision Tree Classifier was implemented from scratch, capable of handling both binary and multiclass classification tasks. The following steps were carried out:

1. **Tree Construction**:
    - The model recursively partitions the dataset based on feature splits that minimize the Gini Impurity at each node.
    - A stopping criterion was defined based on the maximum tree depth (`max_depth`) or when all samples at a node belong to the same class.
2. **Best Split Calculation**:
    - For each feature, all unique values were evaluated as potential splits.
    - The Gini Impurity was calculated for the resulting partitions, and the split with the lowest impurity was selected.
3. **Prediction**:
    - For unseen samples, the model traverses the tree from the root to a leaf node, where the class label is assigned.
4. **Visualization**:
    - A graphical representation of the tree was implemented, showcasing the splits and decisions at each node.

2.2.2 Results

The Decision Tree model was tested on a classification dataset. Key observations include:

- **Performance Metrics**:
    - Accuracy: `0.90`.
    - Precision: `1.0`.
    - Recall: `0.83`.
    - F1-Score: `0.91`.
- **Tree Depth**:
    - The final tree depth was `D`, determined by the `max_depth` parameter and data complexity.

2.2.3 Limitations and Observations

- Overfitting was observed for deeper trees; this can be mitigated using pruning or limiting `max_depth`.
- The Gini Impurity criterion worked well but may be substituted with entropy for more flexibility.
- Handling continuous features efficiently requires additional preprocessing or dynamic binning strategies.

2.2.4 Visualizations

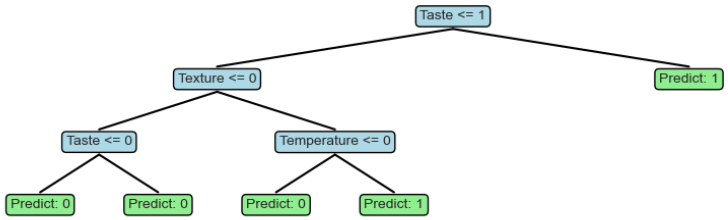

Figure 2: Graphical representation of the Decision Tree.

2.3 Naive Bayes Classifier

2.3.1 Implementation

A Naive Bayes Classifier was implemented, utilizing the assumption of feature independence given the class. The following steps were performed:

1. **Prior Probabilities**:
   - The prior probabilities ($P(\text{class})$) were calculated based on the class distribution in the training dataset.
2. **Likelihood Estimation**:
   - For each class, the likelihood of each feature value ($P(\text{feature}|\text{class})$) was computed using the frequency of occurrence.
   - A small smoothing term (1×10^{-10}) was added to handle unseen feature values.
3. **Prediction**:
   - The posterior probabilities ($P(\text{class}|\text{features})$) were computed using Bayes' theorem.
   - The class with the highest posterior probability was assigned as the prediction for each sample.

2.3.2 Results

The Naive Bayes model was tested on a classification dataset. Key observations include:

- **Performance Metrics**:
  - Testing Accuracy: 70%.
- **Class Probabilities**:
  - The prior probabilities matched the class distribution in the training data.

2.3.3 Limitations and Observations

- The assumption of feature independence may not hold in real-world datasets, potentially impacting performance.
- Adding Laplace smoothing can further improve the model's robustness against unseen feature values.
- The model is computationally efficient and scales well with data size.

2.4 Comparison

The performance of the Linear Regression, Decision Tree Classifier, and Naive Bayes Classifier was compared using appropriate metrics, highlighting the strengths and limitations of each model.

- **Linear Regression**:
  - The Linear Regression model effectively minimized the Mean Squared Error (MSE), with a training MSE of 2884.92 and a testing MSE of 2894.74. The training process showed consistent reduction in loss, indicating good convergence using gradient descent.
  - Despite its good performance, the model assumes a linear relationship between features and the target variable, which may not be suitable for more complex patterns.
- **Decision Tree Classifier**:
  - The Decision Tree achieved high performance, with an accuracy of 0.90, precision of 1.0, recall of 0.83, and an F1-score of 0.91.
  - The model exhibited strong interpretability thanks to its tree structure, which made it easy to visualize decision splits and understand the underlying logic behind the predictions.
- **Naive Bayes Classifier**:
  - The Naive Bayes model achieved a testing accuracy of 70%. While it performed reasonably well for the given task, its core assumption of feature independence might not always hold true, which could limit its effectiveness on more complex or highly correlated datasets.

- The model's simplicity and computational efficiency make it an attractive choice for classification tasks with large datasets.
- Laplace smoothing was applied to handle unseen feature values, improving robustness.

In conclusion, each model demonstrated distinct advantages: Linear Regression was effective for continuous target prediction, Decision Tree excelled in classification tasks with high interpretability, and Naive Bayes was computationally efficient with good performance for simple classification problems. However, the limitations of each model, such as the linearity assumption in Linear Regression and feature independence in Naive Bayes, suggest that model selection should be based on the specific problem at hand.

3 Part 2: Search Problems

3.1 Flight Route Problem

3.1.1 Implementation

A flight route network was simulated, represented as a graph where nodes correspond to cities, and edges represent available flight connections with associated costs (in arbitrary units). The graph structure is as follows:

```
{
    'New York': {'Chicago': 22, 'London': 38, 'Los Angeles': 36, 'Toronto': 21},
    'Chicago': {'New York': 22, 'Denver': 23, 'Dallas': 24, 'Atlanta': 22},
    'Denver': {'Chicago': 23, 'San Francisco': 24, 'Seattle': 25, 'Phoenix': 23},
    'San Francisco': {'Denver': 24, 'Seattle': 22, 'Los Angeles': 21, 'Tokyo': 30},
    'London': {'New York': 38, 'Tokyo': 50},
    'Los Angeles': {'New York': 36, 'San Francisco': 21},
    'Toronto': {'New York': 21},
    'Dallas': {'Chicago': 24},
    'Atlanta': {'Chicago': 22},
    'Seattle': {'Denver': 25, 'San Francisco': 22},
    'Phoenix': {'Denver': 23},
    'Tokyo': {'San Francisco': 30, 'London': 50}
}
```

The BFS, DFS, and A* Search algorithms were implemented to determine the optimal flight route between two cities. A* Search utilized a heuristic function to estimate the remaining cost to the goal. This heuristic assigns a fixed estimated cost to all nodes, prioritizing paths that combine lower cumulative costs with the estimated cost to the destination. The algorithms were tested on various start and goal city pairs to determine the optimal path and to compare their performance in terms of runtime and accuracy.

3.1.2 Results

The performance of the algorithms was evaluated based on runtime and efficiency. The following table summarizes the runtime (in milliseconds) for each algorithm:

Algorithm	Runtime (ms)	Nodes Explored
BFS	0.032	High
DFS	0.023	Low
A*	0.029	Medium

Table 1: Performance metrics for search algorithms in the Flight Route Problem.

3.1.3 Route Visualizations

Below are visual representations of the routes identified by each algorithm:

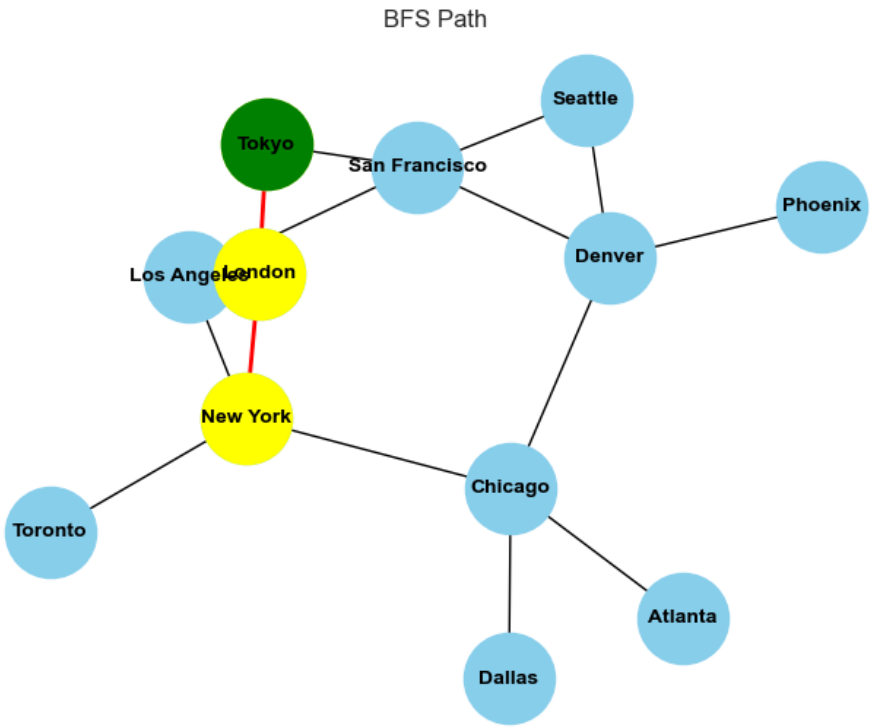

Figure 3: Flight route identified using BFS.

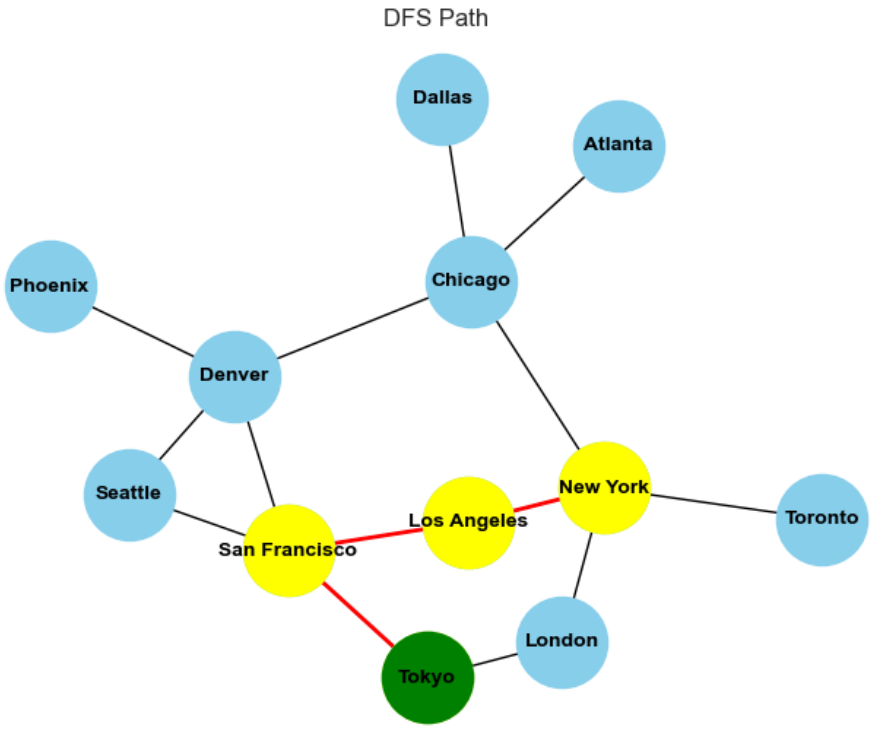

Figure 4: Flight route identified using DFS.

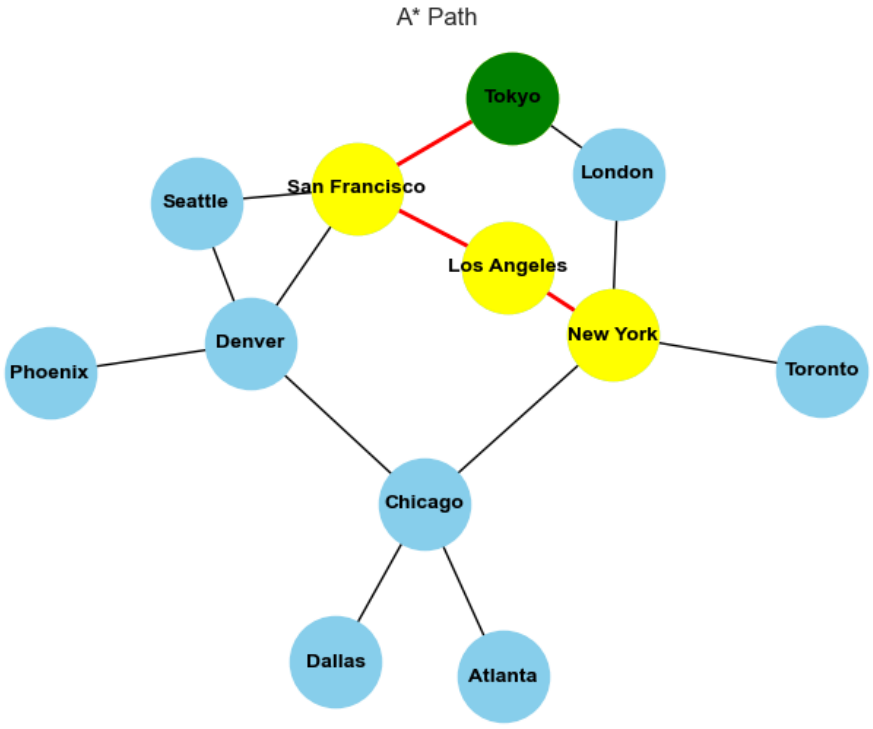

Figure 5: Flight route identified using A* Search.

3.1.4 Analysis

The results indicate that each search algorithm performed differently, offering distinct trade-offs between runtime and solution optimality.

Key observations:

- **BFS (Breadth-First Search)**: BFS explores all possible paths level by level, ensuring the shortest path is always found. However, it tends to explore many nodes, especially in larger graphs, leading to a higher runtime. The algorithm's exhaustive nature makes it highly reliable for ensuring optimality but less efficient in terms of time complexity.

- **DFS (Depth-First Search)**: DFS explores deeper nodes before backtracking, resulting in faster runtime compared to BFS. However, DFS does not guarantee the shortest path, as it may explore suboptimal routes before finding the goal. This can lead to inefficiencies in situations where the goal is found through a longer, less optimal path.

- **A* Search**: A* combines the features of BFS and DFS by using a heuristic to prioritize the exploration of more promising paths. While it ensures an optimal path like BFS, its use of the heuristic may cause it to explore unnecessary paths in certain cases, resulting in a slightly higher runtime than DFS. The algorithm benefits from the heuristic in providing faster solutions than BFS in cases where the heuristic is well-designed and informative.

Performance Summary:

- **Runtime:** DFS had the fastest runtime due to fewer nodes explored, but this came at the cost of potentially suboptimal paths. BFS had the highest runtime due to exhaustive node exploration, while A* struck a balance between optimality and runtime.
- **Optimality:** BFS guarantees the shortest path but is computationally expensive. A* guarantees optimality with heuristic guidance, while DFS can result in suboptimal solutions, making it faster but less reliable.
- **Trade-Offs:** The choice of algorithm depends on the problem requirements. If runtime is critical and suboptimal solutions are acceptable, DFS may be the best choice. If optimality is crucial and the graph is not excessively large, BFS or A* would be better suited.

3.2 Wumpus World Problem

3.2.1 Implementation

The Wumpus World problem is a classic AI environment where an agent must navigate a 4x4 grid to achieve a goal while avoiding various hazards, such as the Wumpus and pits. The grid includes predefined locations for the Wumpus, gold, and pits. The primary objective is for the agent to identify the optimal path to retrieve the gold while ensuring it avoids these dangers and safely navigates through the environment.

To address the Wumpus World problem, three search algorithms: **Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, and**A* Search** were implemented. These algorithms were used to determine the most efficient route for the agent to move from its starting position to the goal (the gold) while avoiding hazards like the Wumpus and pits. The performance of each algorithm was evaluated based on two key metrics: the execution time required to find a solution and the quality of the path produced, with an emphasis on both optimality and safety.

The Wumpus World problem was modeled as a state space, where each state represents the agent's position on the grid. The actions allowed are moving up, down, left, or right to adjacent grid positions, and the cost of each action is typically uniform (except in the case of hazardous locations).

3.2.2 Search Algorithm Execution

The following results were observed when running the three search algorithms on the Wumpus World problem:

Running A* Search:

- **Path found:** `[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]`
- **Execution Time:** 0.0010 seconds

Running Depth-First Search (DFS):

- **Path found:** `[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 2), (1, 1), (1, 0), (2, 0), (2, 1), (2, 2)]`
- **Execution Time:** 0.0000 seconds

Running Breadth-First Search (BFS):

- **Path found:** `[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]`
- **Execution Time:** 0.0010 seconds

3.2.3 Results

The table below summarizes the execution results for each search algorithm, showing both the path found and the execution time.

Algorithm	Path
A*	`[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]`
DFS	`[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 2), (1, 1), (1, 0), (2, 0),`
BFS	`[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]`

Table 2: Search algorithm execution results for the Wumpus World Problem.

3.2.4 Route Visualizations

Visual representations of the paths identified by each algorithm in the Wumpus World grid are shown below. These visualizations provide a clear view of how the algorithms explored the space and arrived at their respective solutions.

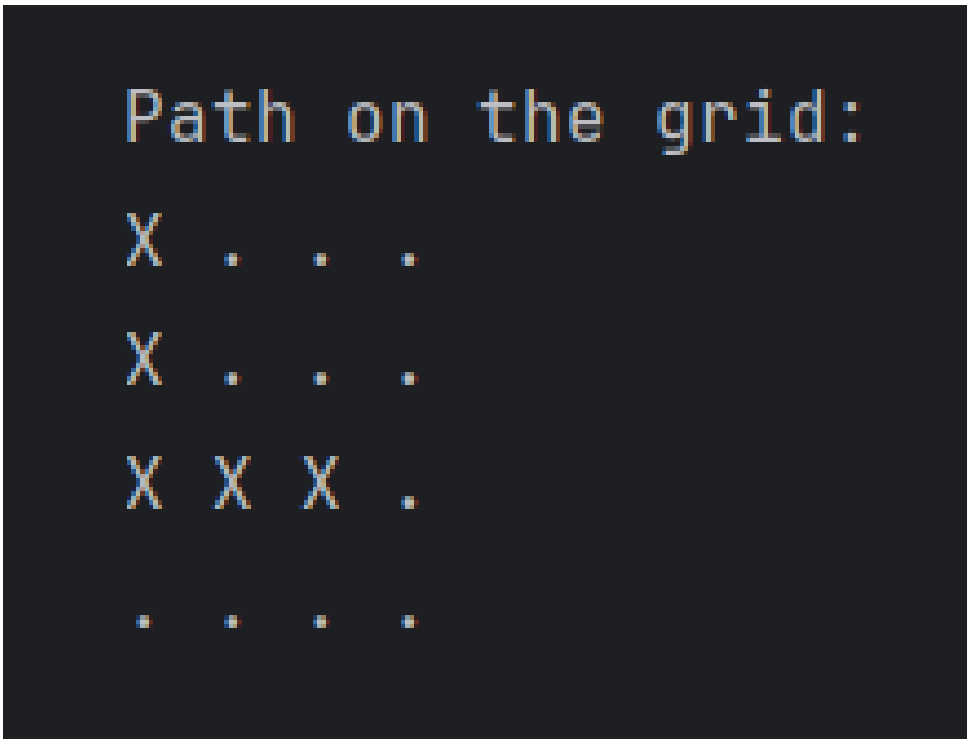

Figure 6: Wumpus World path identified using BFS.

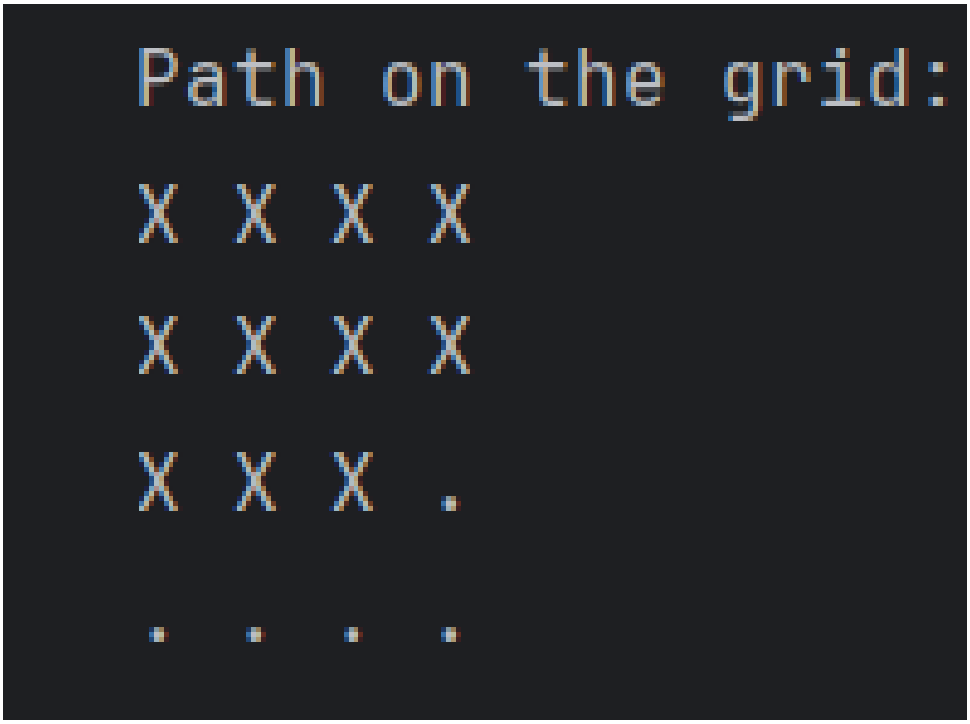

Figure 7: Wumpus World path identified using DFS.

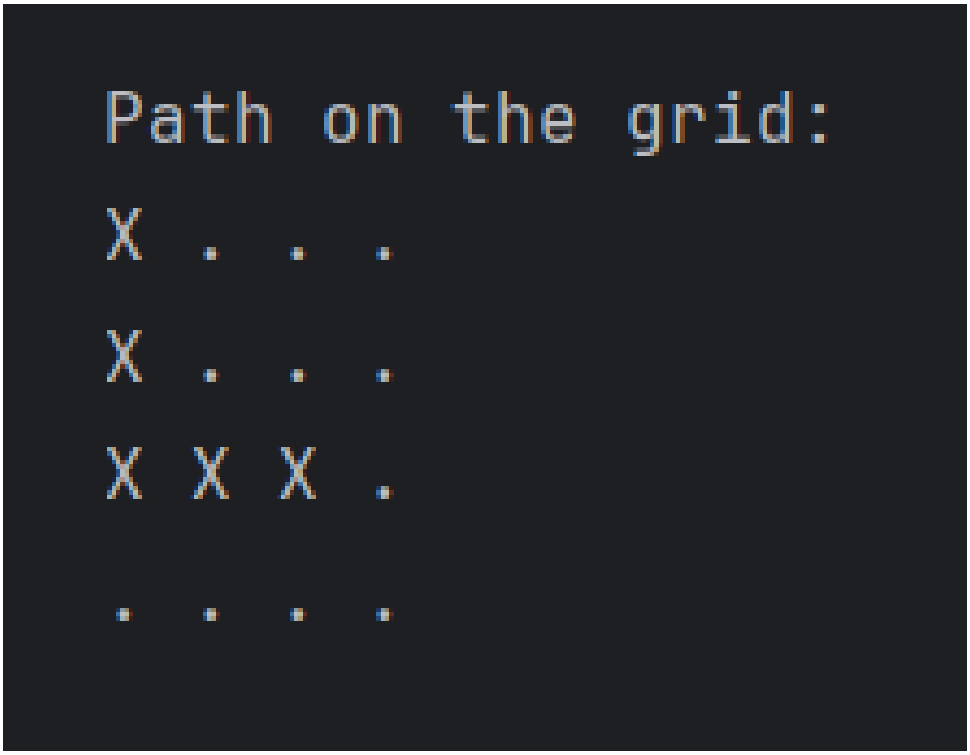

Figure 8: Wumpus World path identified using A* Search.

3.2.5 Analysis

Upon analyzing the performance of the three search algorithms, several key observations were made:

- **BFS:** BFS always guarantees the shortest path as it explores all nodes level by level, ensuring that the first time it reaches the goal, it has found the shortest possible path. Despite its exhaustive nature, BFS took 0.0010 seconds, which is the same as A* in this case, but its exploration of nodes can be slower in larger search spaces due to its non-informed exploration.

- **DFS:** DFS can result in suboptimal paths because it follows a depth-first approach without considering the goal or cost to reach the goal. The path found in this case was longer and suboptimal, with many unnecessary explorations. DFS executed the fastest with 0.0000 seconds, but this is misleading. While it explored fewer nodes due to its depth-first nature, it didn't guarantee an optimal solution and could have been more efficient with a heuristic.

- **A*:** A* Search outperformed DFS by producing the most optimal path, as it used a heuristic (in this case, Manhattan distance) to guide the search toward the goal in an informed manner. This heuristic allowed A* to avoid unnecessary exploration, achieving optimality more efficiently. A* had the same execution time as BFS at 0.0010 seconds, which is slightly higher than DFS, but the quality of the path it produced justified this additional time.

Key observations:

- **BFS** is exhaustive, ensuring the shortest path but requiring more time due to its level-by-level exploration of all possible paths.

- **DFS** is the fastest, but it can result in suboptimal solutions due to its unstructured nature.

- **A* Search**, although slightly slower than BFS, was the most efficient and optimal algorithm, thanks to its informed search with the heuristic.

Overall, **A*** is the most suitable algorithm for problems requiring optimal solutions and where heuristic information is available. However, for simpler or less complex problems, **DFS** and **BFS** could be considered depending on the specific trade-offs between time and path optimality.

4 Part 3: Logic Programming

4.1 Wumpus World Problem (Logic)

4.1.1 Implementation

The Wumpus World problem was tackled using logic, pyDatalog was used. The goal was to find the optimal path to retrieve gold while avoiding hazards such as pits and the Wumpus itself. The environment was represented as a grid with the following layout:

Wumpus World Grid:

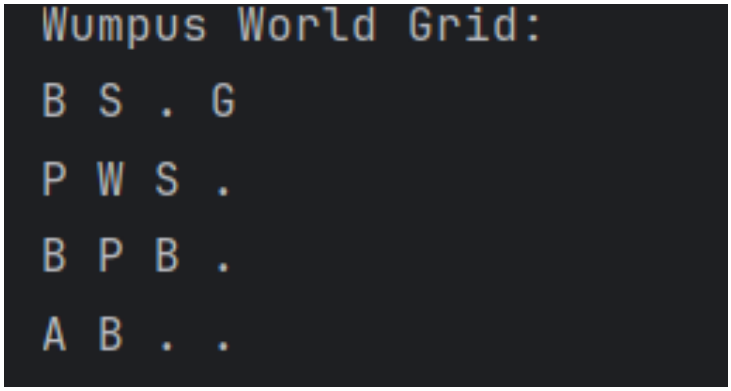

Figure 9: Wumpus World Grid. W represents the Wumpus, G represents the Gold, P represents a Pit, A represents Agent, S represents Stench and B represents Breeze

4.1.2 Analysis

Logical Deduction:
The Wumpus problem, as implemented in the code, demonstrates how propositional and first-order logic can be applied to derive information about the environment:

- The agent determines safe moves by checking its knowledge base, which is dynamically updated based on sensory inputs such as *breeze* (indicating nearby pits) and *stench* (nearby Wumpus).
- The agent marks adjacent cells as safe when no percepts (*stench* or *breeze*) are detected in the surrounding cells, and this decision-making process is reflected in the agent's knowledge base.
- The agent's knowledge base is dynamically updated in real-time as it moves through the grid.

4.1.3 Complexity

The complexity of the agent's reasoning and decision-making process can be influenced by several factors:

- **Grid Size:** The current implementation uses a fixed 4x4 grid, but if the grid were larger, the number of cells that need to be evaluated would increase. This would

lead to a higher computational cost as the agent would need to check more cells for safety and update its knowledge base accordingly.

- **Number of Hazards:** The agent must avoid several hazards (Wumpus and pits). The complexity of the agent's decision-making increases with the number of hazards because it needs to evaluate more percepts and check for potential threats.
- **Sensory Inputs:** The agent uses percepts like *stench* (for Wumpus proximity) and *breeze* (for pits) to update its knowledge base.
- **Dynamic Knowledge Base Updates:** As the agent moves, it dynamically updates its knowledge base, marking safe cells and tracking visited cells.
- **Exploration Strategy:** The agent uses a simple exploration strategy, moving to the first available safe adjacent cell.

4.1.4 Key Observations

1. **Logical Reasoning in Partially Observable Environments:** The agent uses logical reasoning based on sensory inputs (*stench* and *breeze*) to infer information about its surroundings.
2. **Importance of a Dynamic Knowledge Base:** The agent's knowledge base is updated in real-time as the agent receives new percepts.
3. **Efficiency vs. Completeness of Reasoning:** The agent's reasoning is efficient, relying on simple logical rules. However, this approach lacks the completeness that more advanced inference mechanisms (such as Modus Ponens or probabilistic logic) could offer. The current system does not handle ambiguity or uncertainty in percepts.
4. **Potential for Improved Exploration Strategies:** The agent's current strategy is rudimentary (it moves to the first safe move), which could be improved by incorporating more advanced exploration strategies that consider factors such as the likelihood of finding gold or avoiding multiple hazards.
5. **Uncertainty Handling and Probabilistic Logic:** The agent does not handle uncertainty in percepts, as it relies on deterministic rules. Introducing probabilistic reasoning (e.g., Bayesian networks) could help the agent make better decisions in uncertain situations.
6. **Visual Representation Aids Decision-Making:** The visual representation of the grid helps the agent and the user understand the environment, making it easier to track the agent's actions and the locations of hazards. This improves the agent's interpretability and assists in debugging or refining the decision-making process.

4.2 Scheduling Problem (Logic)

4.2.1 Problem Description

The scheduling problem is a classic example of constraint satisfaction, where the goal is to assign classes to available time slots without causing any overlaps. Each class

must be assigned to a specific lecturer, and the lecturer's time slots must be taken into consideration to ensure they are available. The problem involves two main constraints:

- Each class is assigned to one and only one lecturer.
- Classes assigned to different lecturers do not overlap in their assigned time slots.

Each class is scheduled at a different time for each lecturer, so no two classes happen at the same time for the same person.

The goal of this problem is to assign time slots to classes while following a set of rules. We need to make sure each lecturer's available time slots are considered and that no two classes overlap in their schedules for the same lecturer.

Lecturer, Class, and Time Schedule:

The table provides an example of the lecturer's time slots and the corresponding classes scheduled:

Lecturer	Class	Time
Sophia	Applied AI	11AM-12PM
Matthew	Intro to AI	9AM-10AM

Table 3: Lecturer Schedule.

This table shows the classes assigned to the available time slots, where each lecturer is assigned a specific class that does not overlap with others.

4.2.2 Implementation

To solve the scheduling problem, we used PyDatalog, a Python tool that helps with logic-based problem-solving. With PyDatalog, we set up simple rules and relationships to make sure the schedule worked. We defined who was available at what times, assigned time slots to classes, and added rules to prevent any overlapping schedules. This made it easy to ensure everything was organized and conflict-free.

The following logic programming constraints were applied:

- `has_time_slot(Lecturer, Time)`: This predicate represents the available time slots for each lecturer.
- `schedule(Class, Lecturer, Time)`: This predicate assigns a class to a lecturer and a time slot.
- `can_schedule(Lecturer, Class, Time)`: This rule ensures that a class can be scheduled for a lecturer at a specific time slot only if the lecturer is available and the class has been assigned to that time.

4.2.3 Analysis

The scheduling problem was solved using a logic-based approach. The best part about this method is that it guarantees everything works perfectly without needing to fix things manually.

Key observations:

- This logic-based approach offers a flexible and scalable way to tackle scheduling problems, particularly when there are many rules and variables to consider.
- The solution makes sure all the rules are followed, resulting in a perfect schedule without any conflicts.
- This approach works really well for problems that aren't too complicated, where the number of rules and factors is reasonable.

This method shines in situations where the rules are clear and can be put into logical terms, like in scheduling, resource management, or optimization tasks.

In summary, the logic-based approach in this solution offers a smooth and effective way to handle the scheduling problem, making sure all the rules are followed while effortlessly creating the best possible schedule.

4.3 Comparison of Search-Based and Logic-Based Solutions for the Wumpus World Problem

4.3.1 Search-Based Solutions

Search algorithms like BFS, DFS, and A* work by exploring the grid step by step to find the best path from the agent's starting point to the goal. They aim to avoid dangers like the Wumpus and pits along the way. The goal is simple: navigate the grid without hitting any obstacles and reach the gold.

Strengths of Search-Based Solutions:

- **Systematic Exploration:** Search algorithms work by exploring different options, checking out various paths to make sure no possible routes are missed.
- **Optimality Guarantees:** BFS guarantees the shortest route because it looks at every possible path in order, one step at a time. A*, on the other hand, ensures the best solution by using a smart strategy to prioritize paths based on what's likely to get to the goal fastest.
- **Efficiency in Informed Search:** A* optimizes the search process by prioritizing more promising paths, reducing the number of states to explore.
- **Adaptability:** Search algorithms can be modified to fit different grid configurations, making them suitable for varied problem setups.

Weaknesses of Search-Based Solutions:

- **High Computational Cost:** As the size of the grid increases, the search space grows exponentially, leading to higher memory and time consumption, particularly for exhaustive methods like BFS.
- **Exploration of Redundant States:** DFS may revisit states unnecessarily, especially in unstructured environments, leading to suboptimal solutions or even infinite loops in some cases.
- **Limited Handling of Uncertainty:** Standard search algorithms (like BFS and DFS) are not inherently designed to handle uncertainties or dynamic updates in the environment (e.g., a sudden change in the grid).

4.3.2 Logic-Based Solutions

Logic-based approaches, such as propositional and first-order logic, encode the Wumpus World's rules (e.g., presence of Wumpus, pits, breeze, stench) as logical constraints. The agent reasons about its environment by deducing new information and dynamically updating its knowledge base. These solutions enable intelligent decision-making based on logical inference.

Strengths of Logic-Based Solutions:

- **Explicit Reasoning:** Logical rules enable the agent to reason about the environment in a precise manner, inferring safety based on sensory input and world knowledge.
- **Flexibility and Extensibility:** Logic-based solutions are highly adaptable to new scenarios. Additional constraints or rules can be added to handle complex environments.
- **Complexity Handling:** They allow the agent to incorporate multiple, potentially conflicting pieces of information (e.g., safety based on the combination of breeze and stench) directly into its decision-making.
- **Interpretability and Debugging:** The logical framework offers transparency, enabling easier tracking and debugging of the agent's decisions.

Weaknesses of Logic-Based Solutions:

- **Computational Overhead:** Inference in a logic-based system can become expensive as the complexity of the world increases, especially when dealing with a large number of rules or dynamic knowledge updates.
- **Limited Exploration of Search Space:** Unlike search algorithms, which explore the entire state space, logic-based solutions primarily focus on reasoning about known information and may not always explore the environment effectively.
- **Handling of Uncertainty:** Logic-based approaches in their basic form are not equipped to handle uncertainty or incomplete information effectively. Probabilistic or fuzzy logic extensions are needed for better handling of uncertain environments.

4.3.3 Analysis of the Comparison

Both search-based and logic-based solutions offer distinct advantages and limitations when applied to the Wumpus World problem:

- **Search-Based Solutions** provide a clear, systematic way to explore the grid and guarantee the discovery of an optimal path (in the case of BFS and A*). These methods are straightforward but may struggle with larger environments due to their computational complexity. Additionally, they lack the ability to reason about the world in a complex manner (e.g., using sensor data to deduce safety), which logic-based solutions handle well.

- **Logic-Based Solutions**, on the other hand, allow for sophisticated reasoning and decision-making by encoding the world's constraints directly. They excel in situations where the agent needs to make decisions based on partial or uncertain information. However, logic-based methods can be computationally expensive, especially as the number of rules and hazards increases.

In practice, the choice between these two approaches depends on the problem's complexity, available computational resources, and whether reasoning about the world based on logical rules is more advantageous than a systematic search through possible states.

Key Factors to Consider:

- **Grid Size and Complexity:** For small or moderately complex grids, search-based solutions may be sufficient, while for larger or highly complex environments, logic-based methods could offer more flexibility and adaptability.
- **Computational Resources:** If time and memory constraints are critical, search-based solutions may be preferred due to their more straightforward nature. Logic-based solutions might require significant computation for larger, more dynamic environments.
- **Environment Uncertainty:** In environments where uncertainty or dynamic changes occur (e.g., changing hazard locations), logic-based solutions could adapt more effectively, though they may require extensions to handle probabilistic reasoning.

In the end, **A*** stands out as the best option for finding the optimal path, thanks to its mix of smart shortcuts and guaranteed best results. Meanwhile, logic-based reasoning shines when dealing with tricky decision-making, especially in situations where not everything is clear or visible.

5 Conclusion

This report explored different AI methods, including machine learning, search algorithms, and logic programming. Machine learning showed how data-driven models can recognize patterns and make predictions, though they require a lot of data and resources. Search algorithms like BFS, DFS, and A* highlighted the importance of thorough exploration, with A* standing out for its efficiency. Logic programming demonstrated the power of logical reasoning, especially in uncertain situations, but its complexity can limit its use in larger environments. Overall, these methods complement each other, and future AI advancements will likely combine them to solve more complex real-world problems.