**Artificial Intelligence Assignment01 Report**

컴퓨터소프트웨어학부 박제우

1. 함수 설명

제출한 코드에서는 기본적인 bfs, hc, csp함수와 함께 이들 함수에서 사용할 3개의 함수를 추가하였다. 작성한 함수에 대한 설명은 다음과 같다.

**def** checkConflict(arr):

- 각 열에 있는 Queen들의 행 번호들의 배열 arr가 주어지면 모든 Queen이 조건에 맞게 위치할 경우 True, 조건에 맞지 않을 경우 False를 리턴한다. 예를 들어 arr = [2, 4, 3, 1]의 경우 2번 Queen(2열4행)과 3번 Queen(3열3행)이 서로 공격할 수 있으므로 False를 리턴한다. arr = [2, 4, 1, 3]일 경우 모든 Queen들이 서로 공격할 수 없으므로 True를 리턴한다. 이 함수는 bfs에서 마지막에 조건을 모두 충족하는지 확인하는 데에 사용된다.

**def** checkConflictNum(arr):

- checkConflict(arr)함수와 유사하나 이 함수에서는 모든 공격가능한 경우의 쌍의 개수를 제공한다. 예를 들어 arr = [2, 4, 3, 1]의 경우 공격할 수 있는 Queen의 쌍이 2번과 3번 Queen 한 쌍뿐이므로 1을 리턴한다. arr = [2, 4, 1, 3]일 경우 공격할 수 있는 Queen의 쌍이 없으므로 0을 리턴한다. 이 함수는 hc(hill climbing)에서 해당 상황에서 가장 적은 conflict가 발생하도록 Queen을 하나 이동시킬 때 사용한다.

**def** bfs(N):

- N으로 Queen의 개수를 입력하면 bfs방식으로 탐색하여 Queen을 놓을 수 있는 가능한 경우의 수를 하나 찾았을 때 해당 상황에서 각 Queen들의 행 번호의 배열을 반환한다. 각 Queen들이 서로 다른 열에 있다고 문제를 단순화시킬 수 있기 때문에, 예를 들어 배열의 i번째 원소가 j라면 Queen은 i+1열 j행에 놓인 것으로 본다. 만약 Queen을 놓을 수 있는 경우의 수가 없으면 None을 리턴한다. 예를 들어 bfs(4)는 [2, 4, 1, 3]을 리턴할 수 있고, bfs(2)는 N=2 일 때 Queen을 놓을 수 없으므로 반드시 None을 리턴한다.

**def** hc(N):

- 입력과 출력의 형식과 의미는 위의 bfs함수와 동일하나, Queen을 찾을 때 hill climbing search 방식을 사용한다. 해가 없으면 반드시 None을 리턴하지만, hill climbing에서 local optimal에 연속해서 빠질 수 있기 때문에 해가 있어도 운이 나쁘면 해를 찾지 못해 None을 리턴할 수도 있다.

**def** cspFindCase(legalPositionArr, currentArr, N, num):

- csp(constraint satisfaction problem)함수에서 사용하는 보조함수이다. legalPositionArr는 Queen이 현재 상황에서 특정 행에 놓일 수 있는 가를 나타내는 이차원 배열로 legalPositionArr[i][j]가 True이면 i+1번째 퀸이 j번행이 놓일 수 있는 상태이고 False라면 놓일 수 없는 상태를 의미한다. (forward checking방식을 이용한 것으로 자세한 방식은 이후 알고리즘 설명 문단에 작성하였다) currentArr는 현재까지 Queen이 놓인 행번호들의 배열로 처음에는 비었으나 퀸을 놓으면서 하나씩 채워지게 된다. N은 총 놓아야 하는 Queen의 수, num은 현재까지 이미 놓인 Queen의 수이다. 인자로 주어진 조건들에서 Queen이 놓일 수 있는 경우가 있으면 Queen들의 행번호 배열이 리턴되고 없다면 None을 리턴한다.

**def** csp(N):

- 입력과 출력의 형식과 의미는 위의 bfs함수와 동일하며 bfs가 아닌 constraint satisfaction problem으로 푸는 것만 다르다.

2. 알고리즘 설명

(1) BFS (Breadth-First Search)

- Breadth-First Search 방식으로 탐색한 함수인 bfs함수의 코드는 다음과 같이 작성하였다. 제출코드에서는 가능한 상세하게 주석으로 설명을 달아놓았으나 보고서에서는 가독성을 위하여 주석을 최대한 요약하였다.

1 **def** bfs(N):

2

3 bfsQueue = deque() *# (각 Queen의 행번호 배열, 배치된 Queen 수)의 튜플이 담김*

4 bfsQueue.append(([], 0))

5

6 **while** bfsQueue:

7 tempArr, tempNum = bfsQueue.popleft()

8 **if** tempNum == N: *# N개의 퀸들이 조건에 다 맞게 배치된 경우 solution 발견*

9 **if** checkConflict(tempArr):

10 **return** tempArr

11 **else**:

12 **for** i **in** range(1, N+1):

13 bfsQueue.append((tempArr+[i], tempNum+1))

14

15 **return** **None**

- 기본적인 함수동작은 Bread-First Search 방식과 같다. 큐에 넣는 원소는 (각 Queen의 행번호 배열, 현재까지 배치된 Queen 수)의 튜플이다.

처음에는 ([ ], 0) 하나만 들어간 상태에서 탐색을 시작한다(line 3-4). 이후 큐에서 원소를 하나씩 꺼내며(line 7) 만약 Queen이 N개 배치되어 있는 상태라면 checkConflict를 호출하여 모든 퀸들이 조건에 맞게 배치될 경우 바로 그 행번호 배열을 리턴한다(line 8-10). 만약 아직 Queen이 N개 배치되어 있지 않는 상태라면 방문하지 않은 인접한 노드를 큐에 다 넣게 되는데, 이는 행번호 배열에 1~N를 각각 하나씩 넣은 것이다(line 11-13).

수업에서 배운 BFS search에서는 큐에 원소를 넣기 전과 큐에서 원소를 꺼냈을 때 이미 방문한 노드인지 확인하는 절차가 필요하였다. 하지만 여기서는 구조상 큐에 원소를 넣을 때 중복해서 넣지 않기 때문에 별도의 visited 배열 등을 둘 필요가 없다. 또한 큐에 인접한 노드를 넣을 때 각 노드의 배열에 대하여 checkConflict를 호출해서 지금까지 놓은 상태에서 충돌이 일어난다면 굳이 더 탐색하지 않도록, subtree를 가지치기하는 것이 가능하다. 이 경우 훨씬 좋은 성능을 낼 수 있겠지만, 이후 작성할 csp에서도 사용할 방식이기 때문에 성능 비교를 위해서 일부러 모든 state를 탐색하는 방식으로 코드를 작성하였다.

(2) Hill-Climbing Search

- Hill-Climbing Search 방식으로 탐색한 함수인 hc함수의 코드는 다음과 같이 작성하였다. 제출코드에서는 가능한 상세하게 주석으로 설명을 달아놓았으나 보고서에서는 가독성을 위하여 주석을 최대한 요약하였다.

1 **def** hc(N):

2 **for** i **in** range(100): *# 총 100회 random restart*

3 tempArr = [random.randrange(1, N+1) **for** i **in** range(N)]

4

5 **while** **True**:

6 startConflict = checkConflictNum(tempArr) *# 처음 conflict 수*

7 minConflict = startConflict *# 하나 움직였을 때 conflict 최솟값*

8 tempMinArr = tempArr[:] *# 그 때의 Queen 배치*

9

10 **if** startConflict == 0: *# 해를 찾은 경우*

11 **return** tempArr

12

13 **for** j **in** range(N): *# 하나 움직였을 때 conflict 최소가 되는 경우 찾음*

14 **for** j2 **in** range(1, N+1):

15 **if** j2 == tempArr[j]:

16 **continue**

17

18 tempNewArr = tempArr[:]

19 tempNewArr[j] = j2

20 newConflict = checkConflictNum(tempNewArr)

21

22 **if** minConflict > newConflict:

23 minConflict = newConflict

24 tempMinArr = tempNewArr[:]

25

26 **if** minConflict == startConflict: *# 해가 없거나 local minimum*

27 **break**

28 **else**:

29 tempArr = tempMinArr

30

31 **return** **None**

- hc 함수에서는 우선 행번호를 무작위로 배정해서 배열을 만든다(line 3). 이 상태에서 checkConflictNum함수를 이용해서 Queen들이 서로 공격할 수 있는 쌍의 개수를 구한다(line 6). 만약 쌍의 개수가 없다면 조건에 만족하는 solution을 찾은 것이므로 바로 반환한다(line 10-11). 이후 퀸을 하나 움직였을 때의 Queen 공격쌍 개수를 모든 칸에 대하여 계산한다. 계산한 칸들중 가장 공격쌍의 개수가 적은 방향으로 Queen을 한 칸 이동시키고, 같은 과정을 conflict가 0이 될 때까지(즉 해를 찾을 때 까지) 반복한다. 만약 Queen 한 개를 어떻게 이동시켜도 더 conflict 수를 줄일 수 없다면 해가 없거나 0이 아닌 local minimum에 빠져 해를 못 찾은 것이 되므로 이 경우에도 반복이 종료된다(line13-29). 이 때 0이 아닌 local minimum에 빠져 해를 못 찾은 경우가 있을 수 있기 때문에 총 100번의 restart를 시도하여 최대한 높은 확률로 해를 찾을 수 있도록 하였다(line 2).

(3) CSP (Constraint Satisfaction Problem)

- Constraint Satisfaction Problem 방식으로 탐색한 함수인 csp함수와 cspFindCase함수의 코드는 다음과 같이 작성하였다. 제출코드에서는 가능한 상세하게 주석으로 설명을 달아놓았으나 보고서에서는 가독성을 위하여 주석을 최대한 요약하였다.

1 **def** cspFindCase(legalPositionArr, currentArr, N, num):

2 **if** N == num:

3 **return** currentArr

4

5 **for** e **in** [i **for** i **in** range(N) **if** legalPositionArr[num][i]]:

6 tempPositionArr = deepcopy(legalPositionArr)

7 **for** j **in** range(num+1, N):

8 tempPositionArr[j][e] = **False** *# 같은 행 배치 불가*

9 **if** e+(j-num) < N:

10 tempPositionArr[j][e+(j-num)] = **False** *# 대각방향 배치 불가*

11 **if** e-(j-num) >= 0:

12 tempPositionArr[j][e-(j-num)] = **False** *# 대각방향 배치 불가*

13 result = cspFindCase(tempPositionArr, currentArr+[e+1], N, num+1)

14 **if** result != **None**:

15 **return** result

16

17 **return** **None** *# 놓을 수 있는 곳이 empty value set일 경우*

18

19

20 **def** csp(N):

21 legalPositionArr = [[**True** **for** \_ **in** range(N)] **for** \_ **in** range(N)]

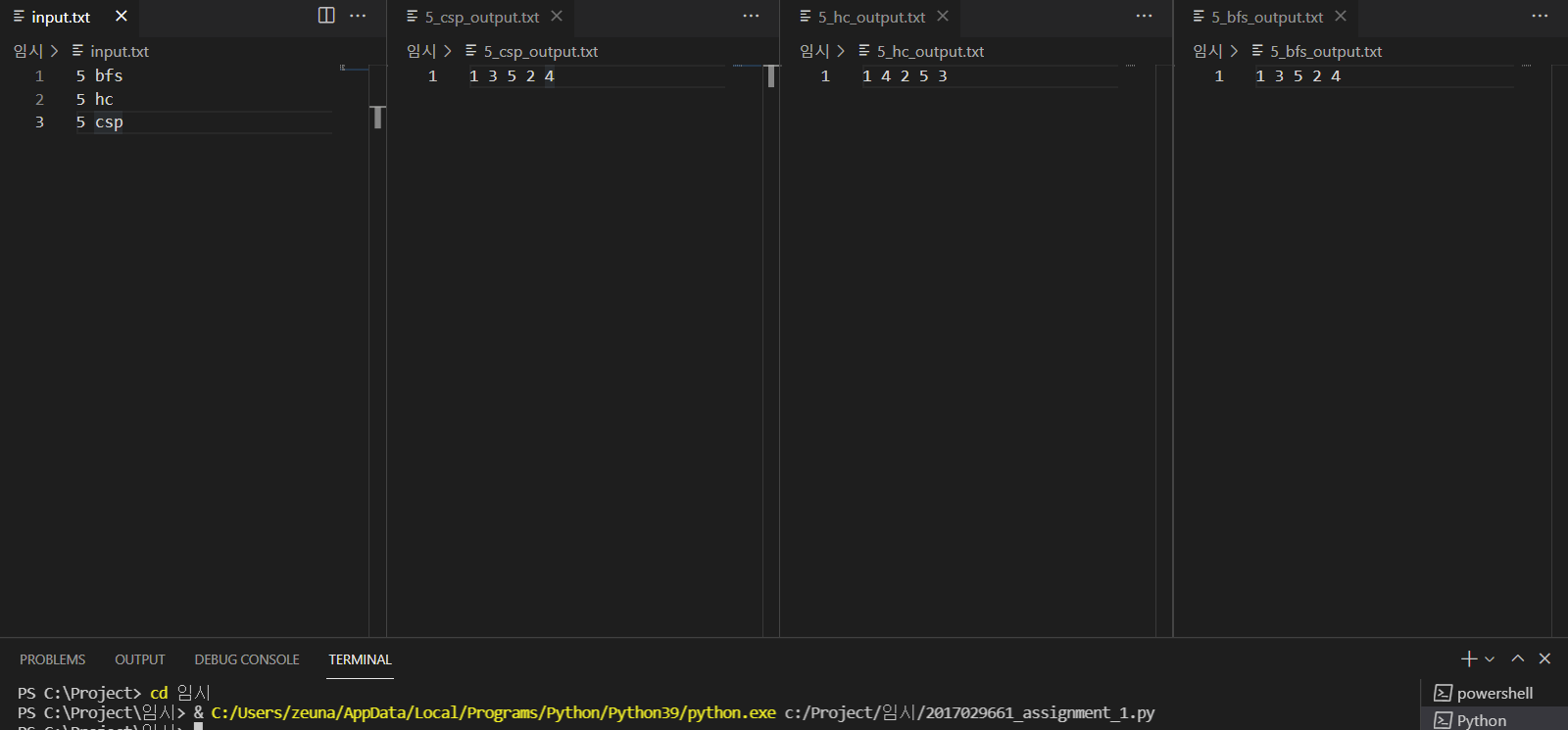
22 **return** cspFindCase(legalPositionArr, [], N, 0);

- csp 풀이 과정에서 주요 연산은 cspFindCase함수에서 진행된다. cspFindCase함수에서는 총 4개의 인자를 받는다. 위의 풀이에서는 수업에서 배운 forward checking 방식을 이용하여 현재 상황에서 배치될 수 있는 legal position을 legalPositionArr에 담도록 하였다. 처음에는 각 Queen에 대하여 가능한 행번호만을 저장하도록 설계하였으나, 배치 불가능해진 행번호 경우를 제거하는 과정에서 O(N)의 시간이 소요되기 때문에 좀 더 메모리를 쓰더라도 행번호 경우를 O(1)에 제거할 수 있도록 가능여부를 True/False로 저장하도록 하였다. 예를 들어 legalPos[a][b]가 True라면 a+1번째 Queen이 b+1번행에 배치될 수 있는 상황이고, False라면 배치될 수 없는 상황임을 의미한다. forward checking에서 처럼 처음에는 legalPos배열이 전부 True로 세팅되었다가(line 21), 불가능해지는 위치가 생길경우 해당 위치에 False로 교체된다. 그 외 나머지 인자인 currentArr는 현재까지 Queen이 놓인 행번호를, N은 총 놓아야 하는 Queen의 개수를, num은 현재까지 놓은 Queen의 개수를 의미한다.

종합하면 cspFindCase(legalPositionArr, currentArr, N, num)은 현재까지 num개의 퀸이 currentArr에 놓인 상태에 있을 때(그 때의 다른 Queen들이 놓일 수 있는 배치는 legalPositionArr) num+1번째 Queen을 놓는 방법을 탐색하여 결과적으로 총 N개의 Queen이 조건에 맞도록 하는 행번호들의 배열을 반환한다. csp함수에서는 line 21-22와 같이 초기값을 넣고 그 결과를 반환한다.

cspFindCase함수에서는 num+1번째 Queen을 놓기 위해 legalPositionArr[num]에서 값이 True인 행번호들에 대하여 for문으로 탐색한다(line 5). 각각의 행번호에 대하여 num+1번째 Queen을 놓은 상태에 대해서 새롭게 함수를 호출하되(line 13) 이 때 num+1번째 Queen을 놓은 것에 대해 이후 놓을 Queen들의 legalPosition이 변했으므로 line 7-12와 같이 설정한 뒤 인자로 넣어준다. Queen을 하나씩 놓는 것을 반복하다가 N개까지 다 놓으면 지금까지의 행번호를 리턴하여 solution을 구하는 것을 완료한다(line 2-3). 만약 도중에 empty value set을 만나는 경우 line 17에 의해 None을 리턴하게 되는데, 이 경우 해당 함수를 불렀던 함수로 백트래킹 된다. 만약 주어진 N에 대해 solution이 없으면 어떤 경우든 결과적으로 None만을 리턴하여 csp함수의 결과도 None이 될 것이다.

3. 실행 결과



- N=5일 때 bfs, hc, csp 함수를 실행한 결과 모두 정상적으로 solution을 찾는 것을 확인할 수 있다. 첫번째 Queen이 1부터인 경우부터 계산하도록 설계했던 bfs와 csp함수와 달리 hc의 경우에는 무작위 상태에서 시작하도록 만들었기에 실행할 때마다 결과가 다를 수 있고, 아주 운이 나쁠 경우 해를 찾지 못할 수도 있다. 하지만 random start를 100번 했기에 매우 높은 확률로 해를 찾아낼 수 있고, csp/bfs와 행번호가 다를 수는 있어도 Queen이 서로 공격 못하도록 하는 solution을 출력하는 건 동일하다.