N-Queens problem with Genetic Algorithm

과목명 : 머신러닝 및 응용

분반 : 060 분반

교수님 : 송 길태 교수님

조교님 : 추 동원 조교님

제출일 : 2020-04-16

00. 목차

장 수현

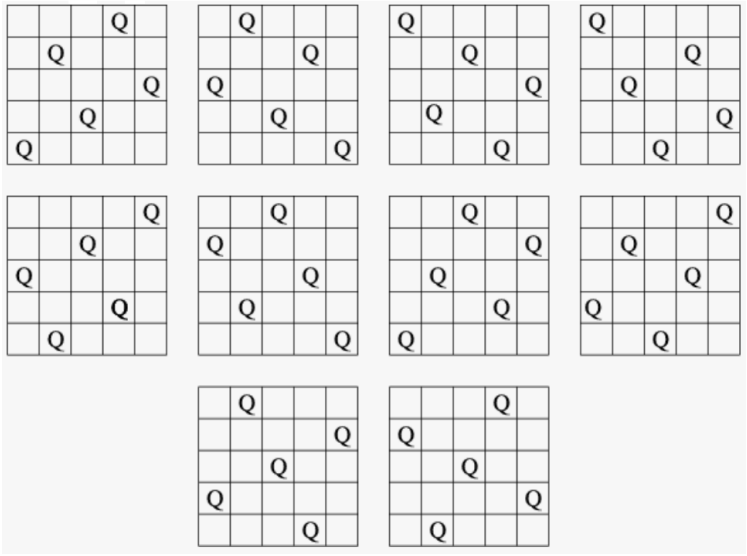
01. 과제 주제

1.1 과제 주제

N-Queen problem 중 5-Queen problem 을 Genetic Algorithm 을 사용하여 해결할 것.

1.2 N-Queen Problem

N-Queen problem 은 N×N 크기의 체스판에 N 개의 퀸을 규칙에 따라 배치하는 문제이다. 배치 규칙은 어떠한 퀸 쌍도 같은 가로, 세로, 대각선 상에 있어 서로를 공격하지 않도록 해야 한다는 것이다. 이번 과제에서는 N 을 5 로 정해 다음과 같이 나올 수 있는 10 가지 해 중 하나를 찾아야 한다.



1.3 Genetic Algorithm

Genetic Algorithm (유전 알고리즘)은 자연 세계의 진화과정에 기초한 계산 모델로서 최적화 문제를 해결하는 기법 중 하나이다. 생물의 진화를 모방하여 만든 알고리즘이고, 최적의 방법이 무엇인지 모르는 상태에서 알고리즘을 통해 최적 상태를 찾기 위한 것으로 진화 방향을 알 수 없다. 따라서 진화 방향을 의도하여 진화를 억제하지 않도록 주의해야한다. 그러므로 한 가지 형태의 답만 찾는 것이 아니라 코드를 실행할 때마다 1.2 에서의 10 개의 답 중 하나를 랜덤하게 찾아낼 수 있어야 한다. 유전자 알고리즘의 각 과정과 순서는 다음과 같다.

1.3.1 Chromosome Design

말 그대로 염색체를 디자인하는 것으로 알고리즘의 구현에 앞서 어떻게 데이터를 가지게 할 것인지 정한다.

1.3.2 Initialization

앞 과정에서 어떻게 데이터를 가질 지 정했다면, 어떤 데이터를 가질지 정하고 염색체를 초기화한다.

1.3.3 Fitness Evaluation

각 염색체의 적합도를 연산하는 과정으로 정해진 기준에 따라 해답에 가까운 지의 여부를 판단한다.

1.3.4 Selection

앞 과정에서 평가된 염색체들 중 다음 세대에 반영할 유리한 것을 선택하는 과정이다.

1.3.5 Crossover

선택된 염색체들끼리 서로 교차하여 새로운 개체를 만들어낸다. 기존 개체를 유지할 지는 선택 사항이다.

1.3.6 Mutation

새로운 세대를 구성하는 과정 중 돌연변이를 발생시켜 진화를 유도하도록 한다.

1.3.7 Update Generation

해를 찾을 때까지 세대를 증가시키는 과정으로 1.3.3~1.3.6 을 반복한다.

02. 과제 해결

2.1 Chromosome Design

2.1.1 코드 구현

```
N = 5
NUM_CHROMOSOME = 16
NUM_SELECTION = 2
NUM_AFTER_CROSS = (NUM_SELECTION*2)

chromosome=[[0 for i in range(N)] for j in range(NUM_CHROMOSOME)]
```

1) 상수 설정

N 은 5 를 대입하여 5 퀸으로 진행했고, 전체 개체 수를 코드 구현 시 빠른 디버깅을 위해 우선 16 으로 설정했다. Fitness evaluation 이후 선택할 개체 수는 2 로 설정했고, 2 개로 crossover 를 하고 기존의 개체를 합하면 그 두배인 4 가 된다.

2) 한 염색체 개체의 자료구조 선택

N 개의 퀸의 위치를 어떻게 넣을 지 선택했어야 했는데, 처음 생각했던 건 Boolean 값을 가지는 2 차원 배열 혹은 x, y 좌표를 가진 1 차원 배열이었다. 정보의 양이 더 많아 효율적일 것이라 생각했지만, 같은 가로 줄과 세로 줄에 있는지 두 가지 경우 모두에 대해 체크해야 하는 번거로움이 있었다. 그래서 위치하고 있는 행의 정보를 가지는 1 차원 배열로 선택했다. 이렇게 했을 때는 무조건 한 열에 하나의 퀸만 들어가므로 같은 열 상에 있는지 따로 확인할 필요가 없다. 그리고 1 차원 배열을 개체 수만큼 가지는 한 세대의 전체 개체를 2 차원 배열로 chromosome 이라고 설정했다.

예를 들어 한 개체가 [4, 1, 3, 2, 0]이라고 하면, 5 개의 퀸이 다음과 같이 위치해 있는 것이다.

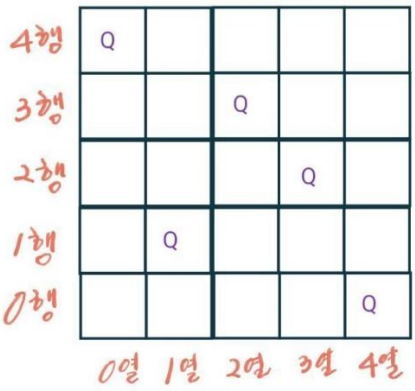


그림 1-1. Chromosome 표현 예시 그림.

2.2 Initialization

2.2.1 코드 구현

```
def initialization():
    for i in range(0,len(chromosome)):
        for j in range(0,len(chromosome[i])):
            chromosome[i][j] = random.randint(0, N-1)
```

1) chromosome 초기화

각 chromosome 을 초기화할 때, 0~4 까지의 값으로 초기화 했다.

2.2.2 함수 실행 결과

```
*** Remote Interpreter Reinitialized  ***
======================== CHROMOSOME START ========================

GENERATION :  0

[ 0 ]   0  :  3     1  :  3     2  :  3     3  :  4     4  :  3
[ 1 ]   0  :  3     1  :  1     2  :  1     3  :  2     4  :  0
[ 2 ]   0  :  3     1  :  2     2  :  3     3  :  1     4  :  2
[ 3 ]   0  :  3     1  :  0     2  :  4     3  :  1     4  :  1
[ 4 ]   0  :  3     1  :  4     2  :  2     3  :  3     4  :  3
[ 5 ]   0  :  0     1  :  2     2  :  1     3  :  1     4  :  1
[ 6 ]   0  :  0     1  :  0     2  :  3     3  :  2     4  :  1
[ 7 ]   0  :  2     1  :  1     2  :  4     3  :  4     4  :  1
[ 8 ]   0  :  3     1  :  3     2  :  0     3  :  1     4  :  2
[ 9 ]   0  :  1     1  :  2     2  :  4     3  :  2     4  :  1
[ 10 ]  0  :  4     1  :  3     2  :  3     3  :  1     4  :  3
[ 11 ]  0  :  0     1  :  1     2  :  1     3  :  0     4  :  0
[ 12 ]  0  :  4     1  :  1     2  :  3     3  :  3     4  :  3
[ 13 ]  0  :  4     1  :  2     2  :  4     3  :  3     4  :  1
[ 14 ]  0  :  3     1  :  1     2  :  4     3  :  4     4  :  3
[ 15 ]  0  :  3     1  :  3     2  :  1     3  :  1     4  :  2

========================= CHROMOSOME END =========================
```

그림 2-2. Initialization 함수 실행 결과. 각 열에 행 숫자가 임의로 들어가 있다.

2.3 Fitness Evaluation

2.3.1 코드 구현

```
BAD_WEIGHT = 3
GOOD_WEIGHT = 5
NUM_PAIR_NEIGHBOR = N*(N-1)//2
MAX_SCORE = (NUM_PAIR_NEIGHBOR*GOOD_WEIGHT)

score_fe=[[0,0]for i in range(NUM_CHROMOSOME)]

def num_duplication(this_chromosome):
    dup_num = 0
    dup_cnt = [0 for i in range(N)]

    for i in range(0,len(chromosome[this_chromosome])):
        dup_cnt[chromosome[this_chromosome][i]] += 1

    for i in range(0,len(dup_cnt)):
        if (dup_cnt[i] > 1):
            dup_num += dup_cnt[i]

    return dup_num


def num_good_neighbor(this_chromosome):
    num_nei = 0;

    for i in range(0,len(chromosome[this_chromosome])-1):
        for j in range(i+1,len(chromosome[this_chromosome])):
            if (not ( chromosome[this_chromosome][i] == chromosome[this_chromosome][j]
                      or j-i == abs(chromosome[this_chromosome][i] -
                      chromosome[this_chromosome][j]) )):
                num_nei += 1

    return num_nei

def fitness_evaluation():
    for i in range(0,len(score_fe)):
        score_fe[i][0] = i
```

```
        bad_score = num_duplication(i) * BAD_WEIGHT;
        good_score = num_good_neighbor(i) * GOOD_WEIGHT;
        score_fe[i][1] = good_score - bad_score; # 점수가 MAX_SCORE 일 경우 해를 찾은 것.

    score_fe.sort(key=lambda x: x[1],reverse=True)
```

1) fitness evaluation 내부 함수 호출

전체 chromosome 에 대해서 각 개체 별 점수 부여를 위해 가지고 있는 이웃한 열과의 관계를 계산한다. Num_duplication 함수를 호출한 뒤 반환 받은 값에 bad 가중치를 곱해 bad_score 에 저장하고, num_good_neighbor 함수를 호출한 뒤 반환 받은 값에 good 가중치를 곱해 good_score 에 저장한다.

2) Num_duplication 함수 수행

Chromosome 의 id 를 받고, 해당 chromosome 의 사이즈 만큼 for 문을 돌면서 같은 숫자를 가지고 있는지 판단한다. 2 개 이상의 수를 가지는 숫자의 개수를 카운트하여 반환한다.

3) num_good_neighbor 함수 수행

Chromosome 의 id 를 받고 해당 chromosome 이 가진 N 의 위치 정보로 같은 행에 있는지, 그리고 같은 대각선에 있는지 체크한다. 비교하는 N 의 쌍은 N*(N-1)//2 로 서로를 공격하지 않는 위치에 있다고 판단되면 좋은 이웃이라 판단하여 카운트하고 최종 카운트 값을 반환한다.

4) fitness evaluation 점수 부여 및 정렬

전달 받은 값에 각 가중치를 곱하여 더한 뒤, score_fe 배열에 두 번째 자리에 저장한다. score_fe 배열의 첫 번째 자리에는 각 chromosome 의 id 가 저장되어 있고, 계산된 점수를 기준으로 내림차순으로 정렬한다. score_fe[0][0]은 optimal 한 값을 가지고 있는 chromosome 의 id 이고, score_fe[0][1]은 해당하는 점수이다. 만일 score_fe[0][1]의 값이 MAX_SCORE 일 경우, 해를 찾은 것이다. MAX_SCORE 는 비교 가능한 N 의 pair 에 가중치를 곱한 것으로 계산한다.

2.3.2 함수 실행 결과

```
Sort!
======================== SCORE_FE START ========================

[ 3 ]  39
[ 13 ]  34
[ 1 ]  29
[ 14 ]  23
[ 5 ]  21
[ 12 ]  21
[ 6 ]  19
[ 8 ]  19
[ 7 ]  18
[ 9 ]  18
[ 4 ]  16
[ 2 ]  13
[ 15 ]  13
[ 10 ]  11
[ 11 ]  5
[ 0 ]  -2

======================== SCORE_FE END ========================
GENERATION :  0
Optimal Chromosome : 3  0  4  1  1
```

그림 3-1. Fitness Evaluation 실행 결과.

```
*** Remote Interpreter Reinitialized  ***
======================== CHROMOSOME START ========================

GENERATION :  0

[ 0 ]   0  :  3     1  :  3     2  :  3     3  :  4     4  :  3
[ 1 ]   0  :  3     1  :  1     2  :  1     3  :  2     4  :  0
[ 2 ]   0  :  3     1  :  2     2  :  3     3  :  1     4  :  2
[ 3 ]   0  :  3     1  :  0     2  :  4     3  :  1     4  :  1
[ 4 ]   0  :  3     1  :  4     2  :  2     3  :  3     4  :  3
[ 5 ]   0  :  0     1  :  2     2  :  1     3  :  1     4  :  1
[ 6 ]   0  :  0     1  :  0     2  :  3     3  :  2     4  :  1
[ 7 ]   0  :  2     1  :  1     2  :  4     3  :  4     4  :  1
[ 8 ]   0  :  3     1  :  3     2  :  0     3  :  1     4  :  2
[ 9 ]   0  :  1     1  :  2     2  :  4     3  :  2     4  :  1
[ 10 ]  0  :  4     1  :  3     2  :  3     3  :  1     4  :  3
[ 11 ]  0  :  0     1  :  1     2  :  1     3  :  0     4  :  0
[ 12 ]  0  :  4     1  :  1     2  :  3     3  :  3     4  :  3
[ 13 ]  0  :  4     1  :  2     2  :  4     3  :  3     4  :  1
[ 14 ]  0  :  3     1  :  1     2  :  4     3  :  4     4  :  3
[ 15 ]  0  :  3     1  :  3     2  :  1     3  :  1     4  :  2

======================== CHROMOSOME END ========================
```

그림 3-2. Initialization 함수 실행 결과.

그림 3-1 은 score_fe 배열의 정렬 결과인데, 39 로 적합도가 제일 높은 chromosome 의 id 는 3 이다. optimal chromosome 의 값은 [3 0 4 1 1] 이고, 이 값은 그림 3-2 의 id 가 3 인 chromosome 의 값과 같으므로 같은 chromosome 임을 확인할 수 있다.

2.4 Selection

2.4.1 코드 구현

```
selected_chromosome=[[0 for i in range(N)] for j in range(NUM_SELECTION)]

def selection():
    for i in range(0,NUM_SELECTION):
        for j in range(0,N):
            selected_chromosome[i][j] = chromosome[score_fe[i][0]][j]
```

1) 적합도가 높은 chromosome 선별

score_fe 배열에서 NUM_SELECTION 만큼 적합도가 높은 순서대로 chromosome 의 id 를 추출하고, 해당하는 chromosome 을 selected_chromosome 에 넣는다. 그리고 새로운 세대를 준비하므로 전역변수로 선언해 두었던 generation 을 +1 증가시킨다.

2.4.2 함수 실행 결과

```
==================== SELECTED CHROMOSOME START ====================

GENERATION :  1

[ 0 ]   0 : 3    1 : 0    2 : 4    3 : 1    4 : 1
[ 1 ]   0 : 4    1 : 2    2 : 4    3 : 3    4 : 1

====================== SELECTED CHROMOSOME END ====================
```

그림 4-1. Selection 함수 실행 결과.

```
======================== SCORE_FE START ========================

[ 3 ]  39
[ 13 ]  34
[ 1 ]  29
[ 14 ]  23
[ 5 ]  21
[ 12 ]  21
[ 6 ]  19
[ 8 ]  19
[ 7 ]  18
[ 9 ]  18
[ 4 ]  16
[ 2 ]  13
[ 15 ]  13
[ 10 ]  11
[ 11 ]  5
[ 0 ]  -2

========================= SCORE_FE END =========================
```

그림 4-2. Fitness Evaluation 실행 결과.

```
*** Remote Interpreter Reinitialized  ***
======================== CHROMOSOME START ========================

GENERATION :  0

[ 0 ]   0 : 3    1 : 3    2 : 3    3 : 4    4 : 3
[ 1 ]   0 : 3    1 : 1    2 : 1    3 : 2    4 : 0
[ 2 ]   0 : 3    1 : 2    2 : 3    3 : 1    4 : 2
[ 3 ]   0 : 3    1 : 0    2 : 4    3 : 1    4 : 1
[ 4 ]   0 : 3    1 : 4    2 : 2    3 : 3    4 : 3
[ 5 ]   0 : 0    1 : 2    2 : 1    3 : 1    4 : 1
[ 6 ]   0 : 0    1 : 0    2 : 3    3 : 2    4 : 1
[ 7 ]   0 : 2    1 : 1    2 : 4    3 : 4    4 : 1
[ 8 ]   0 : 3    1 : 3    2 : 0    3 : 1    4 : 2
[ 9 ]   0 : 1    1 : 2    2 : 4    3 : 2    4 : 1
[ 10 ]  0 : 4    1 : 3    2 : 3    3 : 1    4 : 3
[ 11 ]  0 : 0    1 : 1    2 : 1    3 : 0    4 : 0
[ 12 ]  0 : 4    1 : 1    2 : 3    3 : 3    4 : 3
[ 13 ]  0 : 4    1 : 2    2 : 4    3 : 3    4 : 1
[ 14 ]  0 : 3    1 : 1    2 : 4    3 : 4    4 : 3
[ 15 ]  0 : 3    1 : 3    2 : 1    3 : 1    4 : 2

======================== CHROMOSOME END ========================
```

그림 4-3. Initialization 함수 실행 결과.

그림 4-1 은 selection 후 secte d_chromose 에 들어있는 chromosome 의 출력결과이다. score_fe 배열의 상위 2 개의 id 를 보면 3 과 13 인데, 그림 4-3 에서 id 가 3 과 13 인 chromosome 의 값과 그림 4-1 에서 selected_chromosome 의 값과 같으므로 같은 chromosome 임을 확인할 수 있다.

2.5 Crossover

2.5.1 코드 구현

```
def push_crossover_after():
    for i in range(0,len(selected_chromosome),2):
        cross_begin = random.randint(0, N)
        cross_end = random.randint(0, N)
        if (cross_begin > cross_end):
            cross_begin, cross_end = cross_end, cross_begin #swap


        # cross_begin 이상 cross_end 미만 구간 crossover.
        for j in range(0,cross_begin):
            chromosome[i][j] = selected_chromosome[i][j]
            chromosome[i+1][j] = selected_chromosome[i+1][j]

        for j in range(cross_begin,cross_end):
            chromosome[i][j] = selected_chromosome[i+1][j]
            chromosome[i+1][j] = selected_chromosome[i][j]

        for j in range(cross_end,len(chromosome[i])):
            chromosome[i][j] = selected_chromosome[i][j]
            chromosome[i+1][j] = selected_chromosome[i+1][j]


def push_crossover_before():
    for i,k in zip(range(NUM_SELECTION,NUM_SELECTION+len(selected_chromosome)),
```

```
                range(0,len(selected_chromosome)) ):
        for j in range(0,len(selected_chromosome[k])):
            chromosome[i][j] = selected_chromosome[k][j]

def crossover():
    push_crossover_after()
    push_crossover_before()
```

1) crossover 내부 함수 호출

선별한 chromosome 을 crossover 하여 새로운 세대의 chromosome 에 추가하는 push_crossover_after 함수를 호출한다. 그리고 crossover 하지 않은 기존의 선별된 chromosome 도 추가해주는 push_crossover_before 함수를 호출한다.

2) push_crossover_after 함수 수행

선별한 chromosome 에 crossover 를 진행한다. 교차를 하는 시작점인 cross_begin 과 종점인 cross_end 를 0 부터 N 중에서 임의로 할당한다. 만일, 시작점의 값이 더 클 경우 swap 한다. 이제는 필요 없어진 구세대의 chromosome 2 차원 배열에 교차를 하여 chromosome 의 값을 집어넣는다. 교차는 시작점 이상 종점 미만 구간 동안 시행된다.

3) push_crossover_before 함수 수행

Crossover 하지 않은 기존의 선별된 chromosome 도 교차된 chromosome 이 들어간 이후자리에 값을 집어넣는다.

2.5.2 함수 실행 결과

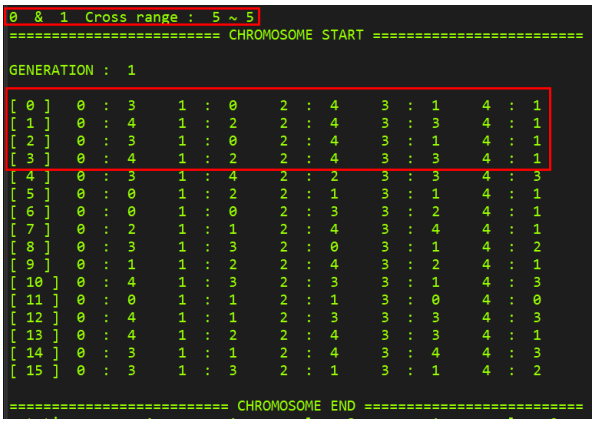


그림 5-1. Crossover 함수 실행 결과.

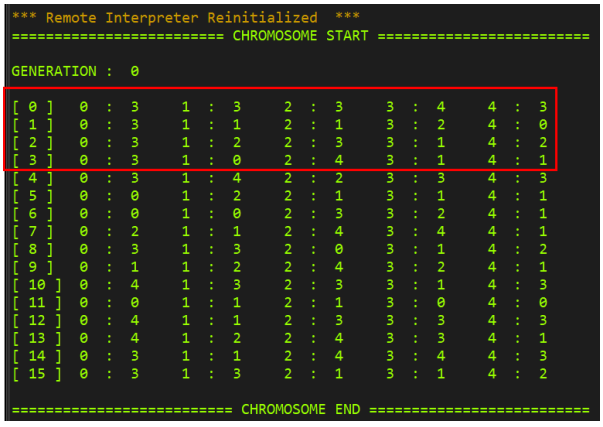


그림 5-2. Initialization 함수 실행 결과.

그림 5-1 은 Crossover 실행 결과 화면인데, 하필 cross 범위가 5~5 가 나와서 crossover 되지 않았다. crossover 시 지금과 같은 경우가 아니라 무조건 crossover 되도록 설정할 수 있지만 나는 모든 것을 임의로 결정되게 했다. 왜냐하면 유전자 알고리즘의 주의사항이 답을 모르는 상태로 진화되면서 답을 찾는 방식이므로 가능한 한 통제하려고 하지 않았다. 그래서 이 경우와 같이 랜덤 값에 의해 crossover 가 되지 않는 경우도 허용했다. 그리고 chromosome 상태를 보면, 그림 5-2 의 0 세대에서 0~3 을 제외하고는 똑같음을 확인할 수 있다. 구세대의 chromosome 에 새로운 chromosome 이 덮어씌워지고 있는 과정임을 확인할 수 있다.

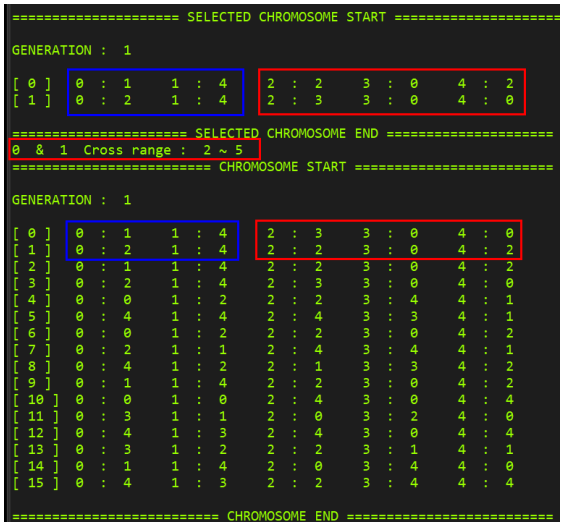


그림 5-3. 또 다른 실행에서의 Crossover 함수 실행 결과.

그림 5-3 은 또 다른 실행에서의 Crossover 실행 결과 화면인데, 그림 5-1 에서 확인하지 못했던 crossover 의 결과를 확인할 수 있다. Cross 범위가 2~5 가 나왔고, 인덱스 2 부터 4 까지 교차를 진행한다. 그림 5-3 을 보면, 교차 범위인 빨간색 테두리 구간에서 교차가 일어났고, 교차 범위 밖인 파란색 테두리 구간에서는 교차가 발생하지 않았음을 확인할 수 있다.

2.6 Mutation

2.6.1 코드 구현

```
def mutate(mutation_chromosome):
    num_mutation = random.randint(0, N)
    for i in range(0,num_mutation):
        pos = random.randint(0, N-1)
        new_val = random.randint(0, N-1)
        mutation_chromosome[pos] = new_val;


def push_mutation(round):
    mutation_chromosome = [0 for i in range(N)]

    for i in range(0,NUM_AFTER_CROSS):
        for j in range(0,len(chromosome[i])):
            mutation_chromosome[j] = chromosome[i][j]

        mutate(mutation_chromosome)

        for j in range(0,len(mutation_chromosome)):
            chromosome[i+round*NUM_AFTER_CROSS][j] = mutation_chromosome[j]


def mutation():
    NUM_ROUND = (NUM_CHROMOSOME - NUM_AFTER_CROSS) // NUM_AFTER_CROSS;
    for i in range(1,NUM_ROUND+1):
        push_mutation(i)
```

1) Mutation 횟수 계산 및 Mutation 내부 함수 호출

전체 chromosome 에서 crossover 결과로 만들어진 chromosome 을 제외하고 남은 chromosome 수를 Mutation 결과로 채워야 한다. 그래서 돌연변이 발생 횟수를 계산하면 NUM_CHROMOSOME - NUM_AFTER_CROSS) // NUM_AFTER_CROSS;이고 이를 NUM_ROUND 에 저장하여 해당 횟수만큼 돌연변이를 발생시킨다. 이때 주의해야 할 점이 있는데, for 문을 0 부터 수행하게 되면 chromosome 을 채워 넣는 과정에서 이미 들어가 있는 crossover 된 chromosome 자리에 덮어씌워지므로 반드시 1 부터 수행해야한다.

2) push_mutation 함수 수행

한 횟수에 crossover 이후 새로 생긴 chromosome 수에 해당하는 만큼 돌연변이 chromosome 이 생성된다. 그만큼 for 문을 수행하면서 임의의 배열 mutation_chromosome 에 crossover 결과로 생성된 모든 chromosome 을 대입한다. 그리고 해당 배열을 mutate 함수에 넘겨주어 돌연변이 발생을 진행한다.

3) mutate 함수 수행

넘겨받은 mutation_chromosome 에 대해 돌연변이 발생을 수행하는데, 돌연변이가 발생할 개수, 돌연변이가 일어날 자리, 돌연변이 후 바뀌는 값 세 가지 모두에 대해 임의의 값을 할당한다. 확률적으로 조정하지 않고, 모든 것을 임의로 지정하는 이유는 2.5 에서 언급한 바와 같이 진화 방향을 의도하지 않기 위함이다. 돌연변이에 의해서 mutation_chromosome 이 모두 변환된 뒤, push_mutation 에 반환하여 chromosome 배열에 채워 넣는다. 처음 시작했던 개체 수만큼 chromosome 이 새로 생성되면 새로운 한 세대가 완성된다.

2.6.2 함수 실행 결과

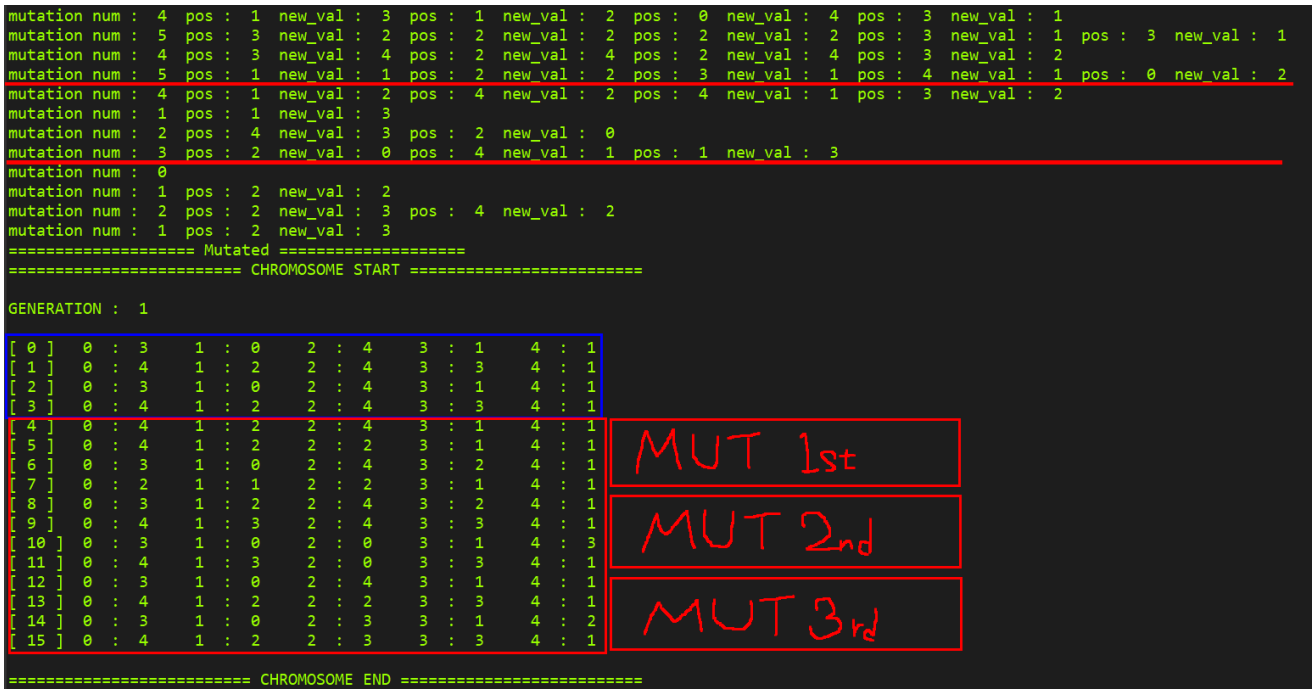


그림 6-1. Mutation 함수 실행 결과.

그림 6-1 은 Mutation 함수 실행 결과로 완성된 새로운 세대의 chromosome 을 모두 출력한 화면이다. 파란색 테두리로 둘러싸인 4 개의 chromosome 은 crossover 결과로 생성된 것이고, 빨간색 테두리로 둘러싸인 12 개의 chromosome 은 mutation 결과로 생성된 것이다. Mutation 은 총 3 번에 걸쳐 발생했고, 각 mutation 에서의 횟수, 자리, 새로운 값도 함께 출력했다.

2.7 Update Generation

2.7.1 코드 구현

```
def update_generation():
    global generation
    generation += 1

    selection()
    crossover()
    mutation()
    fitness_evaluation()

    print_optimal_chromosome()
```

1) update_generation 내부 함수 호출

각 구세대에서 신세대로 세대 갱신이 일어나려면 계산된 적합도를 보고 개체를 선별하는 selection 함수, 선별된 selection 함수를 기반으로 교차를 수행하는 crossover 함수, 돌연변이를 발생시키는 mutation 함수, 한 세대가 완성되고 난 뒤, 적합도를 계산하고 chromosome 을 정렬하는 fitness_evaluation 함수를 차례로 수행한다. 그리고 다음 세대로 넘어가기 전 답이 아니더라도 최적의 chromosome 을 출력하고, 만일 score 가 MAX_SCORE 를 가지는 해 chromosome 이 발견되면 세대 갱신을 그만둔다.

2.7.2 함수 실행 결과

```
GENERATION :  11
Optimal Chromosome : 3  1  4  2  0

The answere of 5-Queen is [3 1 4 2 0].
It's generation is 11.
>>> 우
```

그림 7-1. update_generation 함수 실행 결과.

그림 7-1 은 update_generation 을 반복 수행하다가 답을 찾고 프로그램이 종료된 후의 결과 화면이다. 11 세대만에 해를 찾아냈고 해당하는 chromosome 은 [3 1 4 2 0]임을 알 수 있다.

2.8 Draw Answere

2.8.1 코드 구현

```
def draw_answer():
    cell_text = [['' for i in range(N)] for j in range(N)]

    for i in range(0,N):
        cell_text[(N-1)-chromosome[score_fe[0][0]][i]][i] = 'Q'

    fig, ax = plt.subplots()
    ax.axis('off')

    the_table = ax.table(cellText=cell_text,loc='center',cellLoc='center')
    the_table.set_fontsize(16)
    the_table.scale(1,25/N)

    plt.savefig('201724557_장수현.png', dpi=300)
    plt.show()
```

1) matplotlib 을 사용한 답 출력.

Matplotlib 을 사용하여 최종 답을 보기 쉽게 테이블로 출력하도록 한 함수다. N×N 의 테이블에 넣기 위해 1 차원이었던 chromosome 을 2 차원으로 풀어 Queen 이 존재하는 곳에 'Q'를 출력하도록 했다. 출력되는 이미지 파일은 '201724557_장수현.png'라는 파일명으로 저장된다.

2.8.2 함수 실행 결과

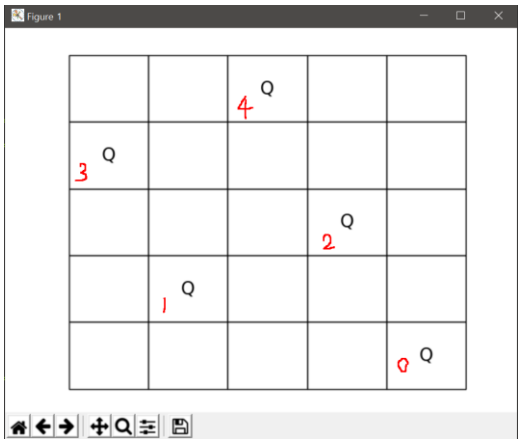


그림 8-1. draw_answer 함수 실행 결과.

03. 전체 코드

python 으로 작성한 이번 과제의 전체 코드는 다음과 같다.

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

N = 5
NUM_CHROMOSOME = 16
NUM_SELECTION = 2
NUM_AFTER_CROSS = (NUM_SELECTION*2)

BAD_WEIGHT = 3
GOOD_WEIGHT = 5
NUM_PAIR_NEIGHBOR = N*(N-1)//2
MAX_SCORE = (NUM_PAIR_NEIGHBOR*GOOD_WEIGHT)

# 전역 선언 #
generation = 0
cross_begin, cross_end = 0, 0
chromosome=[[0 for i in range(N)] for j in range(NUM_CHROMOSOME)]
score_fe=[[0,0]for i in range(NUM_CHROMOSOME)]
selected_chromosome=[[0 for i in range(N)] for j in range(NUM_SELECTION)]

def print_optimal_chromosome():
    global generation
    print("GENERATION : ",generation)
    print("Optimal Chromosome : ",end='')
    for i in range(0,len(chromosome[score_fe[0][0]])):
        print(chromosome[score_fe[0][0]][i]," ",end='')
    print('\n')

def print_answere():
    global generation
    print("The answere of "+str(N)+"-Queen is [",end='')
    for i in range(0,len(chromosome[score_fe[0][0]])-1):
        print(chromosome[score_fe[0][0]][i],"",end='')
    print(chromosome[score_fe[0][0]][-1],end='')
    print("].''\n'"It's generation is "+str(generation)+".")

def initialization():
    for i in range(0,len(chromosome)):
        for j in range(0,len(chromosome[i])):
            chromosome[i][j] = random.randint(0, N-1)


def num_duplication(this_chromosome):
    dup_num = 0
    dup_cnt = [0 for i in range(N)]

    for i in range(0,len(chromosome[this_chromosome])):
        dup_cnt[chromosome[this_chromosome][i]] += 1

    for i in range(0,len(dup_cnt)):
        if (dup_cnt[i] > 1):
            dup_num += dup_cnt[i]

    return dup_num
```

```
def num_good_neighbor(this_chromosome):
    num_nei = 0;

    for i in range(0,len(chromosome[this_chromosome])-1):
        for j in range(i+1,len(chromosome[this_chromosome])):
            if (not ( chromosome[this_chromosome][i] == chromosome[this_chromosome][j]
or
                    j-i == abs(chromosome[this_chromosome][i] -
chromosome[this_chromosome][j]) )):
                num_nei += 1

    return num_nei


def fitness_evaluation():
    for i in range(0,len(score_fe)):
        score_fe[i][0] = i
        bad_score = num_duplication(i) * BAD_WEIGHT;
        good_score = num_good_neighbor(i) * GOOD_WEIGHT;
        score_fe[i][1] = good_score - bad_score;
        # 점수가 MAX_SCORE 일 경우 해를 찾은 것.

    score_fe.sort(key=lambda x: x[1],reverse=True)


def selection():
    for i in range(0,NUM_SELECTION):
        for j in range(0,N):
            selected_chromosome[i][j] = chromosome[score_fe[i][0]][j]


def push_crossover_after():
    for i in range(0,len(selected_chromosome),2):
        cross_begin = random.randint(0, N)
        cross_end = random.randint(0, N)
        if (cross_begin > cross_end):
            cross_begin, cross_end = cross_end, cross_begin #swap


        # cross_begin 이상 cross_end 미만 구간 crossover.
        for j in range(0,cross_begin):
            chromosome[i][j] = selected_chromosome[i][j]
            chromosome[i+1][j] = selected_chromosome[i+1][j]

        for j in range(cross_begin,cross_end):
            chromosome[i][j] = selected_chromosome[i+1][j]
            chromosome[i+1][j] = selected_chromosome[i][j]

        for j in range(cross_end,len(chromosome[i])):
            chromosome[i][j] = selected_chromosome[i][j]
            chromosome[i+1][j] = selected_chromosome[i+1][j]


def push_crossover_before():
    for i,k in zip(range(NUM_SELECTION,NUM_SELECTION+len(selected_chromosome)),
                range(0,len(selected_chromosome)) ):
        for j in range(0,len(selected_chromosome[k])):
            chromosome[i][j] = selected_chromosome[k][j]

def crossover():
```

```
    push_crossover_after()
    push_crossover_before()


def mutate(mutation_chromosome):
    num_mutation = random.randint(0, N)
    for i in range(0,num_mutation):
        pos = random.randint(0, N-1)
        new_val = random.randint(0, N-1)
        mutation_chromosome[pos] = new_val;


def push_mutation(round):
    mutation_chromosome = [0 for i in range(N)]

    for i in range(0,NUM_AFTER_CROSS):
        for j in range(0,len(chromosome[i])):
            mutation_chromosome[j] = chromosome[i][j]

        mutate(mutation_chromosome)

        for j in range(0,len(mutation_chromosome)):
            chromosome[i+round*NUM_AFTER_CROSS][j] = mutation_chromosome[j]


def mutation():
    NUM_ROUND = (NUM_CHROMOSOME - NUM_AFTER_CROSS) // NUM_AFTER_CROSS;
    for i in range(1,NUM_ROUND+1):
        push_mutation(i)


def update_generation():
    global generation
    generation += 1

    selection()
    crossover()
    mutation()
    fitness_evaluation()

    print_optimal_chromosome()

def draw_answer():
    cell_text = [['' for i in range(N)] for j in range(N)]

    for i in range(0,N):
        cell_text[(N-1)-chromosome[score_fe[0][0]][i]][i] = 'Q'

    fig, ax = plt.subplots()
    ax.axis('off')

    the_table = ax.table(cellText=cell_text,loc='center',cellLoc='center')
    the_table.set_fontsize(16)
    the_table.scale(1,25/N)

    plt.savefig('201724557_장수현.png', dpi=300)
    plt.show()

def n_queen_with_genetic_algorithm():
    initialization()
    fitness_evaluation()
```

```
    print_optimal_chromosome()

    while(not(score_fe[0][1]==MAX_SCORE)):
        update_generation()

    print_answere()
    draw_answer()

def main():
    os.system('cls')
    if N > 3:
        n_queen_with_genetic_algorithm()
    else:
        print("It cannot be run. Please change value of N.")

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

04. 결과

4.1 상수 변경 시

4.1.1 전체 개체 수 : 16, 선별 개체 수 : 2

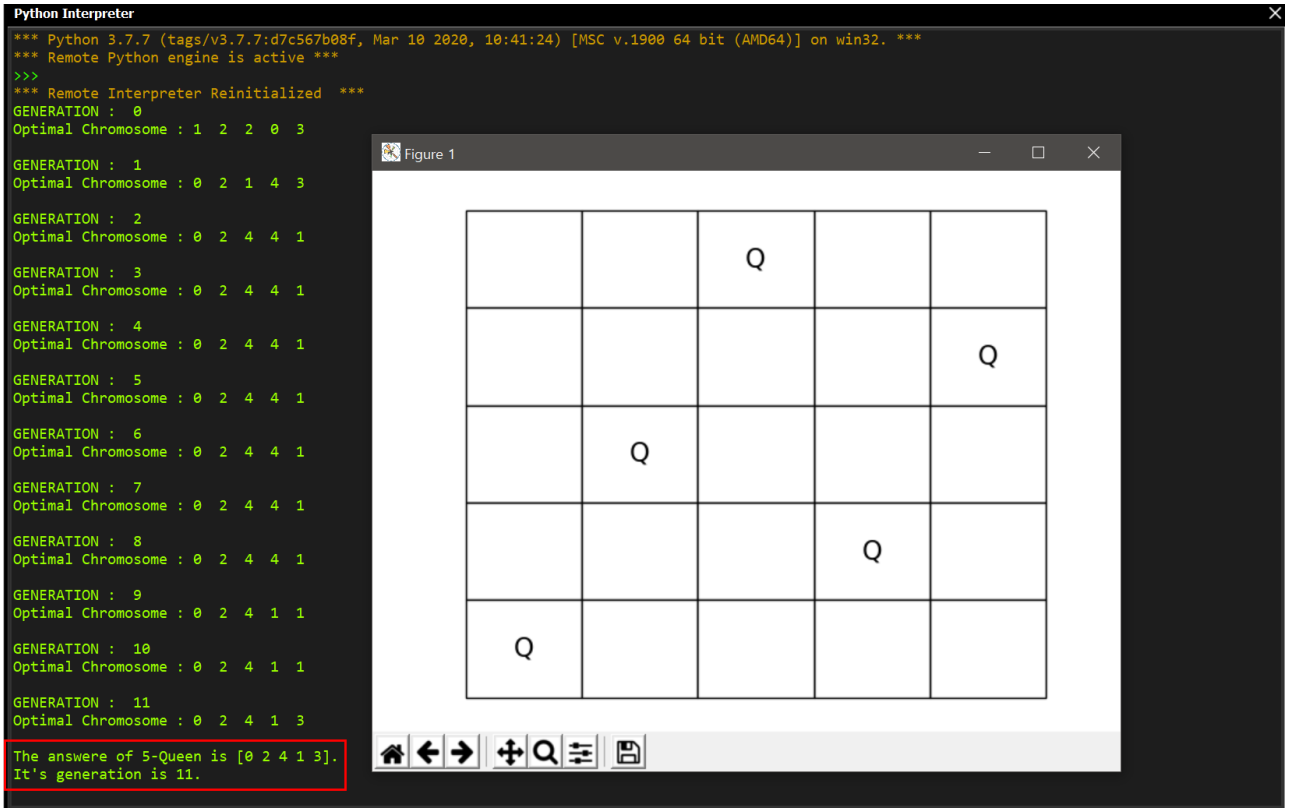


그림 a. 전체 개체 수 : 16, 선별 개체 수 : 2 로 설정 후 실행한 결과.

그림 a 는 전체 개체 수를 16 으로, 선별 개체 수를 2 로 설정했을 때의 프로그램 수행 결과이다. 11 세대 만에 해를 찾았는데, 4.1.2 의 경우보다 비교적 느리게 답을 찾아내는 것을 확인할 수 있었다.

4.1.2 전체 개체 수 : 100, 선별 개체 수 : 10

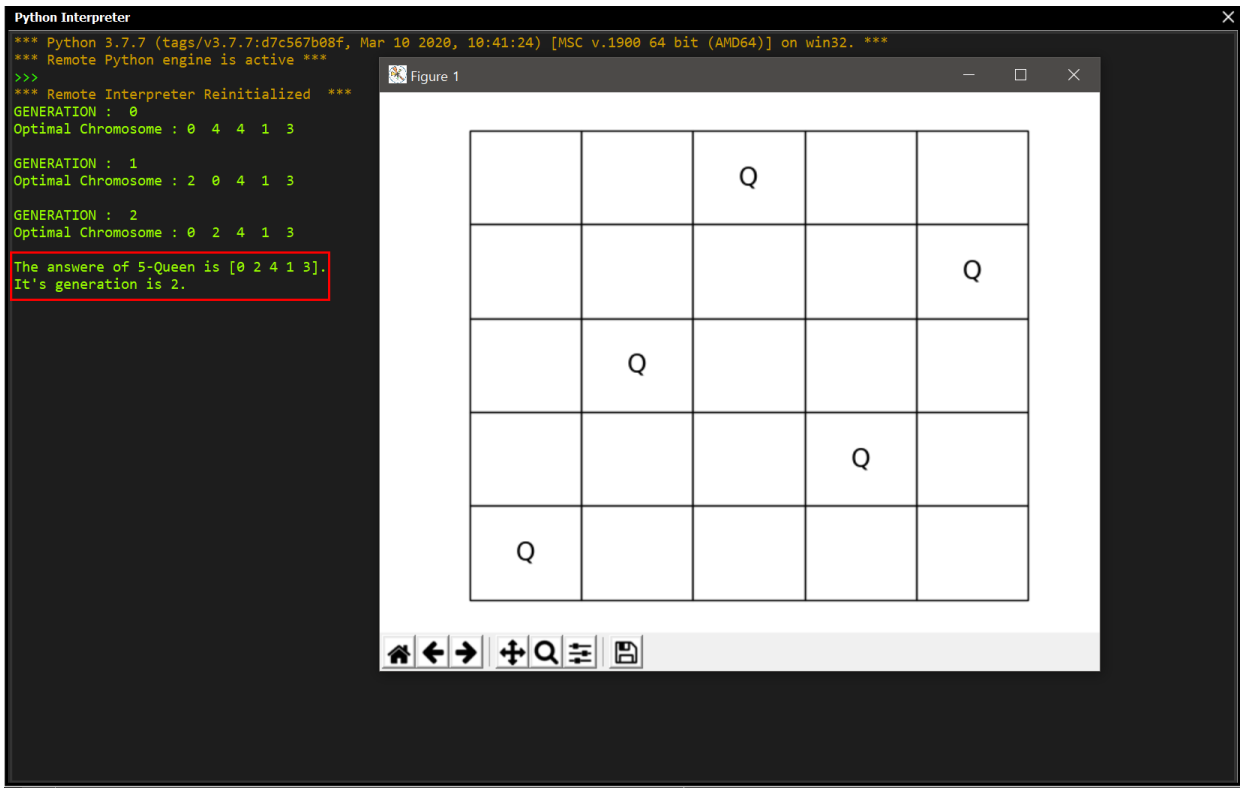


그림 b. 전체 개체 수 : 100, 선별 개체 수 : 10 로 설정 후 실행한 결과.

그림 b 는 전체 개체 수를 100 으로, 선별 개체 수를 10 으로 설정했을 때의 프로그램 수행 결과이다. 2 세대 만에 해를 찾았는데, 4.1.1 의 경우보다 비교적 빠르게 답을 찾아내는 것을 확인할 수 있었다.

4.1.3 N > 5

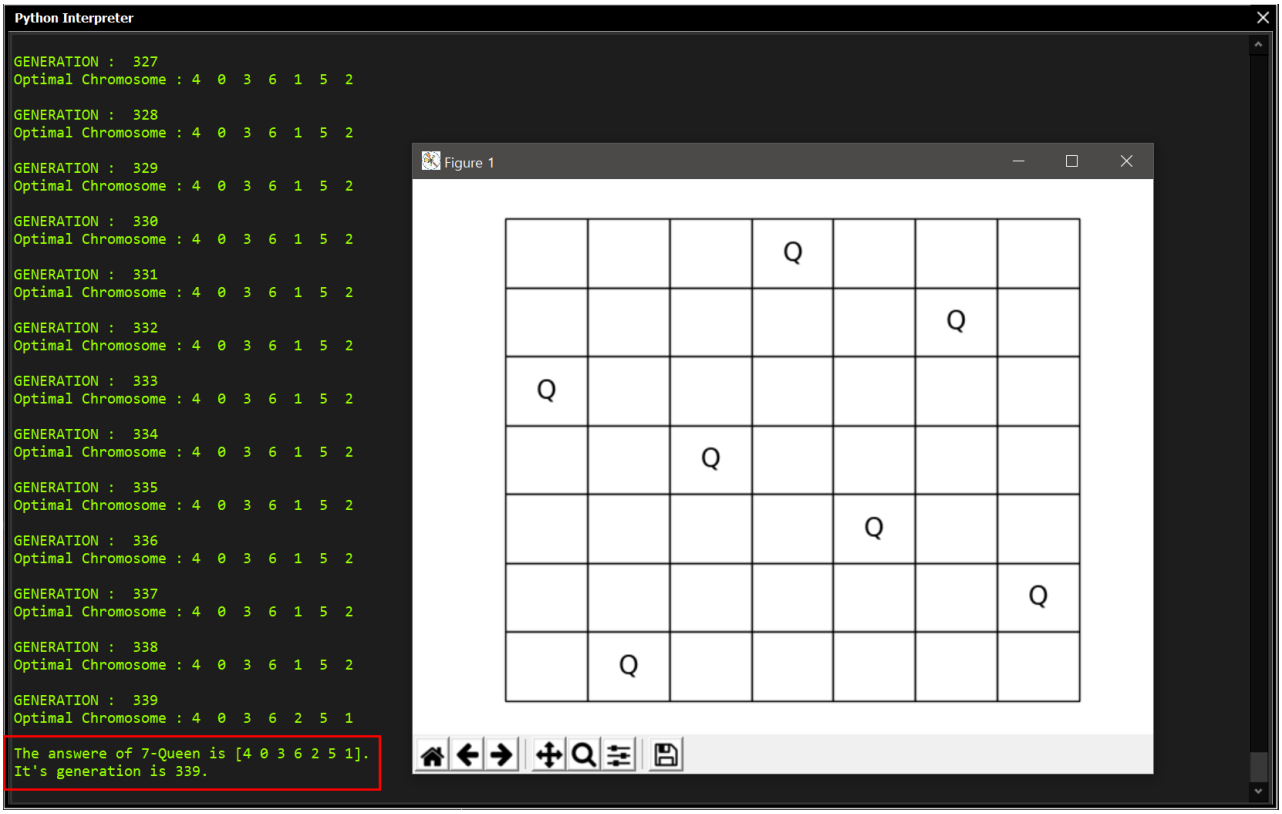


그림 c. N 이 7 일 때 프로그램 실행 결과.

그림 c 는 7-Queen 의 결과이다. 339 세대만에 답을 찾아 5-Queen 와 비교하면 탐색 시간이 훨씬 길다.

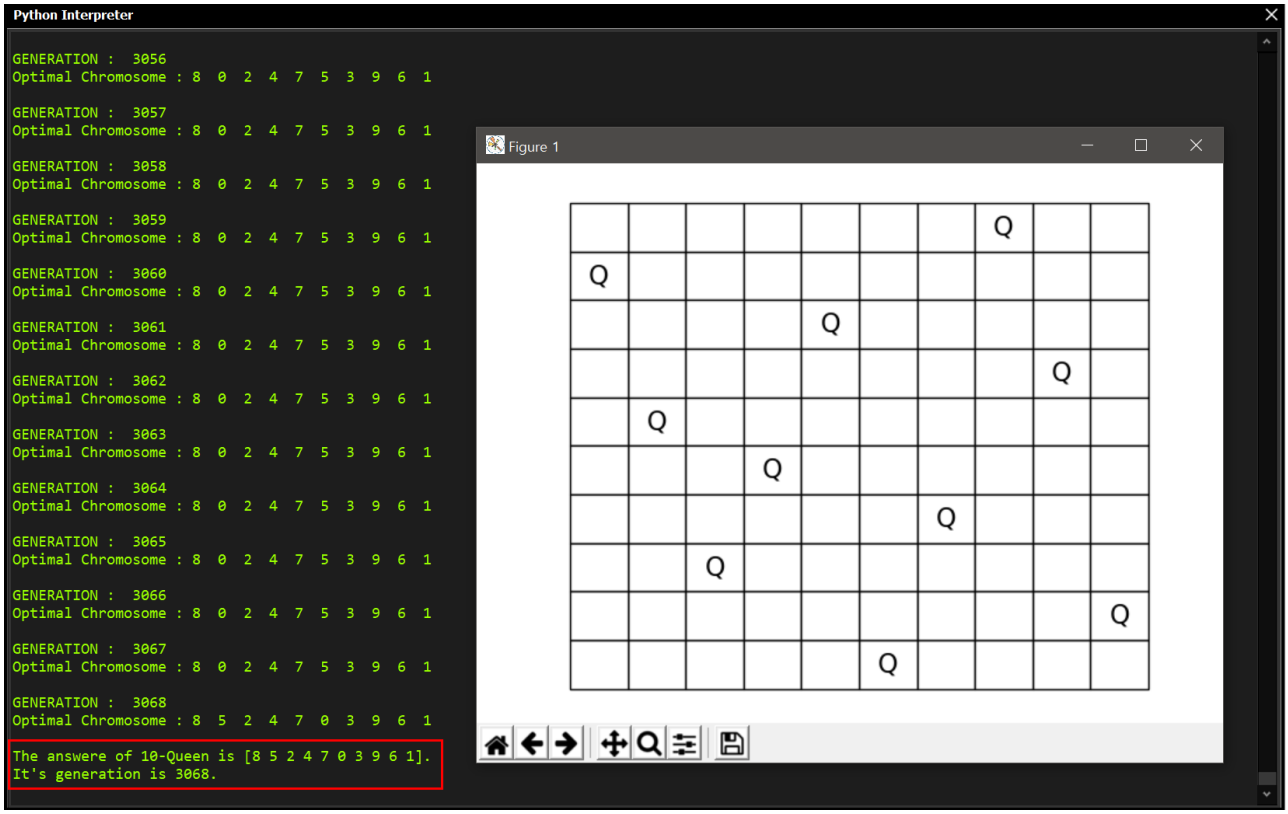


그림 d. N 이 10 일 때 프로그램 실행 결과.

그림 d 는 10-Queen 의 결과이다. 3068 세대만에 답을 찾아 5-Queen 은 물론 7-Queen 과 비교해도 탐색 시간이 훨씬 길다.

4.1.4 N < 4

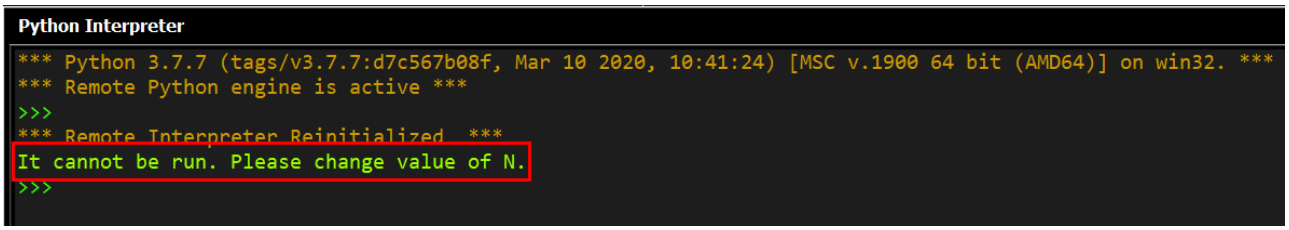
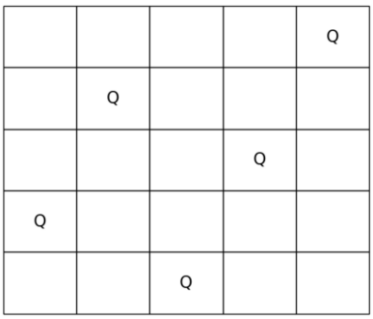


그림 e. update_generation 함수 실행 결과.

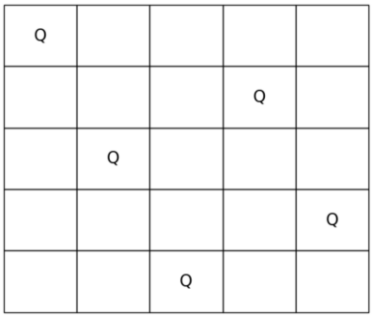
그림 e 는 N 이 4 미만일 경우에 대한 출력 결과이다. N-Queen 문제에서 N 이 3 이하일 경우에 대해서는 해가 없다. 그래서 예외처리를 해주지 않으면 답을 찾을 수 없어서 무한 루프를 돌게 된다.

4.2 5-Queen 의 모든 해 출력

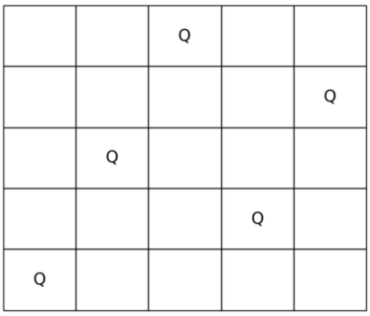
과제의 요구 조건대로 random 하게 시작하여 1 개의 해를 찾고, 프로그램을 실행할 때마다 다른 해를 찾아낸다. Genetic Algorithm 으로 찾은 5-Queen 의 모든 해는 다음과 같다.



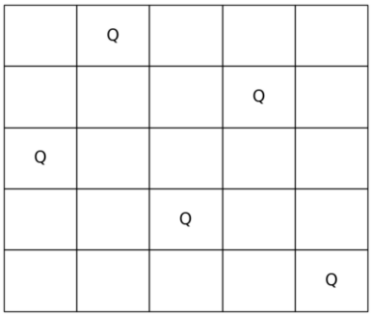
[1 3 0 2 4]



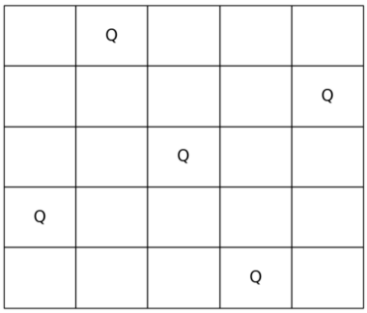
[4 2 0 3 1]



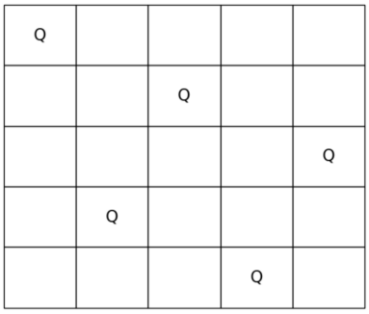
[0 2 4 1 3]



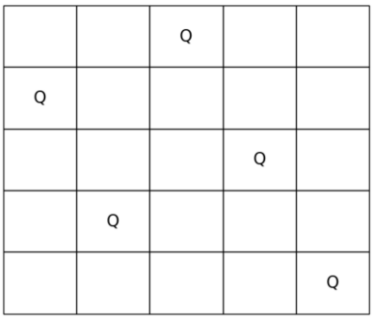
[2 4 1 3 0]



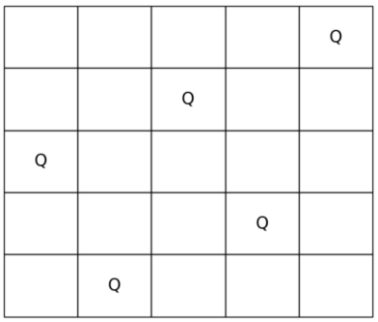
[1 4 2 0 3]



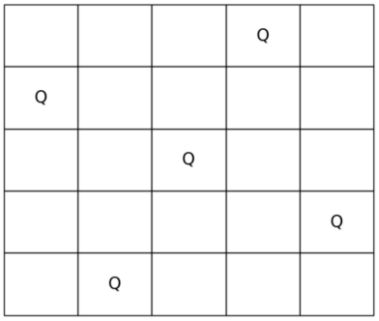
[4 1 3 0 2]



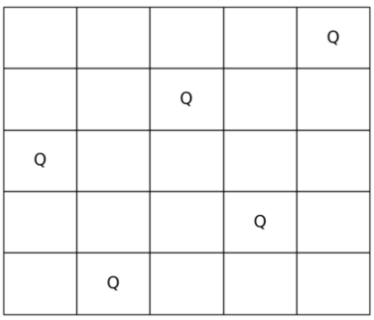
[3 1 4 2 0]



[2 0 3 1 4]



[3 0 2 4 1]



[2 0 3 1 4]