인공지능 Genetic Algorithm 과제 보고서

2016160311 이재윤

본 실험에서는 Genetic Algorithm을 이용하여 4지선다형 영어시험 10문제의 정답을 알아내는 실험을 진행하였다.

문제의 정답은 정수 1부터 4까지의 숫자를 10개 넣은 리스트에 나타내는 것으로 하였으며, population은 10을 유지하였다.

문제의 목표 정답은 임의로 선택하였는데, 여기서는 [3, 1, 4, 3, 4, 1, 4, 1, 2, 2] 로 설정하였다.

초기 population은 빈 리스트를 작성하고 여기에 1부터 4 사이의 임의의 정수를 10개 채워 넣은 서로 다른 리스트를 10개 넣어 만들었다. 작성된 코드는 다음과 같다.

```
for i in range(population_size):
    new_chromosome = []
    for j in range(population_size):
        new_chromosome.append(random.randint(1, 4))
    init_population.append(new_chromosome)
```

문제의 정답률을 구하는 fitness function은 10개의 정답이 나열된 individual을 입력으로 받아 실제 정답과 비교하여 각 문제를 10점으로 하여 모두 맞았을 시 100점이 나오도록 설정하였다. 작성된 코드는 다음과 같다.

```
def fitness_function(individual):
    fitness = 0
    for i in range(0, len(individual)):
        if individual[i] == answer[i]:
            fitness += 10
    return fitness
```

Selection을 하는 방법은 서로 다른 두가지의 전략을 구현하였는데, 이에 사용된 전략은 fitness proportionate selection과 tournament selection이다.

Fitness proportionate selection의 경우, 10개의 정답 리스트가 담긴 population을 입력으로 받아서 각 individual의 정답률을 계산한 뒤, 정답률이 높은 individual일수록 높은 확률로 선택될 수 있도록 구현하였다. 작성된 코드는 다음과 같다.

```
def fitness_proportionate_selection(population):
    fitness_list = []
    fitness_sum = 0
    for i in range(0, len(population)):
        fitness_list.append(fitness_function(population[i]))
        fitness_sum += fitness_function(population[i])
    probability_list = []
    cumulative_value = 0
```

```
    for i in range(0, len(fitness_list)):
        try:
            cumulative_value += fitness_list[i] / fitness_sum
            probability_list.append(cumulative_value)
        except ZeroDivisionError:
            pass
    probability_list[-1] = 1.0
    random_number = random.random()
    for i, probability in enumerate(probability_list):
        if random_number <= probability:
            return population[i]
            break
```

Tournament selection의 경우, population 내의 각 individual의 정답률을 구한 뒤 등수를 매겨서 k 등의 individual이 p*(1-p)^k의 확률로 선택되도록 구현하였다. 여기서 확률 p의 값은 0.7로 고정하였다. 작성된 코드는 다음과 같다.

```
def tournament_selection(population):
    fitness_list = []
    fitness_with_index = []
    probability_list = []
    cumulative_value = 0
    p = 0.7
    max = 0
    for i in range(0, len(population)):
        fitness_list.append(fitness_function(population[i]))
    for i, fitness in enumerate(fitness_list):
        fitness_with_index.append((i, fitness))
    fitness_with_index.sort(key=lambda tuple: tuple[1])
    random_number = random.random()
    for i in range(0, len(population)):
        cumulative_value += p*(1-p)**i
        probability_list.append(cumulative_value)
    probability_list[-1] = 1.0
    for i, probability in enumerate(probability_list):
        if random_number <= probability:
            (index, index_fitness) = fitness_with_index[len(population)-i-1]
            break
    return population[index]
```

부모로부터 받아온 두 개의 individuals를 crossover하는 함수는 single point crossover를 사용하였으며, crossover point는 임의로 선택하였다. 작성된 코드는 다음과 같다.

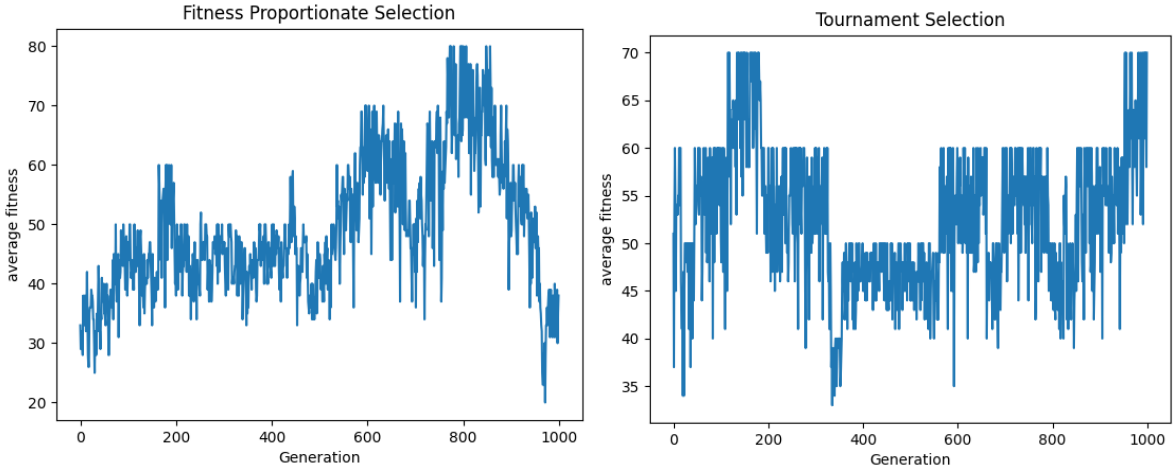
```
def crossover(parent1, parent2):
    crosspoint = random.randint(0, 9)
    child = parent1[crosspoint:] + parent2[:crosspoint]
    return child
```

낮은 확률로 일어나는 변이를 구현한 함수 mutate는 individual 안의 10개의 정답 중 하나를 임의로 선택해서 임의의 정답으로 바꾸는 전략을 선택했다. 작성된 코드는 다음과 같다.

```
def mutate(individual):
    random_index = random.randint(0, 9)
    individual[random_index] = random.randint(1, 4)
    return individual
```

완성된 genetic algorithm은 crossover rate와 mutation rate를 변화시켜가며 1000세대 진행시켰으며, 앞서 설명한 두가지 selection의 전략을 통해 얻은 결과값을 비교하였다.

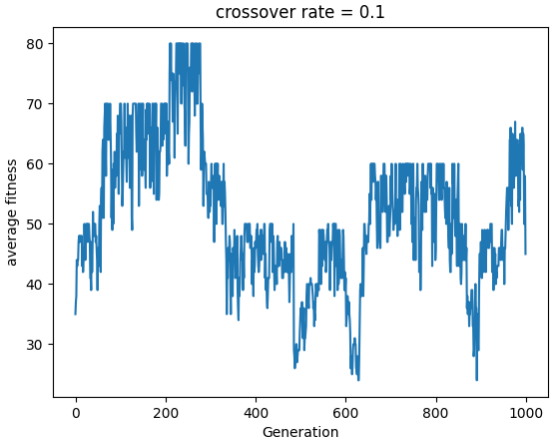
우선 두가지 selection 전략의 비교를 위해 crossover rate = 0.15, mutation rate = 0.005로 고정시켜두고 두 selection을 비교하여 그래프로 나타내었다.

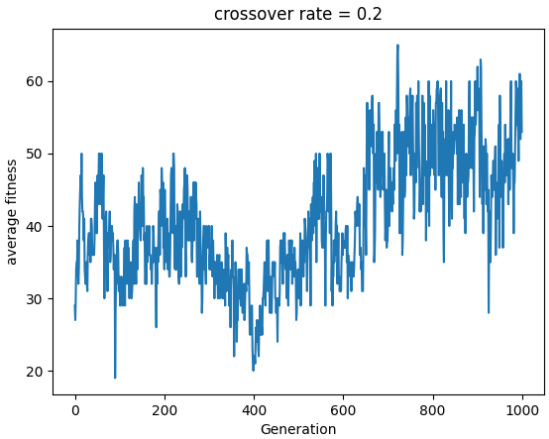


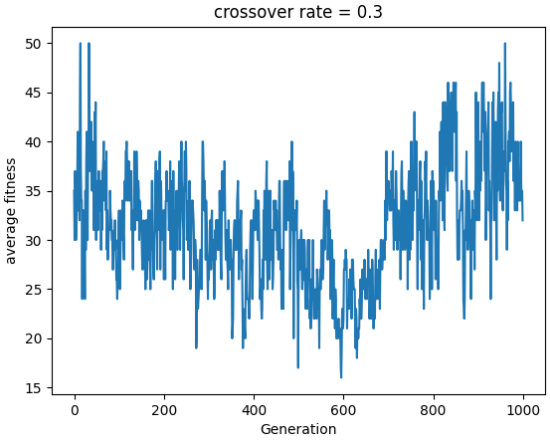
결과부터 보면 우선 두 전략 어느 쪽도 100점의 정답률을 보인 전략은 없었다. 여러 차례 반복하여 그래프를 그려보기도 하고 세대 수를 늘려보기도 했으나 시간만 더 걸릴 뿐 만족할만한 결과값을 도출해내지는 못했다. 또한 명백하게 증가만을 나타내는 그래프도 좀처럼 나타나지 않았으며 대부분 증가하다 감소하는 양상을 보였고 감소세만 보이는 그래프도 존재했다. 이는 global maximum에 도달하지 못하고 local maximum에 빠졌거나 효과적인 알고리즘을 구현하지 못했기 때문인것으로 추측된다. 두 전략의 차이점이라고 한다면 fitness proportionate selection은 비교적 곡선 형태의 추세를 보였으나 tournament selection은 계단형의 추세를 보인 것이 특징이라고 할 수 있겠다.

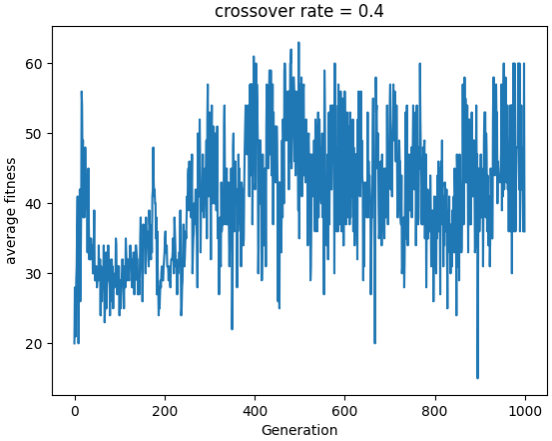
이번엔 selection 전략을 fitness proportionate selection으로 고정해두고 crossover rate와 mutation rate를 조정하면서 비교해보았다.

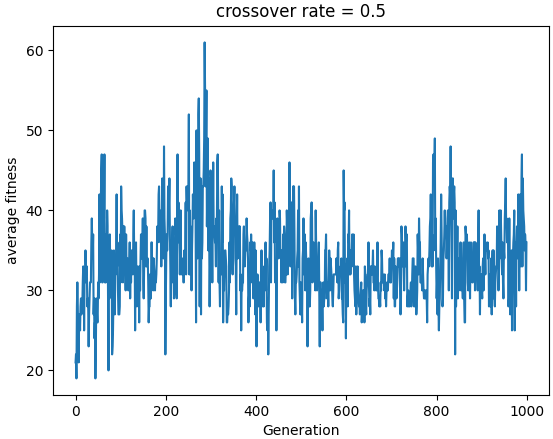
우선 mutation rate는 그대로 0.005로 고정시켜두고 crossover rate를 0.1 단위로 바꿔가며 실험해보았다.

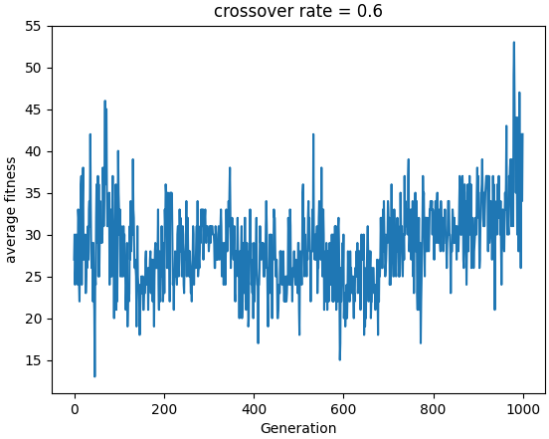

crossover rate = 0.1
average fitness
Generation

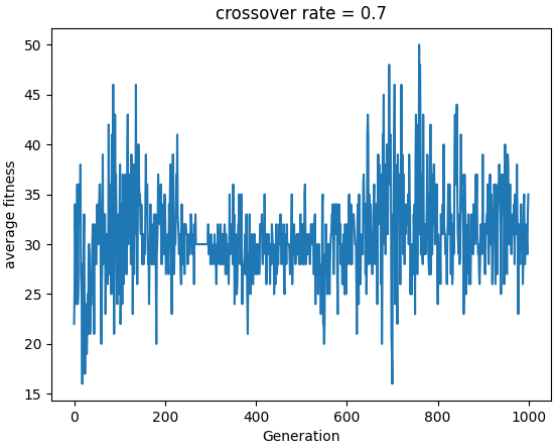

crossover rate = 0.2
average fitness
Generation

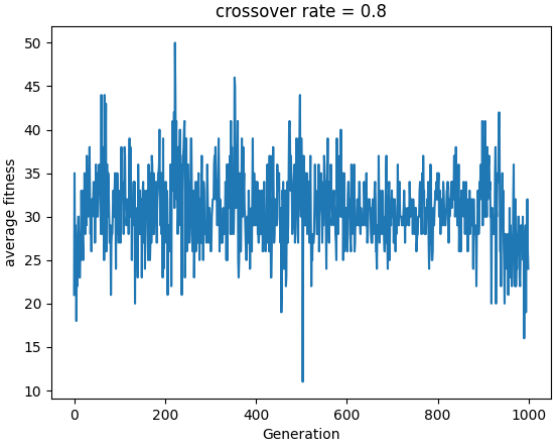

crossover rate = 0.3
average fitness
Generation


crossover rate = 0.4
average fitness
Generation


crossover rate = 0.5
average fitness
Generation

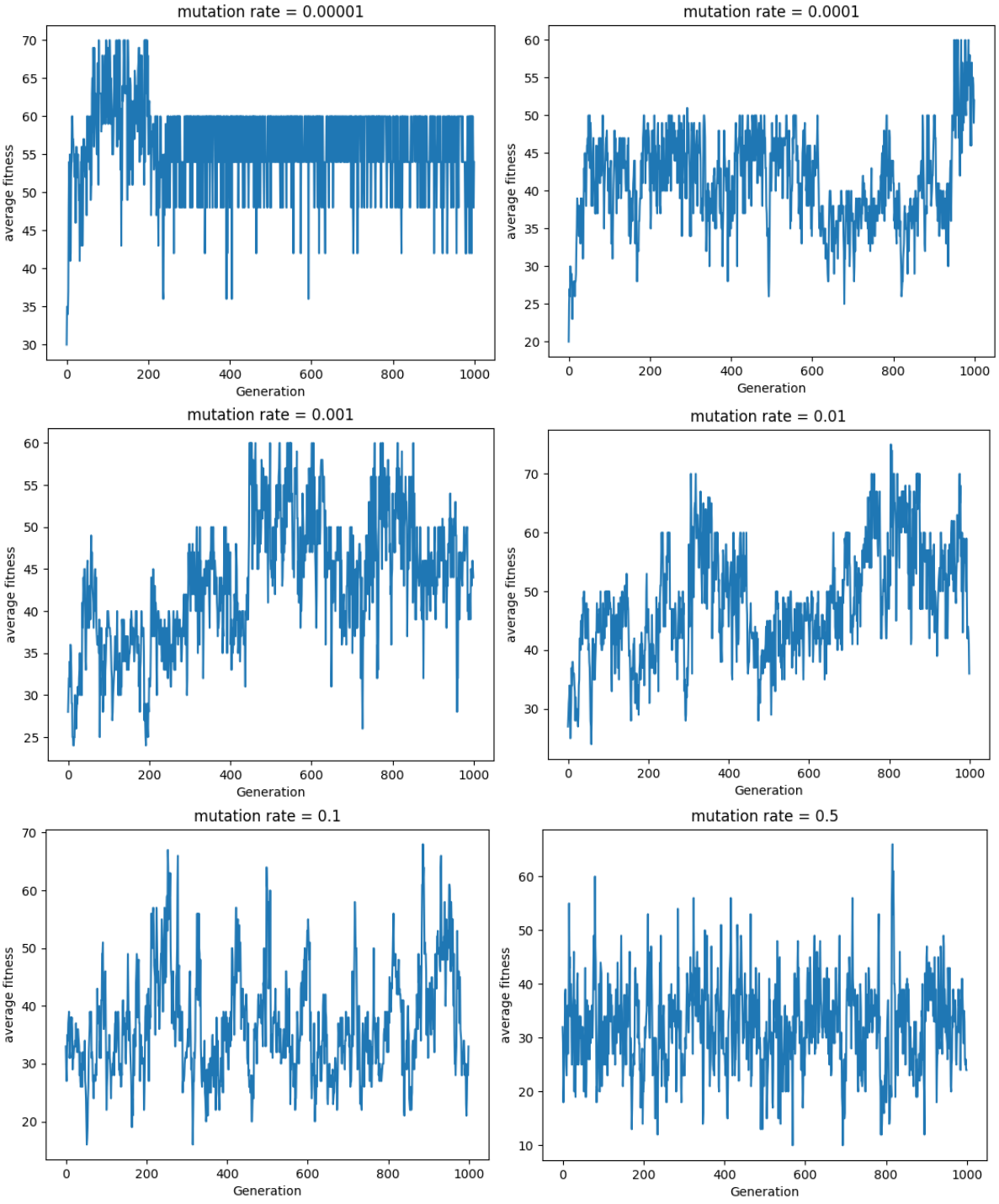

crossover rate = 0.6
average fitness
Generation


crossover rate = 0.7
average fitness
Generation


crossover rate = 0.8
average fitness
Generation

Crossover rate를 변화해가며 실험을 진행하면서 가장 유의미한 차이점은 crossover rate의 값이 증가할수록 정답률이 오히려 떨어진다는 점이었다. 제대로 된 구현에 실패한 것으로 보이기 때문에 올바른 실험 결과라고 보기에는 무리가 있을 것으로 예상되지만, 이러한 차이점이 나타난 이유는 임의의 요소가 더 증가하면서 오히려 불확실성이 커졌기 때문인 것으로 추측된다. 특이한 점은 crossover rate를 높은 값으로 두었을 때, 결과값은 목표치가 100점이 아닌 30점을 향해가는 것처럼 보였다는 것이다.

이번에는 상대적으로 높은 정답률을 보였던 crossover rate를 0.15로 고정해두고 mutation rate를 조정하면서 비교해보았다.



몇가지 주목할만한 차이점은 그래프의 추세와 정답률이었다. Mutation rate 값이 낮을수록 계단형에 가까운 그래프의 모습을 보였으며 상대적으로 높은 정답률을 기록했다. 반대로 mutation rate의 값이 높을수록 그래프는 더 변화가 심했으며 일정 수준 이상으로 mutation rate가 올라가면 앞서 높은 crossover rate의 그래프처럼 30점의 정답률을 유지하는 형태의 그래프를 보였다. 이러한 형태의 그래프가 나타난 이유는 mutation rate가 낮을 경우 주어진 population 내의 정답들이 변화할 확률이 낮아지면서 각 individual의 값이 큰 변화없이 비슷한 형태로 반복될 가능성이 높아졌기 때문인 것으로 보인다. 반대로 mutation rate가 높은 경우는 변이가 자주 일어나 변동성이 커져서 그래프의 변화가 증가한 것으로 보이며 높은 수준의 mutation rate는 변이가 너무 자주 일어나서 불확실성이 커진 탓에 정답률이 오히려 낮아진 것으로 보인다.

주어진 상황에 맞게 적절히 유전 알고리즘을 구현하였다고 생각했으나 어떤 부분에서 문제가 되어 적절한 결과값을 도출해내지 못했는지는 알 수 없었다. Crossover rate나 mutation rate 등의 여러 parameter 값을 조정하면서 일정 수준 정답률을 올리는데에는 성공했으나 역시 정확한 정답을 알아내는 수준까지는 도달하지 못했다. 의도적으로 정답을 포함한 population을 생성하여 유전 알고리즘에 입력해보기도 했으나 초기에는 높은 정답률을 보이다가 세대가 지남에 따라 오히려 정답률이 감소하는 것으로 보아 알고리즘이 의도한 대로 구동되지 않고 있다는 사실은 명확해보였다. 그러나 이 실습을 통해서 어떤 원리에 의해 유전 알고리즘이 정답을 찾아나가는지 확실히 알 수 있었으며 이에 따른 여러 전략들과 그 효과를 알 수 있었다.