인공지능개론 과제 #1

Nqueens problem

Nqueen 문제는 주어진 체스 여왕말의 수(N)와 NxN 체스판에 서로의 공격을 받지 않는 위치에 적절하게 배치하는 것이 핵심이다.

앞서 들어가기 전, 여왕말은 한 차례에 좌우상하 대각선, 모든 방향 체스판의 끝까지 이동할 수 있다,

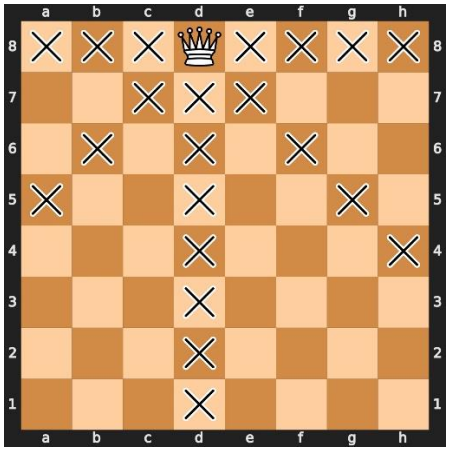

그림 1 한 차례에 여왕말이 이동할 수 있는 경우의 수

이러한 점을 고려하여 아래 3 가지 알고리즘에서 공통적으로 탐색공간을 다음과 같이 정의하였다.

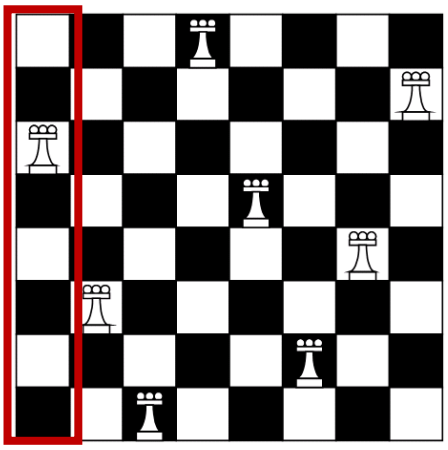

그림 2 한 열의 위치를 값, 행을 인덱스

가용 공간을 줄이기 위해 2 차원 배열이 아닌, 체스판의 각 행의 위치를 값(value)로, 열을 인덱스(index)로 정의하였다. 이는 각 열에서 하나의 여왕말만 존재하므로 다음 그림 2 에서 각 여왕말의 위치를 배열로 표현하면 [3,6,8,1,4,7,5,2]로 표현될 수 있다. 코드 상에서는 클래스 속 self.chessboard 로 표현되었다.

BFS (Breadth First Search)

Breadth First Search 는 너비 우선 탐색으로, 다음 단계의 탐색할 공간보다 현재 앞으로 탐색할 공간들을 우선적으로 탐색하는 알고리즘이다. 정보 없는 검색(Uninformed Search)의 종류 중 하나이며, 현 상태에서 Goal 까지의 Path cost 를 모르는 것이 BFS 의 특징이다. DFS 와는 다르게 중복 경로 생성문제에 대해 고려할 필요가 없으며, 따라서 Complete 하다는 점에서 차이점이 있다, N-queens 문제를 해결하기 위해 Queue(frontier)를 이용하여 queue 의 content 는 각 여왕말의 위치를 배열로 설정하였다.

Code Review - 함수 설명

```
def __init__(self, N):
    self.N = int(N)
    self.chessboard = Queue() # as frontier queue
```

BFS 에서 chessboard 는 Queue 이며, element 는 여왕말의 위치 배열.

```
def isLegal(self, chessboard):
    for yi_idx in range(len(chessboard)):
        for xi_idx in range(yi_idx + 1, len(chessboard)):  # column
            if chessboard[xi_idx] == chessboard[yi_idx]:  # 직선일 경우
                return False
            if abs(chessboard[yi_idx] - chessboard[xi_idx]) == abs(yi_idx - xi_idx):
# 대각선일 경우
                return False
    return True
```

여왕말간 서로 충돌 여부를 확인하며 전도유망(promising)한지 확인한다. 그림 1 참고.

```
def isGoal(self, chessboard):
  if len(chessboard) == self.N:
      return True
  else:
       return False
```

Queue 에 저장되는 chessboard 에 여왕말이 모두 배치되는 것이 목표이므로 여왕말의 위치 배열의 개수가 주어진 N 에 다다르면 종료하게끔 Bool 값 반환

```
def bfs(self): # no argument

  self.chessboard.put([0]) # 여왕말의 위치를 (1,1)에 먼저 배치

  while not self.chessboard.empty():
      queue = self.chessboard.get()
```

```
        if self.isGoal(queue):
            queue = [i+1 for i in queue]
            return queue

        for row_pos in range(self.N):
            new_queue = queue + [row_pos]
            if self.isLegal(new_queue): # promising?
                  self.chessboard.put(new_queue)
```

먼저 여왕 위치를 (1,1)를 놓고 frontier 가 비어지기 전까지 반복문을 계속 돌리게 된다. 본 알고리즘에서 사용하는 Frontier 는 queue 인 선입선출 구조이므로, 먼저 넣은 배열을 꺼내게 된다. 꺼낸 배열이 목표에 도달(isGoal:True)하면, 여왕말의 위치를 인덱스로 저장하였으므로 **전체 배열 속 element 에 1 을 더한 배열을 반환**한다. (이 때는 다른 경우의 수를 고려하지 않고 목표에 도달(isGoal:True)만 하면 반환하게끔 하였다.) 그 외인 경우에는 기존 배열에서 현재 탐색할 공간들을 new_queue 에 추가함으로써 new_queue 에 저장된 상태가 전도유망한 경우라면 chessboard 에 저장하도록 하였다.

Test Result - 실험 결과

그림 3 N=5 일 때 Breadth First Search 알고리즘 결과

초기 상태에서 무작위로 시작하는 것이 아니며, 다양한 경우의 수를 고려하지 않고 무조건 목표에 도달하면 반환하므로 어떤 실행이든 같은 결과를 출력하게 된다.

HC (Hill Climbing)

앞서 제시한 알고리즘과 달리 Hill Climbing(언덕 오르기)은 Path cost 를 무시하고 Goal 자체를 찾는 것이 목표인 알고리즘이다. 해당 알고리즘은 경로에 관심이 없고 현재 위치에서 가장 좋은 선택을 하는 탐욕법에 의해 움직이므로 전체 상황에서 가장 최선의 값인 global value 보다는 현재 상황에서의 최선의 값인 local value 로 수렴하게 된다. 탐욕법에 의해 움직이다 보니, 목표를 빠르게 찾는 장점이 있으나, 어느 시작점 인지에 따라서 최선의 값이 달라지거나, 진척이 불가능한 지점(plateau) 다다르게 되는 단점을

지니고 있다. 여러가지 솔루션 중 본 N-queens 문제에서는 무작위로 재시작하는 방법을 택하였다. N-queens 문제 알고리즘 도식은 아래 이미지와 같다.

그림 4 Nqueens 문제에서 Hill Climbing 알고리즘

Code Review - 함수 설명

```
def __init__(self, N):
    self.N = int(N)
    self.chessboard = [random.randint(0, self.N-1) for _ in range(self.N)] # complete-State formulation #index = xi as column, value = row
    self.heuristicBoard = [[0 for _ in range(self.N)] for _ in range(self.N)]
    self.hCost = float("inf")
```

알고리즘 특성상 초기 상태에 따라 최선의 값이 정해지므로 무작위로 선택된다 (self.chessboard),

체스보드에서 모든 가능한 후행자의 heuristic 값을 저장하는 heuristicBoard, 그리고 현 상황의 heuristic 값 (hCost)까지 선언. 전체 hCost(서로 충돌하는 여왕말의 pair 수)는 최대값으로 0 으로 수렴하는 방향으로 Goal 을 찾아간다.

```
def heuristic(self, chessboard): #heuristic function: 좀 더 좋은 방향으로 생각해볼 것 시간 비용
    currentHCost = 0
    for xi in range(self.N-1):
        for xi_idx in range(xi+1, self.N):  # column
            if chessboard[xi_idx] == chessboard[xi]:  # 직선일 경우
                currentHCost += 1
            if abs(chessboard[xi] - chessboard[xi_idx]) == abs(xi - xi_idx):  # 대각선일 경우
                currentHCost += 1
    return currentHCost
```

Heuristic function 에서는 이전 promising 여부에 대한 조건과 동일하다. 현재 각 chessboard 상태에서 heuristic 값을 반환한다.

```
def getHeuristicBoard(self):
    chessboard = copy.copy(self.chessboard) # temporary chessboard
    for xi in range(self.N):
        originalPos = chessboard[xi]
        for yi in range(self.N):
            chessboard[xi]=yi
            self.heuristicBoard[yi][xi] = self.heuristic(chessboard)
        chessboard[xi] = originalPos
```

Chessboard 의 각 여왕말의 위치를 가정함에 따라 heuristicBoard 가 갱신이 된다. 이 때 chessboard 원 저장 상태를 훼손하지 않아야 하므로 originalPos 에 저장하면서 갱신한다.

```
def hc(self, N): # argument: board size
    steps = 0
    attempts = 0

    while self.hCost > 0: # 목표인 hCost 가 0 이 되기 전까지
        if attempts > 100: # 100 시도 후에는 goal 도달 실패 시 no solution 반환
            break
        if steps > N :
            self.__init__(N)  # generate initial state randomly ( # 무작위 재시작 언덕 오르기)
            steps = 0
            attempts += 1

        self.getHeuristicBoard() # 현 상태에서의 heursitcBoard 갱신

        globalMinXi = -1
        globalMinYi = -1
        globalMin = float('inf')

        for xi in range(0, self.N):
```

```
            columnHeuristic = []
            for yi in range(0, self.N): # 각 열에서 Heuristic 상황을 저장
                columnHeuristic.append(self.heuristicBoard[yi][xi])
            columnMin = min(columnHeuristic) # 최고 추정치 선별
            columnMinYi = columnHeuristic.index(min(columnHeuristic))
            columnMinXi = xi

            if columnMin < globalMin: # 최고 추정치 갱신
                globalMin = columnMin
                globalMinXi = columnMinXi
                globalMinYi = columnMinYi

        self.chessboard[globalMinXi] = globalMinYi # chessboard 갱신
        self.hCost = self.heuristic(self.chessboard) # 현 상태의 h 값 갱신
        steps += 1

    if self.hCost == 0:
        solution = [i+1 for i in self.chessboard]
        return solution
    else:
        pass #return null
```

상세한 설명은 주석 참고. 전체적으로 heursiticBoard 를 매 상태마다 갱신함으로써 각 열에서 여왕말이 선택할 수 있는 최고 추정치에 따라 chessboard 의 value(행의 위치)를 갱신한다.

진척이 불가능한 지점에 다다르는 것을 미연의 방지로 N 번내로 목표에 도달하지 않는다면 초기 상태로 돌아가 다시 무작위로 재시작한다. **전체 시도가 100 번 초과시 불가능한 상태로 판단**하고 **null 을 반환**하며 **성공 시, heuristic cost 가 0 인 chessboard 를 반환**한다.

Test Result - 실험 결과

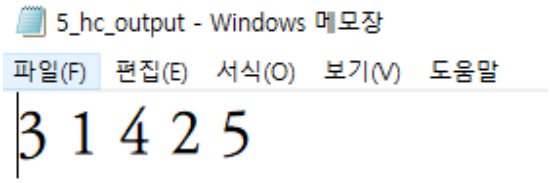

그림 5 N=5 일 때 Hill Climbing 알고리즘 결과

초기 상태는 random 하게 여왕말을 배치하게 되므로, 각 실행마다 결과가 다르게 출력된다

CSP (Constraint Satisfaction Problem)

Constraint Satisfaction Problem (제약 충족 문제)는 특정한 state 에서 변수를 constraint 하는 상황에서 constraint condition 을 만족하는 변수의 pair 을 구하는 문제이다. 위 Local Search 방법인 Hill Climbing 과 다르게 heuristic cost 에 의존하지 않는 특징을 지니고 있다.

제약 충족 문제는 변수(Variable)들의 집합, 변수 정의역(Domain)의 집합, 허용되는 값들을 명시(constraint)하는 제약들의 집합들로 구성되어 있다. 본 N-queens 문제에서 변수는 self.chessboard 의 element, 정의역은 1 부터 N 까지 여왕말의 위치, 제약 조건은 그림 2 와 같이 서로 conflict 하지 않아야 함으로 설정하였다. 다양한 solution 중 DFS 를 이용하여 제약 조건에 걸리면 부모 노드로 후퇴하여 확장을 줄여 나가는(pruning) Backtracking 방법을 이용하였다. 여기서 DFS 함수를 재귀함수로 사용하여 Backtracking 이 되도록 하였다. 해당 알고리즘의 도식은 아래 이미지(그림 6) 참고.

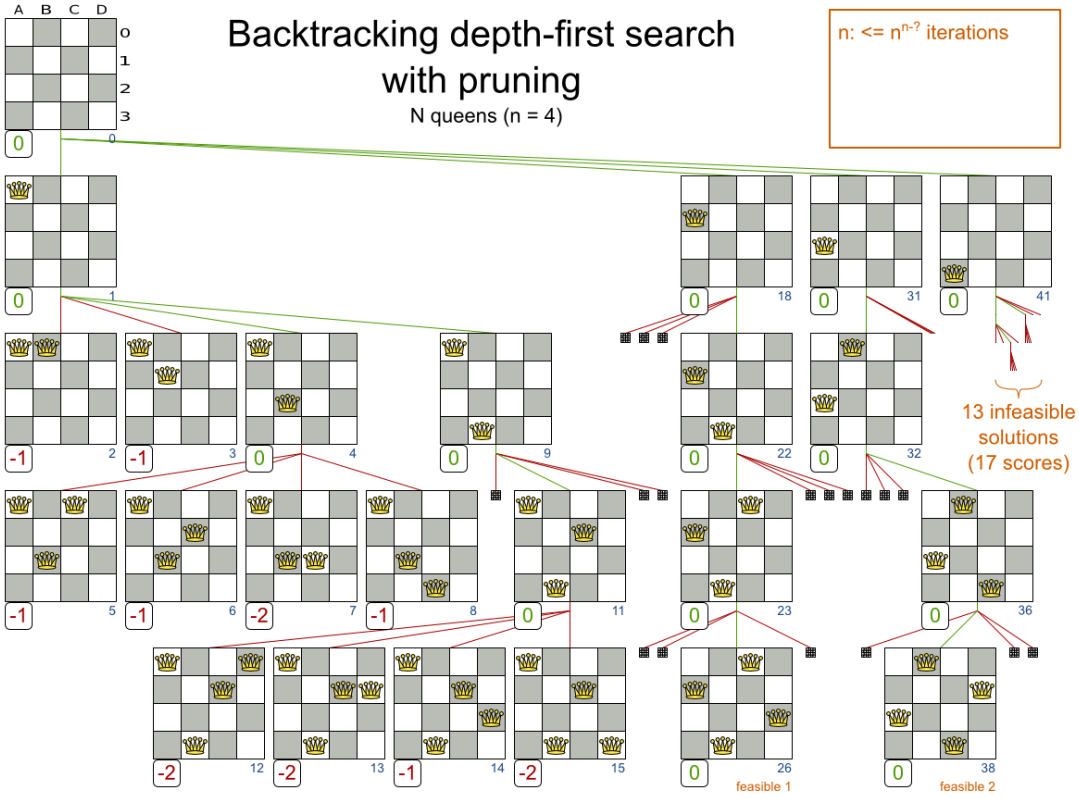

그림 6 Nqueens 문제에서 Constraint Satisfaction Problem 알고리즘

Code Review - 함수 설명

```
def __init__(self, N):
  self.N = int(N)
```

```
    self.solutions=[]
      self.chessboard = [0 for _ in range(N)] #index = xi as column, value = row
```

초기 상태는 위와 동일, Deep copy 문제로 인해 따로 solution 변수를 선언하였다.

```
  def isLegal(self, xi): # 탐색하고자 하는 현 상태
    for xi_idx in range(xi): # column
        if self.chessboard[xi_idx]==self.chessboard[xi]: # 직선일 경우
            return False
        if abs(self.chessboard[xi]- self.chessboard[xi_idx]) == abs(xi - xi_idx): #
대각선일 경우
            return False
      return True
```

탐색하고자 하는 현 상태에서 이전에 배치한 여왕말이 conflict 여부를 확인하는 함수. 앞서 다룬 알고리즘들에서 사용한 함수와 동일하며, 서로 간의 충돌을 확인하며 전도유망(promising)한지 확인한다.

```
  def dfs(self, xi):
    if xi == self.N:
        solution = []
        solution.extend(self.chessboard)
        self.solutions.append([i + 1 for i in solution])
        return True
    for row_pos in range(self.N):
        self.chessboard[xi]=row_pos
        if self.isLegal(xi):
            if self.dfs(xi+1):
                  return True
```

DFS 는 다음 단계의 탐색할 공간을 우선적으로 탐색하므로 **인자는 chessboard 의 인덱스, 열의 위치**로 탐색하게 된다. 만약 열이 끝까지 다다르게 되면 더 이상의 탐색은 불필요하므로 **현재 배치된 chessboard 를 solution 에 얕은 복사를 하여 반환**한다, 그 외에는 현 상태 열에서 행의 위치를 넣음으로써 전도유망한(isLegal:True) 경우 탐색을 이어 나가게 하였다.

```
  def csp(self):
    self.dfs(1)
    # print(self.chessboard)
    if self.solutions:
        return self.solutions
    else:
          pass # return null
```

csp 함수에서는 dfs 함수를 호출하여 **solution 이 존재할 경우 solution 을 반환**하게끔 작성하였다. dfs 함수에서 1 을 인자로 둔 이유는 초기 상태에서 여왕말의 위치를 (1,1) 위치에 고정하기 위함이다. 따라서 여러 번 실행을 해도 같은 결과가 나오게끔 하였다.

Test Result - 실험 결과

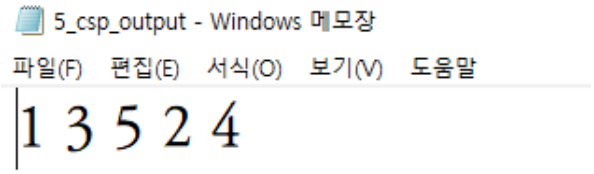

그림 7 N=5 일 때 Constraint Satisfaction Problem 알고리즘 결과

초기 상태는 항상 같으므로 어떤 실행이든 같은 결과를 출력하게 된다.