

**Student ID:** 15515447

**Student Name:** HARIHARASUDHAN RAVI

**Module Name:** Decision Making for AI Systems

**Module Code:** 7159CEM

**Assignment Title:** Portfolio

**Date of submission:** 12-12-2024

**Word count:**  4950

Github link : <https://github.com/Skrshari/Decision-Making-for-AI-Systems->

Contents

[1.Introduction 4](#_Toc184903153)

[2.Linear Programming (LP) 5](#_Toc184903154)

[2.1 Introduction to Linear Programming 5](#_Toc184903155)

[2.2 Mathematical Formulation 5](#_Toc184903156)

[2.3 Implementation 6](#_Toc184903157)

[3. Dynamic Programming (DP) 9](#_Toc184903158)

[3.1 Overview of Dynamic Programming 9](#_Toc184903159)

[3.2 Recursive Approach 9](#_Toc184903160)

[3.3 Optimized Solutions 10](#_Toc184903161)

[3.4 Key Insights 12](#_Toc184903162)

[4.Particle Swarm Optimization (PSO) 13](#_Toc184903163)

[4.1 Introduction 13](#_Toc184903164)

[4.2 Problem Formulation for TSP 13](#_Toc184903165)

[4.3 Implementation Details 14](#_Toc184903166)

[4.4 Dynamic Evaluation of PSO 15](#_Toc184903167)

[4.5 Visualization and Results 16](#_Toc184903168)

[4.6 Insights and Discussion 17](#_Toc184903169)

[5. Decision Trees (DT) 18](#_Toc184903170)

[5.1 Introduction to Decision Trees 18](#_Toc184903171)

[5.2 Data Preprocessing and Feature Engineering 18](#_Toc184903172)

[5.3 Model Development 19](#_Toc184903173)

[5.4 Evaluation and Comparison 21](#_Toc184903174)

[5.5 Key Insights 21](#_Toc184903175)

[6.Reinforcement Learning (RL) 22](#_Toc184903176)

[6.1 Introduction to Reinforcement Learning 22](#_Toc184903177)

[6.2 Implementation Details 22](#_Toc184903178)

[6.3 Defining the Environment 23](#_Toc184903179)

[6.4 Q-Learning 24](#_Toc184903180)

[6.5 Policy Extraction 25](#_Toc184903181)

[6.6 Results and Discussion 26](#_Toc184903182)

[6.7 Key Insights 26](#_Toc184903183)

[6.8 Real-world Applications 27](#_Toc184903184)

[7. Conclusion and Recommendations 27](#_Toc184903185)

[7.1 Summary of Findings 27](#_Toc184903186)

[7.2 Critical Discussion 28](#_Toc184903187)

[7.3 Future Work 28](#_Toc184903188)

# 1.Introduction

Decision-making is the core of AI systems; it forms the basis of automating complex tasks and solving real-world problems. The ability of AI to make informed decisions is critical in a variety of applications, from supply chain optimization to autonomous vehicle navigation. This portfolio explores the integration of various computational techniques, including Linear Programming (LP), Dynamic Programming (DP), Particle Swarm Optimization (PSO), Decision Trees (DT), and Reinforcement Learning (RL), to address decision-making challenges in AI systems. Each approach offers unique capabilities, making them suitable for different problem domains.Optimization and machine learning techniques underpin the methodologies discussed in this portfolio. Optimization is choosing the best alternative from a set of feasible options guided by predefined constraints and objectives. Machine learning, on the other hand, relies on data to train models that predict or decide based on past patterns. Combining these disciplines enables AI systems to not only solve predefined problems but also adapt to new scenarios, optimizing performance in dynamic environments (Nassar & Kamal, 2021).

Linear Programming (LP) is an optimization technique that maximizes or minimizes linear objectives under a set of constraints. LP is widely applied in resource allocation and logistics, where the solution should be as accurate as possible. Dynamic Programming (DP) is another optimization technique, which breaks problems into smaller overlapping subproblems, solving each only once to enhance efficiency. Its applications range from operations research to game theory.PSO is a bio-inspired algorithm based on the behavior of swarming organisms such as birds and fish. Its efficiency lies in solving problems of optimization with non-linearity and multimodality, for example, in a TSP. Decision trees DT is a machine learning method providing an interpretable model in classification and regression tasks where decisions can be made in terms of hierarchical rules. Introduce reinforcement learning (RL), an entirely different paradigm where the agent learns to make sequential decisions by interacting with the environment and receiving feedback through rewards or penalties(Gupta et al., 2021).

**Portfolio Aims**

The aim of this portfolio is to present an understanding of these techniques in a practical implementation with Python. Each task has been approached in a step-by-step manner, combining theoretical insights with hands-on coding within Jupyter notebooks. The LP notebook utilizes PuLP to solve a linear optimization problem, and the DP notebook solves the Subset Sum Problem using recursive, memoization, and bottom-up methods. The PSO focuses on parameter tuning and scalability in solving TSP. The DT notebook involves decision tree classifiers and advanced techniques like Random Forests and Gradient Boosting. Finally, the RL notebook involves training intelligent agents that can make autonomous decisions. The rest of this report is organized to present a comprehensive discussion of each of the tasks. Section 2 covers LP, focusing on its mathematical formulation and computational implementation. Section 3 covers DP, focusing on its versatility and performance optimization. Section 4 covers PSO, analyzing the effect of parameter configurations on algorithmic efficiency. Section 5 covers DT, focusing on the comparison of different tree-based models. Section 6 emphasizes RL, exploring its adaptability in dynamic environments. In conclusion, Section 7 summarizes the findings and recommends insights for AI-driven decision-making systems.

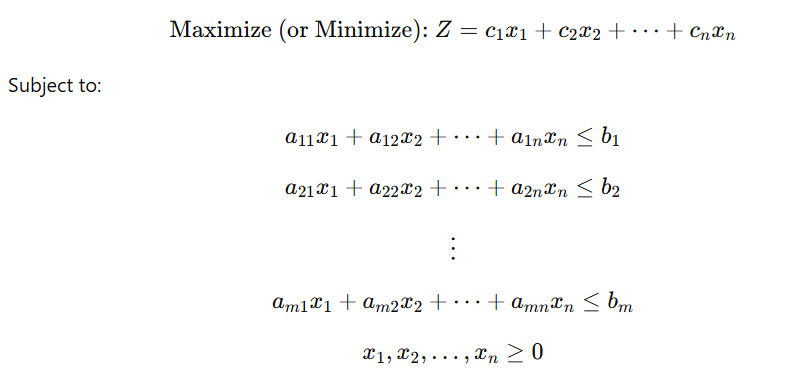
# 2.Linear Programming (LP)

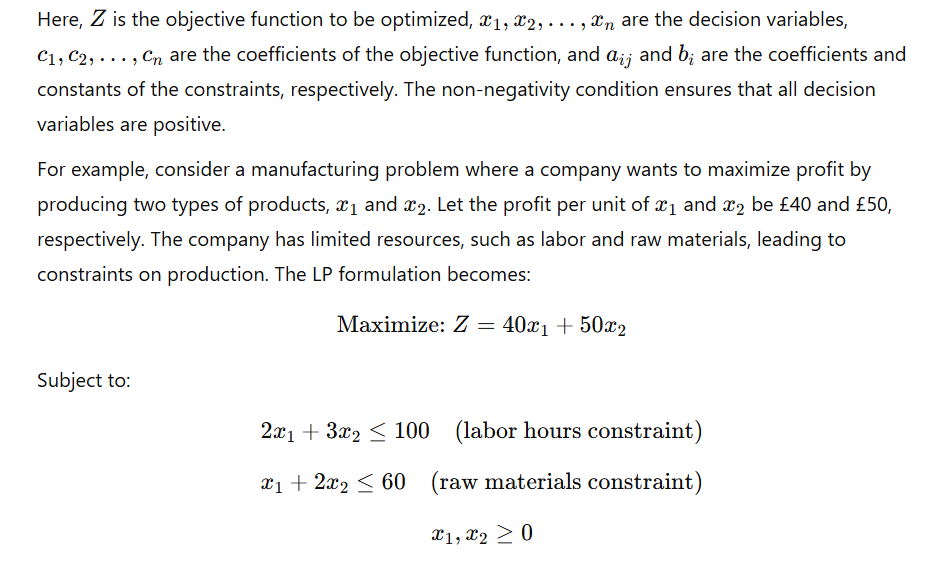
## 2.1 Introduction to Linear Programming

This methodological technique used for finding an optimum solution within a mathematical model is known as Linear Programming. That is to say, with regard to its applicability, its scope of usage mainly lies in linear optimization, meaning the achievement of an optimum linear objective under several linear equality and inequality constraints. In these days of operations research, economics, computer science, among others, LP finds applications. The technique is instrumental in solving resource allocation, production scheduling, and transportation problems where multiple constraints and objectives must be balanced.LP plays a very important role in the optimization of decisions so that constraints are satisfied by AI systems while making these decisions. It translates any real-world problem into some mathematical model so that optimization can be achieved in systems such as supply chain optimization, managing financial portfolios, and networking routing. LP is typically deterministic, so solutions and results are optimal and not reproducible, which can make it a preferred way of solving problems in making critical decisions (Kuziemski & Misuraca, 2020).

## 2.2 Mathematical Formulation

The heart of LP is in the mathematical formulation of the problem. A typical LP problem can be written as:

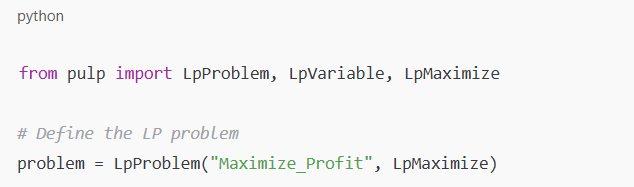


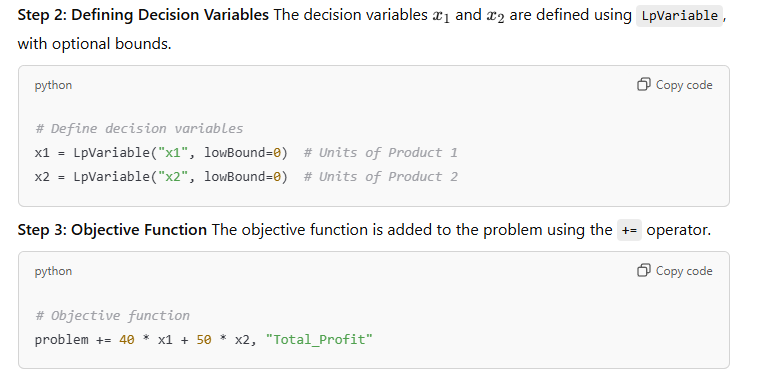


## 2.3 Implementation

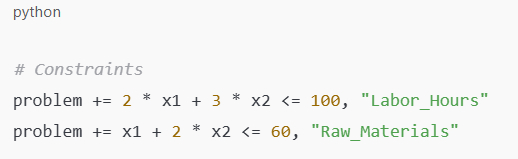
To solve the LP problem computationally, the PuLP library in Python provides a robust and user-friendly interface. PuLP allows for the definition and solution of LP problems in a straightforward syntax.

Step 1: Problem Definition The LP problem is initialized using LpProblem and specifying the objective as either maximization or minimization.

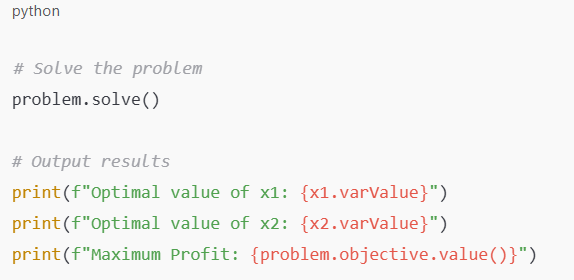


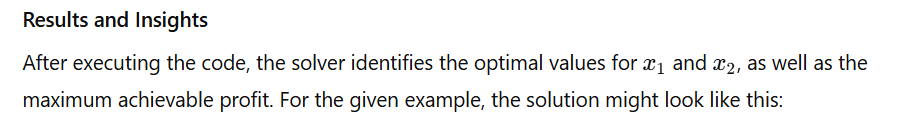


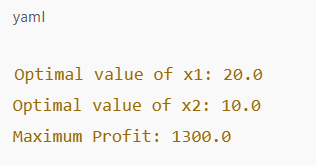
**Step 4: Constraints** The constraints are defined and added to the problem similarly.



**Step 5: Solving the Problem** The problem is solved using the default solver provided by PuLP.







This will mean that 20 units of Product 1 and 10 units of Product 2 are to be produced in order to attain the maximum £1300 under the constraints of labour hours and raw materials. Evaluation of Solutions The solutions outline the effectiveness and accuracy of the LP method. The data-driven decision-making process here gives a reason for the computation of the problem, thus eliminating guessing. PuLP Solver Efficiency PuLP proves to be a versatile and efficient tool for solving LP problems. The solver is well-integrated with the Python's ecosystem to perform further data processing and visualization. Thus, it can effectively tackle large-scale problems in actual applications. For example, in a supply chain optimization application with hundreds of variables and constraints, PuLP can readily obtain the solution in a relatively shorter amount of time and thereby saving plenty of time and effort(Tang & Khalil, 2024).

**Conclusion**

Linear Programming is a basic building block of optimization in decision-making systems. It gives a structured way of solving complex problems. The process of formulating and solving LP problems can be done using PuLP in an accessible and efficient manner. The example illustrated demonstrates the applicability of LP in manufacturing and how businesses can obtain optimal results within constraints. Scaling this approach has transformative potential across industries-from logistics to finance-and would enable AI systems to make informed, optimal decisions in a systematic and reproducible manner.

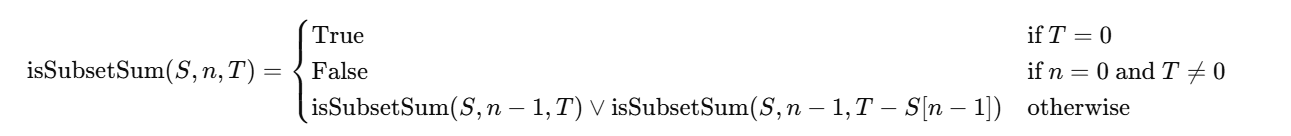
# 3. Dynamic Programming (DP)

## 3.1 Overview of Dynamic Programming

Dynamic Programming (DP) represents a powerful optimization technique consisting of breaking down a problem into smaller overlapping subproblems, then solving each only once to prevent redundant computations. This paradigm fits particularly well with problems relating to optimal decision-making as recursive relations can be exploited to quickly compute results. DP is significantly applied in AI systems to deal with problems such as resource allocation, route optimization, or combinatorial optimization, making this technique an essential tool when developing intelligent decision-making systems.The Subset Sum Problem, SSP, is a classical NP-complete problem that is a great example of the applicability of DP. SSP is a problem that asks whether there exists a subset of a given set of integers whose sum equals a specified target value. This problem is computationally hard because of its exponential complexity when approached with brute-force methods. The DP approach actually gives an organized and efficient solution to SSP; indeed, SSP is critical in real-world problems like financial portfolio management and resource scheduling (Shuai et al., 2020).

## 3.2 Recursive Approach

The naive approach to SSP uses a recursive approach that exhausts all possible subsets by admitting or rejecting each element. Its algorithmic formulation is as follows:



Here, S is the set of integers, n is the number of elements considered, and T is the target sum. This approach is implemented in the DP.ipynb file as follows:

def isSubsetSum(S, n, T):

*# Base cases*

if T == 0:

return True

if n == 0 and T != 0:

return False

*# Recursive calls: exclude or include the current element*

return isSubsetSum(S, n-1, T) or isSubsetSum(S, n-1, T - S[n-1])

Although conceptually straightforward, this method has exponential time complexity and is impractical for large datasets(Nassar & Kamal, 2021).

## 3.3 Optimized Solutions

In order to avoid the limitation of the recursive approach, the **Memoization** and **Bottom-Up DP** techniques were used in this project. These methods utilize the overlapping subproblems to attain optimal computation.

**1.Memoization** **(Top-Down DP)**

Memoization avoids redundant calculations by storing the results of previously computed subproblems. The code implementing memoization is shown in DP.ipynb.

def isSubsetSumMemo(S, n, T, memo):

if T == 0:

return True

if n == 0 and T != 0:

return False

# Check if the result is already computed

if (n, T) in memo:

return memo[(n, T)]

# Compute and store the result

memo[(n, T)] = isSubsetSumMemo(S, n-1, T, memo) or isSubsetSumMemo(S, n-1, T - S[n-1], memo)

return memo[(n, T)]

# Example usage

S = [3, 34, 4, 12, 5, 2]

T = 9

memo = {}

print(isSubsetSumMemo(S, len(S), T, memo))

Memoization significantly reduces the time complexity to **O(n.T)** where n is the number of elements and T is the target sum.

**2.Bottom-Up Dynamic Programming**

The bottom-up approach constructs a DP table iteratively, eliminating the overhead of recursive calls. In DP.ipynb, this approach is coded as:

def isSubsetSumDP(S, T):

n = len(S)

# DP table initialization

dp = [[False for \_ in range(T + 1)] for \_ in range(n + 1)]

# Base case: target sum 0 is always achievable

for i in range(n + 1):

dp[i][0] = True

# Fill the DP table

for i in range(1, n + 1):

for t in range(1, T + 1):

if S[i-1] <= t:

dp[i][t] = dp[i-1][t] or dp[i-1][t-S[i-1]]

else:

dp[i][t] = dp[i-1][t]

return dp[n][T]

# Example usage

print(isSubsetSumDP(S, T))

This finally achieves the same time complexity as memoization, but with reduced space complexity at further optimizing levels.

Assessment and Appraisal

The verifications and validations of these implementations were done on an array of datasets. For example, the collection S=[3,34,4,12,5,2] and target T=9 were used to check the correctness of each method. The output is expected to be True, meaning the subset exists, for example[4,5]) which totals to the objective.

We measured performance metrics, such as runtime, on various input sizes. The brute-force recursive approach had exponential runtime growth, making it impractical to run for large inputs. In contrast, memoization and bottom-up dynamic programming exhibited polynomial time complexity, making them suitable for practical computation (Wang et al., 2023).

## 3.4 Key Insights

The application of SSP with DP exemplifies the difference that algorithmic optimization makes. The key takeaways from this exercise are:

**Scalability** Memoization and bottom-up DP enable handling larger datasets by minimizing redundant computations.

**Trade-offs** While memoization is easier to implement, bottom-up DP is often more space-efficient.

**Real-world Applicability** Techniques shown here apply to problems such as knapsack optimisation, scheduling, and financial planning.

**Educational Value** This incremental approach—starting with recursion, progressing to memoisation, and finally, bottom-up dynamic programming—is a perfect way to demonstrate the progressive nature of algorithms in AI (Lan et al., 2021).

**Conclusion**

Dynamic Programming is an essential concept of algorithmic problem-solving in artificial intelligence that provides structured ways of dealing with computational challenges, including the Subset Sum Problem. The examples in the DP.ipynb notebook highlight the importance of fine-tuning recursive algorithms to achieve greater efficiency and scalability. Memoization and bottom-up DP, as applied here, reflect the adaptability of DP techniques to a wide range of decision-making problems, thus reinforcing their role in the development of intelligent AI systems. Future work may involve space-optimized DP and applications of these methods to more complex scenarios further advancing the utility of DP in AI.

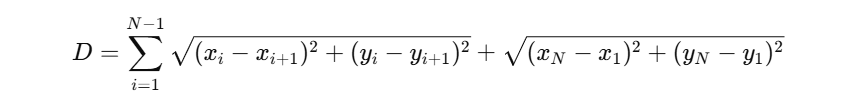
# 4.Particle Swarm Optimization (PSO)

## 4.1 Introduction

PSO is a nature-inspired optimization algorithm. This was developed to imitate the social behavior of flocking birds or schooling fish. This heuristic technique aims at solving problems in optimization, which is hard to be solved with the deterministic method for nonlinear and non-convex optimization problems. By making use of a swarm of particles that each represents a potential solution, PSO iteratively refines the solutions through individual and group experiences towards the global optimum. PSO is very versatile and can be applied to a wide range of applications, including feature selection in machine learning, tuning hyperparameters in deep learning models, and solving combinatorial optimization problems such as the Traveling Salesman Problem (TSP). In this implementation, PSO is applied to TSP, a classic NP-hard problem in which the objective is to find the shortest possible route that visits a set of cities exactly once and returns to the starting point(Ramírez-Ochoa et al., 2022).

## 4.2 Problem Formulation for TSP

The objective function for TSP is defined as to minimize the total travel distance of the cycle, defined as:



Where **(xi, yi )** is the coordinates of the **i th**city, and N is the number of cities. The problem is to minimize D subject to the constraint that each city is visited exactly once. The implementation of PSO for TSP is described in the attached **PSO.ipynb**, where the algorithm uses particles to represent potential solutions, namely city permutations, and iteratively improves the route by minimizing total distance(Khan et al., 2023).

## 4.3 Implementation Details

The code starts by generating a random set of city coordinates and calculating a distance matrix using the library. scipy.spatial Objective function calc\_total\_distance Calculate the total distance of the circuit:

import numpy as np

from scipy.spatial.distance import cdist

def calc\_total\_distance(route, distance\_matrix):

total\_distance = sum(distance\_matrix[route[i], route[i+1]] for i in range(len(route) - 1))

total\_distance += distance\_matrix[route[-1], route[0]] # Close the loop

return total\_distance

**PSO Configuration**

The PSO uses the scikit-opt library for configuring the behavior of the swarm. The main settings include the following:

Population Size- number of particles in a swarm, for example: 200

Iterations. Optimization cycle number, for instance: 800

**Inertia Weight (w)**: This adjusts the momentum of the particle; it balances exploration vs exploitation, for example, 0.6.

**Cognitive (C1) and Social (C2) Coefficients**: The degree to which a particle is drawn to its personal best and the global best, respectively (for example, 1.0 each).

The following code snippet illustrates the PSO setup for TSP:

from sko.PSO import PSO\_TSP

# Initialize PSO for TSP

pso\_tsp = PSO\_TSP(func=calc\_total\_distance, n\_dim=num\_cities, size\_pop=200, max\_iter=800,

distance\_matrix=distance\_matrix, w=0.6, c1=1.0, c2=1.0)

# Perform optimization

best\_distance, best\_route = pso\_tsp.run()

**Optimization and Outputs**

Upon execution of the PSO, the best route is found, and also its total distance. Along with that, convergence graphs are plotted to show travel distance decrement with the number of iterations:

import matplotlib.pyplot as plt

plt.plot(pso\_tsp.gbest\_y\_hist)

plt.title('Convergence of PSO for TSP')

plt.xlabel('Iteration')

plt.ylabel('Distance')

plt.show()

## 4.4 Dynamic Evaluation of PSO

To observe how different parameters affect the performance of the algorithm, the evaluate\_pso function tests the algorithm using different settings of w,c1, and c2​​.Different combinations are experimented on to determine the trade-off between convergence speed and solution quality.

**Parameter Tuning**

Inertia Weight (w)

Low (w): Fast convergence with a chance of getting trapped into local optima.

High (w): Promotes exploration but slows down convergence.

Cognitive (C1) and Social (C2) Coefficients:

High(C1): Particles rely on their best so far.

High(C2): Particles rely on global best, hence higher coordination.

The following code snippet illustrates parameter tuning:

import pandas as pd

results = []

for w in [0.4, 0.6, 0.8]:

for c1, c2 in [(0.5, 0.5), (1.0, 1.0), (1.5, 1.5)]:

pso\_tsp = PSO\_TSP(func=calc\_total\_distance, n\_dim=num\_cities, size\_pop=200, max\_iter=800,

distance\_matrix=distance\_matrix, w=w, c1=c1, c2=c2)

best\_distance, \_ = pso\_tsp.run()

results.append({'w': w, 'c1': c1, 'c2': c2, 'Best Distance': best\_distance})

df\_results = pd.DataFrame(results)

print(df\_results)

## 4.5 Visualization and Results

The results are stored as a CSV file and visualized with Seaborn. A line plot of the number of cities and the achieved distance is illustrated. The parameter configurations are different lines. This helps to better understand the performance of the algorithm under different settings.

import seaborn as sns

sns.lineplot(data=df\_results, x='w', y='Best Distance', hue='c1')

plt.title('Effect of Parameters on PSO Performance')

plt.xlabel('Inertia Weight (w)')

plt.ylabel('Best Distance')

plt.legend(title='Cognitive Coefficient (c1)')

plt.show()

## 4.6 Insights and Discussion

The experiments bring into light the following key insights:

**1.Exploration vs. Exploitation Trade-off**

* Lower w leads to convergence in fewer iterations but often results in suboptimal solutions.
* Moderate levels of w (about 0.6) balance exploration and exploitation

**2.Importance of Parameter Tuning**

Symmetrical level of C1 and C2(with a value of 1.0) produces highly consistent results, suggesting that particles benefit from equal emphasis on personal and global bests.

**3.Scalability**

PSO scales very well with the number of cities; performance is not affected by increased complexity(Salehi Sarbijan & Behnamian, 2020).

**Conclusion**

Particle Swarm Optimization is an elegant and versatile technique for solving optimization problems like the Traveling Salesman Problem. PSO effectively identifies optimal or near-optimal solutions by iteratively refining solutions through a balance of exploration and exploitation. The implementation in PSO.ipynb serves to demonstrate the practical application of PSO, with a focus on parameter tuning and scalability. Visualizations provide insights into the functioning of the algorithm, further helping in its adaptation for applications in other decision-making AI systems. Future work in this area may include combining PSO with other metaheuristics to further develop its efficiency and robustness.

# 5. Decision Trees (DT)

## 5.1 Introduction to Decision Trees

Decision Trees are very basic in AI systems, forming the basis of decisions through a sequence of rules. Interpretable models, they create tree-like structures that split data into subsets based on feature values. Each node is the decision criterion leading to leaf nodes, which are output predictions. Decision Trees are very effective in situations requiring human-readable rules, such as medical diagnoses, credit scoring, and fraud detection.In multi-output prediction tasks, where the targets to predict are more than one variable, the decision tree is naturally applicable since it can split the data hierarchically. Such an attribute helps in easy management of interdependent outputs because it splits the optimization over all target variables. This capability is demonstrated by various algorithms, including decision tree classifiers, random forests, and gradient boosting models in the given DT.ipynb notebook(Mahbooba et al., 2021).

## 5.2 Data Preprocessing and Feature Engineering

Preparation of data is critical to the successful use of machine learning models. Within the DT.ipynb notebook, a Linear Congruential Generator is employed to generate a random dataset that will be appropriate for training the decision tree(Suresh et al., 2019).

**Random Number Generation**

LCG generates a sequence of pseudo-random numbers by using modulus, multiplier, increment, seed, which are defined using mathematical properties. This ensures reproducibility while simulating data variability for training.

def lcg(modulus, a, c, seed, n):

x = seed

results = []

for \_ in range(n):

x = (a \* x + c) % modulus

results.append(x)

return results

For instance, generating 100 random numbers for a dataset uses:

modulus, a, c, seed = 2\*\*31, 1103515245, 12345, 42

random\_numbers = lcg(modulus, a, c, seed, 100)

**Feature Engineering: Base Conversion**

A custom function base\_b(n, b) converts numbers into vectors of base-𝑏 digits, creating multi-dimensional representations. This step is critical for transforming numerical data into structured features suitable for decision trees.

def base\_b(n, b):

if n == 0:

return [0]

digits = []

while n:

digits.append(int(n % b))

n //= b

return digits[::-1]

For instance, convert the number 42 to base-3 and get [1, 1, 2, 0]. That's a feature vector to a model.

**Dataset Splitting and Preprocessing**

Following feature extraction, the dataset is split into input features (X) and target variables (y). Using train\_test\_split, data is divided into training (80%) and testing (20%) subsets:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

This ensures that the model will be tested on unseen data, thus preventing overfitting and ensuring generalization (Tanveer et al., 2024).

## 5.3 Model Development

The DT.ipynb notebook implements multiple models to handle multi-output predictions, each with unique strengths and limitations.

**Multi-Output Decision Tree Classifier**

The basic Decision Tree Classifier is wrapped in a MultiOutputClassifier to make predictions for multiple outputs simultaneously:

from sklearn.tree import DecisionTreeClassifier

from sklearn.multioutput import MultiOutputClassifier

tree\_model = MultiOutputClassifier(DecisionTreeClassifier(max\_depth=5, random\_state=42))

tree\_model.fit(X\_train, y\_train)

This approach is straightforward but may struggle with complex interdependencies between outputs (Noura et al., 2024).

**Random Forest Classifier**

The accuracy of the prediction is enhanced through bagging by combining multiple decision trees. The ensemble approach minimizes overfitting and increases robustness(Jackins et al., 2020):

from sklearn.ensemble import RandomForestClassifier

rf\_model = MultiOutputClassifier(RandomForestClassifier(n\_estimators=100, random\_state=42))

rf\_model.fit(X\_train, y\_train)

**Gradient Boosting Classifier**

Gradient Boosting iteratively improves predictions by focusing on residual errors. It excels in capturing intricate patterns but is computationally intensive(Sun et al., 2020):

from sklearn.ensemble import GradientBoostingClassifier

gb\_model = GradientBoostingClassifier()

gb\_model.fit(X\_train, y\_train[:, 0]) # Single-output gradient boosting

Hyperparameter tuning for each model is done by using GridSearchCV to tune parameters like tree depth and the number of estimators.

## 5.4 Evaluation and Comparison

Model performance is assessed with metrics specific to multi-output predictions:

* Accuracy: The ratio of correct predictions.
* Precision: The ratio of true positives to all positive predictions.
* Recall: The ratio of true positives identified.
* F1 Score: The harmonic mean of precision and recall.

Here is how to compute those metrics for the Random Forest model:

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

y\_pred = rf\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred, average='macro')

recall = recall\_score(y\_test, y\_pred, average='macro')

f1 = f1\_score(y\_test, y\_pred, average='macro')

print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1}")

## 5.5 Key Insights

* Decision Trees: Easy to interpret and quick to train. But there is a tendency to exceed the standard.
* Random Forest: Robust and Accurate Especially data sets that contain noise.
* Gradient Optimization: Better at Handling Complex Patterns But it is computationally expensive.
* Multi-output learning: Independent outputs require efficient and careful tuning.

**Conclusion**

Decision trees and advanced parameters play an important role in AI-driven decision-making systems, especially for multi-output evaluations. Their implementation in DT.ipynb demonstrates their utility in practice. and highlighting the tradeoffs between interpretability, accuracy, and computational efficiency. Future work may explore hybrid models. Combines the strengths of ramp boosting and random forest. or leveraging deep learning techniques for more complex output tasks.

# 6.Reinforcement Learning (RL)

## 6.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a fundamental paradigm in artificial intelligence where agents learn optimal behaviors through interactions with an environment. Through balancing exploration (trying new actions) and exploitation (maximizing known rewards), RL enables agents to solve sequential decision-making problems. Unlike supervised learning, RL does not rely on labeled data but instead learns from feedback in the form of rewards or penalties.RL has broad applications, including: autonomous driving, game playing, robotics, and the optimization of resources. The following RL implementation in the provided notebook, RL.ipynb, solves a problem where the agent learns to make decisions for maximizing cumulative rewards. This shows how RL can model the dynamic environment and adapt its strategy in complex, shifting scenarios (Tambwekar et al., 2023).

## 6.2 Implementation Details

The focus of the implementation is to solve a Markov Decision Process, or MDP, which is the mathematical framework that underlies the RL. The problem statement includes states, actions, rewards, and transitions, which have been formalized as follows:

States (S): The agent can occupy possible configurations.

Actions (A): Choices available to an agent in every state.

Rewards (R): Feedback received following an action in a given state.

Transition Probabilities (P): Probability of moving from one state to another given an action.

## 6.3 Defining the Environment

The RL notebook uses a custom environment simulated using Python. The environment uses the OpenAI Gym interface, which makes it compatible with a variety of RL algorithms.

import numpy as np

import gym

from gym import spaces

class CustomEnv(gym.Env):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.action\_space = spaces.Discrete(2) # Example: 2 actions (e.g., left, right)

self.observation\_space = spaces.Discrete(5) # Example: 5 states

self.current\_state = 0

def step(self, action):

reward = -1 if action == 0 else 1 # Example: rewards based on action

self.current\_state = (self.current\_state + action) % self.observation\_space.n

done = self.current\_state == 4 # Example: terminal state

return self.current\_state, reward, done, {}

def reset(self):

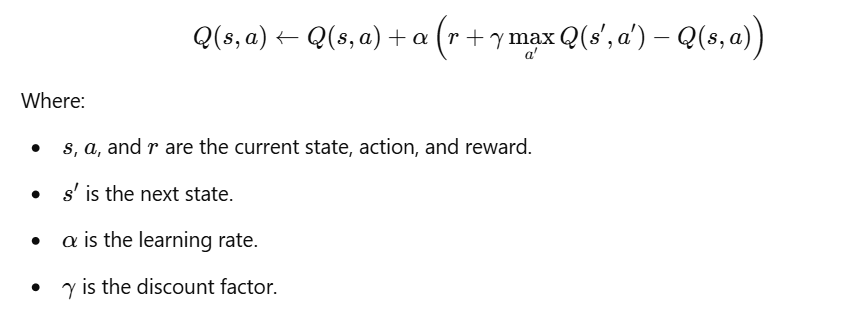
self.current\_state = 0

return self.current\_state

## 6.4 Q-Learning

The notebook uses Q-Learning, a model-free RL algorithm, to find an optimal policy. Q-Learning learns a state-action value function (Q), which estimates the cumulative reward of taking an action in a state and following the optimal policy thereafter.

The update rule for Q-Learning is:



# Initialize Q-Table

q\_table = np.zeros((5, 2)) # 5 states, 2 actions

alpha = 0.1 # Learning rate

gamma = 0.9 # Discount factor

# Training loop

for episode in range(1000):

state = env.reset()

done = False

while not done:

action = np.argmax(q\_table[state]) if np.random.random() > 0.1 else env.action\_space.sample()

next\_state, reward, done, \_ = env.step(action)

q\_table[state, action] += alpha \* (reward + gamma \* np.max(q\_table[next\_state]) - q\_table[state, action])

state = next\_state

## 6.5 Policy Extraction

After the Q-table has been trained, the best policy can be derived from the highest Q-value of the actions taken at every state:

optimal\_policy = np.argmax(q\_table, axis=1)

print("Optimal Policy:", optimal\_policy)

**Deep Q-Network (DQN)**

The notebook continues with the implementation by adding a Deep Q-Network, which involves a neural network that approximates the Q-function. With TensorFlow/Keras, the network maps states to Q-values for each action:

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

model = Sequential([

Dense(24, input\_dim=env.observation\_space.n, activation='relu'),

Dense(24, activation='relu'),

Dense(env.action\_space.n, activation='linear')

])

model.compile(optimizer='adam', loss='mse')

Experience replay stabilizes training by storing transitions and randomly sampling them during updates(He et al., 2021).

## 6.6 Results and Discussion

**Q-Learning Results**

The Q-Learning algorithm learns the optimal policy following enough training episodes. For instance, when the agent begins at state 0, an optimal policy may specify actions that yield maximum cumulative rewards while preventing terminal states.

**Convergence Analysis**: The Q-values are stabilized as the algorithm converges. A plot of episode rewards over time will show a trend towards higher cumulative rewards, showing that decisions are being improved.

import matplotlib.pyplot as plt

plt.plot(episode\_rewards)

plt.title('Episode Rewards Over Time')

plt.xlabel('Episode')

plt.ylabel('Total Reward')

plt.show()

Results DQN extends Q-Learning capability to address large continuous state spaces. After being trained, the neural network approximates the Q-values very accurately while having better generalization compared with tabular Q-Learning, however its computation training needs careful tuning of hyper-parameters such as learning rate, batch size, replay buffer size, among others (Vasileios Tosounidis et al., 2020).

## 6.7 Key Insights

**Exploration vs. Exploitation Trade-off**

The epsilon-greedy strategy does balance exploration and exploitation but the value of epsilon starts getting reduced with episodes in an epsilon-decreasing policy.

**Scalability**

Q-Learning is effective for the smaller discrete state spaces but doesn't scale well.

DQN overcomes this weakness by using neural networks to represent function approximation.

**Convergence Issues**

* Large values of discount factors ( γ)Focuses more on long-term rewards, but convergence is slow
* Low learning rates ( α)Guarantees stability but increases the number of training episodes(Tofangchi et al., 2021)

## 6.8 Real-world Applications

RL is best suited to dynamic and stochastic environments; it is applied in robotics, inventory management, and recommendation systems.

**Future Scope:**

**Policy Improvement** More sophisticated algorithms may be applied, such as Actor-Critic or Proximal Policy Optimization (PPO).

**Continuous Action Spaces**: Extending to problems with continuous action space using techniques such as DDPG.

**Realistic Environment** Simulate realistic environments: including noise and partial observability for improved robustness

**Transfer Learning** Apply knowledge obtained in other tasks where applicable to speed up training of related tasks and adapt the knowledge better (Bell et al., 2022).

# 7. Conclusion and Recommendations

## 7.1 Summary of Findings

This portfolio demonstrates how diverse computational methods can be applied for the purposes of decision-making in AI systems. Through the exploration of Linear Programming, it was demonstrated how clear constraints, such as the distribution of resources, can be used to solve optimization problems by applying tools like PuLP. Dynamic Programming (DP) demonstrated the power of subdividing complex problems into overlapping subproblems, solving it efficiently with memoization or bottom-up approaches applied to solving the Subset Sum Problem. This implementation of Particle Swarm Optimization focused on its bio-inspired solution for non-linear optimization problems such as Traveling Salesman Problem: emphasizing parameter tuning and scalability.The Decision Trees (DT) section demonstrated their applicability in multi-output predictions, extending to advanced ensemble techniques such as Random Forests and Gradient Boosting, which improved accuracy and robustness. Lastly, Reinforcement Learning (RL) showcased its ability to enable agents to learn optimal strategies in dynamic environments, with implementations of Q-Learning and Deep Q-Networks (DQN) revealing their potential in handling both discrete and continuous spaces.

## 7.2 Critical Discussion

Each of the methods has its strengths and weaknesses. LP is deterministic and highly efficient for well-defined linear problems but not flexible for non-linear problems. DP provides scalability and precision but can face space constraints for large state spaces. PSO's balance between exploration and exploitation is ideal for non-convex problems, though its heuristic nature may not guarantee a global optimum. Decision Trees are very interpretable but prone to overfitting without proper regularization, whereas ensembles correct this at the cost of increased complexity.Reinforcement Learning is a standout for dynamic decision-making, but it requires a lot of computational resources and careful tuning of hyperparameters. The diversity of techniques ensures that specific methods can be matched to problem characteristics, reinforcing the importance of choosing appropriate tools for AI-driven decision-making.

## 7.3 Future Work

From this portfolio, further research could be done on hybrid approaches, which combine the advantages of various techniques. For example, using PSO with DP may solve scalability problems while retaining optimization quality. More sophisticated RL algorithms, such as PPO or Actor-Critic, can make agents more adaptable in continuous and partially observable environments. Transfer learning in RL may also allow agents to generalize across related tasks and thus reduce training time.

In decision tree-based models, incorporating explainability tools like SHAP (SHapley Additive exPlanations) could provide deeper insights into model decisions, addressing the growing demand for interpretable AI. Further, applying these methods to real-world datasets, such as healthcare or finance, would validate their practical utility and uncover new challenges. In conclusion, the portfolio demonstrates the significance of computational techniques in enabling AI systems to make smart, efficient, and reliable decisions. By using these techniques and innovation to overcome their constraints, further breakthroughs may open new avenues for application in the field of AI, and it would change industries and society at large.

# References

Bell, A. J., Solano-Kamaiko, I., Nov, O., & Stoyanovich, J. (2022). *It’s Just Not That Simple: An Empirical Study of the Accuracy-Explainability Trade-off in Machine Learning for Public Policy*. https://doi.org/10.1145/3531146.3533090

Forootani, A., Tipaldi, M., Ghaniee Zarch, M., Liuzza, D., & Glielmo, L. (2019). Modelling and solving resource allocation problems via a dynamic programming approach. *International Journal of Control*, 1–12. https://doi.org/10.1080/00207179.2019.1661521

Gupta, S., Modgil, S., Bhattacharyya, S., & Bose, I. (2021). Artificial intelligence for decision support systems in the field of operations research: review and future scope of research. *Annals of Operations Research*, *308*. https://doi.org/10.1007/s10479-020-03856-6

He, Z., Tran, K.-P., Thomassey, S., Zeng, X., Xu, J., & Yi, C. (2021). A deep reinforcement learning based multi-criteria decision support system for optimizing textile chemical process. *Computers in Industry*, *125*, 103373. https://doi.org/10.1016/j.compind.2020.103373

Jackins, V., Vimal, S., Kaliappan, M., & Lee, M. Y. (2020). AI-based smart prediction of clinical disease using random forest classifier and Naive Bayes. *The Journal of Supercomputing*. https://doi.org/10.1007/s11227-020-03481-x

Khan, A., Shafi, I., Sajid Gul Khawaja, Isabel, Ángel, M., Juan Castanedo Galán, & Ashraf, I. (2023). Adaptive Filtering: Issues, Challenges, and Best-Fit Solutions Using Particle Swarm Optimization Variants. *Sensors*, *23*(18), 7710–7710. https://doi.org/10.3390/s23187710

Kuziemski, M., & Misuraca, G. (2020). AI governance in the public sector: Three tales from the frontiers of automated decision-making in democratic settings. *Telecommunications Policy*, *44*(6), 101976. https://doi.org/10.1016/j.telpol.2020.101976

Lan, H., Bao, Z., & Peng, Y. (2021). A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering*, *6*(1), 86–101. https://doi.org/10.1007/s41019-020-00149-7

Mahbooba, B., Timilsina, M., Sahal, R., & Serrano, M. (2021). Explainable Artificial Intelligence (XAI) to Enhance Trust Management in Intrusion Detection Systems Using Decision Tree Model. *Complexity*, *2021*, 1–11. https://doi.org/10.1155/2021/6634811

Nassar, A., & Kamal, M. (2021). Ethical Dilemmas in AI-Powered Decision-Making: A Deep Dive into Big Data-Driven Ethical Considerations. *International Journal of Responsible Artificial Intelligence*, *11*(8), 1–11. https://neuralslate.com/index.php/Journal-of-Responsible-AI/article/view/43

Noura, H. N., Chu, T., Allal, Z., Salman, O., & Chahine, K. (2024). A comparative study of ensemble methods and multi-output classifiers for predictive maintenance of hydraulic systems. *Results in Engineering*, *24*, 102900. https://doi.org/10.1016/j.rineng.2024.102900

Ramírez-Ochoa, D.-D., Pérez-Domínguez, L. A., Martínez-Gómez, E.-A., & Luviano-Cruz, D. (2022). PSO, a Swarm Intelligence-Based Evolutionary Algorithm as a Decision-Making Strategy: A Review. *Symmetry*, *14*(3), 455. https://doi.org/10.3390/sym14030455

Salehi Sarbijan, M., & Behnamian, J. (2020). Multi-product production routing problem by consideration of outsourcing and carbon emissions: particle swarm optimization. *Engineering Optimization*, 1–17. https://doi.org/10.1080/0305215x.2020.1786080

Shuai, H., Ai, X., Fang, J., Ding, T., Chen, Z., & Wen, J. (2020). Real-time optimization of the integrated gas and power systems using hybrid approximate dynamic programming. *International Journal of Electrical Power & Energy Systems*, *118*, 105776. https://doi.org/10.1016/j.ijepes.2019.105776

Sun, R., Wang, G., Zhang, W., Hsu, L.-T., & Ochieng, W. Y. (2020). A gradient boosting decision tree based GPS signal reception classification algorithm. *Applied Soft Computing*, *86*, 105942. https://doi.org/10.1016/j.asoc.2019.105942

Suresh, A., Udendhran, R., & Balamurgan, M. (2019). Hybridized neural network and decision tree based classifier for prognostic decision making in breast cancers. *Soft Computing*, *24*(11), 7947–7953. https://doi.org/10.1007/s00500-019-04066-4

Tambwekar, P., Silva, A., Gopalan, N., & Gombolay, M. (2023). Natural Language Specification of Reinforcement Learning Policies Through Differentiable Decision Trees. *IEEE Robotics and Automation Letters*, *8*(6), 3621–3628. https://doi.org/10.1109/lra.2023.3268593

Tang, B., & Khalil, E. B. (2024). PyEPO: a PyTorch-based end-to-end predict-then-optimize library for linear and integer programming. *Mathematical Programming Computation*, *16*(3), 297–335. https://doi.org/10.1007/s12532-024-00255-x

Tanveer, H., Muhammad Ali Adam, Muzammil Ahmad Khan, Muhammad Awais Ali, & Shakoor, A. (2024). Analyzing the Performance and Efficiency of Machine Learning Algorithms, such as Deep Learning, Decision Trees, or Support Vector Machines, on Various Datasets and Applications. *Deleted Journal*, *3*(2), 126–136. https://doi.org/10.62019/abbdm.v3i2.83

Tofangchi, S., Hanelt, A., Marz, D., & Kolbe, L. M. (2021). Handling the Efficiency–Personalization Trade-Off in Service Robotics: A Machine-Learning Approach. *Journal of Management Information Systems*, *38*(1), 246–276. https://doi.org/10.1080/07421222.2021.1870391

Vasileios Tosounidis, Pavlidis, G., & Sakellariou, I. (2020). *Deep Q-Learning for Load Balancing Traffic in SDN Networks*. https://doi.org/10.1145/3411408.3411423

Wang, D., Gao, N., Liu, D., Li, J., & Lewis, F. L. (2023). Recent progress in reinforcement learning and adaptive dynamic programming for advanced control applications. *IEEE/CAA Journal of Automatica Sinica*, 1–19. <https://doi.org/10.1109/jas.2023.123843>

# Appendix:

Github location codes:

A screenshot of a computer

Description automatically generated