پروژه هوش مصنوعی - سپهر فصیحی

فاز اول - حل پروژه با استفاده از الگوریتم ژنتیک :

- **واردات کتابخانه‌ها**

```
import numpy as np
import random
from copy import deepcopy
```

`numpy` برای کار با آرایه‌های چند بعدی.

`random` برای تولید اعداد تصادفی.

`deepcopy` برای ایجاد کپی عمیق از آبجکت‌ها.

- تعریف کلاس `SudokuSolver`

```
class SudokuSolver:
```

این کلاس شامل روش‌ها و توابع لازم برای حل مسئله سودوکو با استفاده از الگوریتم ژنتیک است.

- متد `__init__`

```
    def __init__(self, target, population_size=1000, generations=5000, mutation_rate=0.2):
        self.target = target
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.fixed_positions = self._get_fixed_positions(target)
        self.population = self._initialize_population()
```

```
print("Initialization complete")
```

`target`: سودوکوی اولیه که باید حل شود.

`population_size`: تعداد افراد در جمعیت.

`generations`: تعداد نسل‌هایی که الگوریتم اجرا می‌شود.

`mutation_rate`: نرخ جهش.

`fixed_positions`: موقعیت‌های ثابت در سودوکو که تغییر نمی‌کنند.

`population`: جمعیت اولیه.

- متد `_get_fixed_positions`

```
def _get_fixed_positions(self, board):
    fixed = np.zeros_like(board, dtype=bool)
    for i in range(9):
        for j in range(9):
            if board[i][j] != 0:
                fixed[i][j] = True
    print("Fixed positions identified:\n", fixed)
    return fixed
```

این متد موقعیت‌های ثابت در سودوکو را مشخص می‌کند.

`np.zeros_like(board, dtype=bool)` یک آرایه‌ی بولین با همان اندازه و شکل سودوکو ایجاد می‌کند که تمام خانه‌های آن مقدار `False` دارند.

در صورت غیر صفر بودن مقدار هر خانه، مقدار `True` در آن خانه قرار می‌گیرد.

- متد initialize_population_

```
def _initialize_population(self):
    population = []
    for _ in range(self.population_size):
        individual = self._generate_individual()
        population.append(individual)
    print("Initial population generated")
    return population
```

این متد جمعیت اولیه را ایجاد می‌کند.

با استفاده از متد generate_individual_، افراد جدید تولید و به جمعیت اضافه می‌شوند.

- متد generate_individual_

```
def _generate_individual(self):
    board = np.copy(self.target)
    for i in range(9):
        missing_nums = [num for num in range(1, 10) if num not in board[i]]
        random.shuffle(missing_nums)
        for j in range(9):
            if board[i][j] == 0:
                board[i][j] = missing_nums.pop()
    return board
```

این متد یک فرد جدید ایجاد می‌کند.

np.copy(self.target) یک کپی از سودوکوی هدف ایجاد می‌کند.

برای هر سطر، اعداد گم‌شده را پیدا و تصادفی می‌چینیم و به جای صفرها قرار می‌دهیم.

- متد _fitness

```
  def _fitness(self, board):
      score = 0
      for row in range(9):
          score += len(np.unique(board[row]))
      for col in range(9):
          score += len(np.unique(board[:, col]))
      for box_row in range(3):
          for box_col in range(3):
              box = board[box_row*3:(box_row+1)*3,
box_col*3:(box_col+1)*3]
              score += len(np.unique(box))
      return score
```

این متد امتیاز برازش یک برد را محاسبه می‌کند.

تعداد اعداد یکتا در هر سطر، ستون و بلوک 3x3 را محاسبه و به امتیاز

اضافه می‌کند.

- متد _crossover

```
  def _crossover(self, parent1, parent2):
      child = np.copy(parent1)
      for i in range(9):
          if random.random() > 0.5:
              child[i] = parent2[i]
      return child
```

این متد ترکیب دو والد را انجام می‌دهد.

به صورت تصادفی سطرهای والد دوم را جایگزین سطرهای والد اول

می‌کند.

- متد mutate_

```
def _mutate(self, child):
    for i in range(9):
        if random.random() < self.mutation_rate:
            swap_indices = [j for j in range(9) if not self.fixed_positions[i][j]]
            if len(swap_indices) >= 2:
                idx1, idx2 = random.sample(swap_indices, 2)
                child[i][idx1], child[i][idx2] = child[i][idx2], child[i][idx1]
    return child
```

این متد جهش را روی فرزند اعمال می‌کند.

به صورت تصادفی اعداد غیر ثابت در یک سطر را جابجا می‌کند.

- متد solve

```
def solve(self):
    best_board = None
    best_fitness = 0

    for generation in range(self.generations):
        self.population = sorted(self.population, key=self._fitness,
reverse=True)
        current_best_fitness = self._fitness(self.population[0])
        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_board = self.population[0]

        print(f"Generation {generation}: Best fitness = {best_fitness}")

        if best_fitness == 243:
            print("Optimal solution found!")
            break

        next_generation = self.population[:10]
        for _ in range(self.population_size - 10):
```

```
            parent1 = random.choice(self.population[:50])
            parent2 = random.choice(self.population[:50])
            child = self._crossover(parent1, parent2)
            child = self._mutate(child)
            next_generation.append(child)
        self.population = next_generation

        if generation % 100 == 0:
            print(f"Generation {generation}: Best board so
far\n{best_board}")

    return best_board, best_fitness
```

این متد مسئله سودوکو را حل می‌کند.

در هر نسل، جمعیت براساس امتیاز برازش مرتب شده و بهترین‌ها انتخاب

می‌شوند.

ترکیب و جهش روی والدین اعمال می‌شود تا نسل بعدی ایجاد شود.

هر ۱۰۰ نسل، بهترین بورد تا آن زمان چاپ می‌شود.

اگر امتیاز برازش 243 (بهترین امتیاز ممکن) برسد، حل مسئله متوقف می‌شود.

- **اجرای برنامه**

```
target = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
```

```
])

solver = SudokuSolver(target)
solution, solution_fitness = solver.solve()
print("Solved Suduku :")
print(solution)
print(“Best fitness :", solution_fitness)
```

سودوکوی هدف را تعریف می‌کند.

نمونه‌ای از SudokuSolver را ایجاد و مسئله را حل می‌کند.

سودوکوی حل شده و امتیاز برازش را چاپ می‌کند.

فاز دوم - حل پروژه با استفاده از CSP:

- کتابخانه‌ها:

```
import numpy as np
import random
```

این خطوط کتابخانه‌های `numpy` و `random` را وارد می‌کنند که برای مدیریت آرایه‌ها و تولید اعداد تصادفی استفاده می‌شوند.

- تعریف کلاس `Sudoku`:

```
class Sudoku:
```

این کلاس شامل تمامی متدهای لازم برای تولید، بررسی و حل پازل سودوکو است.

- متد `__init__`:

```
def __init__(self, board=None)
    self.board = np.array(board, dtype=int)
```

این متد سازنده کلاس است. برد ورودی را به عنوان برد فعلی تنظیم می‌کند.

- متد `is_valid`:

```
def is_valid(self, row, col, num):
    for i in range(9):
        if self.board[row][i] == num or self.board[i][col] == num:
            return False
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if self.board[start_row + i][start_col + j] == num:
                return False
    return True
```

این متد بررسی می‌کند که آیا یک عدد می‌تواند در یک خانه خاص قرار گیرد یا خیر. این بررسی شامل ردیف، ستون و جعبه 3x3 است.

- متد `find_empty_location`:

```
def find_empty_location(self):
    for i in range(9):
        for j in range(9):
            if self.board[i][j] == 0:
                return i, j
    return None
```

این متد اولین خانه خالی را پیدا می‌کند و مختصات آن را برمی‌گرداند. اگر خانه خالی پیدا نشود، `None` برمی‌گرداند.

- متد `solve`:

```
def solve(self):
    empty_loc = self.find_empty_location()
    if not empty_loc:
        return True
    row, col = empty_loc

    for num in range(1, 10):
        if self.is_valid(row, col, num):
            self.board[row][col] = num
            if self.solve():
                return True
            self.board[row][col] = 0
    return False
```

این متد پازل سودوکو را با استفاده از الگوریتم backtracking حل می‌کند.

ابتدا یک خانه خالی پیدا می‌کند، سپس اعداد 1 تا 9 را امتحان می‌کند تا ببیند کدام یک معتبر است. اگر هیچ عدد معتبری پیدا نشود، به خانه قبلی بازمی‌گردد و عدد را تغییر می‌دهد.

- متد `display`:

```
def display(self):
    for row in self.board:
        print(" ".join(map(str, row)))
```

این متد برد سودوکو را نمایش می‌دهد. هر ردیف را به صورت یک رشته نمایش می‌دهد.

- **اجرای برنامه:**

```
initial_board = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
])

sudoku_solver = Sudoku(initial_board)
print("Initial Sudoku board:")
sudoku_solver.display()

if sudoku_solver.solve():
    print("\nSudoku solved successfully:")
    sudoku_solver.display()
else:
    print("No solution exists.")
```

در این بخش، برد اولیه تعریف می‌شود، یک شیء از کلاس سودوکو با برد اولیه ایجاد می‌شود، برد اولیه نمایش داده می‌شود و سپس پازل حل می‌شود. اگر حل شد، برد نهایی نمایش داده می‌شود، در غیر این صورت پیام "هیچ راه‌حلی وجود ندارد" چاپ می‌شود.