HW03

수리과학부 2019-16022 박채연

1. Describe how your program works and how you made your algorithm as efficient as possible.

Iterative backtracking 알고리즘을 수행하기 위해 스택 자료구조를 활용하였는데 C언어에서는 스택과 관련된 STL을 제공하지 않기 때문에 C++로 구현하였다. 'main.cc' 파일에 main 프로그램을 구현하고 'g++ -std=gnu++17 -O2 main.cc -o main' 명령어를 통해 컴파일하면 'main'이라는 실행파일이 생성된다. 이는 실행모드, input 파일 경로, output 파일 경로를 순서대로 인자로 입력받는다. 실행 모드에는 총 두 가지가 있는데, 1은 iterative backtracking algorithm을 수행하고, 2는 recursive backtracking algorithm을 수행한다.

```
fscanf(fpin, "%d %d", &n, &holenum);
makeMap(fpin, n, holenum);
cnt = 0;
switch(mode){
    case 1:
        iterative_N(n);
        break;
    case 2:
        recursive_N(0, 0, 0, n);
        break;
}

fprintf(fpout, "%d\n", cnt);
fclose(fpin);
fclose(fpout);
return 0;
```

위의 이미지는 main 함수 내의 핵심 로직이다. input 파일로부터 n 값과 hole의 개수를 입력받고 makeMap 함수를 통해 nxn map에서 hole의 위치에 'H'를, 나머지 위치에 '*'를 저장한다. 만약 입력받은 실행 모드가 1이면 iterative_N 함수를 호출하여 iterative backtracking 알고리즘을 실행시키고 2이면 recursive_N 함수를 호출하여 recursive backtracking 알고리즘을 실행시킨다. 마지막으로 알고리즘 실행 결과 알게 된 Queen을 놓을 수 있는 가짓수를 output 파일에 기록한다.

Recursive backtracking 알고리즘을 먼저 구현한 후 정상적으로 동작하는 것을 확인하고 이를 Iterative 버전으로 변환하는 과정을 거쳤다. Iterative backtracking 알고리즘에서 스택을 사용한다는 점을 제외하면 완전히 동일한 함수를 호출하는 등 전체적인 로직이 거의 비슷하다. 따라서 recursive backtracking 알고리즘을 기준으로 전체적인 프로그램의 로직을 설명하면 아래와 같다.

```
void recursive_N(int i, int k, int cur, int n){
```

recursive backtracking 알고리즘을 수행하는 함수의 원형은 위의 이미지와 같다. 내부에서 for 반복문을 활용하여 i행 k열부터 i행 n-1열까지의 위치 가운데 Queen을 놓을 수 있는 위치가 있다면 재귀 호출을 통해 다음 Queen을 놓을 수 있는 위치를 찾고자 하였다. 또한, 중복되는 경우를 줄이기 위해 k=0일 때만 recursive_N(i+1, 0, cur, n)을 호출하여 i행에 Queen을 놓지 않고 i+1행에 Queen을 놓을 수 있는지 여부를 확인하였다. 만약 모든 행을 다 지나오며(i = n) Queen을 배치하

였으나 배치한 Queen의 개수가 n개가 아니라면 return 하고 배치한 Queen의 개수가 n개라면 전역 변수인 cnt를 1만큼 증가시킨다.

backtracking 알고리즘을 보다 효율적으로 구현하기 위해 다양한 방법을 시도해보았는데, 이는 다음과 같다. 우선, (i, j) 위치에 Queen을 놓아도 되는지 여부를 확인할 때, 위와 왼쪽 대각선, 오른쪽 대각선을 매번 일일이 확인하면서 다른 Queen이 있는지 여부를 확인한다면 시간을 많이 잡아먹게 된다. (i, j) 를 지나는 세로선에 있는 모든 위치 (x, y)는 y 값이 동일하다는 특징이 있다. 또한, (i, j)를 지나는 왼쪽 대각선에 있는 모든 위치 (x, y)는 x – y + n이 모두 같은 값을 가진다는 특징이 있다. 예를 들어, n = 5일 때 (3, 3)을 지나는 왼쪽 대각선에 놓인 모든 위치를 나열하면 (0,0), (1, 1), (2, 2), (3, 3), (4, 4)이다. 이때, x – y + n이 모두 5로 동일한 값을 가진다. 비슷하게, (i, j)를 지나는 오른쪽 대각선에 있는 모든 위치 (x, y)는 x + y가 모두 같은 값을 가진다. 따라서 배열 v, l, r을 사용하여 (i, j) 위치에 Queen을 놓기 전 v[j], l[i-j+n], r[i+j]가 1인지(=세로선 혹은 왼쪽 대각선 혹은 오른쪽 대각선에 Queen이 이미 놓여있는지)를 확인하고, 만약 놓여있지 않다면 v[j], l[i-j+n], r[i+j]를 모두 1로 지정하였다. 한편, (i, j)에 hole이 있다면 v[j], l[i-j+n], r[i+j]에 'H'를 기록하였기 때문에 만약 (x, y) 위치에 Queen을 놓을 수 있는지 여부를 확인할 때, v[y], l[x-y+n], r[x+y] 가 'H'라면 세로선 혹은 왼쪽 대각선 혹은 오른쪽 대각선에 (x, y)위치를 공격하는 Queen이 있는지 혹은 hole이 모든 공격을 막아주는지 직접 조사해야 한다는 단점이 있다.

아래의 표가 5x5 map이라고 하자. 이때, '*'는 빈 공간을 의미하고 'H'는 hole을 의미한다(0≤i<5, 0≤j<5).

*	*	H	*	*
*	*	*	*	*
*	*	*	H	*
*	*	*	*	*
*	H	*	*	*

만약 (1, 0) 위치에 Queen을 놓는다면 1행에는 hole이 없기 때문에 1행의 나머지 열에는 queen이 놓을 수 없다. 따라서 바로 2행으로 넘어가서 Queen을 놓을 수 있는 위치를 찾으면 된다. 그러나 만약 (0, 0)위치에 Queen을 놓는다면 0행에는 hole이 있기 때문에 0행 나머지 열에 Queen을 놓을 수도 있다. 따라서 배열 h를 사용하여 i행에 hole이 있는지 여부를 처음에 hole의 좌표를 입력받는 과정에서 기록해두고, h[i]=1이라면 (= i행에 hole이 있다면) for 반복문을 이용하여 j+1열부터 n-1열까지 hole이 있는지 여부를 조사하였다. 만약 i행 x열에 hole이 있다면 (i, x+1) 위치에 Queen을 놓을 수 있는지 여부를 조사하는 식으로 알고리즘을 작성하였다.

또한, Iterative backtracking 알고리즘을 구현할 때 Stack 자료구조를 사용하였는데, 원소를 insert할 때 push()대신 속도가 더 높다고 알려진 emplace()를 사용하였다.

지금까지 설명한 방법을 적용하여 Autograder를 통해 채점해보았을 때 Correctness : 20/20, Optimization: 10/20을 기록하였다. 그러나 h[i] = 1일 때 매번 for 반복문을 통해 (i, j) 위치의 오른

쪽에 hole이 있는지 여부를 매번 탐색하는 것은 비효율적이라고 생각되었고, 해당 문제를 해결하기 위해 holeh 배열을 도입하였다. holeh 배열은 2차원 배열인데, holeh[i][0]는 i행에 놓인 hole의 개수를 의미하고, holeh[i][1]부터 holeh[i][3]까지는 i행에 놓인 hole의 열번호를 기록한다. 따라서 아래의 코드를 통해 holeh[i][1] 부터 holeh[i][holeh[i][0]]만을 조사하여 만약 (i, j)보다 오른쪽에 위치한 hole이 있다면 해당 hole의 바로 오른쪽에 Queen을 놓을 수 있는지 여부를 조사하였다. 앞선 방법은 worstcase 시간복잡도가 O(n)인 반면 이 방법은 worstcase 시간복잡도가 O(1)이다. 해당 방법을 적용하여 다시 Autograder로 채점을 진행해본 결과 Correctness : 20/20, Optimization: 12/20으로 약간 성능이 개선된 모습을 확인할 수 있었다.

```
if (h[i]){
    // find the nearest hole on the right side of (i, j)
    int num = holeh[i][0];
    for (int idx = 1; idx <= num; idx++){
        if (holeh[i][idx] <= j)
            continue;
        //recursive call
        recursive_N(i, holeh[i][idx]+1, cur+1, n);
        break;
    }
}
```

recursive_N(i, j, cur, n)은 (i, j)위치에 Queen을 놓을 수 있는지 여부를 조사하는데, cur은 지금까지 놓은 Queen의 개수, 놓아야 하는 총 Queen의 개수를 의미한다. 원래 hole이 없다면 하나의 행마다 1개의 Queen만 둘 수 있는데, hole이 있다면 해당 행에는 최대 hole의 개수만큼 더 Queen을 둘 수 있는 가능성이 있다. 이는 최대 hole개의 행에는 Queen이 놓이지 않을 수 있다는 것을 의미한다. 그러나 만약 i - cur 값이 hole의 개수보다 크다면 이는 남은 행에 아무리 Queen을 놓아도 n 개의 Queen을 모두 둘 수 없다는 것을 의미하기 때문에 return 하여 시간복잡도를 줄이고자 하였다. 해당 아이디어를 Iterative backtracking 알고리즘과 Reursive backtracking 알고리즘 모두에 적용하여 Autograder를 돌려보았으나 예상 외로 오히려 성능이 떨어져 원상복구하였다.

현재까지 개선된 알고리즘에서 checkL, checkR, checkV 함수의 시간복잡도를 줄이면 더욱 더 성능 향상이 이루어질 수 있을 것이라고 예상한다. 현재는 (i, j) 위치에 Queen을 놓을 때, 만약 (i, j)를 지나는 세로선이나 왼쪽 대각선, 오른쪽 대각선에 hole이 위치한다면 직접 해당 선을 따라가면서 (i, j) 위치를 공격하는 Queen이 있는지 혹은 hole이 해당 공격을 막아주는지를 for 반복문이나 while문을 이용하여 조사한다. 이는 굉장히 비효율적이다. C++ STL 가운데 우선순위 큐를 이용하여 해당 부분을 최적화할 수 있을 것이라고 예상하였다. 예를 들어, 각 열에 대하여 int를 다루는, 내림차순 정렬하는 우선순위 큐를 만들고 hole의 위치가 (i, j)라면 j열에 대한 우선순위 큐에 i를 insert 한다. 또한, (x, y) 위치에 Queen을 놓을 때에도 y열에 대한 우선순위 큐에 x를 insert한다. (2, 5) 위치에 Queen을 놓고자 할 때 5번째 열에 대한 우선순위 큐가 비어있다면 이는 5번째 열에 Queen과 hole 모두 존재하지 않는다는 의미이므로 해당 위치에 Queen을 놓을 수 있다. 만약 비어있지 않다면 해당 우선순위 큐의 첫번째 원소는 5번째 열에 놓여있는, (2, 5)에 가장 가까이 위치한 Queen 또는 hole의 행번호를 의미한다. 이를 x라고 했을 때, map[x][5]가 'H'라면

Queen을 놓을 수 있고, 'Q'이면 Queen을 놓을 수 없다. 해당 아이디어를 가지고 실제로 구현을 시도해보았지만 정답이 다르게 출력되었고, 디버깅을 하기에는 현실적으로 여러 시험 등으로 인해 시간이 부족하다고 생각하여 개선하지 못했다.

2. Compare the running time of your iterative backtracking algorithm and that of your recursive backtracking algorithm, and discuss the results.

n 값을 4부터 12까지 1씩 증가시키고 hole의 개수를 0개부터 3개까지 1개씩 증가시키며 input 파일을 임의로 생성하고 Iterative backtracking 알고리즘과 Recursive backtracking 알고리즘의 수행시간을 조사하였다.

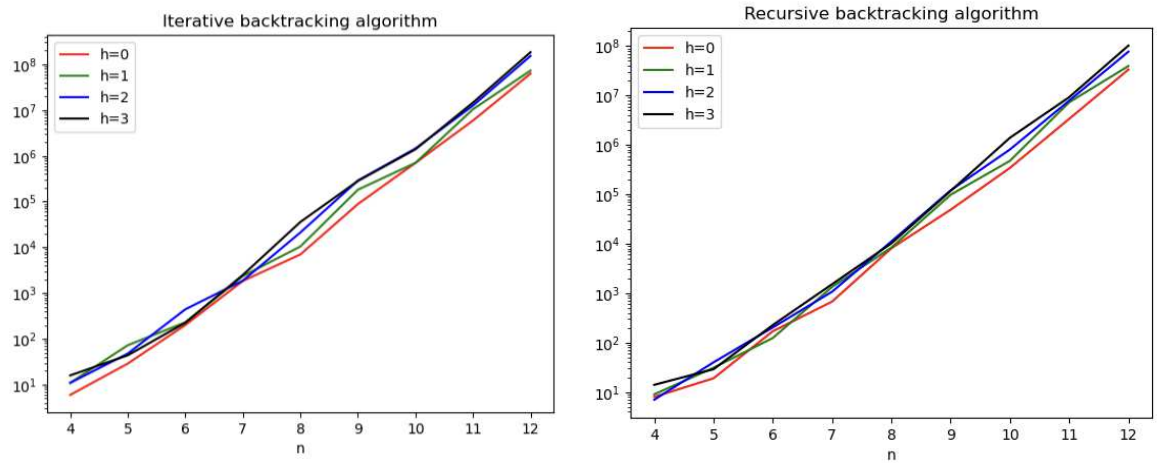
Iterative backtracking 알고리즘의 수행시간을 정리하면 아래의 표와 같다. 단위는 밀리초이다.

	n = 4	**n = 5**	**n = 6**	**n = 7**	**n = 8**	**n = 9**	**n = 10**	**n = 11**	**n = 12**
H = 0	6	29	202	1855	7027	88744	704622	5929461	62786567
H = 1	11	73	231	2367	10492	182727	704089	10478589	73138379
H = 2	11	48	445	1880	21216	290187	1453227	12594392	152796787
H = 3	16	44	226	2504	36000	283518	1394167	14645608	184588668

recursive backtracking 알고리즘의 수행시간을 정리하면 아래의 표와 같다. 단위는 밀리초이다.

	n = 4	**n = 5**	**n = 6**	**n = 7**	**n = 8**	**n = 9**	**n = 10**	**n = 11**	**n = 12**
H = 0	8	19	170	676	8108	48818	342383	3360245	33511120
H = 1	9	31	124	1341	8374	97905	476223	7268069	39434849
H = 2	7	40	203	1072	11140	120442	806618	7853163	77006352
H = 3	14	29	227	1520	10172	117046	1393478	9230221	102155848

파이썬의 matplotlib 라이브러리를 이용하여 Iterative backtracking 알고리즘과 Recursive backtracking 알고리즘의 실행시간을 각각 그래프로 아래와 같이 나타내보았다. 이때, yscale을 'log'로 지정하였다.



Iterative bakctracking 알고리즘과 Recursive backtracking 알고리즘 모두 n 값이 커질수록 실행시간이 급격하게 증가하는 것을 확인할 수 있었다. 또한, n 값이 커질수록 hole 개수 별 수행시간의 차이가 뚜렷해졌는데, Iterative backtracing 알고리즘과 Recursive backtracking 알고리즘 모두 hole의 개수가 많아질수록 수행시간이 증가하였다. 또한, 예상과는 다르게 Iterative 알고리즘보다 Recursive 알고리즘의 수행시간이 일반적으로 더 적게 걸렸으며 n 값이 커질수록 그 차이는 뚜렷했다. Recursive 알고리즘과는 다르게 Iterative 알고리즘에서는 Stack 자료구조를 사용하는데, insert(=emplace)하고 pop하는 등의 오버헤드가 전체 수행시간에 영향을 미쳤을 가능성이 있다.

3. Write down the environment you run your program and how to run your program.

Window OS에서 VS code를 이용하여 프로그램을 작성하였고, 도커 환경 내에서 hole의 개수와 n 값에 따른 iterative backtracking 알고리즘과 recurive backtracking 알고리즘의 수행 시간을 비교하기 위해 테스트를 진행하였다. Docker Desktop에서 'docker pull yehyun/cta:assignment' 명령어를 통해 도커 이미지를 불러오고 'docker run -it yehyun/cta:assignment bash' 명령어를 통해 실행하였다.

과제 공지사항에서 명시된 바와 같이 'g++ -std=gnu++17 -O2 main.cc -o main'명령어로 main.cc 파일을 컴파일 하고, './main x 1.in 1.out'과 같은 명령어로 프로그램을 실행하였다. 이때, 'x' 자리에는 1이나 2 가운데 하나의 값이 올 수 있는데, 1은 iterative backtracking 알고리즘을 실행시키고, 2는 recursive backtracking 알고리즘을 실행시킨다. '1.in' 파일로부터 n 값의 크기와 hole의 개수, hole의 위치를 순서대로 입력받고 x 자리에 오는 값에 따라 적절한 알고리즘을 실행시켜 Queen이 놓일 수 있는 총 개수를 구한 후 '1.out' 파일에 해당 값을 기록하게 된다.