컴퓨터공학실험1　8주차 결과보고서

전공: 컴퓨터공학과　　학년: 2　　학번: 20191559　　이름: 강상원

1. 실습 시간에 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술하시오. 완성한 알고리즘(추가 구현하게 되는 효율성을 고려한 tree도 포함)의 시간 및 공간 복잡도를 보이시오.

```
typedef struct _RecNode* NodePTR;
typedef struct _RecNode {
    int lv;
    //int accumulatedScore; -> modified_recommend 로 인해 필요 없음
    char recField[HEIGHT][WIDTH];
    //NodePTR* child; -> modified_recommend 로 인해 필요 없음
} RecNode;
```

추천 트리는 위의 구조체를 하나의 노드로 한 트리이다. 트리의 레벨은 tetris.c의 depth_of_travel에 의해 결정되며, 기본 3으로 설정되어 있다. 각 블록의 종류마다 블럭의 회전 방향, x축 위치에 따라 자식 노드의 경우가 나뉜다. 자식 노드들의 경우의 수는 nextBlock[i]에 저장된 블럭의 종류에 따라 결정된다. 기존 실습의 자료구조에서는 자식 노드를 회전 방향(4)*x축 경우의 수 만큼 생성하고 비교하였지만, 8주차 숙제에서의 modified_recommend 구현 때는 각 블럭마다 가능한 회전의 개수를 미리 계산해두어 이를 활용한다.

```
int rotPossible[7] = {2, 4, 4, 4, 1, 2, 2}; //블럭의 종류마다 가능한 회전의 수
```

추가로, 트리의 노드를 나타내는 RecNode 구조체에서 accumulatedScore와 child는 생략할 수 있다. 이는 modified_recommend 함수에 accumulatedScore 단 한 개의 변수를 설정하여 연산을 수행할 수 있고, 재귀문을 통해 연결할 수 있기 때문이다.

```
if (child->lv < depth_of_travel) accumulatedScore += modified_recommend(child);
```

modified_recommend 소스 코드를 기준으로 추천 알고리즘에 대해 설명하고자 한다.

```
int modified_recommend(NodePTR root) {
    RecNode node;
    NodePTR child = &node;
    int Xpos, Ypos, rot;
    int accumulatedScore; //최대 점수
    int result = -1; //결괏값 음수: 오류 방지


    int rotPossible[7] = {2, 4, 4, 4, 1, 2, 2}; //블럭의 종류마다 가능한 회전의 수


    //for (rot = 0; rot < 4; rot++) {
    for (rot = 0; rot < rotPossible[nextBlock[root->lv]]; rot++) {
        for (Xpos = -2; Xpos < 13; Xpos++) {
            if (!CheckToMove(root->recField, nextBlock[root->lv], rot, 0,
Xpos)) continue;
            for (int i = 0; i < HEIGHT; i++)
                for (int j = 0; j < WIDTH; j++)
```

```
                    child->recField[i][j] = root->recField[i][j];

            child->lv = root->lv + 1;
            Ypos = 0; //seg fault 방지
            Ypos = returnY(child->recField, Ypos, Xpos, nextBlock[root->lv],
rot);

            //Ypos=FinalShapeDrop()

            //점수 계산
            accumulatedScore = AddBlockToField(child->recField, nextBlock[root-
>lv], rot, Ypos, Xpos);
            accumulatedScore += DeleteLine(child->recField);

            accumulatedScore += (Ypos * Ypos); //추천 효율을 대폭 향상시키는 식 발견


            //재귀
            if (child->lv < depth_of_travel) accumulatedScore +=
modified_recommend(child);
            //최댓값 찾기
            if (result <= accumulatedScore) {
                if (root->lv == 0) {
                    if (Y_rec < Ypos || result < accumulatedScore) {
                        X_rec = Xpos;
                        Y_rec = Ypos;
                        R_rec = rot;
                    }
                }
                result = accumulatedScore;
            }

        }
    }
    return result;
}
```

↑modified_recommend 함수 전문

먼저 이 알고리즘을 구현하는데 필수적인 recommend(modified_recommend)함수는 트리의 루트 노드를 가리키는 포인터를 입력으로 받는다. 블럭의 종류마다 가능한 회전의 수를 미리 정의하고, 계산하고자 하는 블럭의 회전 가능 수에 따라 1차 for문을 돌게 된다. 그 안에 2차 for문에서는 x좌표(WIDTH) 만큼 탐색을 하여 블럭의 놓일 좌표값 경우를 나눈다. 이러한 이중 for문 안에서, 부모노드의 field 정보를 자식노드로 갱신시켜주는 명령문을 작성하였다. 이후에는, x좌표와 회전수가 fix된 해당 블럭이 떨어질 수 있는 y좌표를 returnY 함수로 계산한다. 이로서 완성된 해당 노드의 블럭의 정보 (블럭의 종류, 회전수, x좌표, y좌표)에 따른 누적 점수를 계산한다.

```
//점수 계산
accumulatedScore = AddBlockToField(child->recField, nextBlock[root->lv], rot,
Ypos, Xpos);
accumulatedScore += DeleteLine(child->recField);

accumulatedScore += (Ypos * Ypos); //추천 효율을 대폭 향상시키는 식 발견
```

accumulatedScore를 계산하는데 있어, 기존 recommend 함수에 비해 개선된 점은 블럭의 y좌표의 제곱을 가중치로 더한다는 점이다. 이렇게 하면 기존 recommend 함수를 따를 시 10만점을 넘

기지 못하던 것에 비해, 개선된 이후에는 100만점 이상까지 (거의 무한정) recommended play가 진행된다. 이렇게 개선할 수 있었던 이유는 기존 recommend 함수는 한 번에 여러 줄이 지워지는 경우를 높이 치기 때문에 (100*(지워진 줄 수)^2 + 10*맞닿은 면적) 한 번에 여러 줄을 지우기 위해서 줄을 바로 지우지 않고 쌓는 과정에서 빈 공간이 생기고, 그로 인해 비효율성이 발생하였었지만, y좌표의 제곱을 가중치로서 더해주게 된다면 블럭이 되도록 아래쪽으로 쌓이고자 하는 경향이 강해져 빈 공간 없이 쌓이게 된다. 이로써 game over가 쉽사리 나지 않는 방법을 찾아내었다.

추천 트리의 노드에 블록 정보 갱신을 끝마치고, 중간 점수 계산을 완료한 이후에는 재귀로 레벨이 3에서 0이 될 때까지 반복 수행한다. 블럭 하나당 각각의 경우의 수 노드를 확인한 후 점수를 구해야 하므로 시간 복잡도는 (블럭의 회전 방향 수)*(x 위치 (WIDTH))*(다음 깊이에 있는 블럭의 회전 가능 수)*(WIDTH)씩 재귀적으로 반복되며 실행된다. 공간 복잡도는 (노드의 수)^3에 비례한다.

2. **모든 경우를 고려하는 tree 구조와 비교해서 어떤 점이 더 향상되고, 어떤 점이 그렇지 않은지 아울러 기술하시오.**

   모든 경우를 고려하는 tree 구조에 비해 향상된 점은 각 블럭마다 무조건 4가지 회전의 경우를 고려하지 않고 블럭마다의 회전의 경우의 수를 생각해 준다는 것이다. 기존 경우의 수 28(7*4)보다 19(2+4+4+4+1+2+2)가지로 줄었다.

   기존에는 각 노드에 int형 변수 accumulatedScore와 자식 노드에 대한 포인터를 저장해 두었지만, 이를 modified_recommend 함수 내의 변수로 처리해 공간 낭비를 줄였다. 하지만 pruning과 같이 시간, 공간 복잡도 개선이 극적으로 이루어지지 못했다는 점이 아쉽다.

3. **테트리스 프로젝트 3주 과정을 통해 습득한 내용이나 느낀점을 기술하시오. (기술적인 측면 중심으로 서술)**

   교재에 나와있는 추천 알고리즘을 그대로 구현한다면 일부 변수를 불필요하게 중복적으로 사용하고, 필요치 않는 경우의 수를 고려하는 일이 발생하였다. (상세 내용은 상기함) 효율성 개선 측면에서 처음에는 pruning 기법을 사용하고자 하였으나, 최종적으로 최선의 경우의 수를 제할 수 있다는 가능성 때문에 효율적이지 못할 것 같아서 회전 경우의 수와 각 노드의 데이터 필드를 최적화하는 방향으로 진행하였다.

   테트리스 프로젝트를 구현하는 데 있어 중간에 segmentation fault와 같은 오류들이 숱하게 떠서 애를 먹기도 했지만 이를 디버깅하고 수정하는 과정을 통해 얼마 전 자료구조 시간에 배운 tree 구조에 대한 이해를 확실시 할 수 있었고, C언어의 메모리 할당 구조에 대해서도 보다 능숙해진 것 같다. 이번 프로젝트에서 구현하진 않았지만 구현하면서 트리의 다양한 구조, 탐색 방법에 대해 알아보게 되어 BFS, DFS, 세그먼트 트리와 같은 트리의 구조, 구현 방법을 습득하는 계기가 되었다.

   트리 이외에도 rank를 관리하는데 쓰인 linked list를 구현할 때 중간에 원소를 삽입/삭제하는 과정에서 오류가 떠서 이를 처리하는데도 조금 시간이 소요되었었지만 그 과정에서 링크드 리스트 원소 삽입의 예외 처리, 노드의 선형 탐색을 확실히 다룰 수 있는 능력을 갖추게 되었다.