Assignment 1 – N Queens Problem

CSE4007 2021-2학기 (13347)

기계공학부 ********** 손상현

---

1. 함수 설명

본 과제 코드는 총 5개의 모듈을 포함합니다.

- text 입출력 및 실행과 관련된 ************_assignment_1.py** (실행파일)
- 알고리즘들에서 공통적으로 적용되는 로직을 포함하는 **utils.py**
- 각 알고리즘의 구현 함수를 포함하는 **BFS.py, HC.py, CSP.py**

a. **********_assignment_1.py

************_assignment_1.py** 파일을 실행하면 아래 세 함수가 연속으로 호출되어 작업을 수행합니다.

함수 명	리턴 값	설명
read_inputs(path)	[[N, 'algorithm_type'], ...]	이 함수는 input.txt를 읽어 [N, 'algorithm_type']를 가지는 리스트를 입력 값의 개수만큼 생성하고, inputs 리스트에 담아 반환합니다.
run_search(inputs)	[[N, 'algorithm_type', solution], ...]	이 함수는 read_inputs에서 반환한 리스트 정보를 입력받아, 지정한 알고리즘으로 N-Queens의 솔루션을 1개 탐색합니다. 탐색 결과들은 [[N, 'algorithm_type', solution] ...] 형태를 갖는 results 리스트를 생성해 반환합니다.
write_output(results)	없음	run_search() 함수의 탐색 결과를 입력받아 지정된 양식의 .txt 파일로 출력합니다. *이때 solution이 비어있는 list인 경우, no solution을 출력합니다.*

b. utils.py

utils.py 파일에 포함된 함수들은 **BFS.py, HC.py, CSP.py** 모듈의 알고리즘에서 공통적으로 호출되는 로직을 포함하고 있습니다. 또한 실험 결과를 출력하기 위한 함수를 포함합니다.

함수 명	리턴 값	설명
cvtIndex(queens)	[int, ...]	이 함수는 (0 ~ N-1)의 인덱스로 계산된 queen들의 좌표를 (1 ~ N) 범위로 변환합니다. 각 알고리즘 함수들이 solution 값을 구하게 되면 이 함수를 호출해 값을 변환하고, return

		값을 할당합니다.
isPair(q0, q1)	Boolean	두 개의 queen을 입력받아 서로가 이동위치 상에 있는지 검사합니다. 입력값은 [x, y] 리스트 형태의 좌표 2개를 입력받습니다. 아래 세 가지 요소를 체크하고, 최소 한개의 쌍이 존재하면 True를, 그렇지 않으면 False를 반환합니다. 1. x 좌표가 같은지 2. y 좌표가 같은지 3. 두 퀸이 대각선 방향에 있는지 abs(x0-x1), abs(y0-x1) 두 값이 같은 경우 대각선 방향임을 판단합니다.
checkNoPair(queens)	Boolean	0번 인덱스부터 순차적으로 채워진 퀸들간의 공격쌍이 있는지 검사합니다. 리스트에 있는 모든 퀸 쌍들에 대해 isPair() 함수로 검사하며, 한 개의 쌍이라도 공격쌍인 경우, False를 반환합니다.
visualize(solution, verbose=2)	없음	입력받은 solution을 명령창에 출력합니다. verbose=0인 경우 solution의 존재 여부만을, verbose=1인 경우 각 퀸들의 index를, verbose=2인 경우 각 퀸들의 위치를 체스판을 그려 함께 출력합니다.

c. BFS.py

탐색 함수들의 자세한 동작은 뒤의 알고리즘 설명에서 다룹니다.

함수 명	리턴 값	설명
bfs(N)	goal_queens [int, ...]	N 값을 입력받아 N-Queens 문제에 Breadth First Search를 수행합니다. BFS를 수행하기 위해 python의 내장 자료구조인 queue.Queue를 사용합니다. BFS를 실행하는 중 첫 solution을 찾으면, 탐색을 중지하고 solution을 반환합니다.

d. HC.py

탐색 함수들의 자세한 동작은 뒤의 알고리즘 설명에서 다룹니다.

함수 명	리턴 값	설명
countPairs(queens)	int	입력받은 퀸들의 현재 상태에서 공격쌍의 수를 반환합니다. isPair() 함수를 사용합니다.
hc(N)	goal_queens [int, ...]	N 값을 입력받아 N-Queens 문제에 Hill Climbing을 수행합니다. N이 2, 3인 경우를 제외하고는 local minimum에 도달할 경우 optimal solution을 찾을 때 까지 random restart를 반복합니다. solution을 찾으면, 탐색을 중지하고 solution을 반환합니다.

e. CSP.py

탐색 함수들의 자세한 동작은 뒤의 알고리즘 설명에서 다룹니다.

함수 명	리턴 값	설명
csp(N)	goal_queens [int, ...]	N 값을 입력받아 N-Queens 문제에 CSP 문제 해결을 위한 알고리즘 중 하나인 DFS 기반의 Backtracking Search를 수행합니다. Backtracking Search를 수행하기 위한 stack으로 python의 기본 list 자료형을 사용합니다. Backtracking Search를 실행하는 중 첫 solution을 찾으면, 탐색을 중지하고 solution을 반환합니다.

2. 알고리즘 설명

a. 공통 적용 (공격쌍 판정)

isPair(q0, q1) 함수 원형 (utils.py)

```
6   # return True when two queens are attacking
7   def isPair(q0, q1):
8       # check queen pair
9       if (q0[0] == q1[0] or                    # check x position
10          q0[1] == q1[1] or                    # check y position
11          abs(q0[0]-q1[0]) == abs(q0[1]-q1[1])):  # check diagonal difference
12          return True
13      else:
14          return False
```

두 퀸이 주어졌을 때 이동경로 상에 있어 서로 공격이 가능한지 여부를 판단합니다. 입력값으로는 q0=[x, y], q1=[x, y] 좌표를 받습니다. 두 퀸이 같은 직선상에 있는 경우는 x 위치가 같거나, y 위치지가 같은 경우입니다. 대각선 상에 있는 경우는 x위치 차이의 절대값과 y위치 차이의 절대값이 같은 경우입니다. 직선 혹은 대각선 상에 존재하는 경우 공격이 가능하므로 True를 반환합니다.

checkNoPair(queens) 함수 원형 (utils.py)

```
17  # return True when no pairs
18  def checkNoPair(queens):
19      N = len(queens)
20
21      # iterating all pairs in the list
22      for i in range(N):
23          for j in range(i+1, N):
24              q0 = [i, queens[i]]
25              q1 = [j, queens[j]]
26
27              # check queen pair
28              if isPair(q0, q1):
29                  # return False if any of pairs is invalid
30                  return False
31
32      # no pairs found
33      return True
```

이 함수는 퀸들의 상태를 입력받아, 2차원 반복문으로 모든 쌍들에 대해 isPair()함수로 공격쌍인지 조사합니다. 이때 한 개의 쌍이라도 공격쌍이 되는 경우 바로 False를 반환합니다. 모든 쌍들이 서로 공격하지 않으면 True를 반환합니다.

b. Breadth First Search

bfs() 함수 원형 (BFS.py)

```python
def bfs(N):
    # using queue for BFS
    queue = Queue()

    # initial state: start filling from left
    # state can have N values, within range from 0 to N
    init_queens = []
    queue.put(init_queens)

    # the list will be assigned when goal state is found
    # if not, empty list will be returned.
    goal_queens = []

    # perform bfs until queue is empty (all states have been reached)
    while not queue.empty():
        state_queens = queue.get()

        # check no queens pair is attacking
        if checkNoPair(state_queens):
            # N queens with no pair (solution)
            if len(state_queens) == N:
                # assign goal state
                # clear queue (to stop search here)
                goal_queens = cvtIndex(state_queens)
                queue.queue.clear()

            # less queens with no pair (valid state)
            else:
                # add successors to the queue
                # branching factor of N
                for i in range(N):
                    # generating successor (copy state and add a queen)
                    successor_queens = list(state_queens)
                    successor_queens.append(i)
                    queue.put(successor_queens)

        # if some queens are attacking each other
        else:
            # backtrack (do nothing)
            pass

    # return empty list when bfs fails
    return goal_queens
```

bfs() 함수는 퀸의 개수 N을 입력받아, N-Queens를 만족하는 해 1개를 찾아 반환하는 함수입니다. Breadth First Search 탐색을 위해 queue.Queue 자료형을 사용합니다. First In, First Out 동작으로 인해 queue에 먼저 삽입된 state node들이 먼저 확장될 수 있도록 합니다. 이 결과 먼저 queue에 먼저 삽입된 상위 level을 모두 확장하고, 그 다음 level의 node 확장하게 진행됩니다. 큐에 삽입되는 node의 개수는 brute force라면 branching factor가 N이므로 매 level에서 N^d로 증가합니다. 그러나 이 함수는 BFS를 하되, pruning을 적용했습니다. 뒤의 문단에서 설명합니다.

BFS를 수행함에 있어 왼쪽의 퀸부터 한 level에서 한 column씩 배치하도록 문제를 reduction했습니다. 총 N 개의 column을 가지므로, BFS는 최대 N의 depth를 가집니다. 각 상태의 child state는 현재 퀸 배치에서 다음 column에 새로운 퀸을 놓는 것으로 생성합니다.

또한 이 N-queens 문제에서만은 reduction으로 인해, 한 column에서 배치할 수 있는 여러 state들은 절대로 겹치지 않는 child state를 가지므로 한번 expand 한 상태에 다시 도달했는지 여부를 체크하지 않아도 됩니다. 다른 일반적인 BFS에서는 체크를 해야 하는 부분입니다. 단, 체스판의 대칭으로 인해 겹치는지 여부는 알지 못합니다.

매 while 루프에서는 queue에서 한 개의 state를 가져와서 확장하고, successor를 queue에 다시 넣습니다. depth N에 도달하여 queen을 담고 있는 state 배열의 길이가 N이 된 경우, checkNoPair() 함수로 퀸들간의 공격쌍이 있는지 체크하고, 없는 경우에는 goal_state 변수에 solution을 할당합니다. 그 다음 queue를 비워 while 루프를 탈출합니다. 혹은 모든 상태를 다 탐색을 해도 solution을 찾지 못한 경우에는 queue가 비워져 루프를 탈출해도 goal_state 변수가 빈 리스트로 남아있게 됩니다. 함수의 마지막에서는 goal_state 변수를 반환합니다. N=2, 3인 경우입니다.

중요한 변경점으로는, 이 함수에서는 BFS를 하되 완전한 N 깊이에 도달하지 않았을 때도, 매 while 루프에서는 if checkNoPair(queens) 구문으로 현재 상태의 퀸들이 서로 공격하지 않는지를 체크하도록 구현하였습니다. 이 경우 완성되지 않은 현재 퀸들의 배치에서도 서로 공격하는 쌍이 발생하는 경우, successor를 만들지 않고 그 subtree를 pruning 합니다. 뒤의 CSP 알고리즘에서 다시 한번 설명합니다. 이 분기는 원래 후술할 CSP의 Backtracking Search를 위해 구현되었습니다. 다만 BFS에서도 함수의 큰 수정 없이 바로 적용이 가능하므로 채용하였습니다.

c. Hill Climbling

countPairs(queens) 함수 원형 (hc.py)

```
5   def countPairs(queens):
6       N = len(queens)
7
8       count = 0
9       # iterating all pairs in the list
10      for i in range(N):
11          for j in range(i+1, N):
12              q0 = [i, queens[i]]
13              q1 = [j, queens[j]]
14
15              # check queen pair
16              if isPair(q0, q1):
17                  count += 1
18
19      return count
```

이 함수는 입력받은 queens 상태에서 몇 개의 공격쌍이 발생하는지를 계산합니다. queens의 모든 쌍들에 대해 isPair()를 이용해 공격쌍 여부를 검사하고, True일 경우 카운팅합니다. 카운팅이 끝나면 개수를 반환합니다. hc() 함수에서 heuristic을 계산하기 위해 사용합니다. (뒷장에 계속)

hc(N) 함수 원형 (hc.py)

함수 길이가 길어 중간의 반복문 일부(47~66번 줄)을 나눠서 설명합니다.

```
22  def hc(N):
23      # boundary value, no solution exist
24      if 2 <= N and N <= 3:
25          return []
26
27      # goal states to be returned
28      goal_queens = []
29
30      local_minimum_h = -1
31      while local_minimum_h != 0:
32          # init queens with random state
33          state_queens = [random.randrange(0,N) for i in range(N)]
34
35          # init heuristic with worst case value
36          predecessor_h = N*(N-1)/2 + 1
37          state_h = N*(N-1)/2
38
39          # depth = 0
40          # stop loop when heuristic value do not decrease
41          while state_h < predecessor_h:
42              # init minimum h value
43              successor_h_min = state_h
44              successor_h_min_positions = []
45
46              # search heuristic values
47              for x in range(N):      # x index…
67
68              # randomly select one successor
69              successor_x, successor_y = random.choice(successor_h_min_positions)
70              state_queens[successor_x] = successor_y
71
72              # update heuristic value
73              predecessor_h = state_h
74              state_h = successor_h_min
75
76
77          if state_h == 0:
78              goal_queens = cvtIndex(state_queens)
79
80          local_minimum_h = state_h
81
82
83      return goal_queens
```

hc() 함수는 퀸의 개수 N을 입력받아, N-Queens를 만족하는 해 1개를 찾아 반환하는 함수입니다. 이 알고리즘은 탐색한 상태공간을 모두 저장하지 않는 이상, solution의 존재 여부를 판별하지 못하므로 solution이 없는 2, 3에 대해서는 예외값 처리를 해두었습니다.

가장 바깥의 while 반복문은 hill climbing으로 탐색한 상태의 local minimum heuristic 값이 0에 도달할 때 까지 계속 random restart를 반복합니다. random restart는 0 ~ N-1 인덱스의 퀸을 0 ~ N-1 사이의 값으로 랜덤하게 초기화합니다. heuristic 값은 현재 상태가 가지는 퀸 공격쌍의 개수로 정의합니다. 1과 4 이상의 숫자에 대해서는 해가 존재함이 알려져있으므로, hill climbing 알고리즘이 global minimum인 heuristic=0에 도달할 때 까지 반복 시행합니다.

다음 안쪽의 while문은 hill climbing의 move를 반복합니다. 현재의 퀸 상태의 heuristic 값을 predecessor_h으로 저장합니다. 그리고 현재 상태에서 움직일 수 있는 모든 칸들에 대해 successor의 heuristic값을 조사합니다. 이 중 가장 heuristic이 낮아지는 위치들을 list에 저장해 주었다가, random하게 한 개를 선택해 퀸을 움직이고, 그 heuristic 값을 state_h에 저장합니다. 만약 move를 반복했는데도 state_h가 predecessor_h보다 작아지지 않았다면 local_minimum에 도달했음을 의미합니다.

```
46              # search heuristic values
47              for x in range(N):      # x index
48                  for y in range(N):  # y index
49                      # generate successor
50                      successor = state_queens.copy()
51                      successor[x] = y
52                      h_value = countPairs(successor)
53
54                      # if h value of current position is smaller than minimum
55                      if (h_value < successor_h_min):
56                          # renew h_min and positions
57                          successor_h_min = h_value
58                          successor_h_min_positions = [[x, y]]
59                      # if h value of current position equals minimum
60                      elif (h_value == successor_h_min):
61                          # keep tracking positions
62                          successor_h_min_positions.append([x, y])
63                      # if h value is higher
64                      else:
65                          # do nothing
66                          pass
```

위의 코드에서 접어둔 47번 줄의 반복문 부분입니다. 2차원 반복문은 현재 상태에서 한개의 퀸을 움직일 수 있는 모든 경우의 수로 successor를 생성합니다. 그리고 바로 위에서 설명한 countPairs()함수로 successor의 heuristic 값을 조사합니다. 가장 낮은 heuristic값을 가지는 successor들만을 기억하고, 이 중 하나를 골라 랜덤하게 이동합니다.

이러한 전체 과정을 시행함으로서 hill climbing은 N-Queens 문제를 풀 수 있습니다.

d. Constraint Satisfaction Problem

csp() 함수 원형 (CSP.py)

```
3   # domain: 0~N column queens
4   # value: 0~N row position
5   def csp(N):
6       # based on DFS
7       stack = []
8
9       # initial state: start filling from left
10      # state can have N values, within range from 0 to N
11      init_queens = []
12      stack.append(init_queens)
13
14      goal_queens = []
15
16      while len(stack) != 0:
17          state_queens = stack.pop()
18
19          # check no queens pair is attacking
20          if checkNoPair(state_queens):
21
22              # N queens with no pair (solution)
23              if len(state_queens) == N:
24                  # assign goal state
25                  # clear stack (to stop search here)
26                  goal_queens = cvtIndex(state_queens)
27                  stack.clear()
28
29              # less queens with no pair (valid state)
30              else:
31                  # add successors to the stack
32                  # branching factor of N
33                  for i in range(N):
34                      # generating successor (copy state and add a queen)
35                      successor_queens = list(state_queens)
36                      successor_queens.append(i)
37                      stack.append(successor_queens)
38
39          # if some queens are attacking each other
40          else:
41              # backtrack (do nothing)
42              pass
43
44      return goal_queens
```

(뒷장에 계속)

N-Queens의 CSP 문제는 퀸이 놓일 수 있는 column N개를 variable, 각 column에서의 queen의 위치를 value로 정의했습니다. 그리고 constraint로는 모든 퀸들이 서로를 공격할 수 없어야 합니다.

csp() 함수는 퀸의 개수 N을 입력받아, N-Queens를 만족하는 해 1개를 찾아 반환하는 함수입니다. Backtracking Search를 구현하였으며, Depth First Search 기반의 알고리즘이므로 stack 자료구조로서 파이썬의 내장 list 자료형을 pop과 append로 접근하는 방식으로 사용하였습니다. stack은 Last-In, First-Out 동작을 하므로 늦게 생성된 깊은 level의 노드들을 계속 깊게 확장합니다.

앞선 BFS 알고리즘과 마찬가지로, Backtracking Search를 수행함에 있어 왼쪽의 퀸부터 한 level에서 한 column씩 배치하도록 문제를 reduction해서 variable과 value를 정의했습니다. 총 N 개의 column을 가지므로, 최대 N의 depth를 가집니다. limited-depth이므로 무한 깊이에 빠지지 않고, solution이 존재하는 경우 무조건 도달하게 됩니다. (reduction으로 인한 completeness) 또한, 이 문제에서만은 reduction을 통해 각 subtree들은 절대로 겹치지 않으므로 이미 한번 expand한 state인지 여부를 검사하지 않아도 됩니다. 단, 체스판의 대칭으로 인해 겹치는지 여부는 알지 못합니다.

아무 퀸도 배치하지 않은 초기 상태를 stack에 집어넣어 탐색을 시작합니다. While 루프에서 stack의 노드 한 개를 꺼내 탐색을 하면, 그 상태가 constraint를 만족하는지를 checkNoPairs() 함수로 항상 테스트합니다. 만약 퀸들이 N개가 다 배치 되기도 전에 constraint에 위배 된다면, subtree를 탐색할 필요 없이 바로 backtracking을 합니다. 각 상태의 child state는 현재 퀸 배치에서 다음 column에 새로운 퀸을 놓는 것으로 생성합니다. constraint를 만족한다면 다음 column에 0~N-1 칸에 퀸을 놓아 successor들을 생성하고 stack에 삽입합니다.

이 외, CSP에서 사용되는 Most Constrained/Constraining Variable, Least Constraining Value와 같은 휴리스틱과 Forward Checking은 구현하지 않았습니다.

(뒷장에 계속)

3. 실험결과

입력값

input.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
5 bfs
5 hc
5 csp
```

출력값

text 출력 (Index 1부터 시작)

5_bfs_output.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
1 3 5 2 4
```

5_hc_output.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

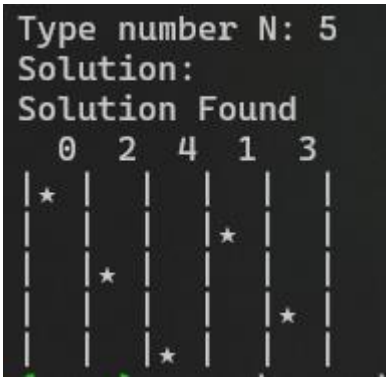
```
4 1 3 5 2
```

5_csp_output.txt - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

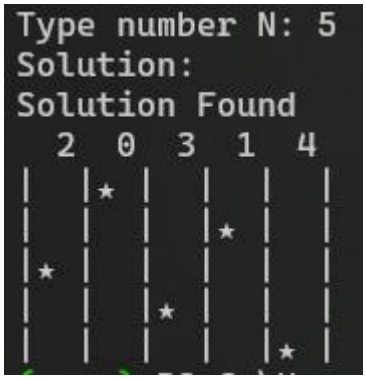
```
5 3 1 4 2
```

명령창 출력 (Index 0부터 시작)

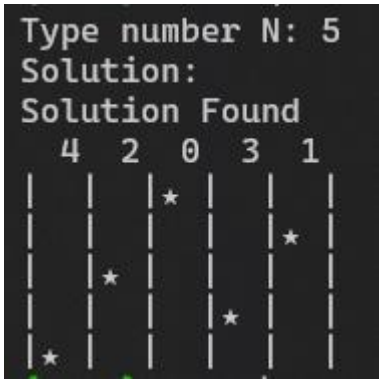
BFS



Hill Climbing



CSP



- Hill Climbing은 매 실행마다 random으로 인해 처음 찾게되는 solution이 바뀜
- BFS와 CSP는 successor를 queue/stack에 넣는 순서에 따라 leftmost 혹은 rightmost 솔루션을 탐색함.