پیاده سازی و بررسی Genetics Algorithm با انتخاب Tournament و کد گذاری اعداد حقیقی

درس رایانش تکاملی – مدرس: دکتر روحانی

نویسنده: آریا عربان

هدف و راه حل

در این تمرین همانند تمرین سری قبل، خواسته شده که الگوریتم GA پیاده سازی شود و برروی چندین تابع تست ارزیابی شود. تنها تفاوت این تمرین با تمرین سری قبل در این است که به جای کد گذاری دودویی، از کد گذاری اعداد حقیقی استفاده می‌شود، و جهت انجام عمل انتخاب از روش Tournament استفاده می‌شود.

برای انجام این کار، ده تابع تست در کد پیاده سازی شده است(توابع تست F1 الی F10 که در اسلایدهای 38 و 39 چپتر3 درس آمده‌اند)، و الگوریتم GA را بر روی هر کدام از این توابع 10 دفعه اجرا می‌کنیم و بررسی می‌کنیم که تغییر پارامترهای الگوریتم (تعداد جمعیت, احتمال تقاطع و احتمال جهش) به چه میزان بر روی نتیجه اثر می‌گذارد.

برای پیاده‌سازی الگوریتم GA، دقت می‌شود که هر عضوی از جمعیت، خود شامل N عدد است. هر کدام از این N عدد به صورت حقیقی هستند. برای انجام تقاطع احتمالی دو والد انتخاب شده، روش تقاطع حسابی استفاده شده است. همچنین برای عمل جهش، فرزند انتخاب شده با بردار استاندارد تصادفی به میانگین 0 و واریانس σ (که قابل تنظیم است) جمع می‌شود.

شرح نتایج

در ابتدا، با استفاده از یکی از تابع های هدف، مقادیر بهینه برای پارامترها را تعیین می‌کنیم، تا برای مراحل بعدی از آنها استفاده شود. در این قسمت از تابع هدف f5 برای این کار استفاده می‌کنیم، و مقدار N را برابر 10 در نظر می‌گیریم.

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \quad \text{where} \quad \mathbf{x} = (x_1, \ldots, x_N) \in \mathbb{R}^N$$

ابتدا با در نظر گرفتن مقدار ثابت احتمال تقاطع 0.9 و احتمال جهش 0.1، تابع فوق را برای مقادیر مختلف جمعیت بررسی می‌کنیم:

population_size	objective_function_score
50	9.56
100	8.67
250	6.7
500	7.29

بنابرین بنظر می‌آید که مقدار مناسبی برای جمعیت برابر 250 می‌باشد.

در ادامه، با در نظر گرفتن عدد 250 برای جمعیت، و مقدار ثابت احتمال جهش برابر 0.1، تابع را برای مقادیر مختلف احتمال تقاطع بررسی می‌کنیم:

p_crossover	objective_function_score
0.2	10.3
0.5	8.58
0.7	9.62
0.9	8.74

بنظر می‌آید که مقدار مناسبی برای احتمال تقاطع برابر 0.5 (یا همان 50%) می‌باشد.

با در نظر گرفتن مقدار 250 برای جمعیت، و احتمال تقاطع برابر 0.5، تابع را برای مقادیر مختلف احتمال جهش بررسی می‌کنیم:

p_mutation	objective_function_score
0.01	9.17
0.05	8.72
0.1	7.6
0.2	9.49
0.5	7.94

بنظر می‌آید که مقدار مناسبی برای احتمال جهش برابر 0.1 (یا همان 10%) می‌باشد.

با در نظر گرفتن مقدار 250 برای جمعیت، احتمال تقاطع برابر 0.5، و احتمال جهش برابر 0.1، تابع را برای مقادیر مختلف تعداد فرزندان شرکت کننده در هر Tournament(k) را بررسی می‌کنیم:

k	objective_function_score
1	61.27
3	8.99
5	9.31
7	9.97

بنظر می‌آید که مقدار مناسبی برای تعداد فرزندان شرکت کننده در هر Tournament برابر 3 می‌باشد.

در نهایت، با در نظر گرفتن مقادیر پارامتر های بدست آمده در قسمت‌های قبل، به دنبال مقدار مناسب برای پارامتر جهش σ می‌گردیم:

σ	objective_function_score
0.01	19.3
0.05	9.57
0.1	7.31
0.2	8.28
0.5	8.72

بنظر می‌آید که مقدار مناسبی برای پارامتر جهش σ برابر 0.1 می‌باشد.

بنظر می‌آید که مقدار مناسبی برای تعداد فرزندان شرکت کننده در هر Tournament برابر 3 می‌باشد.

با مشخص شدن مقادیر مناسب برای پارامترها، حالا به بررسی نتایج حاصل از اعمال الگوریتم GA بر روی هرکدام از توابع تست به ازاء N=10, 30, 50 پرداخته می‌شود. دقت می‌شود که در حال 10 دفعه اجرا گرفته می‌شود، و نهایتا جواب نهایی را میانگین جواب‌ها حساب می‌کنیم.

	N=10	N=20	N=50
f1	0.0009	0.072	0.507
f2	0.058	0.324	2.231
f3	0.0097	60.272	1782.247
f4	0.032	8.248	10.324
f5	7.688	33.285	246.729
f6	0.0007	0.0499	29.469
f7	1.259	8.674	17.239
f8	-2340.051	-4728.528	-5480.22
f9	0.0005	0.0324	21.533
f10	1.844	2.392	3.525

در تمامی مواردی که بالا بررسی شده‌اند، تعداد Iterationها را برابر 1000 هستند.

مشاهده می‌شود که در اکثر این موارد، هرچه N بزرگتر باشد، گرفتن جواب بهینه سختتر خواهد بود. علت آن است که الگوریتم باید تعداد بیشتری عدد را بهینه‌سازی کند.

اگر نتایجی که در این قسمت بدست آمده را با نتایج تمرین سری قبل مقایسه کنیم، دیده می‌شود که نتایج این قسمت به مقدار زیادی بهتر از نتایج قسمت قبل هستند.

پس می‌توانیم نتیجه بگیریم که در توابع مورد بررسی واقع شده، استفاده از روش انتخاب Tournament به همراه کد گذاری اعداد حقیقی، نتایج بسیار قوی‌تری نسبت به روش انتخاب Roulette Wheel به همراه کد گذاری باینری می‌دهد.

علت می‌تواند در این باشد که در روش انتخابی Tournament، شانس بیشتری برای انتخاب عضوهای ضعیف‌تری در جامعه فعلی وجود دارد، زیرا در این روش انتخاب مقادیر مطلق تابع هدف مهم نیستند، و به تعبیری میزان اختلاف بین تابع هدف عضوهای جامعه تاثیری در شانس برنده شدن آنان ندارد. این امر باعث می‌شود تا اکتشاف در جهت پیدا کردن جواب بهینه به نحو بهتری انجام شود.

همچنین در هنگام انجام عمل جهش بر روی اعضا دارای کدگزاری حقیقی، از تغییرات ناگهانی فرد پس از جهش جلوگیری می‌شود، که این امر می‌تواند در گرفتن نتایج بهتری نقش مهمی داشته باشد.

ضمیمه (کد) :

Github Repo

پیاده سازی توابع هدف:

```python
def f1(v):
    total = 0
    for i in range(len(v)):
        xi = v[i] ** 2
        total = total + xi
    return np.abs(total)


def f2(v):
    return np.sum(np.abs(v)) + np.prod(np.abs(v))


def f3(v):
    total = 0
    for i in range(len(v)):
        total = total + (np.sum(v[:(i+1)]) ** 2)
    return total


def f4(v):
    return np.max(np.abs(v))


def f5(v):
    ### ROSENBROCK ###
    total = 0
    for i in range(len(v) - 1):
        xi = v[i]
        x_next = v[i + 1]
        new = 100 * (x_next - xi ** 2) ** 2 + (xi - 1) ** 2
        total = total + new
    return total
```

```python
def f6(v):
    total = 0
    for i in range(len(v)):
        total = total + ((v[i] + 0.5) ** 2)
    return total


def f7(v):
    total = 0
    for i in range(len(v)):
        total = total + (i+1) * (v[i] ** 4) + random.uniform(0, 1)
    return total


def f8(v):
    total = 0
    for i in range(len(v)):
        total = total - (v[i] * math.sin(math.sqrt(np.abs(v[i]))))
    return total


def f9(v):
    total = 0
    for i in range(len(v)):
        total = total + ((v[i] ** 2) - (10 * math.cos(2 * math.pi * v[i])) + 10)
    return total


def f10(v):
    ttl1 = 0
    ttl2 = 0
    for i in range(len(v)):
        ttl1 = ttl1 + (v[i] ** 2)
        ttl2 = ttl2 * math.cos(2 * math.pi * v[i])

    total = -20 * math.exp(-0.2 * math.sqrt(ttl1)) - math.exp((1 / len(v)) * ttl2) + 20 + math.e
    return total
```

پیاده سازی الگوریتم GA:

```python
# tournament selection
def _tournament_selection(pop, scores, k):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k - 1):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    # print(pop[selection_ix])
    return pop[selection_ix]


# crossover two parents to create two children
def _crossover(p1, p2, r_cross):
    # children are copies of parents by default
    c1, c2 = p1.copy(), p2.copy()
    # check for recombination
    if rand() < r_cross:
        c1 = (p1 + random.uniform(0, 1) * (np.subtract(p2, p1))).tolist()
        c2 = (p2 + random.uniform(0, 1) * (np.subtract(p1, p2))).tolist()

    return [c1, c2]


# mutation operator
def _mutation(child, r_mut, sigma_mut):
    # check for a mutation
    if rand() < r_mut:
        return child + np.random.normal(loc=0.0, scale=sigma_mut, size=len(child))

    return child
```

```python
# genetic algorithm
def genetic_algorithm(objective, bounds, n_iter, n_pop, n_inside_parent, r_cross, r_mut, sigma_mut,k_tournament):
    # initial population of random numbers in bounds
    pop = [[randint(bounds[0], bounds[1]) for _ in range(n_inside_parent)] for _ in range(n_pop)]
    # keep track of best solution
    best, best_eval = 0, objective(pop[0])
    # enumerate generations
    for gen in range(n_iter):
        # evaluate all candidates in the population
        scores = [objective(d) for d in pop]
        # check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
            # print(">%d, new best f(%s) = %f" % (gen, decoded[i], scores[i]))
        # select parents
        selected = [_tournament_selection(pop, scores, k_tournament) for _ in range(n_pop)]
        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i + 1]
            # crossover and mutation
            for c in _crossover(p1, p2, r_cross):
                # mutation

                chd = _mutation(c, r_mut, sigma_mut)
                # store for next generation

                children.append(chd)
        # replace population
        pop = children
    return [best, best_eval]
```

```python
def GA_main(population_size, n_inside_parent, p_crossover, p_mutation, sigma_mutation, k_tournament, func_num):
    # define the total iterations
    n_iter = 1000
    # define the population size
    n_pop = population_size
    # crossover rate
    r_cross = p_crossover
    # mutation rate
    r_mut = p_mutation
    # perform the genetic algorithm search
    sigma_mut = sigma_mutation

    n_inside_parent = n_inside_parent

    objective_func = funcs[func_num]  # g1t objective function and bound pair from the funcs dictionary, can be 1 to 10

    best, score = genetic_algorithm(objective_func['function_name'], objective_func['bounds'], n_iter, n_pop,
                                    n_inside_parent, r_cross, r_mut, sigma_mut, k_tournament)
    # print('Done!')
    # decoded = decode(objective_func['bounds'], n_bits, best)
    # print('f(%s) = %f' % (decoded, score))
    return best, score
```

```python
def main():
    for i in range(10):
        print(f"FUNCTION {i + 1}")
        for N in [10, 30, 50]:
            print(f"N = {N}")
            sols = []
            obj = []
            for k in range(10):
                best_solution, objective_function_score = GA_main(population_size=100,
                                                                  n_inside_parent=N, p_crossover=0.5,
                                                                  p_mutation=0.1, sigma_mutation=0.1, k_tournament=3,
                                                                  func_num=i+1)
                obj.append(objective_function_score)
                sols.append(best_solution)

            print(f"best_solution:\n {np.mean(sols,axis=0)}")
            print(f"objective_function:\n {np.mean(obj)}")
            print("------------")


if __name__ == '__main__':
    main()
```