

Python Implementation

Python, specifically focusing on decision-making, path planning, and machine learning techniques. Following implementations demonstrate practical applications relevant to real-world autonomous systems:

Example 1: Decision-Making with Greedy Algorithm

The greedy algorithm is a fundamental concept where decisions are made by choosing the best immediate option, with the hope that this leads to an overall optimal solution. This is useful in real-time decision-making, especially in resource-limited autonomous systems.

Problem: Find the minimum number of coins needed to make a certain amount of change.

```
def coin_change(coins, amount):
    coins.sort(reverse=True)
    result = []
    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)
    return result

# Example usage:
coins = [1, 5, 10, 25]
amount = 63
change = coin_change(coins, amount)
print(f"Coins used: {change}")
```

Explanation:

Step 1: The coin denominations are sorted in descending order to prioritise using the largest coin first.

Step 2: The algorithm checks if the current coin can be used by comparing it to the remaining amount.



Step 3: If the coin can be used, it is subtracted from the amount, and the coin is added to the result.

Step 4: This process repeats until the amount is reduced to zero.

Purpose: The greedy approach ensures a solution that minimises the number of coins used.

Example 2: Path Planning with A Algorithm*

Path planning is essential in autonomous systems like robots and self-driving cars, where the goal is to navigate from one point to another while avoiding obstacles. The A* algorithm is a popular path planning algorithm that finds the shortest path based on heuristics.

```
import heapq

def a_star(graph, start, goal):
    queue = []
    heapq.heappush(queue, (0, start))
    costs = {start: 0}
    came_from = {start: None}

    while queue:
        current_cost, current = heapq.heappop(queue)

        if current == goal:
            break

        for neighbor, cost in graph[current].items():
            new_cost = costs[current] + cost
            if neighbor not in costs or new_cost < costs[neighbor]:
                costs[neighbor] = new_cost
                priority = new_cost
                heapq.heappush(queue, (priority, neighbor))
                came_from[neighbor] = current

    return came_from, costs

# Graph representing connections and distances
```

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# Example usage:
start = 'A'
goal = 'D'
path, cost = a_star(graph, start, goal)
print(f"Path: {path}")
```

Explanation:

Step 1: A priority queue (min-heap) is initialised with the start node having a priority of 0.

Step 2: At each step, the node with the lowest cost (priority) is selected.

Step 3: The neighbors of the current node are evaluated, and the total cost to reach them is calculated.

Step 4: If a neighbor's new cost is lower than the previously recorded cost, the path is updated, and the neighbor is pushed onto the priority queue.


Step 5: The algorithm continues until the goal node is reached.

Purpose: A* is used in path planning for autonomous systems as it efficiently finds the shortest path by balancing exploration and cost.

Example 3: Machine Learning for Perception and Control

Machine learning is often employed in autonomous systems for perception tasks, such as object detection. Here's a basic Python implementation of a decision tree classifier for a perception task using the popular **scikit-learn** library.

```
from sklearn.datasets import load_iris
```



```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Load dataset (example: iris dataset)
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data,
data.target, test_size=0.3)

# Train Decision Tree
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Evaluate the model
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

Explanation:

Step 1: Training data consists of features (X_{train}) and corresponding labels (y_{train}), which are provided to the KNN classifier.

Step 2: The KNN algorithm stores the training data and labels.

Step 3: For the new input data point (X_{test}), the algorithm calculates the distance between the test point and each point in the training data.

Step 4: The K nearest neighbors (in this case, 3) are identified based on the shortest distances.

Step 5: The algorithm assigns the most common label among the nearest neighbors to the test point.

Purpose: KNN is a simple and effective machine learning algorithm for classification tasks in autonomous systems.