**인공지능 assignment 1 Report**

2019023436 김현수

1. 함수 설명
2. **main()**: 2019023436\_assignment\_1.py를 실행했을 때 실행되는 메인 함수이다.
   * input.txt 파일로부터 한 줄씩 읽어와서 bfs, hc, csp 함수를 실행시킨다.
3. **bfs(N)**: main으로부터 넘겨받은 N을 가지고 bfs 알고리즘을 이용해 문제를 해결한다. 답이 없는 경우 no solution을 {N}\_bfs\_output.txt 파일에 출력한다.
   * **is\_ok(queen, x, y)** 함수를 호출해 체스판의 (x, y) 좌표에 퀸을 놓아도 될 지 체크하며 자식 노드를 만들어나간다.
   * **write(N, queen)** 함수를 이용하여 solution을 {N}\_bfs\_output.txt 파일에 출력한다.
   * **class chess** 클래스를 만들어서 현재 노드의 level과 놓인 queen 상태를 저장한다.
4. **hc(N)**: main으로부터 넘겨받은 N을 가지고 hill-climbing을 이용해 문제를 해결한다. 답이 없는 경우 no solution을 {N}\_hc\_output.txt 파일에 출력한다. while문에서 local minimum 혹은 global minimum에 도달할 때까지 루프를 돌려 climbing(queen)을 실행한다. climbing 함수로부터 0을 반환 받은 경우 global minimum에 도달한 것이고 -1을 반환 받은 경우 local minimum에 도달한 것이다.
   * **climbing(queen)** 함수에서는 체스판의 모든 칸 (x, y)에 퀸을 놓았을 때 heuristic값을 heuristic함수를 호출하여 계산하고 그 중 가장 값이 작은 칸에 실제로 퀸을 놓는다.
   * **heuristic(queen, x, y)** 함수가 climbing에서 호출되면 체스판의 (x, y)에 퀸을 놓았을 때 서로 잡아먹는 pair의 개수를 계산해서 반환한다.
   * **write(N, queen)** 함수를 이용하여 solution을 {N}\_hc\_output.txt 파일에 출력한다.
5. **csp(N)**: main으로부터 넘겨받은 N을 가지고 forward checking을 이용해 문제를 해결한다. 1부터 N까지 들어있는 리스트 N개가 들어있는 리스트를 생성하여 dfs함수에 넘겨준다. dfs 함수가 False를 반환하면 no solution을 {N}\_csp\_output.txt 파일에 출력한다.
   * **class chess** 클래스를 만들어서 현재 column에 놓을 수 있는 row 리스트들(self.queen)과 row가 확정된 column 리스트(self.confirmed)를 저장한다.
   * **dfs(parent)** 함수에서는 맨 처음에 모든 column에 모든 row가 전부 들어있는 chess 클래스 parent를 받아서 각 column의 row를 확정한 결과를 다시 dfs에 넘겨준다. 넘겨받은 parent.confirmed의 size가 N이라면 write 함수를 호출한다.
   * **constrain(queen, col, row)** 함수를 이용해 col번째 column의 값을 row 하나로 확정하고 이로 인해 영향 받는 나머지 column들의 row들을 제거한다.
   * **write(N, queen)** 함수를 이용하여 solution을 {N}\_csp\_output.txt에 출력한다.
6. 알고리즘 설명
   1. bfs

bfs tree의 노드는 퀸을 하나도 놓지 않은 level 0에서 시작해서 1번부터 N번 column 순서대로 퀸을 놓아 나간다. 이때 앞선 column에 놓인 퀸으로 인해 잡아 먹히게 되는 자리에는 새 퀸을 놓을 수 없다. level k인 노드는 1번부터 k번 column까지 퀸이 놓여있는 상태이다. 이러한 현재 상태와 현재 level은 class chess로 만든 객체에 저장되어 움직인다.

1번째 column의 1~N번째 row에 퀸을 놓은 체스 board들을 차례로 queue에 넣는다. queue에서 board하나를 pop하여 2번째 column의 1~N번째 row 중 1번째 column에 이미 놓여있는 퀸에게 영향을 받지 않는 row만을 골라 퀸을 놓은 board를 차례로 queue에 넣는다. 이때 지금 퀸을 놓으려는 (row, col)이 이미 놓여있는 퀸들에게 잡아 먹히는지를 **is\_ok**함수를 이용하여 검사한다. 이 과정을 queue가 빌 때까지 반복한다.

**is\_ok**함수에서는 퀸들이 서로 같은 row에 놓여있거나 (row끼리의 거리 차)/(column끼리의 거리 차) 가 1인 경우가 존재한다면 퀸들끼리 서로 잡아먹는 것 이므로 False를 반환하고 존재하지 않는다면 True를 반환한다.

사실 queue에 새 board들을 넣기 전에 이 board의 level이 N에 도달했는지 검사한다. 도달했다면 이는 solution이므로 **write**함수에 N과 함께 넘겨 파일에 출력한다. 도달하지 못한 board라면 queue에 넣는다.

queue가 전부 빌 때까지 solution을 찾지 못했다면 no solution을 출력한다.

* 1. hill-climbing

각 column에 퀸 하나씩 random 모듈을 이용해서 놓는다.

**climbing** 함수에서 현재 상태에서 퀸 하나를 같은 column의 다른 row로 옮겼을 때 서로 잡아먹는 pair의 개수를 **heuristic**함수를 이용해 구한다. 가장 heuristic 값이 작은 칸으로 퀸을 실제로 옮긴다. 그리고 그 heuristic 값을 반환한다. 이때 옮기기 전 상태의 heuristic 값과 옮기고 난 후의 heuristic 값이 차이가 없다면 더 이상 나아지지 못하는 local minimum 상태에 도달한 것이므로 -1을 반환한다.

**hc**함수에서는 while으로 무한루프를 돌리며 **climbing** 함수가 heuristic값으로 0 또는 -1을 반환할 때까지 **climbing** 함수를 호출한다. 0을 반환 받았다면 **write**함수를 호출하여 파일에 solution을 출력한다. -1을 반환 받았다면 초기 상태가 애초에 global minimum에 도달할 수 없는 상태였던 것이므로 다시 random 모듈을 이용하여 새 초기 상태를 만들고 이 과정을 반복한다.

* 1. csp

Forward checking 방식을 이용한다. 모든 column에 모든 row값을 다 넣은 체스판을 만든다. 이를 **dfs**함수에 넘겨주고 False를 반환 받는다면 no solution을 출력한다.

**dfs** 함수에서는 각 column별 row를 하나씩 확정해 나가는 과정을 재귀적으로 처리한다. 인자로 넘겨받은 parent 상태의 column들의 길이를 전부 구하고 이미 confirm되지 않은 column중 row 개수가 가장 적게 남은 column들을 구한다. (most constrained variable)

그런 column이 여러 개라면 각 column들의 row를 확정했을 때 가장 다른 column들의 row set이 많이 제거되는 column을 찾는다. (most constraining variable) 이는 **constrain**함수에 board와 확정해볼 col, row를 넘겨줌으로써 찾을 수 있다. **constrain**함수가 호출되면 (col, row)에 배치된 퀸이 잡아먹을 수 있는 다른 column들의 row들을 전부 제거하게 된다. 그리고 그렇게 제거가 끝난 board의 size, 즉 남은 row의 개수들의 합이 가장 작을수록 constraining 정도가 크다고 할 수 있다.

그렇게 고른 column의 row를 확정할 때에는 **constrain**함수를 이용해 확정한 board의 size가 가장 큰, 즉 다른 column들에 가하는 constrain정도가 가장 작은 row를 고른다. (least constraining value) 이 board를 재귀적으로 **dfs**에 넘겨준다. 이때 empty set이 있는 column이 하나라도 있다면 넘기지 않고 그 다음으로 작은 row를 골라 넘겨준다.

위의 모든 heuristic은 heapq 모듈을 이용해 최대 혹은 최솟값을 찾고 backtracking으로 되돌아오면 이를 이용해 그 다음 최대 혹은 최솟값을 찾는다.

heap이 다 빌 때까지 solution을 찾지 못했다면 False를 return한다. 이는 csp함수에서 받아서 no solution을 출력한다. solution을 찾았다면 **write**함수를 호출해 출력하고 True를 return한다.

1. 실험 결과

   