NQueens Problem and Local Search Analysis Report

2017320114_컴퓨터학과_최재원

Local Search Algorithm

Hill Climbing과 Stochastic Hill Climbing에 대해 알고리즘 비교 및 원리를 설명하기 이전에 이 두 알고리즘을 Local Search Algorithm에 대해서 설명을 하겠다. Local Search Algorithm이란 정확한 조건보다는 주변에 상황과 변수들을 예상하여 만들 수 있는 Optimal Solution을 제시한다는 것이다. 이전의 배웠던 알고리즘은 Objective Function, 즉 최단 경로나 Optimal한 Case가 주어져 있고 이를 바탕으로 Optimal Solution을 찾아가는 방식이었다. 하지만 Local Search는 말 그대로 지역 즉, 주변 상황을 판단해서 그것에 의한 좋은 해답을 찾아 나가는 것이다. 실제로 이 함수에서는 지나왔던 Path, Cost 등은 필요하지 않다. 앞으로 나오는 알고리즘 비교 및 분석에서는 이 그래프에 비유를 해서 사용을 할것이다. 우선적으로 목표로 두는 것이 바로 Global Maximum에 도달하는 것이고, 알고리즘을 구현 함에 따라 Local Maximum에도 빠질 수가 있다.

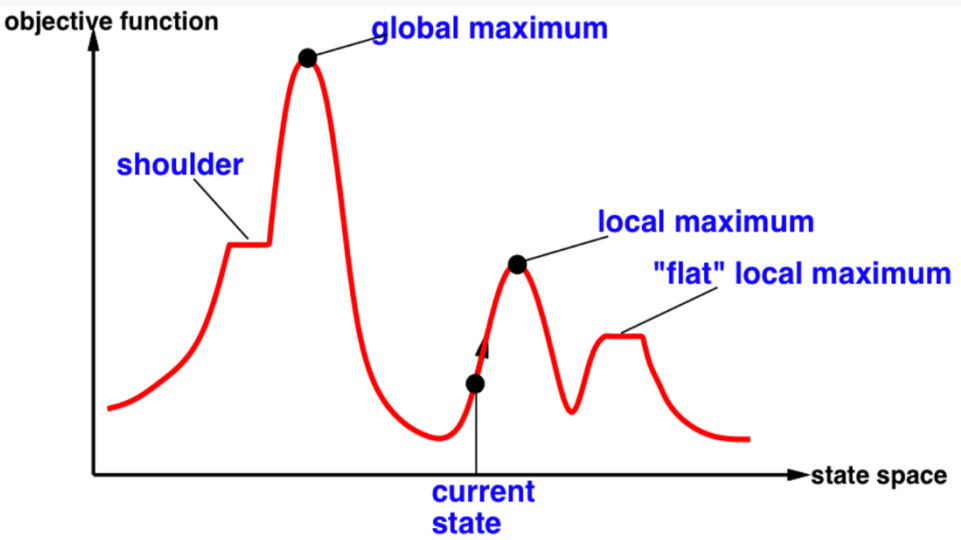


Table of Contents

Hill Climbing vs. Stochastic Hill-Climbing

Hill Climbing 알고리즘은 현재 상황에서 주변 노드 중에 올라갈 곳이 있는지 확인 후 더 올라갈 수 있는 곳이 있으면 이동 하는 방식이다. 더 올라갈 곳이 없으면 그것이 Optimal State라고 생각하고 알고리즘은 멈춘다. 하지만 이 알고리즘 같은 경우에는 위의 그래프에서도 보이다시피 더 올라가다가 Local Maximum에 도달하면 Global Maximum이랑 구분이 안되기 때문에 그대로 멈춘다. 또한 위 그래프에서 shoulder라고 할 수 있는 평평한 부분에 state이 오게 되면 어디로 갈지 판단을 못하는 상태가 된다. 따라서 Local Maximum에 빠질 확률이 있다. 실습에서 구현된 코드에서는 findBestNeighbor의 코드를 보면 이런 식으로 bestNeighbor를 찾으려는 것이 보인다.

```
for neigh in neighbors:
    value = neigh.getValue()
    if value > bestValue:
        bestNeighs = [neigh]
        bestValue = value
```

Local Maximum에 대한 하나의 해결 방안으로 제시된 것이 바로 Stochastic hill climbing이다. 이 방법은 Hill Climbing와는 다르게 매번 움직일 neighbor를 best로 찾는 것이 아니라 현재보다는 나은 neighbor중에서 임의로 하나를 선택하는 것이다. 즉, 현재보다는 괜찮지만 최고로 좋은 neighbor은 아닌 것이다. 하지만 better neighbor 중에서 랜덤으로 고르는 것이 아니라 괜찮은 정도에 따라서 확률적인 값을 부여 해준다. 따라서 더 좋은 neighbor가 선택될 확률을 높이는 것이다.

```
for neigh in neighbors:
    value = neigh.getValue()
    if currValue <= value:
        bestNeighs.append(neigh)
```

<finding better neighbor>

```
deltaValues = [neigh.getValue() - currValue for neigh in bestNeighs]
bestPos = rouletteSelect(deltaValues)
return bestNeighs[bestPos]
```

<giving neighbors different chance to be picked stochastically>

실습 실행 결과

아래 결과에서는 여러가지 변수들의 변화를 줘서 실험을 진행해 보았다.

Practice 1)

```
====================================
Running Tests on <function hillClimb at 0x7f945a5e0dc0>
------------------
Size= 4
Run 1 : quality =  5.0 out of 6.0 count = 1 time =  0.00028
Run 2 : quality =  5.0 out of 6.0 count = 1 time =  0.00035
Run 3 : quality =  5.0 out of 6.0 count = 2 time =  0.00057
Run 4 : quality =  6.0 out of 6.0 count = 4 time =  0.00068
Run 5 : quality =  5.0 out of 6.0 count = 1 time =  0.00050
------------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 7 time =  0.00400
Run 2 : quality =  14.0 out of 15.0 count = 8 time =  0.00377
Run 3 : quality =  14.0 out of 15.0 count = 1000 time =  0.39253
Run 4 : quality =  14.0 out of 15.0 count = 3 time =  0.00124
Run 5 : quality =  14.0 out of 15.0 count = 6 time =  0.00261
========================================
```

<5 repetition on Hill Climb Alg.>

```
====================================
Running Tests on <function stochHillClimb at 0x7fb5e8c20670>
------------------
Size= 4
Run 1 : quality =  6.0 out of 6.0 count = 32 time =  0.00434
Run 2 : quality =  6.0 out of 6.0 count = 11 time =  0.00239
Run 3 : quality =  6.0 out of 6.0 count = 7 time =  0.00094
Run 4 : quality =  6.0 out of 6.0 count = 13 time =  0.00225
Run 5 : quality =  5.0 out of 6.0 count = 1000 time =  0.12342
------------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 1000 time =  0.19103
Run 2 : quality =  14.0 out of 15.0 count = 1000 time =  0.17679
Run 3 : quality =  13.0 out of 15.0 count = 1000 time =  0.17038
Run 4 : quality =  14.0 out of 15.0 count = 1000 time =  0.17149
Run 5 : quality =  14.0 out of 15.0 count = 1000 time =  0.17329
========================================
```

<5 repetition, 5 numNeighs on Stochastic Hill Climb>

실행 환경을 설명하자면, 처음에는 5번의 repetition으로 결과를 보았다. Stochastic 함수 같은 경우에는 numNeigh로 설정된 neighbor를 계속 찾으면서 Local Maxima에 대한 확률을 줄일려고 하기 때문에 계산의 양이 늘게 되고 이로 인해서 시간의 차이가 확연히 들어난다는 것을 볼 수 있다. 또한 Average Quality에서는 Stochastic Hill Climb이 4-Queens에서 Global Maximum(6 out of 6)를 Hill Climb보

다는 비교적 많이 달성한 것을 볼 수 있다.

5 repetition 5 numNeighs	Hill Climbing	Stochastic Hill Climb
Average Time	4-Queen: 0.000476 6-Queen: 0.08083	4-Queen: 0.026668 6-Queen: 0.176596
Average Quality	4-Queen: 5.2 6-Queen: 14	4-Queen: 5.8 6-Queen: 13.8

<Time Complexity and Quality Compare>

그러면, "Quality는 비슷하지만 Time에서 압도적으로 차이가 많이 난다면 과연 Hill Climb이 Stochastic보다 효율성이 좋다고 할 수 있는가?" 라는 것에 대한 확실한 답을 얻기 위해 quality를 더 높일 수 있는 Stochastic의 numNeigh를 10으로 올렸다. 또한 실험에서의 보다 정확한 결론을 내리기 위해 반복횟수 또한 5회에서 10회로 올렸다. 아래에 실행 결과 및 분석 테이블을 보자.

Practice 2) [*5 repetition*] ➔ *[10 repetition], [5 numNeighs]* ➔ *[10 numNeighs]*

```
Running Tests on <function hillClimb at 0x7ffad4333040>
------------------
Size= 4
Run 1 : quality =  6.0 out of 6.0 count = 2 time =  0.00030
Run 2 : quality =  6.0 out of 6.0 count = 2 time =  0.00030
Run 3 : quality =  5.0 out of 6.0 count = 1 time =  0.00035
Run 4 : quality =  6.0 out of 6.0 count = 4 time =  0.00061
Run 5 : quality =  5.0 out of 6.0 count = 1 time =  0.00038
Run 6 : quality =  5.0 out of 6.0 count = 2 time =  0.00182
Run 7 : quality =  5.0 out of 6.0 count = 3 time =  0.00092
Run 8 : quality =  6.0 out of 6.0 count = 4 time =  0.00908
Run 9 : quality =  5.0 out of 6.0 count = 1 time =  0.00081
Run 10 : quality =  6.0 out of 6.0 count = 4 time =  0.00124
------------------
Size= 6
Run 1 : quality =  15.0 out of 15.0 count = 74 time =  0.06126
Run 2 : quality =  14.0 out of 15.0 count = 4 time =  0.00409
Run 3 : quality =  14.0 out of 15.0 count = 4 time =  0.00422
Run 4 : quality =  12.0 out of 15.0 count = 1000 time =  0.36148
Run 5 : quality =  14.0 out of 15.0 count = 3 time =  0.00124
Run 6 : quality =  14.0 out of 15.0 count = 14 time =  0.00485
Run 7 : quality =  14.0 out of 15.0 count = 4 time =  0.00183
Run 8 : quality =  13.0 out of 15.0 count = 2 time =  0.00147
Run 9 : quality =  14.0 out of 15.0 count = 2 time =  0.00131
Run 10 : quality =  12.0 out of 15.0 count = 1000 time =  0.28763
```

<10 repetition on Hill Climb Alg.>

```
Running Tests on <function stochHillClimb at 0x7fa9a3b45670>
------------------
Size= 4
Run 1 : quality =  6.0 out of 6.0 count = 4 time =  0.00117
Run 2 : quality =  5.0 out of 6.0 count = 1000 time =  0.24701
Run 3 : quality =  5.0 out of 6.0 count = 1000 time =  0.22462
Run 4 : quality =  6.0 out of 6.0 count = 2 time =  0.00058
Run 5 : quality =  6.0 out of 6.0 count = 2 time =  0.00061
Run 6 : quality =  6.0 out of 6.0 count = 3 time =  0.00107
Run 7 : quality =  5.0 out of 6.0 count = 1000 time =  0.22364
Run 8 : quality =  5.0 out of 6.0 count = 1000 time =  0.22988
Run 9 : quality =  6.0 out of 6.0 count = 18 time =  0.00454
Run 10 : quality =  5.0 out of 6.0 count = 1000 time =  0.24546
------------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 1000 time =  0.36470
Run 2 : quality =  14.0 out of 15.0 count = 1000 time =  0.35394
Run 3 : quality =  14.0 out of 15.0 count = 1000 time =  0.34043
Run 4 : quality =  14.0 out of 15.0 count = 1000 time =  0.34878
Run 5 : quality =  14.0 out of 15.0 count = 1000 time =  0.35151
Run 6 : quality =  12.0 out of 15.0 count = 1000 time =  0.33456
Run 7 : quality =  14.0 out of 15.0 count = 1000 time =  0.33606
Run 8 : quality =  14.0 out of 15.0 count = 1000 time =  0.33611
Run 9 : quality =  14.0 out of 15.0 count = 1000 time =  0.35032
Run 10 : quality =  14.0 out of 15.0 count = 1000 time =  0.33539
==========================================
```

<10 repetition, 10 numNeighs on Stochastic Hill Climb>

10 repetition 10 numNeighs	Hill Climbing	Stochastic Hill Climb
Average Time	4-Queen: 0.001581 6-Queen: 0.044175	4-Queen: 0.117858 6-Queen: 0.34518
Average Quality	4-Queen: 4.9 6-Queen: 13.6	4-Queen: 5.5 6-Queen: 13.8

<Time Complexity and Quality Compare>

따라서 이러한 변수의 변화 역시 Stochastic Hill Climb이 낫다는 결과를 도출해내지 못했다. 결론적

으로 Hill Climb와 Stochastic Hill Climb을 비교했을 때 비록 Local Maxima에 빠질 확률이 더 높더라도 계산의 부하가 적은 *"Hill Climb이 비교적 더 좋은 알고리즘이다"* 라고 결론을 내릴 수 있습니다.

Hill Climbing vs. Simulated Annealing

앞서 Hill Climbing 알고리즘에 대해서는 충분한 설명을 했으니 여기서는 Simulated Annealing 알고리즘에 대해서 설명하겠다. 알고리즘의 이름 자체는 금속 공학에서 쓰이는 방법인 안정적인 금속 하나를 가열한 뒤에 천천히 온도를 낮춰서 초기보다 낮은 에너지를 가지는 결정의 형태로 금속 원자들을 결합하는 Annealing 기법에서 따온 것이고, 이와 비슷한 원리로 알고리즘에도 적용을 한다. 이 알고리즘 또한 Local Maximum에 빠져도 다시 내려가 다른 봉우리로 이동해서 Global Maximum에 도달할 가능성을 높이는 것을 중점으로 했다.

실습에서 알고리즘으로 고안한 것은 initTemp가 0보다 크고 한번 씩 돌아갈 때마다 온도를 0.1을 낮췄다. 즉 종료될 조건은 0이 되거나 최댓값을 찾는 것이다. 만약 이웃한 state가 현재 state보다 좋으면 바로 이동을 한다. 그렇지 않을 때는 안좋은 노드라고 해서 이동하지 않는 것이 아니다. 온도가 더 안좋아지는 정도를 판단하고 이동을 할지 말지를 정한다. 그 식은 실습환경에서는 e^(difference between next and current value / current Temperature)로 정했다.

```
threshold = math.exp(difference/currTemp)
random_value = random.random()
if random_value <= threshold:
    currState = nextState
    currValue = nextValue
```

<set threshold>

실습 실행 결과

아래 결과에서 변수들의 변화를 줘서 실험을 진행해 보았다. (Hill Climb의 경우 같은 결과를 사용)
Practice 1)

```
==================================
Running Tests on <function hillClimb at 0x7f945a5e0dc0>
----------------
Size= 4
Run 1 : quality =  5.0 out of 6.0 count = 1 time =  0.00028
Run 2 : quality =  5.0 out of 6.0 count = 1 time =  0.00035
Run 3 : quality =  5.0 out of 6.0 count = 2 time =  0.00057
Run 4 : quality =  6.0 out of 6.0 count = 4 time =  0.00068
Run 5 : quality =  5.0 out of 6.0 count = 1 time =  0.00050
----------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 7 time =  0.00400
Run 2 : quality =  14.0 out of 15.0 count = 8 time =  0.00377
Run 3 : quality =  14.0 out of 15.0 count = 1000 time =  0.39253
Run 4 : quality =  14.0 out of 15.0 count = 3 time =  0.00124
Run 5 : quality =  14.0 out of 15.0 count = 6 time =  0.00261
==================================
```

<5 repetition on Hill Climb Alg.>

```
==========================================
Running Tests on <function simAnnealing at 0x7fca95a45790>
--------------------
Size= 4
Run 1 : quality =  5.0 out of 6.0 count = 51 time =  0.00141
Run 2 : quality =  3.0 out of 6.0 count = 51 time =  0.00148
Run 3 : quality =  4.0 out of 6.0 count = 51 time =  0.00155
Run 4 : quality =  5.0 out of 6.0 count = 51 time =  0.00175
Run 5 : quality =  4.0 out of 6.0 count = 51 time =  0.00133
--------------------
Size= 6
Run 1 : quality =  9.0 out of 15.0 count = 51 time =  0.00214
Run 2 : quality =  11.0 out of 15.0 count = 51 time =  0.00268
Run 3 : quality =  13.0 out of 15.0 count = 51 time =  0.00291
Run 4 : quality =  12.0 out of 15.0 count = 51 time =  0.00330
Run 5 : quality =  10.0 out of 15.0 count = 51 time =  0.00426
==========================================
```

<5 repetition, initTemp = 5 on Simulated Annealing >

Simulated Annealing 같은 경우에는 주변 노드가 현재 노드의 상태보다 안좋더라도 확인을 해보고 State을 옮기기 때문에 Average Time 즉 Time Complexity 측면에서 Hill Climb보다 훨씬 열세일 것으로 예측을 했다. 결과는 아래에 테이블에서도 보이듯이 4-Queens에서는 엄청난 차이를 보였다. 하지만 6-queens에서는 Hill Climb은 계속해서 child node를 탐색하기 때문에 주변에 노드 하나를 고르는 simulated annealing보다는 average time이 높게 나왔다.
Average Quality도 현재 Nqueen 보드에서는 Hill Climb 알고리즘이 더 좋은 quality 즉 non-attacking

pair를 가지고 있다.

5 repetition InitTemp = 5 Decrease by 0.1	Hill Climbing	Simulated Annealing
Average Time	4-Queen: 0.000476 6-Queen: 0.08083	4-Queen: 0.001504 6-Queen: 0.003058
Average Quality	4-Queen: 5.2 6-Queen: 14	4-Queen: 4.2 6-Queen: 11

<Time Complexity and Quality Compare>

Practice 1에 대한 결과로 인해 생기는 의문증, "과연 initTemp를 높이고 Temperature를 더욱 느리게 낮아지게 한다면 Average Quality의 값은 Hill Climbing 보다 좋아질까?" 라는 의문을 가지고 initTemp를 50으로 높이고 아래에 세팅되어 있는 currTemp의 값을 0.05로 바꾸어 보았다.

```
currTemp -= 0.1
count += 1
```

<change to 0.05>

Practice 2) [*initTemp = 5*] ➔ *[initTemp = 50], [decrease by 0.1]* ➔ *[decrease by 0.05]*

```
====================================
Running Tests on <function hillClimb at 0x7f945a5e0dc0>
----------------
Size= 4
Run 1 : quality =  5.0 out of 6.0 count = 1 time =  0.00028
Run 2 : quality =  5.0 out of 6.0 count = 1 time =  0.00035
Run 3 : quality =  5.0 out of 6.0 count = 2 time =  0.00057
Run 4 : quality =  6.0 out of 6.0 count = 4 time =  0.00068
Run 5 : quality =  5.0 out of 6.0 count = 1 time =  0.00050
----------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 7 time =  0.00400
Run 2 : quality =  14.0 out of 15.0 count = 8 time =  0.00377
Run 3 : quality =  14.0 out of 15.0 count = 1000 time =  0.39253
Run 4 : quality =  14.0 out of 15.0 count = 3 time =  0.00124
Run 5 : quality =  14.0 out of 15.0 count = 6 time =  0.00261
====================================
```

<5 repetition on Hill Climb Alg.>

```
====================================
Running Tests on <function simAnnealing at 0x7ff5674468b0>
----------------
Size= 4
Run 1 : quality =  6.0 out of 6.0 count = 74 time =  0.00380
Run 2 : quality =  6.0 out of 6.0 count = 153 time =  0.00823
Run 3 : quality =  6.0 out of 6.0 count = 466 time =  0.02040
Run 4 : quality =  6.0 out of 6.0 count = 128 time =  0.00454
Run 5 : quality =  6.0 out of 6.0 count = 242 time =  0.00789
----------------
Size= 6
Run 1 : quality =  13.0 out of 15.0 count = 1001 time =  0.05130
Run 2 : quality =  13.0 out of 15.0 count = 1001 time =  0.05137
Run 3 : quality =  15.0 out of 15.0 count = 838 time =  0.03827
Run 4 : quality =  12.0 out of 15.0 count = 1001 time =  0.04155
Run 5 : quality =  15.0 out of 15.0 count = 926 time =  0.03799
====================================
```

<5 repetition, initTemp = 50 decrease by 0.05 on Simulated Annealing>

5 repetition InitTemp = 50 Decrease by 0.05	Hill Climbing	Simulated Annealing
Average Time	4-Queen: 0.000476 6-Queen: 0.08083	4-Queen: 0.008972 6-Queen: 0.044096
Average Quality	4-Queen: 5.2 6-Queen: 14	4-Queen: 6 6-Queen: 13.6

<Time Complexity and Quality Compare>

4-Queens에서는 Quality 즉 Optimal Value 값을 찾아내는 데 있어서는 충분히 높은 온도와, 천천히 줄어드는 온도를 설정해주면 결과에서 보이는 바와 같이 Global Maximum을 찾아준다. 하지만 이 경우에 time을 보게 되면 너무 느리다. 보통은 100대의 횟수를 기록했으며 걸린 시간을 보면 확연하게 차이가 난다. 따라서 최적해를 반드시 구할 때 사용을 하면 된다. 하지만 6-Queens

에서는 Quality의 우세함을 가져가지 못했다. 오히려 quality가 더 낮게 나올 뿐만 아니라, 걸린 시간에서도 확연히 차이가 났다. 이에 대해서는 아마도 훨씬 경우의 수가 많아진 6-Queen에서는 좀 더 충분한 온도와 온도 감소율을 제시해야 할 것 같다. 하지만 6-queens에서도 고무적인 부분은 simulated Annealing의 목적인 local maxima에 빠지지 않고 Global maximum을 찾으려고 한다라는 것이 보여진 것은 중간 중간 Global Maximum(15 out of 15)에서 보여주듯이 달성했다고 볼 수 있다.

결론적으로는 효율성 측면에서 즉 "주어진 시간 안에 어느 정도의 quality를 보장하느냐" 라고 했을 때, Hill Climb이 Simulated Annealing보다 좋은 효율을 보여주고 있다. 하지만 Simulated Annealing도 적절히 온도와 감소율을 잘 조절해준다면, 최적해를 반드시 찾는다는 것은 보장을 못하겠지만, "어느 정도 수준이 보장되는 해답은 줄 수 있다"라고 할 수 있다.

Hill Climbing vs. Beam Search

Beam Search에서는 BFS Search로 다음 상태의 해 집합을 구한 뒤에, 해 집합을 Goal에 가까운 순서대로 정렬한다. 그리고 미리 설정된 수 만큼의 해 집합을 유지하고 나머지는 잘라내는 방법이다. 이 알고리즘 또한 Local Search이기 때문에 어느 정도의 해를 구하기만 하면 되기 때문에 이러한 방법이 나왔다. 즉 무작위로 생성된 State들인 K에서 시작을 하고, 각 단계에서 모든 K의 자손 노드를 생성한다. 자손 노드 중에서 어느 하나가 목표인 경우 알고리즘은 해를 찾은 것이다. 만약 이게 안된다면 목록에서 K의 최선의 Child 노드를 선택하고 반복한다. Beam 탐색은 가치가 없는 자손 노드를 빠르게 제거하고 가장 최선의 노드로 이동할 수 있게 해준다. 확률적인 Beam Search에서는 유지된 자손 노드들이 그들의 효율성을 기반으로 선택된다. 실습에서는 아래의 코드(KeepNNeighbor)가 계속해서 최선의 리스트를 update 하도록 해준다.

```
while i < len(neighs):
    nextNeigh = neighs[i]
    if len(bestSoFar) == n:
        worstOfBest = bestSoFar[-1]
        if nextNeigh.getValue() < worstOfBest.getValue():
            break
    insertState(bestSoFar, nextNeigh, n)
    i = i + 1
```

실습 실행 결과

아래에 실습을 진행해 보았다.

Practice 1)

```
==========================================
Running Tests on <function hillClimb at 0x7f945a5e0dc0>
------------------
Size= 4
Run 1 : quality =  5.0 out of 6.0 count = 1 time =  0.00028
Run 2 : quality =  5.0 out of 6.0 count = 1 time =  0.00035
Run 3 : quality =  5.0 out of 6.0 count = 2 time =  0.00057
Run 4 : quality =  6.0 out of 6.0 count = 4 time =  0.00068
Run 5 : quality =  5.0 out of 6.0 count = 1 time =  0.00050
------------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 7 time =  0.00400
Run 2 : quality =  14.0 out of 15.0 count = 8 time =  0.00377
Run 3 : quality =  14.0 out of 15.0 count = 1000 time =  0.39253
Run 4 : quality =  14.0 out of 15.0 count = 3 time =  0.00124
Run 5 : quality =  14.0 out of 15.0 count = 6 time =  0.00261
==========================================
```

<5 repetition on Hill Climb Alg.>

```
Running Tests on <function beamSearch at 0x7f8d4de21940>
------------------
Size= 4
Run 1 : quality =  6.0 out of 6.0 count = 1 time =  0.00147
Run 2 : quality =  6.0 out of 6.0 count = 3 time =  0.00537
Run 3 : quality =  6.0 out of 6.0 count = 2 time =  0.00426
Run 4 : quality =  6.0 out of 6.0 count = 2 time =  0.00402
Run 5 : quality =  6.0 out of 6.0 count = 2 time =  0.00400
------------------
Size= 6
Run 1 : quality =  14.0 out of 15.0 count = 500 time =  2.12442
Run 2 : quality =  15.0 out of 15.0 count = 3 time =  0.01989
Run 3 : quality =  15.0 out of 15.0 count = 5 time =  0.02682
Run 4 : quality =  15.0 out of 15.0 count = 4 time =  0.01978
Run 5 : quality =  14.0 out of 15.0 count = 500 time =  2.24825
```

<5 repetition on Beam Search>

5 repetition popSize 10	Hill Climbing	Beam Search
Average Time	4-Queen: 0.000476 6-Queen: 0.08083	4-Queen: 0.003824 6-Queen: 0.887832
Average Quality	4-Queen: 5.2 6-Queen: 14	4-Queen: 6 6-Queen: 14.6

<Time Complexity and Quality Compare>

실습을 진행한 결과 Time Complexity 측면에서는 Hill Climb이 Beam Search보다 월등하게 우월했다. 이는 앞서 서술한 Beam Search의 알고리즘 특성 때문에 나온 결과이며, 다른 알고리즘들과는 다르게 Beam Search의 quality면에서는 거의 최고의 Optimality를 보여줬다. 또한 4-queens에서는 모든 시도마다 Global maximum을 찾았으며, 6-queens에서도 거의 대부분 Global Maximum을 구해냈다. 따라서 다소 시간이 오래 걸리더라도 높은 수준의 최적의 해를 찾고 싶다면 Beam Search를 이용하는 것이 좋다.

Genetic Algorithm vs. Beam Search

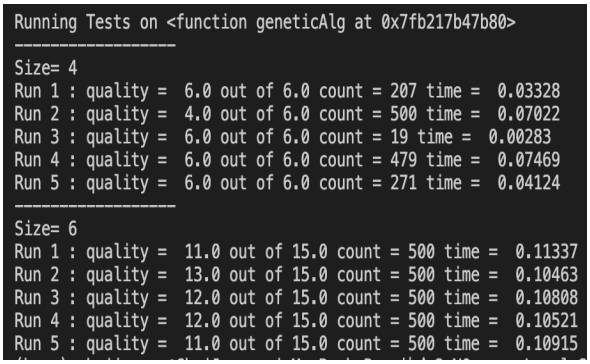
Genetic Algorithm은 이름 그대로 자연계의 유전적인 현상을 그대로 계산한 것이다. 가장 좋은 유전자만 남기는 습성을 가지고 또 돌연변이 유전자로 통해서 유전자 변형을 일으킬 수도 있다. 각 state는 유전자로 비유가 되고 다음 자손으로 세대가 교체될 때 모든 유전자들은 Fitness Function에 의해 평가가 된다. Fitness가 높게 측정되는 유전자에 한정되어 번식이 가능하게 되는 것이다. 또한 Crossover Point를 지정을 해서 그 기준으로 Crossover를 한다. 마지막으로 mutation 과정을 통해 임의의 유전자를 swap을 하고 돌연변이를 만들어 주면 된다.

```
parentPool = selectParents(currStates, fits)
currStates = mateParents(parentPool, crossPerc, mutePerc)
```

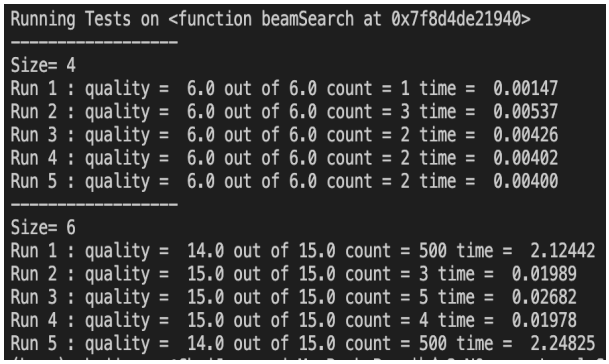
<Crossover & Mutation Process>

실습 실행 결과

아래에 실습을 진행해 보았다. 앞서 진행하였던 Beam Search와 비슷한 조건을 형성 시키기 위해서 기존의 *PopSize를 10으로 줄였고, Max Generation 또한 500으로 줄였다. Crossover 확률은 0.5, Mutation 확률은 0.01로 설정해서 실습을 진행해 보았다. Repetition도 마찬가지로 5번으로 유지를 했다.*



<5 repetition on Genetic Algorithm>



< 5 repetition on Beam Search>

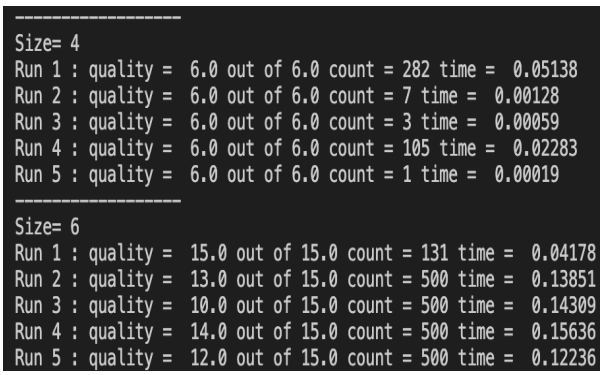
5 repetition popSize 10	Genetic Algorithm	Beam Search
Average Time	4-Queen: 0.044452 6-Queen: 0.108088	4-Queen: 0.003824 6-Queen: 0.887832
Average Quality	4-Queen: 5.6 6-Queen: 11.8	4-Queen: 6 6-Queen: 14.6

<Time Complexity and Quality Compare>

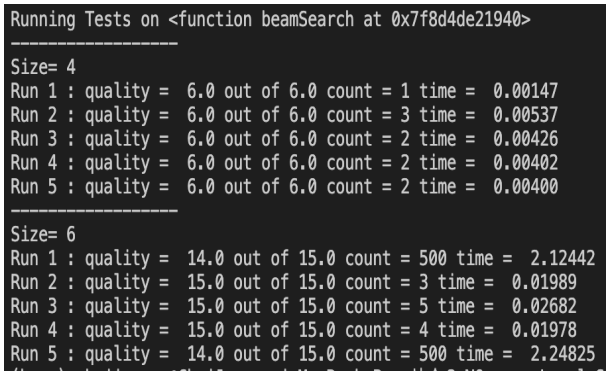
이론적으로만 생각을 해보면 Beam Search는 주어진 주변 노드들을 모두 확인을 해서 좋은 노드로 옮기고 또 다른 노드가 생기면 업데이트를 하면서 계속적으로 설정된 주변 노드들을 모두 확인해서 진행되는 반면에, Genetic Algorithm은 Parent Generation의 Crossover를 해서 진행하기 때문에 Time Complexity 측면에서 Genetic Algorithm이 이점을 가질 것이라 생각을 했지만 실제 실습을 진행한 결과 4-Queens 에서는 Beam Search가 더 효율적으로 보였다. 이는 실행 결과에서도 볼 수 있듯이 4-Queens Beam Search에서 모든 repetition결과가 비교적 이른 시간에 maxValue를 찾았기 때문이다. 반면에 6-Queens 같은 경우에는 Genetic Algorithm이 예측한 대로 Time Complexity 측면에서 이점을 가졌다. 나아가, Quality 측면에서는 Beam Search가 우세를 보였다.

그렇다면 과연 Genetic Algorithm의 Crossover percentage와 Mutation Percentage를 조정을 해주면 조금 더 좋은 Quality를 보여줄 수 있는가?에 대한 의문으로 위의 두가지 조건에 변화를 줘서 다시 실습을 진행해 보았다. `crossPerc=0.6, mutePerc=0.1)`

Practice 2) [*crossPerc = 0.5] ➔ [crossPerc = 0.6], [mutePerc = 0.01] ➔ [mutePerc = 0.1]*



<5 repetition on Genetic Algorithm>



<5 repetition on Beam Search>

5 repetition popSize 10	Genetic Algorithm	Beam Search
Average Time	4-Queen: 0.015254 6-Queen: 0.12042	4-Queen: 0.003824 6-Queen: 0.887832
Average Quality	4-Queen: 6 6-Queen: 12.8	4-Queen: 6 6-Queen: 14.6

<Time Complexity and Quality Compare>

Crossover percentage, Mutation Percentage 모두 상향 조절을 해서 실습을 진행한 결과 의도한 바와 같이 average quality가 높아진 것을 확인할 수 있다. 하지만 역시 quality를 높이려고 진행한 두번째 실습이기 때문에 time complexity 측면에서는 여전히 열세를 보였다. 그래도 고무적인 부분은 4-queens에서 첫번째 실험과는 달리, 비교적 적은 횟수(times)안에 quality 6를 맞췄다는 것이다.