KWANGWOON UNIVERSITY

알 고 리 즘

Homework #3

(C++ / Visual Studio 2015)

담당 교수님 : 김용혁 교수님

제 출 날 짜 : 2018. 06. 06.

학 과 : 컴퓨터소프트웨어학과

학 번 : 2014707040

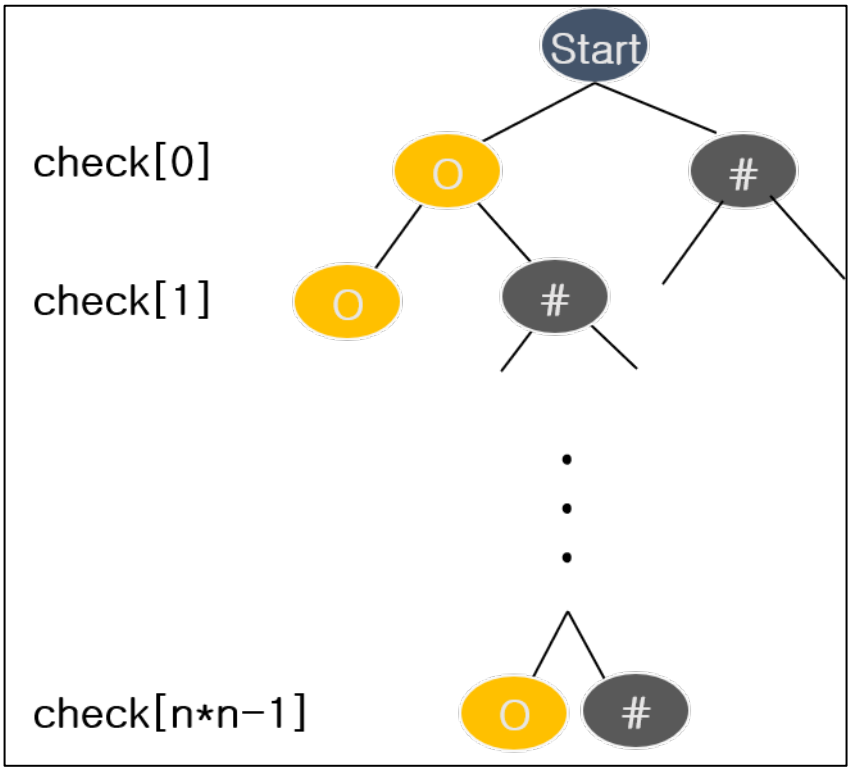
이 름 : 유 진 혁

1. 알고리즘 소개

백트래킹 알고리즘을 이용하여 구현하였다. 과제 설명에선 모든 불이 꺼져 있는 상태를 초기 상태로 지정하였는데, 편의상 반대로, 입력 받은 전구의 상태를 초기 상태로 하고 모든 불을 꺼지게 하는 솔루션을 찾았다. 누르는 스위치 위치는 똑같고, 별도의 배열을 만들어 모든 칸을 끈 상태로 만들어주는 작업이 필요 없기 때문이다.

문제를 간단히 정리하자면, n X n 2차원 배열에서 각 스위치들은 눌리거나 눌리지 않은 상태 두 상태 중 하나를 가지고, 각 스위치들의 상태를 조합하여 모든 불이 꺼진 상태로 만드는 것이다. 백트래킹 알고리즘을 이용하려면 자식 노드를 탐색할 조건, 즉 promising 여부를 판단할 기준이 필요한데, 현재 누르려는 스위치의 주변 스위치 하나의 상태를 가지고 판단하기로 했다. 여기서는 누르려는 스위치의 위쪽 스위치 상태를 참고하여 그 스위치가 꺼져 있으면 promising, 켜져 있으면 non-promising으로 하였다. 왜냐하면 이 알고리즘은 2차원 배열의 왼쪽 위부터 오른쪽으로, 그리고 아래로 순차적으로 탐색하며 이동하는데 위쪽 스위치를 켜 두고 다음 자식 노드로 이동하면 그 스위치를 다시 끌 방법이 없기 때문이다. 따라서 위쪽 스위치가 꺼져 있는 상태일 때만 다음 자식 노드를 봐야한다.

위에서 설명한 것처럼 각 스위치들의 상태를 아래와 같은 트리구조로 나타낼 수 있다. 누를 스위치 위치정보를 1차원화하여 check라는 배열에 저장한다. (0, 0) 위치의 스위치는 check[0], (1, 0) 위치의 스위치는 check[n]에 정보가 저장된다. 따라서 트리의 깊이는 n X n이 된다.



각 노드가 promising인지 판단하여 DFS를 하고 이는 재귀적으로 구현하여 non-promising일 경우 백트래킹 하게 하였다.

2. 코드 설명

```
int n;  // 배열 크기
bool** input;   // 입력 배열
bool** output;  // 출력 배열 (누를 스위치)
bool* check;    // 누르는 스위치 수 최소화를 위해
bool isSolved = false;  // 솔루션 유무
int minCount;   // 누르는 스위치 수 최솟값
```

전역변수로 선언한 변수 목록이다. 각 변수의 역할은 주석문을 통해 알 수 있다. check 배열은 누르는 스위치 수가 최소일 때에 출력하기 위해 output 으로 넣기 전 중간 단계의 배열이라 생각하면 된다. 다음 노드로의 순차적 탐색을 편리하게 하기 위해 1 차원 배열로 선언하였다.

```
int main()
{
    char ch;

    /*** 입력 및 할당 ***/
    cin >> n;
    input = new bool*[n];
    output = new bool*[n];
    minCount = n*n;
    for (int i = 0; i < n; i++)
    {
        input[i] = new bool[n];
        output[i] = new bool[n];
    }
    check = new bool[n * n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            cin >> ch;
            if (ch == 'O')
                input[i][j] = true;
            else if (ch == '#')
                input[i][j] = false;
        }

    TurnOff(-1);
```

main 함수의 초반부이다. 사용자로부터 배열 크기를 입력 받고 그 값을 이용하여 배열들을 동적 할당한다. minCount 는 누르는 스위치 수의 최솟값이 들어가는 변수인데, 초기값으로 최댓값인 배열의 원소의 수를 넣어준다. 코드를 간단하게 짜기 위해 사용할 배열들을 bool 형 배열로 만들었다. 그렇기 때문에 사용자가 O 를 입력했으면 true 값을, #을 입력했으면 false 값을 input 배열에 넣어준다. 즉, true 가 켜진 상태, false 가 꺼진 상태이다. 그리고 입력된 배열의 상태를 불이 다 꺼진 상태로 바꿔주는 TurnOff 함수를 호출한다. TurnOff 함수의 인자로는 check 배열의 인덱스가 들어가는데 루트 노드부터 시작하기 위해 -1 을 인자로 넘겨준다.

```
/*** 모든 칸의 불을 끄는 함수 ***/
void TurnOff(int index)
{
    int count = 0;

    if (Promising(index))
    {
        if (Solve() == false)
        {
            if (index + 1 < n * n)
            {
                // 스위치 눌렀을 때
                check[index + 1] = true;
                Switch(index + 1);
                TurnOff(index + 1);

                // 스위치 누르지 않았을 때
                check[index + 1] = false;
                Switch(index + 1);
                TurnOff(index + 1);
            }
        }
```

```
/*** 윗 칸의 불이 꺼져있으면 Promising ***/
bool Promising(int index)
{
    if (index == -1)    // 루트 노드
        return true;

    int i = index / n;  // 행
    int j = index % n;  // 열

    if (i == 0 || input[i - 1][j] == false)
        return true;
    else
        return false;
}
```

TurnOff 함수와 Promising 함수의 정의 부분이다. 우선 TurnOff 함수에 대해서 보면, 눌린 스위치의 개수를 담는 변수 count 를 선언한다. 이와 관련된 작업은 TurnOff 함수 뒷부분에서 솔루션을 찾았을 때 수행한다. 그 다음, 인자로 받은 index 가 promising 인지 확인한다. promising 여부는 오른쪽 Promising 함수에서 볼 수 있듯, index 를 이용하여 행과 열을 구하고 위쪽 스위치의 상태를 참고하여 꺼져 있는 상태면 true 를 반환한다. 현재 보고 있는 스위치가 input 의 첫 번째 행에 있는 경우에도 true 를 반환한다. 또, 알고리즘을 처음 시작할 때, 즉, 루트 노드일 땐 무조건 promising 하도록 true 를 반환한다. 이렇게 promising 여부를 판단하여 promising 일 경우 input 배열의 현재 상태를 보고 모든 불이 꺼져 있는 상태인지 Solve 함수로 확인한다. Solve 함수의 정의는 아래와 같다. promising 한데 모든 불이 꺼져 있는 상태가 아니라면 다음 스위치를 확인하여 스위치를 누르는 작업을 해준다. 먼저 다음 스위치를 눌렀을 때 TurnOff 함수를 재귀 호출하여 promising 여부를 판단하고, 다시 원래 위치로 돌아오면 눌렀던 스위치를 다시 눌러 스위치를 누르지 않았을 때 상태를 검사한다. 스위치를 눌러주는 함수 Switch 의 정의는 아래와 같다.

```
/*** 모든 칸의 불이 꺼져 있으면 문제 해결 ***/
bool Solve()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
        {
            if (input[i][j] == true)
                return false;
        }
    return true;
}
```

```
/*** 스위치 켜기/끄기 ***/
void Switch(int index)
{
    int i = index / n;  // 행
    int j = index % n;  // 열

    input[i][j] = !input[i][j];
    if (i - 1 >= 0)
        input[i - 1][j] = !input[i - 1][j];
    if (i + 1 < n)
        input[i + 1][j] = !input[i + 1][j];
    if (j - 1 >= 0)
        input[i][j - 1] = !input[i][j - 1];
    if (j + 1 < n)
        input[i][j + 1] = !input[i][j + 1];
}
```

Solve 함수는 input 배열의 모든 원소를 확인하여 불이 하나라도 켜져 있으면 false 를 반환한다. Switch 함수는 index 로 행과 열을 구한 뒤, 이를 이용하여 input 에서의 현재 스위치와 이웃 스위치 상태를 바꿔준다.

```
else
{
    isSolved = true;
    for (int i = 0; i < n * n; i++)
        if (check[i] == true)
            count++;
    if (minCount > count)
    {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                output[i][j] = check[i*n + j];
        minCount = count;
    }
}
```

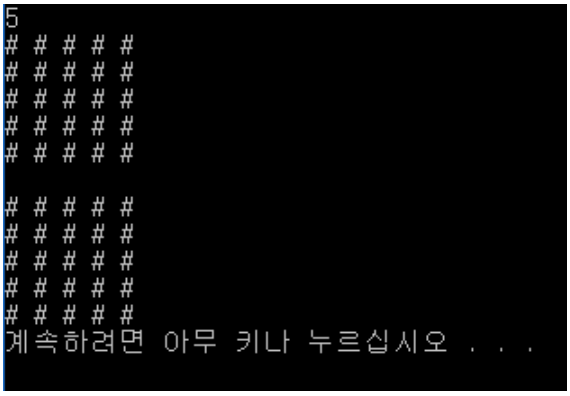
TurnOff 함수의 뒷부분이다. 모든 불을 다 끄는 솔루션을 찾으면 솔루션이 존재한다고 변수에 표시하고 누른 스위치의 개수를 센다. 이 개수가 현재까지의 누른 스위치 개수 최솟값보다 작다면 누른 스위치 정보를 output 에 저장하고 최솟값을 바꿔준다.

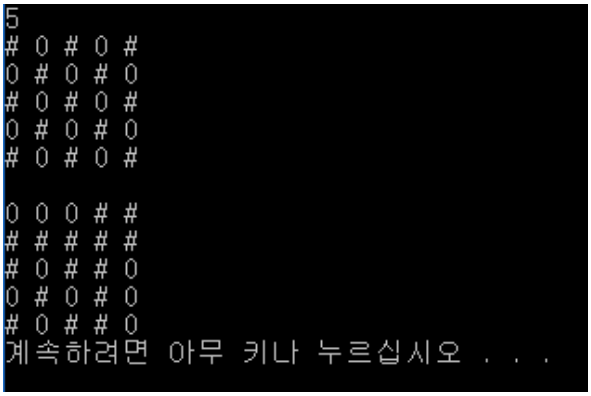
```
/*** 출력 ***/
if (isSolved)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (output[i][j] == true)
                cout << "0 ";
            else
                cout << "# ";
        }
        cout << endl;
    }
}
else
    cout << "no solution.₩n";
```

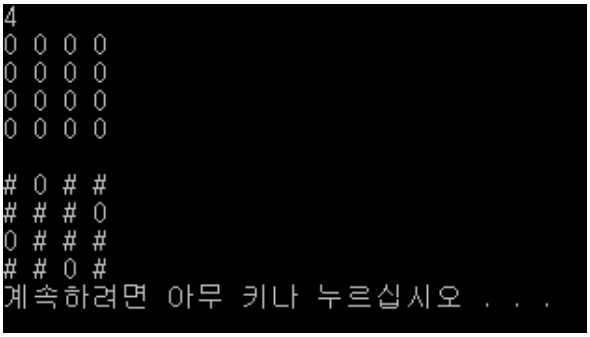
```
/*** 메모리 해제 ***/
for (int i = 0; i < n; i++)
{
    delete[] input[i];
    delete[] output[i];
}

delete[] input;
delete[] output;
delete[] check;

return 0;
```

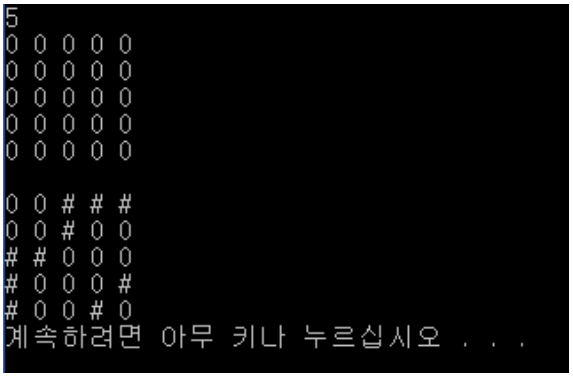
main 함수의 뒷부분이다. TurnOff 함수 작업이 끝나고 솔루션이 존재하면 저장된 output 값을 O 와 #으로 변경하여 화면에 출력해주고, 솔루션이 존재하지 않는다면 솔루션이 없다고 출력해준다. 마지막으로 동적 할당한 메모리를 해제해준다

3. 실행 테스트









누른 스위치의 개수가 최소인 솔루션이 출력된다.

4. 고찰

백트래킹 알고리즘을 사용하기 위해선 기준을 정하여 promising과 non-promising을 결정해야 하는데 이 기준을 생각해내는 것이 쉽지 않았다. N-queen 문제의 경우, 두 퀸이 같은 행과 열, 같은 대각선 상에 있으면 안된다는 명확한 non-promising 조건이 있는 반면, 이 문제는 그런 조건이 제시되어 있지 않기 때문이다. 같이 첨부된 엑셀 파일로 여러 경우를 따져보다가 모든 불을 켜기 위해선 위쪽 스위치의 상태를 확인해서 스위치를 누르면 된다는 교수님의 말씀이 떠올라 이를 활용해보기로 했다. 상태 트리를 어떻게 짤지도 고민을 많이 하였다. DFS로 노드를 접근하는 만큼 트리를 잘 짜야 그 만큼 더 좋은 성능이 나오기 때문이다. 그 결과, 한 스위치의 두 가지 상태에서 그 다음 스위치 상태로 뻗어 나가는 깊이 n X n의 트리 구조 외엔 생각이 안 나지 않았다. 이를 토대로 구현하면서도 백트래킹 알고리즘은 재귀 호출이 자주 발생하여 과정을 생각하고 디버깅을 하기가 매우 까다로웠다. 현재 이 알고리즘은 본인 PC 기준 n이 16일 때까진 비교적 빠른 속도로 결과가 바로 출력되지만 n이 이 값을 초과하게 되면 시간이 꽤 걸리게 된다. n이 20일 땐 1분 30초가 걸린다. 반면 8초가 걸리는 알고리즘을 구현했다는 다른 학생이 있는 것을 보면, 백트래킹으로 구현하였음에도 분명 더 빠른 알고리즘이 존재하는 것으로 보인다. 아마 그래프 구조와 선형식을 이용한 알고리즘이거나 내 알고리즘의 트리구조보다 깊이가 깊지 않은 트리구조를

만들어 구현했을 것이라 생각된다. 그래도 이 알고리즘도 promising 조건에 따라 (위쪽 스위치의 상태를 보고) 탐색하지 않는 노드가 존재하므로 시간 복잡도는 $O(n^2 2^n)$가 될 것으로 생각한다.