حل مسئله برج هانوی

Tower Of Hanoi

به کمک الگوریتم ژنتیک

مینا خسروی

درس هوش محاسباتی

بعد از خواندن دو مقاله زیر به این فکر افتادم که برای حل مسئله برج هانوی الگوریتم ژنتیک بنویسم:

/http://www.algorithmha.ir/%D8%A8%D8%B1%D8%AC-%D9%87%D8%A7%D9%86%D9%88%DB%8C

و

https://programstore.ir/%D8%A8%D8%B1%D8%AC-%D9%87%D8%A7%D9%86%D9%88%DB%8C-%DA%86%DB%8C%D8%B3%D8%AA%D8%9F-%D8%A8%D8%B1%D8%B1%D8%B3%DB%8C-%D9%85%D8%B3%D8%A6%D9%84%D9%87-%D8%A8%D8%B1%D8%AC-%D9%87%D8%A7%D9%86%D9%88%DB%8C-tower/

پس در ادامه توضیحات لازم برای تشبیه مسئله در الگوریتم ژنتیک رو خواهم آورد:

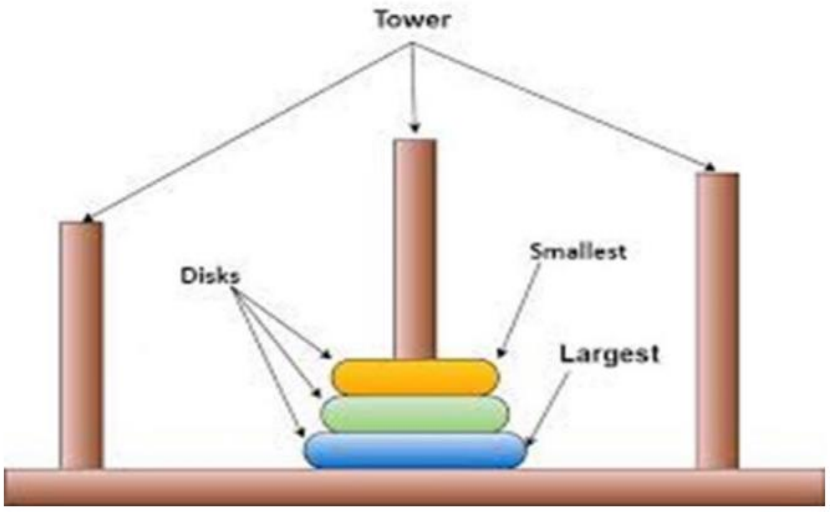
برج هانوی چیست؟

معمای برج های هانوی یه بازی فکری و معمای قدیمیه. برج های هانوی از چند تا دیسک (حلقه، جعبه یا...) که از کوچیک تا بزرگ روی هم قرار گرفتن تشکیل شده .این دیسک هارو باید از محل مبداء به مقصد منتقل کنین با چند تا شرط:

1 هر بار فقط می تونین یک دیسک رو جابجا کنیم.

2 هیچوقت نباید دیسک بزرگتر روی دیسک کوچیکتر قرار بگیره.

3 تمام دیسک ها به جز دیسکی که حمل میشه باید روی میله قرار داشته باشند.



Population یا جمعیت مسئله

هر جمعیت شامل چند نسل میشه که نسل ها رو کرموزوم در نظر میگیریم .هر کروموزوم متشکل از چندتا ژن هست .در مسئله ی ما ژن ها را حرکت دیسک ها در نظر میگیریم .

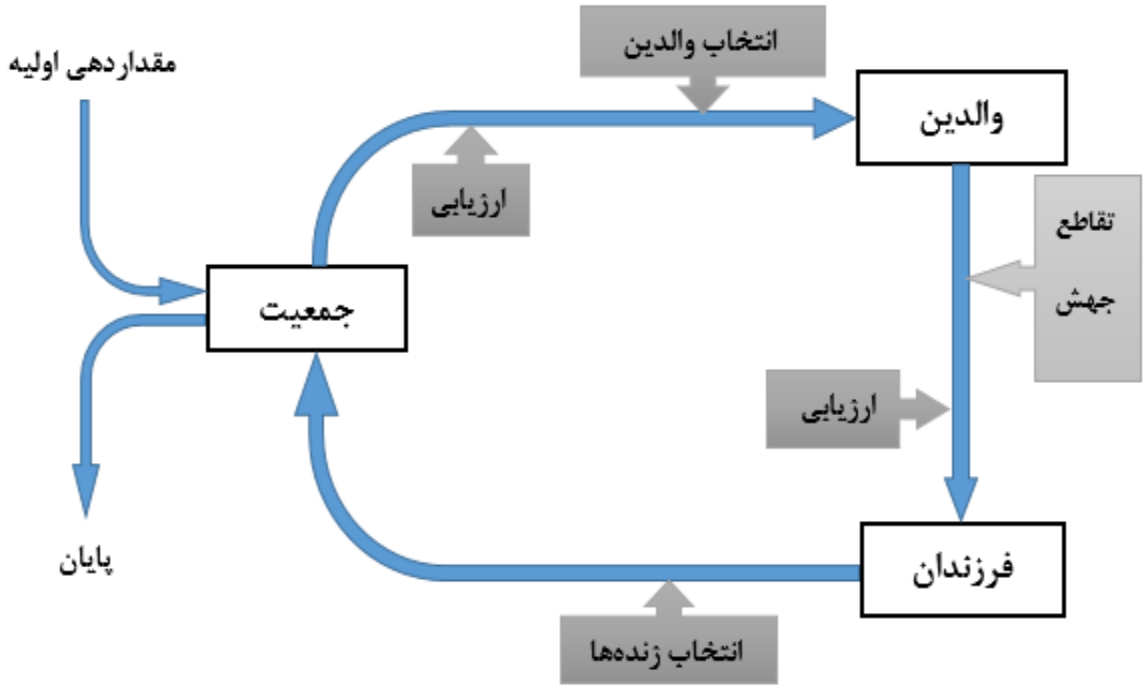
میله های به ترتیب به صورت ۱و ۲و ۳ نمایش داده میشوند .مثلا حرکت از میله ی به صورت ۱۲ و حرکت از میله ی با ۱۳ نمایش داده میشود .

طول کروموزوم های هر جمعیت رو برابر با کمترین(بهینه ترین) دفعات جابه جایی در نظر میگیریم :

$$2^n -1$$

که مقدار n برابر با تعداد دیسک هاست.

نمایی از کلیت مراحل یک الگوریتم ژنتیک:



جمعیت اولیه

برای اولین جمعیت،الگوریتم جمعیتی رندوم تولید میکند .

سپس (تابع برآزندگی) برای همه ی کروموزوم های(نسل های) یک جمعیت (اینجا جمعیت اولیه) محاسبه میشه، الگوریتم هرحرکاتی که برخلاف قوانین مسئله باشد را پیدا کرده و از مقدار برآزندگی کسر میکند، نسلی با کمترین تعداد حرکت اشتباه یعنی بالاترین برآزندگی میتواند به جواب مسئله نزدیک باشد.

اگر الگوریتم نتواند بهترین برآزندگی را در جمعیت موجود پیدا کند، جمعیت جدیدی تولید میشود تا دوباره برای تمام کرموزوم های آن، مقدار برآزندگی محاسبه شود.

هدف پیدا کردن راه حل بهتر است،پس در هر جمعیت به دنبال نسلی با برازش بالاتر هستیم.

تابع تولید جمعیت اولیه

اولین جمعیت را رندوم ایجاد میکنیم

جمعیت این مسئله همان حرکات دیسک هاست!

مقدار برگشتی تابع لیست جمعیت ماست

```python
def generate_initial_population(n_pop, n_bits, num_towers=3):  #تعریف داده های ورودی اگر داریم به صورت رشته،در اینجا رندوم ایجاد شده
    pop = []
    for i in range(n_pop):
        specimen = []
        min_length = 2 ** n_bits - 1  # طول کروموزوم رو کمترین تعداد جابه جایی در نظر میگیریم

        max_length = min_length + 50   #حداکثر 50 حرکت پس از بهینگی انجام بشه
        specimen_size = random.randint(min_length, max_length)

        while len(specimen) < specimen_size:
            origin_tower = random.randint(1, num_towers)  # به صورت رندوم یک برج رو برای مبدا انتخاب می کنیم
            if len(specimen) == 0:
                origin_tower = 1

            destination_tower = random.randint(1, num_towers)  # به صورت رندوم یک برج رو برای مقصد انتخاب می کنیم

            element = (origin_tower, destination_tower)  # هر المنت برجمون یک مبدا و مقصدی داره که حرکت رو نشون میده
            specimen.append(element)

        pop.append(specimen)

    return pop
```

تابع برازندگی

ابتدا مقدار فیتنس رو یک عدد ثابت در نظر میگیریم (اینجا برابر با یک قرار دادیم) و به ازای حرکات درست و غلط (ژن های خوب و بد) مقداری اضافه یا کسر میشه.

حرکات(جا به جایی های) اشتباهی که از مقدار فیتنس کسر شده :

-اگر حرکتی از یک میله برج خالی از دیسک شروع شده باشه .

-اگر دیسک بزرگتر روی دیسک کوچکتر قرار بگیره.

همچنین اگر با یک حرکت کل تعداد دیسک ها از میله مبدا به میله ی مقصد منتقل شود (با یک اتفاق نادر یا اگر فقط یک دیسک داشته باشیم) عددی بزرگ به برازش اضافه میشود.

```python
def fitness(specimen, n_bits, print_state=lambda x: None):  #تابع برازندگی
    #به انجام معتبر حرکات با بازی باید
    state = [[i + 1 for i in range(n_bits)], [], []]
    print_state(state)
    fitness_value = 1
    for movement in specimen:
        origin_tower = movement[0] - 1
        destination_tower = movement[1] - 1

        # به انجام مجاز حرکات فقط تا نامعتبر حرکات گرفتن نادیده
        if len(state[origin_tower]) == 0:
            # نامعتبر حرکت  کنیم شروع رو حرکت خالی برج یک از نمیتونیم
            fitness_value = fitness_value - 5
            continue

        origin_peg = state[origin_tower][0]
        if len(state[destination_tower]) > 0 and state[destination_tower][0] < origin_peg:
            #نامعتبر حرکت  بدیم قرار کوچکتر دیسک یک روی رو بزرگتر دیسک نمیتونیم
            fitness_value = fitness_value - 5
            continue

        state[origin_tower].pop(0)
        state[destination_tower].insert(0, origin_peg)
        print_state(state)

        if len(state[1]) == n_bits or len(state[2]) == n_bits:
        # بیکنیم اضافه فیتنس ارزش به زیاد مقدار یک صورت این در (باشیم داشته دیسک یک فقط اگه یا نادر حالت یک) بشن منتقل مقصد به مبدا برج از ها دیسک کل حرکت یک یا اگر
            fitness_value = fitness_value + 10000
            break
        fitness_value = fitness_value + 1

    # بیکنیم کم ازش برازش از شونده شروع برج در هنوز که هایی دیسک تعداد براساس
    fitness_value = fitness_value - len(state[0]) * 10
    # بیکنیم کم ازش برازش از داده انجام که حرکاتی تعداد براساس
    fitness_value = fitness_value - len(specimen)

    fitness_value = fitness_value + len(state[2]) * 5
    fitness_value = fitness_value + sum(state[2]) * 8
    fitness_value = fitness_value - sum(state[0]) * 15
    fitness_value = fitness_value - len(state[1]) * 5

    if len(state[2]) and state[2][-1] == n_bits:
        fitness_value = fitness_value + 400
    return fitness_value
```

تابع تولید نسل جدید

بین والدین برگزیده جفت گیری جهت ایجاد کروموزوم جدید انجام میشود .جفت گیری با عملگر هم برش(crossover تک نقطه ای) انجام میشه و ژن های والدین با برازندگی بیشتر به فرزندان و نسل جدید انتقال پیدا میکنه.

یک نقطه به صورت تصادفی برای برش کروموزوم ها از اون نقطه انتخاب میشه و بعد بخش اول از والد اول و بخش دوم از والد دوم فرزند اول رو تشکیل میدن،

و بخش اول از والد دوم و بخش دوم از والد اول فرزند دوم رو تشکیل میده.

در نهایت جمعیت اولیه که به ان فرزندان اضافه شده از تابع برمیگردد

```
def crossover(parents, population_len, n_bits):  #تابع تولید نسل جدید
    for i in range(population_len - 1):
        specimen = []
        while len(specimen) < 2 ** n_bits - 1:
            # با ترکیب تصادفی والدین فرزندان جدید ایجاد می کنیم

            parent_1, parent_2 = random.sample(parents, 2)

            pt1 = len(parent_1) // 2
            pt2 = len(parent_2) // 2

            part_1 = parent_1[:pt1]
            part_2 = parent_2[pt2:]

            random.shuffle(part_2)

            specimen = part_1
            specimen.extend(part_2)

        pop.append(specimen)
    return pop
```

تابع جهش

در اخر با اعمال جهش روی فرزندان، با تغییرِ رندوم ژن های موجود؛ یک جهش در نسل جدید برای ایجاد ژن های جدید انجام میدیم.

ورودی تابع جمعیت و نرخ جهش هست و در اخر جمعیتی که بعضی ژن هایش به صورت رندوم تغییر کرده برمیگردد

```
def mutate(pop, mut_r):  #جهش تابع
    for specimen in pop:
        specimen_len = len(specimen)
        mutable_genes = random.randint(1, specimen_len)
        for gene in range(mutable_genes):
            if random.uniform(0, 1) >= mut_r:
                continue

            index = random.randint(1, specimen_len - 1)

            origin_tower = 0
            destination_tower = 0

            while origin_tower == destination_tower:
                origin_tower = random.randint(1, 3)
                destination_tower = random.randint(1, 3)

            element = (origin_tower, destination_tower)

            specimen[index] = element
    return pop
```

فراخوانی ها و نمایش جمعیت ها و فیتنس ها (لوپ اصلی)

```
for generation in range(num_generations):
    population_fitness = [fitness(specimen, n_bits) for specimen in pop]
    population_fitness, pop = zip(*sorted(zip(population_fitness, pop), reverse=True))

    pop = list(pop)

    best_index = np.argmax(population_fitness)
    best_specimen = copy.deepcopy(pop[best_index])
    best_fitness = fitness(best_specimen, n_bits)
    generational_fitness.append(best_fitness)
    print('Generation {} best index: {}. Fitness: {}'.format(generation, best_index, best_fitness))
    print('Generation {} best specimen'.format(generation))
    print(pop[best_index])
    print('Generation {} best specimen final state'.format(generation))
    fitness(best_specimen, n_bits, lambda x: print(x))

    # فقط بهترین والدین برای تولید نسل بعد انتخاب میشوند
    parents = pop[:crossover_parents_len]

    # ساخت جمعیت جدید
    pop = crossover(parents, n_pop - new_specimens_size - 1, n_bits)
    # جهش در جمعیت جدید
    pop = mutate(pop, mut_r)

    # بهترین نمونه از همانگونه که هستند به نسل بعد میرسند برای تضمین عدم کاهش برازندگی
    pop.append(best_specimen)

    # معرفی نمونه های برتر
    new_specimens = generate_initial_population(new_specimens_size, n_bits)
    pop.extend(new_specimens)
```

استفاده از کتابخانه ها

از کتابخانه numpy برای عملیات ماتریسی استفاده میکنیم و بهش نام اختصاری npمیدیم (برای راحت تر صدا زدن در طول برنامه)

از کتابخانه matplotlib برای رسم نمودار استفاده میکنیم با نام اختصاری plt.

از کتابخانه random برای عملیات های مختلفی که به اعداد تصادفی نیاز دارن استفاده میکنیم.

استفاده از ماژول copy

```
import numpy as np
import matplotlib.pyplot as plt
import random
import copy
```

نمایش نمودار

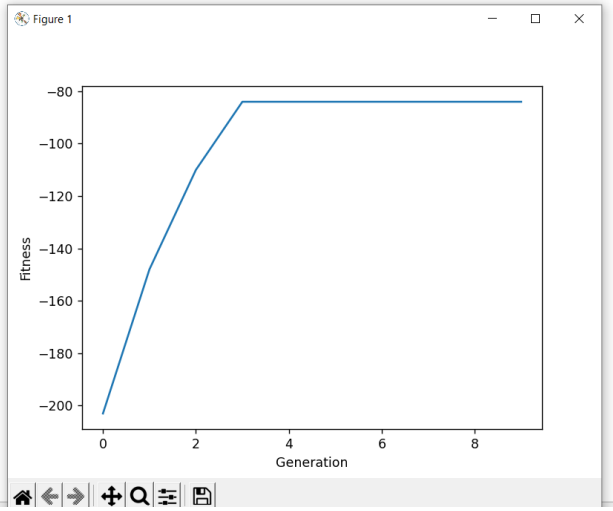
```
plt.plot(generational_fitness)
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.show()
```

مقدار دهی ها

```
n_bits = 5  #ها دیسک تعداد یا رشته طول
n_pop = 100 #جمعیت سایز
crossover_parents_len = 10
new_specimens_size = 20  # نسل تولید هر در تولیدی های کروموزوم تعداد
num_generations = 10   #ها نسل تعداد
mut_r = 0.50 + n_bits / 15  #جهش نرخ
pop = generate_initial_population(n_pop, n_bits) #جمعیت ساخت تابع فراخوانی
#populationn_bits
len(pop)
generational_fitness = []
```

خروجی برنامه

```
[[5], [2, 3], [1, 4]]
[[1, 5], [2, 3], [4]]
[[1, 5], [3], [2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [1, 3], [2, 4]]
[[1, 5], [3], [2, 4]]
[[5], [3], [1, 2, 4]]
Generation 9 best index: 0. Fitness: -84
Generation 9 best specimen
[(1, 1), (1, 2), (3, 1), (1, 3), (3, 3), (2, 2), (3, 2), (1, 1), (1, 2), (2, 1), (3, 1), (1, 3), (1, 2), (1, 1), (2, 3), (1, 1), (3, 1), (1, 2), (1, 1), (2, 1), (2, 3), (3, 2), (1, 2),
(1, 2), (2, 2), (1, 3), (2, 1), (3, 3), (1, 2), (2, 3), (1, 3), (3, 1), (3, 2), (2, 3), (1, 3), (2, 2), (3, 3), (3, 2), (1, 2), (2, 1), (1, 3), (1, 3)]
Generation 9 best specimen final state
[[1, 2, 3, 4, 5], [], []]
[[1, 2, 3, 4, 5], [], []]
[[2, 3, 4, 5], [1], []]
[[3, 4, 5], [1], [2]]
[[3, 4, 5], [1], [2]]
[[3, 4, 5], [1], [2]]
[[3, 4, 5], [1], [2]]
[[1, 3, 4, 5], [], [2]]
[[3, 4, 5], [], [1, 2]]
[[4, 5], [3], [1, 2]]
[[4, 5], [3], [1, 2]]
[[4, 5], [3], [1, 2]]
[[1, 4, 5], [3], [2]]
[[4, 5], [1, 3], [2]]
[[4, 5], [1, 3], [2]]
[[1, 4, 5], [3], [2]]
[[1, 4, 5], [2, 3], []]
[[4, 5], [1, 2, 3], []]
[[4, 5], [1, 2, 3], []]
[[5], [1, 2, 3], [4]]
[[1, 5], [2, 3], [4]]
[[1, 5], [2, 3], [4]]
[[5], [1, 2, 3], [4]]
[[5], [2, 3], [1, 4]]
[[1, 5], [2, 3], [4]]
[[1, 5], [3], [2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [3], [1, 2, 4]]
[[5], [1, 3], [2, 4]]
[[1, 5], [3], [2, 4]]
[[5], [3], [1, 2, 4]]
```



پایان ^^