Cuttlefish Algorithm

Members

6410110326 นางสาวปานิสรา สังข์ทอง

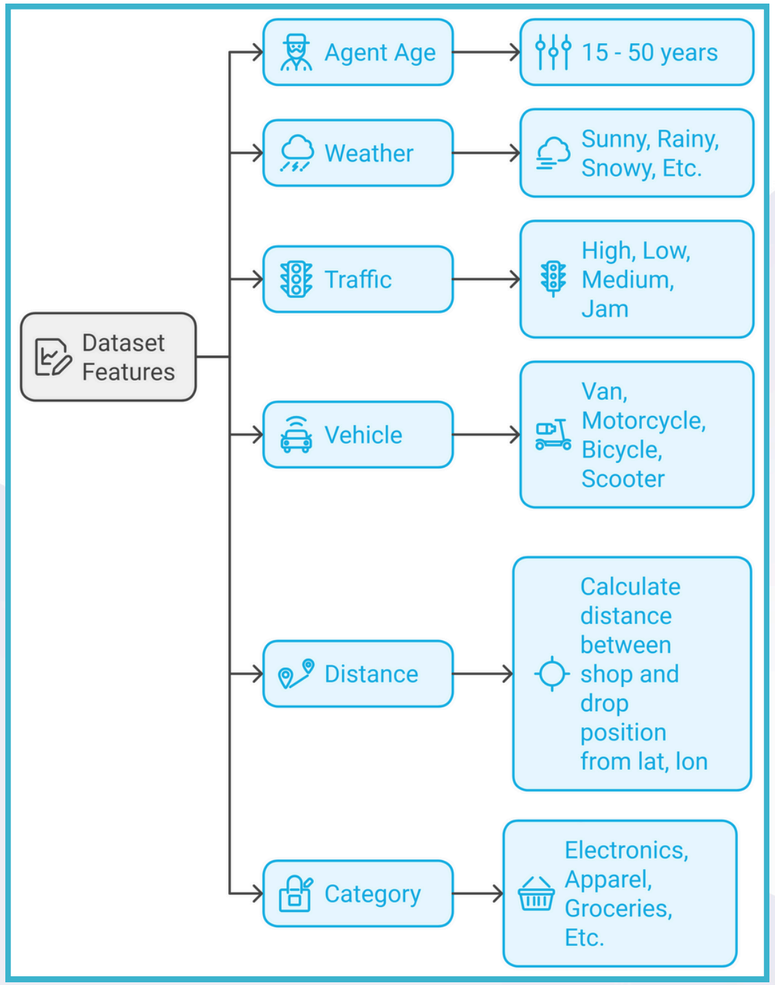
6410110706 นางสาวมาเรียนี สารอเอง

6410110721 นางสาวศศิธร ศักดิ์แก้ว

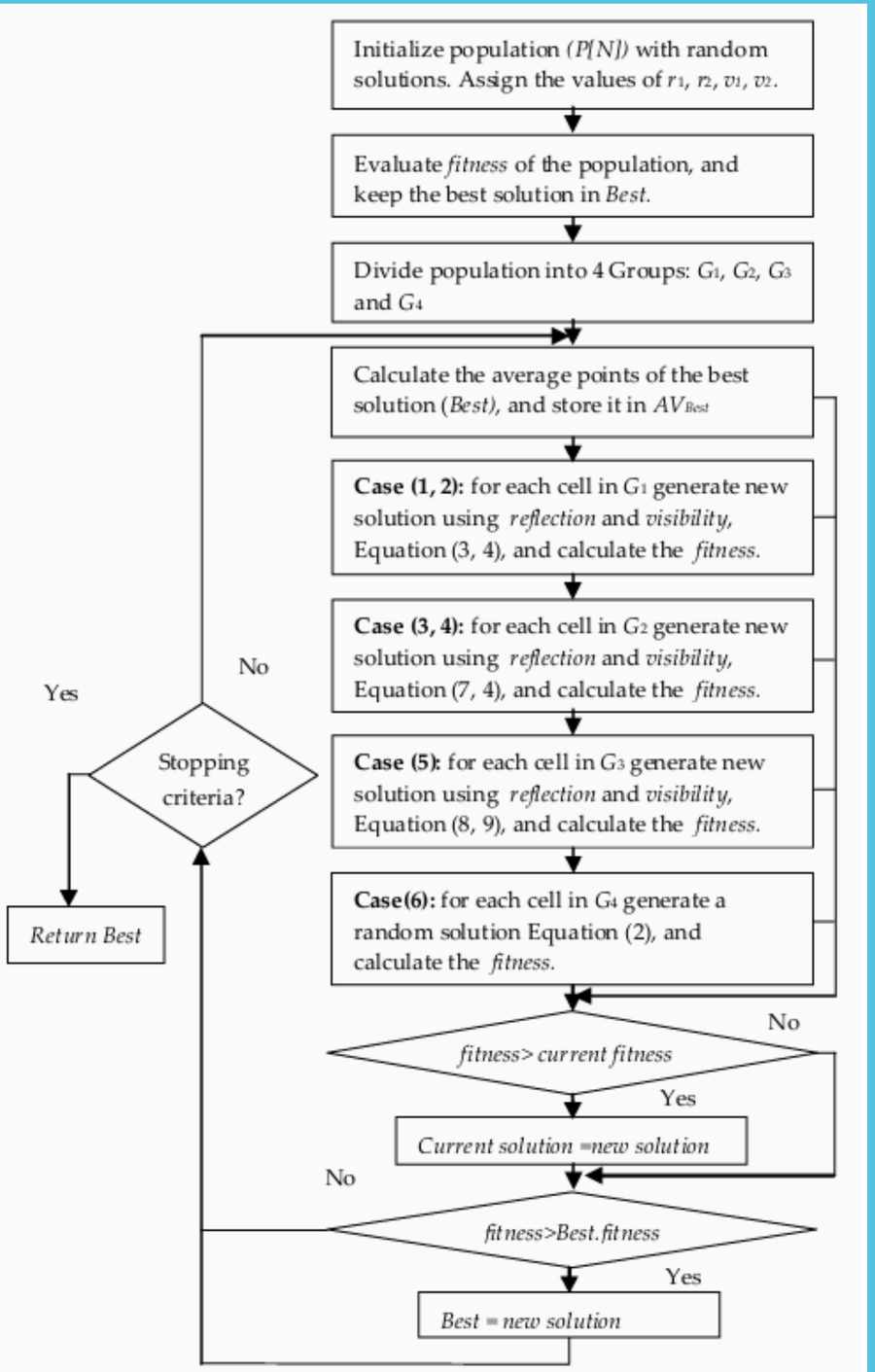
Problems & Dataset

- **Predicting delivery times**

The purpose of this problem is to predict the delivery time in advance from the available data to improve the efficiency of transportation management and delivery planning.



Flowchart of the basic Cuttlefish algorithm



Group 1 Simulation Case 1&2

- **Caculate reflection**

$$\text{reflection}_j = R \cdot G_1[i].Points[j]$$

Equation 3

- **Caculate visibility**

$$\text{visibility}_j = V \cdot (\text{BestPoints}_j - G_1[i].Points[j])$$

Equation 4

Group 2 Simulation Case 3&4

- **Caculate reflection**

$$\text{reflection}_j = R \cdot \text{BestPoint}_j$$

Equation 7

- **Caculate visibility**

$$\text{visibility}_j = V \cdot (\text{BestPoints}_j - G_1[i].Points[j])$$

Equation 4

Group 3 Simulation Case 5

- **Caculate reflection**

$$\text{reflection}_j = R \cdot \text{BestPoints}_j$$

Equation 8

- **Caculate visibility**

$$\text{visibility}_j = V \cdot (\text{BestPoints}_j - AV_{Best})$$

Equation 9

Group 4 Simulation Case 6

$$F[i].points[j] = random^{*}\,(upperLimit - lowerLimit) + lowerLimit$$

$$i = 1, 2, \ldots, N;\ j = 1, 2, \ldots, d$$

Equation 2

Program

- **Import libraries**

```
# import libraries
import pandas as pd
import numpy as np
import random
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from math import radians, cos, sin, asin, sqrt
```

Program (Cont.)

- **Load and prepare data**

```
# Load data (assumed to be in a CSV file named 'amazon_delivery_data.csv')
df = pd.read_csv('amazon_delivery.csv')

# Remove rows where Weature and Traffic is NaN
df = df[df['Weather'] != 'NaN']
df = df[df['Traffic'] != 'NaN ']

# Prepare data
le = LabelEncoder()
for col in ['Weather', 'Traffic', 'Vehicle', 'Category']:
    df[col] = le.fit_transform(df[col])

# Remove rows with zero coordinates
df = df[(df['Store_Latitude'] != 0) & (df['Store_Longitude'] != 0) &
        (df['Drop_Latitude'] != 0) & (df['Drop_Longitude'] != 0)]

def haversine_distance(lat1, lat2, lon1, lon2):
    lon1, lon2, lat1, lat2 = map(radians, [lon1, lon2, lat1, lat2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * asin(sqrt(a))
    r = 3956  # Radius of earth in miles
    return r * c

df['Distance'] = df.apply(lambda row: haversine_distance(row['Store_Latitude'], row['Drop_Latitude'],
                                                         row['Store_Longitude'], row['Drop_Longitude']), axis=1)

# Remove rows where distance > 100
df = df[df['Distance'] <= 100]
```

Program (Cont.)

- **Define objective function**

```
# Define objective function
def objective_function(params):
    params = np.array(params).reshape(-1)  # Reshape params
    y_pred = np.dot(X_train_scaled, params)
    mse = np.mean((y_pred - y_train) ** 2)
    return mse
```

Program (Cont.)

- **Cuttlefish algorithm function**

```python
def cuttlefish_algorithm(objective_func, dim, pop_size, max_iter, lb, ub):
    population = np.random.uniform(lb, ub, (pop_size, dim))
    fitness = np.array([objective_func(ind) for ind in population])
    best_solution = population[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        groups = np.array_split(population, 4)
        r1, r2 = 1, -1
        v1, v2 = 0.5, -0.5

        R = random.random() * (r1 - r2) + r2
        V = random.random() * (v1 - v2) + v2

        # Group 1
        for i in range(len(groups[0])):
            for j in range(dim):
                reflection_j = R * groups[0][i][j]
                visibility_j = V * (best_solution[j] - groups[0][i][j])
                new_position_j = reflection_j + visibility_j
                new_position_j = np.clip(new_position_j, lb[j], ub[j])
                groups[0][i][j] = new_position_j

            new_fitness = objective_func(groups[0][i])
            if new_fitness < fitness[i]:
                population[i] = groups[0][i]
                fitness[i] = new_fitness
                if new_fitness < best_fitness:
                    best_solution = groups[0][i]
                    best_fitness = new_fitness
```

Group 1

Program (Cont.)

- **Cuttlefish algorithm function (Cont.)**

```
# Group 2
for i in range(len(groups[1])):
    for j in range(dim):
        # R = 1
        reflection_j = 1 * best_solution[j]
        visibility_j = V * (best_solution[j] - groups[1][i][j])
        new_position_j = reflection_j + visibility_j
        new_position_j = np.clip(new_position_j, lb[j], ub[j])
        groups[1][i][j] = new_position_j

    new_fitness = objective_func(groups[1][i])
    if new_fitness < fitness[len(groups[0]) + i]:
        population[len(groups[0]) + i] = groups[1][i]
        fitness[len(groups[0]) + i] = new_fitness
        if new_fitness < best_fitness:
            best_solution = groups[1][i]
            best_fitness = new_fitness
```

Group 2

Program (Cont.)

- **Cuttlefish algorithm function (Cont.)**

```
# Group 3
AV_Best = np.mean(groups[2], axis=0)

for i in range(len(groups[2])):
    for j in range(dim):
        # R = 1
        reflection_j = 1 * best_solution[j]
        visibility_j = V * (best_solution[j] - AV_Best[j])
        new_position_j = reflection_j + visibility_j
        new_position_j = np.clip(new_position_j, lb[j], ub[j])
        groups[2][i][j] = new_position_j

    new_fitness = objective_func(groups[2][i])
    index_in_population = len(groups[0]) + len(groups[1]) + i
    if new_fitness < fitness[index_in_population]:
        population[index_in_population] = groups[2][i]
        fitness[index_in_population] = new_fitness
        if new_fitness < best_fitness:
            best_solution = groups[2][i]
            best_fitness = new_fitness
```

Group 3

Program (Cont.)

- **Cuttlefish algorithm function (Cont.)**

```
        # Group 4
        for i in range(len(groups[3])):
            # Randomly assign a new position within bounds using the specified formula
            groups[3][i] = np.random.rand(dim) * (ub - lb) + lb  # Randomly initialize position

            new_fitness = objective_func(groups[3][i])  # Evaluate fitness for the new position
            index_in_population = len(groups[0]) + len(groups[1]) + len(groups[2]) + i
            if new_fitness < fitness[index_in_population]:
                population[index_in_population] = groups[3][i]  # Update population
                fitness[index_in_population] = new_fitness
                if new_fitness < best_fitness:
                    best_solution = groups[3][i]
                    best_fitness = new_fitness

    return best_solution, best_fitness
```

Group 4

Program (Cont.)

- **Prepare data for predict and evaluation**

```
# Select features for prediction
features = ['Agent_Age', 'Weather', 'Traffic', 'Vehicle', 'Distance', 'Category']
X = df[features]
y = df['Delivery_Time']

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


# Set parameters for Cuttlefish Algorithm
dim = len(features)
pop_size = 100
max_iter = 200
lb = np.full(dim, -20)
ub = np.full(dim, 20)

# Use Cuttlefish Algorithm
best_solution, best_fitness = cuttlefish_algorithm(objective_function, dim, pop_size, max_iter, lb, ub)
```

Program (Cont.)

- **Output result**

```python
# Output results
print("Best coefficients for each factor:")
for feature, coef in zip(features, best_solution):
    print(f"{feature}: {coef:.4f}")

print(f"\nMean Squared Error: {best_fitness:.4f}")

# Test the model on the test set
y_pred_test = np.dot(X_test_scaled, best_solution)
mse_test = np.mean((y_pred_test - y_test) ** 2)
print(f"\nMean Squared Error on test set: {mse_test:.4f}")

# Analyze the importance of each factor
importance = np.abs(best_solution)
importance_normalized = importance / np.sum(importance)

print("\nImportance of each factor:")
for feature, imp in sorted(zip(features, importance_normalized), key=lambda x: x[1], reverse=True):
    print(f"{feature}: {imp:.2%}")
```

```
Best coefficients for each factor:
Agent_Age: -3.6989
Weather: -5.4605
Traffic: 15.8902
Vehicle: -19.0301
Distance: 17.9075
Category: 0.6642

Mean Squared Error: 17747.3780

Mean Squared Error on test set: 18881.8171

Importance of each factor:
Vehicle: 30.37%
Distance: 28.58%
Traffic: 25.36%
Weather: 8.72%
Agent_Age: 5.90%
Category: 1.06%
```

Thank You

Cuttlefish Algorithm