مقدمه

مسئله ژنتیک یکی از مسائل کلاسیک و مشهور دنیای هوش مصنوعی میباشد که روش ها و الگوریتم های متنوعی برای حل آن ارائه شده است. در این گزارش ما این مسئله را با یکی از الگوریتم های تکاملی یعنی الگوریتم ژنتیک حل خواهیم کرد و به شرح کامل پارامتر ها، تعریف مسئله بر اساس مفاهیم ژنتیک و نحوه پیاده سازی آن خواهیم پرداخت.

معرفی الگوریتم ژنتیک

الگوریتم ژنتیک (GA | Genetic Algorithms)، خانواده‌ای از «مدل‌های محاسباتی» (Computational Models) است که از مفهوم «تکامل» (Evolution) الهام گرفته شده‌اند. این دسته از الگوریتم‌ها، «جواب‌های محتمل» (Potential Solutions) یا «جواب‌های کاندید» (Candidate Solutions) و یا «فرضیه‌های محتمل» (Possible Hypothesis) برای یک مسأله خاص را در یک ساختار داده‌ای «کروموزوم مانند» (Chromosome-like) کدبندی می‌کنند. الگوریتم ژنتیک از طریق اعمال «عملگرهای بازترکیب» (Recombination Operators) روی ساختارهای داده‌ای کروموزوم مانند، اطلاعات حیاتی ذخیره شده در این ساختارهای داده‌ای را حفظ می‌کند.

در بسیاری از موارد، از الگوریتم‌های ژنتیک به عنوان الگوریتم‌های «بهینه‌ساز تابع» (Function Optimizer) یاد می‌شود؛ یعنی، الگوریتم‌هایی که برای بهینه‌سازی «توابع هدف» (Objective Functions) مسائل مختلف به کار می‌روند. البته، گستره کاربردهایی که از الگوریتم ژنتیک، برای حل مسئله در دامنه کاربردی خود استفاده می‌کنند، بسیار وسیع است.

پیاده‌سازی الگوریتم ژنتیک، معمولا با تولید جمعیتی از کروموزوم‌ها (جمعیت اولیه از کروموزوم‌ها در الگوریتم‌های ژنتیک، معمولا تصادفی تولید می‌شود و مقید به حد بالا و پایین متغیرهای مسأله هستند) آغاز می‌شود. در مرحله بعد، ساختارهای داده‌ای تولید شده (کروموزوم‌ها) «ارزیابی» (Evaluate) می‌شوند و کروموزوم‌هایی که به شکل بهتری می‌توانند «جواب بهینه» (Optimal Solution) مسأله مورد نظر (هدف) را نمایش دهند، شانس بیشتری برای «تولید مثل» (Reproduction) نسبت به جواب‌های ضعیف‌تر پیدا می‌کنند. به عبارت دیگر، فرصت‌های تولید مثل بیشتری به این دسته از کروموزوم‌ها اختصاص داده می‌شود. میزان «خوب بودن» (Goodness) یک جواب، معمولا نسبت به جمعیت جواب‌های کاندید فعلی سنجیده می‌شود.

الگوریتم ژنتیک و مسئله هشت وزیر

برای بیان این مسئله در قالب این الگوریتم ما هر کروموزوم را یک آرایه هشت تایی در نظر گرفته ایم که اندیس های این آرایه معرف ستون های خانه شطرنج و هر مقدار در این آرایه شماره سطر می باشند. مقادیر این آرایه غیر تکراری و جایگشتی از اعداد یک تا هشت هستند. در اینجا ما جمعیت اولیه را 100 در نظر گرفته ایم اما به دلخواه میتواند هر عددی انتخاب شود. فقط این نکته باید توجه شود که تعداد جایگشت های متفاوت 8 عدد 40320 می باشد و جمعیت اولیه باید مقداری کمتر از این تعداد باشد. شرط توقف الگوریتم اتمام 10،000 دور و یا پیدا شدن جواب مسئله می باشد. نحوه انتخاب والدین برای تولید فرزندان به این صورت است که 5 کروموزوم را به صورت رندم انتخاب

میکنیم و بعد از این پنج کروموزوم دو کروموزوم که تابع سودمندی (Fitness Function) بیشینه دارند انتخاب میشوند. نحوه محاسبه سودمندی یک کروموزوم به این صورت است که برای هر ژن یا وزیر در یک کروموزوم تعداد تهدید ها محاسبه شده و از جمع آنها و سپس معکوس کردن آن، مقدار تابع سودمندی برای یک کروموزوم محاسبه میشود. هدف ماکزیمم کردن تابع سودمندی است. برای تولید فرزندان نیز یک نقطه رندم در کروموزوم انتخاب میشود. فرزند اول ژن های سمت چپ این نقطه در والد اول و فرزند دوم ژن های سمت چپ این نقطه در والد دوم را به ارث میبرند. سپس برای پر کردن باقی ژن ها در سمت راست این نقطه در فرزند اول، والد دوم را از سمت چپ به راست پیمایش کرده و هر مقداری که در فرزند اول وجود نداشت را به همان ترتیب پیمایش در این فرزند درج میکنیم. برای پر کردن سمت راست فرزند دوم نیز همین کار را با والد اول تکرار میکنیم.

برای هر یک از این دو فرزند نیز به احتمال 80 درصد جهش انجام میشود. برای جهش دادن فرزندان دو اندیس رندم انتخاب شده و مقدار موجود در این اندیس ها با هم جابجا میشوند. سپس کل جمعیت به ترتیب تابع سودمندی مرتب شده و دو کروموزوم آخر که جز بدترین ها هستند را با فرزندان جابجا میکنیم.

شرح کد

Init()

اولین تابعی که در پیاده سازی این الگوریتم صدا زده میشود تابع init برای تولید جمعیت اولیه می باشد که یک ورودی سایز جمعیت را دریافت میکند. به تعداد ورودی این تابع یک لیست از جایگشت های اعداد یک تا هشت بصورت رندم تولید میشود.

```
def init(pop_size):
  import itertools
  permutations = itertools.permutations([1, 2, 3, 4, 5 ,6 ,7 ,8], r=None)
  start = random.randint(0, 40320 - pop_size)
  stop = start + pop_size
  return list(itertools.islice(permutations, start, stop))
```

Print_chrom(chrom)

این تابع برای چاپ یک کروموزوم بر روی صفحه شطرنج استفاده می شود.

```
def print_chrom(chrom):
  for i in range(8):
    print("|", end='')
    for j in range(8):
      if j+1 == chrom[i]:
        print(" Q |", end='')
      else:
        print("   |", end='')
    print()
    print("-----------------")
```

one_queen_penalty(index, chrom)

این تابع تعداد تهدید های موجود برای یک وزیر را محاسبه کرده و بر میگرداند. ورودی آن شماره ستون وزیر مورد نظر و کورموزم مربوط به آن است. نحوه محاسبه برخورد دو وزیر با هم بر اساس فاصله منهتن است. اگر به اندازه فاصله ستون دو وزیر بر اساس جلوتر یا عقب تر بودن وزیر دیگر از شماره ردیف آن کم یا به آن اضافه کنیم باید دو وزیر روبروی هم قرار بگیرند. اگر این اتفاق افتاد این دو وزیر یکدیگر را بصورت قطری تهدید میکنند.

```
def one_queen_penalty(index, chrom):
  col = index
  row = chrom[index]
  penalty = 0
  for i in range(len(chrom)):
    if i == col:
      continue
    if chrom[i] < row and chrom[i] + math.fabs(col - i) == row:
      penalty = penalty + 1
    elif chrom[i] > row and chrom[i] - math.fabs(col - i) == row:
      penalty = penalty + 1
  return penalty
```

configuration_penalty(chrom)

این تابع مقدار کل تهدیدها در یک کروموزوم را با محسابه این مقدار هر یک از وزیر ها یا ژن ها به کمک تابع one_queen_penalty و جمع آن ها، بر میگرداند.

```
def configuration_penalty(chrom):
  sum = 0
  for i in range(len(chrom)):
    sum = sum + one_queen_penalty(i, chrom)
  return sum
```

chrom_fitness_calculator(chrom)

این تابع محاسبه سودمندی برای یک کرومزوم را بر عهده دارد. با محاسبه مقدار کل تهدیدها در یک کروموزوم و معکوس کردن آن، سودمندی یک کروموزوم بدست می آید.

اگر در کروموزوم وزیر ها یکدیگر را تهدید نکنند یا به اصطلاح کروموزوم مورد نظر، جواب مسئله باشد سودمندی آن مقدار 2 که ماکزیمم است، می باشد.

```
def chrom_fitness_calculator(chrom):
  penalty = configuration_penalty(chrom)
  if penalty > 0:
    return 1/penalty
  else:
    return 2
```

selection(population, number)

این تابع 5 کروموزوم را به صورت رندم برای تولید مثل انتخاب میکند. ورودی آن لیست جمعیت کل و تعداد والد برای انتخاب می باشد.

```
def selection(population, number):
  randomlist = random.sample(range(0, len(population)), number)
  selected = []
  for i in randomlist:
    selected.append(population[i])
  return selected
```

get_two_parents(population)

این تابع دو والد از کروموزوم های انتخاب شده در تابع selection با سودمندی ماکزیمم را بر میگرداند.

```
def get_two_parents(population):
  population.sort(reverse=True, key=chrom_fitness_calculator)
  # print(population)
  return population[0:2]
```

cross_over(parent1, parent2)

این تابع عمل تولید 2 فرزند با دو والد انتخاب شده توسط تابع get_two_parents را با روش ذکر شده انجام میدهد.

```python
def cross_over(parent1, parent2):
  parent1 = list(parent1)
  parent2 = list(parent2)
  position = random.randint(1,6)
  # print("Cross over Position = " , position)
  child1 = parent1[0:position]
  child2 = parent2[0:position]
  for i in range(len(parent1)):
    if parent1[i] not in child2:
      child2.append(parent1[i])
    if parent2[i] not in child1:
      child1.append(parent2[i])
  child1 = tuple(child1)
  child2 = tuple(child2)
  return [child1, child2]
```

mutation(childs) و mutate(chrom)

این دو تابع مسئول انجام عمل جهش با روش ذکر شده در فرزندان تولید شده هستند. تابع mutation دو فرزند را گرفته و برای هر کدام یک احتمال تولید میکند. اگر زیر 80 درصد باشد با تابع mutate یک جهش در فرزند ایجاد میکند در غیر اینصورت جهشی ایجاد نمیشود.

```
def mutation(childs):
  mutated =[]
  for child in childs:
    prob = random.randint(1, 100)
    # print("Mutation Prob = ", prob)
    if prob < mutation_prob:
      mutated.append(mutate(child))
    else:
      mutated.append(child)
  return mutated

def mutate(chrom):
  position1 = random.randint(0, 7)
  position2 = random.randint(0, 7)
  # print("Mutation Pos = ", position1, " , " , position2)
  chrom = list(chrom)
  temp = chrom[position1]
  chrom[position1] = chrom[position2]
  chrom[position2] = temp
  chrom = tuple(chrom)
  return chrom
```

survival_selection(population, childs)

این تابع نیز با روش ذکر شده در لیست جمعیت کل، فرزندان تولید شده را جایگزین میکند.

```
def  survival_selection(population, childs):
  population.sort(reverse=True, key=chrom_fitness_calculator)
  population[-1] = childs[0]
  population[-2] = childs[1]
  if chrom_fitness_calculator(childs[0]) == 2:
    return (1, population)
  if chrom_fitness_calculator(childs[1]) == 2 :
    return (2, population)
  return (0, population)
```

پارامتر ها

- pop_size : تعداد جمعیت اولیه
- select_random : تعداد کروموزومی که بصورت رندم برای انتخاب والدین، انتخاب میشوند.
- mutation_prob : احتمال جهش
- rounds : تعداد تکرار الگوریتم

نتیجه نهایی

الگوریتم با توجه به جمعیت انتخاب شده بعد از یک تا بیشتر از 200 تکرار میتواند به جواب برسد. در یکی از اجرا های این الگوریتم بعد از 214 تکرار به جواب رسیدیم.

- ده کروموزوم برتر:

```
[(4, 1, 5, 8, 2, 7, 3, 6), (6, 3, 1, 2, 5, 8, 4, 7),
 (6, 3, 1, 4, 8, 5, 2, 7), (6, 3, 1, 2, 5, 8, 4, 7),
 (6, 4, 7, 3, 8, 2, 5, 1), (6, 3, 5, 2, 8, 1, 7, 4),
 (1, 3, 6, 2, 5, 8, 4, 7), (4, 7, 5, 8, 2, 1, 3, 6),
 (7, 6, 3, 5, 8, 1, 4, 2), (4, 2, 5, 8, 1, 7, 6, 3)]
```

- کروموزوم برتر و جواب مسئله

```
(4, 1, 5, 8, 2, 7, 3, 6)
```

- شکل این کروموزوم در صفحه شطرنج