دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

درس هوش مصنوعی و کارگاه

گزارش ۶: پیاده سازی مسئله n-وزیر با استفاده از الگوریتم تکامل تدریجی

نگارش

کیارش مختاری دیزجی

۹۸۳۰۰۳۲

استاد اول

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

خرداد ۱۴۰۲

چکیده

در این گزارش سعی شده است مسئله n-وزیر با استفاده از الگوریتم ژنتیک که یکی از الگوریتم‌های تکامل تدریجی می‌باشد پیاده سازی شود و در انتها سعی شده است تا با اضافه کردن متدهای دیگر به مسئله مثل روش جستجوی محلی سرعت این الگوریتم را افزایش داد.

واژه‌های کلیدی:

مسئله n-وزیر، الگوریتم تکامل تدریجی، الگوریتم ژنتیک، الگوریتم جستجوی محلی

لینک پروژه:

github project 6

فهرست مطالب

صفحه

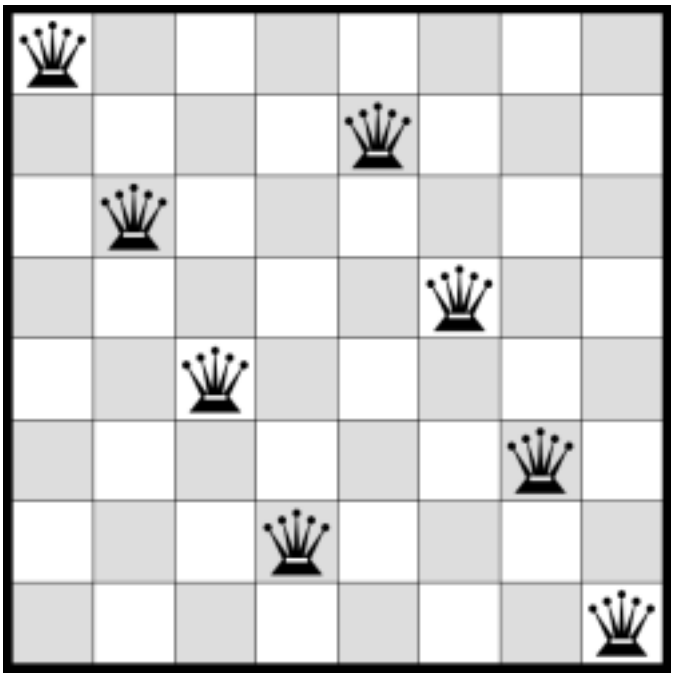
فهرست اشکال

صفحه

۱. فصل اول

مقدمه

مسئله چند وزیر یک معمای شطرنجی و ریاضیاتی است که بر اساس آن باید n وزیر شطرنج در یک صفحه n×n شطرنج به‌گونه‌ای قرار داده شوند که هیچ‌یک زیر ضرب دیگری نباشند. با توجه به اینکه وزیر به‌صورت افقی، عمودی و اُریب حرکت می‌کند، باید هر وزیر را در طول، عرض و قطر متفاوتی قرار داد.



۱- نمونه‌ای از پاسخ مسئله ۸-وزیر

۲. فصل دوم

پیاده سازی مسئله n-وزیر با استفاده از الگوریتم ژنتیک

به طور کلی برای پیاده سازی الگوریتم ژنتیک باید چند تابع که در هر الگوریتم ژنتیکی می‌باشد را متناسب با مسئله که اینجا مسئله n-وزیر می‌باشد پیاده سازی کرد:

۱. **initialize_population**: این تابع برای ساخت یک population اولیه از individualها به صورت رندوم را می‌سازد.

```
def initialize_population(population_size: int, board_size: int) -> list:
    population = []
    for _ in range(population_size):
        individual = [random.randint(0, board_size-1) for _ in
range(board_size)]
        population.append(individual)

    return population
```

۲. **evaluate_fitness**: این تابع تعداد برخوردهای هر وزیر را با وزیرهای دیگر در یک individual چک می‌کند.

```
def evaluate_fitness(individual):
    conflicts = 0
    size = len(individual)

    for i in range(size):
        for j in range(i+1, size):
            if individual[i] == individual[j] or abs(individual[i] -
individual[j]) == abs(i - j):
                conflicts += 1

    fitness = 1 / (conflicts + 1)

    return fitness
```

۳. selection: این تابع individualهایی را که fitness بالاتری دارند را انتخاب می‌کند. متد استفاده شده در اینجا روش tournament_selection می‌باشد که در هر iterate یک تعداد رندوم از individualهای population را انتخاب می‌کند و سپس individualای که بیشترین مقدار fitness را دارد در یک لیست جدید ذخیره می‌کند.

```
def tournament_selection(population, tournament_size):
    selected = []

    for _ in range(len(population)):
        sub_population = random.sample(population, tournament_size)
        winner = max(sub_population, key=evaluate_fitness)
        selected.append(winner)

    return selected
```

۴. crossover:این تابع یک ترکیب جدیدی با استفاده از والدهای انتخاب شده ایجاد می‌کند.

```
def crossover(parent1, parent2):
    size = len(parent1)
    crossover_point = random.randint(1, size - 1)

    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]

    return crossover_point, child1, child2
```

۵. mutation: این تابع تغییرات یا جهش‌های تصادفی را به جمعیت فرزندان معرفی می‌کند. این تنوع که به جمعیت اضافه می‌شود باعث کشف مناطق مختلف فضای راه‌حل می‌شود. برای مسئله n-وزیر، جهش می تواند شامل تغییر تصادفی موقعیت یک ملکه باشد.

```
def mutation(individual, mutation_rate):
    size = len(individual)
    mutated_individual = individual.copy()
    for i in range(size):
        if random.random() < mutation_rate:
            new_position = random.randint(0, size - 1)
            mutated_individual[i] = new_position
    return mutated_individual
```

۶. replacement: این تابع افراد را از جمعیت والدین و جمعیت فرزندان انتخاب می‌کند تا نسل بعدی را تشکیل دهند. معیارهای انتخاب می‌تواند بر اساس fitness باشد، که به افراد با fitness بالاتر اجازه می‌دهد باقی بمانند.

```
def replacement(parent_population, offspring_population, population_size):
    combined_population = parent_population + offspring_population

    combined_population.sort(key=evaluate_fitness, reverse=True)

    next_generation = combined_population[:population_size]

    return next_generation
```

۷. termination_condition: این تابع زمان توقف الگوریتم را تعیین می‌کند. این می‌تواند بر اساس حداکثر تعداد نسل، رسیدن به آستانه fitness مطلوب، یا سطح خاصی از همگرایی باشد. که در این تابع ما زمانی توقف می‌کنیم که به مقدار fitness یک برسیم.

```
def termination_condition(population, max_fitness):
    max_population_fitness = max(evaluate_fitness(individual) for
individual in population)
    return max_population_fitness == max_fitness
```

۸. solve_n_queen: این تابع الگوریتم ژنتیک می‌باشد که با استفاده از توابع قبلی تعریف شده مسئله n-وزیر را حل می‌کند. ابتدا یک population ساخته و سپس تا زمانی که به شرایط توقف الگوریتم نرسد وارد لوپ می‌شود و عملیات crossover، mutation، replacement را به ترتیب انجام می‌دهد و در آخر جواب مسئله به همراه fitness آن را برمیگرداند.

```
def solve_n_queen(population_size, board_size, mutation_rate):
    population = initialize_population(population_size, board_size)
    current_generation = 0

    while not termination_condition(population, 1.0):
        parent_population = tournament_selection(population, 3)
        offspring_population = []

        while len(offspring_population) < population_size:
            parent1, parent2 = random.sample(parent_population, 2)
            crossover_point, child1, child2 = crossover(parent1, parent2)
            mutated_child1 = mutation(child1, mutation_rate)
            mutated_child2 = mutation(child2, mutation_rate)
            offspring_population.extend([mutated_child1, mutated_child2])

        population = replacement(parent_population, offspring_population,
population_size)
        population = [local_search(individual) for individual in
population]
        current_generation += 1

    best_individual = max(population, key=evaluate_fitness)
    best_fitness = evaluate_fitness(best_individual)
    return best_individual, best_fitness
```

۳. فصل سوم

نمونه خروجی پیاده‌سازی و ارزیابی

با اجرا کردن قطعه کد زیر خروجی های مسئله برای الگوریتم ژنتیک را می‌توان دید:

```
def print_individual(individual):
    board_size = len(individual)

    for row in range(board_size):
        line = ""
        for col in range(board_size):
            if individual[row] == col:
                line += "Q "
            else:
                line += "- "
        print(line.strip())

# Example for 8-Queen
population_size = 1000
board_size = 8
mutation_rate = 0.1

start_time = time.time()
best_individual, best_fitness = solve_n_queen(population_size, board_size,
mutation_rate)
end_time = time.time()

execution_time = end_time - start_time

print("8-Queen Problem:")
print("Best Individual:", best_individual)
print_individual(best_individual)
print("Best Fitness:", best_fitness)
print("Execution Time:", execution_time, "seconds")

print("\n")

# Example for 16-Queen
population_size = 100
board_size = 16
mutation_rate = 0.3

start_time = time.time()
best_individual, best_fitness = solve_n_queen(population_size, board_size,
mutation_rate)
end_time = time.time()

execution_time = end_time - start_time
```

```
print("16-Queen Problem:")
print("Best Individual:", best_individual)
print_individual(best_individual)
print("Best Fitness:", best_fitness)
print("Execution Time:", execution_time, "seconds")
```

خروجی نمونه:

```
8-Queen Problem:
Best Individual: [5, 2, 6, 1, 3, 7, 0, 4]
- - - - - Q - -
- - Q - - - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -
- - - - - - - Q
Q - - - - - - -
- - - - Q - - -
Best Fitness: 1.0
Execution Time: 0.7532246112823486 seconds


16-Queen Problem:
Best Individual: [10, 2, 7, 5, 13, 0, 14, 6, 15, 3, 8, 4, 12, 9, 11, 1]
- - - - - - - - - - Q - - - - -
- - Q - - - - - - - - - - - - -
- - - - - - - Q - - - - - - - -
- - - - - Q - - - - - - - - - -
- - - - - - - - - - - - - Q - -
Q - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - Q -
- - - - - - Q - - - - - - - - -
- - - - - - - - - - - - - - - Q
- - - Q - - - - - - - - - - - -
- - - - - - - - Q - - - - - - -
- - - - Q - - - - - - - - - - -
- - - - - - - - - - - - Q - - -
- - - - - - - - - Q - - - - - -
- - - - - - - - - - - - - Q - - -
- Q - - - - - - - - - - - - - -
Best Fitness: 1.0
Execution Time: 315.4497609138489 seconds
```

لازم به ذکر است که در ابتدا با همین توابع که در بخش قبل تعریف شد خروجی مسئله ۱۶-وزیر نزدیک به ۲۰ دقیقه تا ۳۰ دقیقه زمان می‌برد اما پس از اعمال روش local-search و پیاده سازی آن در الگوریتم ژنتیک زمان حل مسئله ۱۶-وزیر را توانستیم به ۵ دقیقه کاهش دهیم.

```
def local_search(individual):
    best_fitness = evaluate_fitness(individual)
    best_individual = individual
    size = len(individual)

    for i in range(size):
        for j in range(size):
            if individual[i] != j:
                new_individual = individual[:]
                new_individual[i] = j
                new_fitness = evaluate_fitness(new_individual)
                if new_fitness > best_fitness:
                    best_fitness = new_fitness
                    best_individual = new_individual

    return best_individual
```

به طور کلی این الگوریتم پیاده سازی چندان مشکلی ندارد و به راحتی قابل پیاده سازی می‌باشد اما پیچیدگی زمانی بسیار زیادی دارد که البته با استفاده کردن از برخی روش‌های بهینه سازی مانند local search که نوعی هیوریستیک است که روش کارش به این صورت است که آیا یک individual در مراحل بعدی می‌تواند مقدار fitness را افزایش دهد یا خیر، می‌توان زمان حل مسئله را بسیار کاهش داد. البته لازم است به این نکته نیز اشاره کرد که مسئله n-وزیر برای n های بزرگ حالات بسیاری دارد و همین موضوع باعث افزایش زمان اجرا الگوریتم نیز می‌شود.

منابع

[1] https://blog.faradars.org/genetic-algorithm/

[2]https://fa.wikipedia.org/wiki/%D9%85%D8%B3%D8%A6%D9%84%D9%87_%DA%86%D9%86%D8%AF_%D9%88%D8%B2%DB%8C%D8%B1#:~:text=%D9%85%D8%B3%D8%A6%D9%84%D9%87%20%DA%86%D9%86%D8%AF%20%D9%88%D8%B2%DB%8C%D8%B1%20%DB%8C%DA%A9%20%D9%85%D8%B9%D9%85%D8%A7%DB%8C,%D9%88%20%D9%82%D8%B7%D8%B1%20%D9%85%D8%AA%D9%81%D8%A7%D9%88%D8%AA%DB%8C%20%D9%82%D8%B1%D8%A7%D8%B1%20%D8%AF%D8%A7%D8%AF.

[3] chatgpt

لینک پروژه:

github project 6