8주차 결과보고서

전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181294 이름: 임승섭

**1.**

실습 시간에는 따로 효율성을 높이기 위한 장치를 만들진 않았고, 강의자료에 있는 알고리즘 그대로 코드로 구현하였다.

1 - 1. 우선 트리를 구성하는 노드 하나를 구조체로 선언하였다.

typedef struct \_RecNode{

    // must-have elements

    int level;

    int acc\_score;

    char recField[HEIGHT][WIDTH];

    struct \_RecNode \*c[CHILDREN\_MAX];

    /\*

    // optimal elements

    int curBlockID;

    int recBlockX, recBlockY, recBlockRotate;

    struct \_RecNode \*parent;

    \*/

} RecNode;

이름은 RecNode로 하였고, 강의자료에는 optimal elements들도 있었지만, 알고리즘을 구현하는데 따로 필요가 없어서 사용하지 않았다. must elements들로는 tree 내에서 node의 level, field에 node 정보의 블록이 쌓일 때 얻게 되는 점수(acc\_score), field에 node 정보의 블록이 쌓일 때의 field 정보(recField), 그리고 그 노드의 자식 노드들을 가리키는 포인터들의 배열(\*c[CHILDREN\_MAX])로 구성하였다. 사실 이 중에서도, acc\_score와 \*c[]는 실제 코드에서 굳이 사용하지 않더라도 정상적으로 작동하게 할 수 있었다.

1 - 2. DrawRecommend()

reccommend 함수를 통해 얻은 위치에 ‘R’ 표시를 하게 만들었다. DrawBlock() 함수를 이용하였다.

void DrawRecommend(int y, int x, int blockID,int blockRotate){

    // user code

    DrawBlock(y, x, blockID, blockRotate, 'R');

}

1 - 3. recommend(RecNode \*root) : 실습 시간에 구현한 recommend 함수.

int recommend(RecNode \*root){

    int max=-1;

    int max\_level = VISIBLE\_BLOCKS;

    int rot[7] = {2, 4, 4, 4, 1, 2, 2};

    int r = rot[nextBlock[root->level]];

    RecNode \*child = (RecNode\*)malloc(sizeof(RecNode));

    for (int i = 0; i < r; i++) {

        for (int j = -1; j < WIDTH; j++) {

            if (CheckToMove(root->recField, nextBlock[root->level], i, 1, j) == 0)

                continue;

            int k = 0;

            while (CheckToMove(root->recField, nextBlock[root->level], i, k+1, j) != 0)

                k++;

            int c\_score = 0;

            for (int p = 0; p < HEIGHT; p++) {

                for (int q = 0; q < WIDTH; q++) {

                    child->recField[p][q] = root->recField[p][q];

                }

            }

            c\_score += AddBlockToField(child->recField, nextBlock[root->level], i, k, j);

            c\_score += DeleteLine(child->recField);

            child->acc\_score = c\_score;

            child->level = root->level + 1;

            if (child->level < max\_level) {

                c\_score += recommend(child);

            }

            if (c\_score > max) {

                max = c\_score;

                if (root->level == 0) {

                    recBlockX = j;

                    recBlockY = k;

                    recBlockR = i;

                }

            }

        }

    }

    return max;

}

<코드에 대한 설명>

(1). 변수 초기화 (최대 점수를 나타내는 max 값을 음수로 초기화하고, 트리의 최대 레벨을 나타내는 max\_level 값은 global vaiable로 선언되어 있는 VISIBLE\_BLOCKS로 선언하였다)

(2). 블록의 rotate 구분. (사실 강의자료에 따로 나와있지 않아서 나중에 효율성 고려한 코드에서 구현해도 괜찮았던 것 같다. 아무튼, 블록에 따라 회전에 대한 경우의 수가 모두 다른데, 이걸 굳이 4개의 loop을 돌 필요가 없다고 생각하여, index는 블록의 ID, 값은 블록의 회전수를 나타내는 array를 하나 생성하였다(rot[]). nextBlock은 블록의 ID를 반환하고, root->level은 현재 내려가고 있는 트리의 level을 나타내기 때문에 rot[nextBlock[root->level]]을 하면 현재 블록의 rotation(회전수)의 개수를 정확하게 알 수 있다.

(3). rotation에 대한 for loop (int i= 0; i < r; i++). 이 이후부터 i가 블록의 현재 rotate 상태를 나타낸다.

(4). x좌표에 대한 for loop (int j = -1; j < WIDTH; j++). 이 이후부터 j가 블록의 현재 x좌표를 나타낸다.

(5). 첫번째 라인(높이가 1)에서 블록이 내려갈 수 있는지 확인하고, 불가능하다면 loop을 그냥 넘어갔다. 높이 1에 블록을 쌓게 되면 곧 죽기 때문에 의미가 없기 때문이다. CheckToMove 함수를 이용했다.

(6). recommend로 얻을 y좌표를 k로 선언하고, 마찬가지로 CheckToMove 함수를 이용해 블록이 쌓이는 위치의 y좌표를 확인했다. 이 이후부터 k가 블록의 y좌표를 나타낸다.

(7). 현재 정보들을 저장할 노드 child (위에 child노드 생성하는 곳을 빼먹었다. RecNode 를 이용해 선언하였다)의 RecField를 root의 RecField와 같게 만들어준다. 이중 loop을 이용하여 Field의 모든 부분을 같게 만든다.

(8). 업데이트된 RecField에 블록을 쌓고, 지울 수 있는 라인은 삭제한다. AddBlockToField와 DeleteLine 함수를 이용한다. 이 과정에서 평소와 똑같이 점수를 계산한다(c\_score). 계산 전 점수는 0으로 초기화한다.(c\_score = 0).

(9). 계산한 점수를 노드의 acc\_score element로 설정한다. 또한 노드의 level은 현재 함수의 인자로 받은 root의 level보다 1 더해준다.

(10). 재귀함수 부분이다. 만약 현재 노드의 level이 max\_level(VISIBLE\_BLOCKS)보다 작다면, 트리의 level을 더 늘릴 수 있기 때문에 다시 현재 노드(child)에 대해 recommend 함수를 실행한다. 이후에 나오겠지만 함수의 return 값은 최대점수이기 때문에 c\_score에 그 값을 더해서 저장해준다.

(11). max 점수를 가질 때의 위치를 저장한다. 우선, 현재 점수가 저장된 max값보다 크면 max를 c\_score로 바꿔줌으로써 점수를 업데이트 해주고, 그 때 root->level이 0이라면 현재 돌고 있는 loop이 nextBlock[0]의 정보를 갖고 있다는 뜻이므로 추천\_블록\_좌표(recBlockX, recBlockY, recBlockR)을 현재 노드의 정보(i, j, k)로 업데이트 시켜준다.

(12). loop이 모두 끝나면 max 값을 return 한다.

1 - 4. recBlockDown

void recBlockDown(int sig) {

    if (CheckToMove(field, nextBlock[0], blockRotate, blockY+1, blockX) == 0 )

            gameOver = 1;

    else {

        score += AddBlockToField(field, nextBlock[0], recBlockR, recBlockY, recBlockX);

        score += DeleteLine(field);

        PrintScore(score);

        nextBlock[0] = nextBlock[1];

        nextBlock[1] = nextBlock[2];

        nextBlock[2] = rand()%7;

        recRoot = (RecNode\*)malloc(sizeof(RecNode));

        recRoot->level = 0;

        recRoot->acc\_score = 0;

        for (int i = 0; i < HEIGHT; i++) {

            for (int j = 0; j < WIDTH; j++) {

                recRoot->recField[i][j] = field[i][j];

            }

        }

        recBlockX = 0;

        recBlockY = 0;

        recBlockR = 0;

        recommend(recRoot);

        blockRotate = 0;

        blockY = -1;

        blockX = (WIDTH/2) - 2;

        DrawNextBlock(nextBlock);

    }

    DrawField();

    DrawBlockWithFeatures(blockY, blockX, nextBlock[0], blockRotate);

    timed\_out = 0;

}

<코드 설명>

BlockDown 함수와 거의 유사하지만, CheckToMove를 통해 블록이 내려갈 수 없음을 확인하면 여기선 아예 게임을 끝내버렸다. 이유는 블록이 한 칸씩 내려가는 형태가 아니라, 블록이 내려갈 수 있으면 바로 위의 recommend 함수에서 얻은 위치(recBlockX, recBlockY, recBlockR)에 블록을 바로 쌓아버리기 때문이다. 이를 else 부분에서 구현했다. 일단 AddBlockToField와 DeleteLine 함수를 이용해서 score를 계산하고 필드를 업데이트한 후, 이전과 마찬가지로 score를 화면에 출력한다. nextBlock 배열을 이용해서 새로운 블록들을 업데이트해주고, recRoot를 초기화한다. recRoot가 가지고 있는 element 값들(level, acc\_score)를 0으로 초기화하고, recField는 현재 Field와 동일하게 만들어준다. recommend함수를 들어가기 전에 recBlockX, recBlockY, recBlockR 값을 다시 0으로 초기화하고 recoomend 함수를 실행한다. 이후 내용은 BlockDown함수와 동일하다.

여기까지가 효율성을 고려하지 않고 구현한 실습 내용이다. 이후에는 더 높은 효율성을 내기 위한 알고리즘을 설명하겠다.

1 - 5. 더 높은 효율성을 내기 위한 방법으로는 1. running time을 줄인다 2. memory size를 줄인다 3. 더 높은 점수를 내는 위치를 추천한다. 로, 총 3가지 방법이 있다. 나는 이 중 3번에 가장 집중하였다. time과 memory를 줄여도 게임이 금방 죽으면 결코 좋은 결과는 아니기 때문에, recommend 함수에서 어떻게 해야 더 높은 점수를 받을 수 있을지 고민하였다. 내가 내린 결론은 1. 최대한 블록들이 아래에 쌓이도록 한다, 2. 블록과 블록 사이에 빈칸을 최대한 줄인다 로 두가지였다. 높은 점수를 내기 위한 노력으로는 이렇게 두가지 방법을 적용하였고, 강의자료에 나온 pruning과 data simplication도 이용하였다. 이 과정에서 각각의 정도를 바꿔가며 여러번 시행하면서 최상의 효율이 나올 수 있도록 변수들을 설정하였다. 사실 시간과 공간을 넉넉하게 쓸수록 그 이상으로 점수가 높게 나오는 점을 확인할 수 있어서, 높은 점수를 내는 데 집중했던 방법이 괜찮은 방법이라고 생각했다.

1 - 6. modified\_recommend

int modified\_recommend(RecNode \*root) {

    int max = -1;

    int max\_level = VISIBLE\_BLOCKS;

    m\_size += sizeof(\*root);

    int rot[7] = {2, 4, 4, 4, 1, 2, 2};

    int r = rot[nextBlock[root->level]];

    RecNode \*child = (RecNode\*)malloc(sizeof(RecNode));

    for (int i = 0; i < r; i++) {

        for (int j = -1; j < WIDTH; j++) {

            if (CheckToMove(root->recField, nextBlock[root->level], i, 1, j) == 0)

                continue;

            int k = 0;

            while( CheckToMove(root->recField, nextBlock[root->level], i, k+1, j) ) {

                k++;

            }

            int c\_score = 0;

//

            for (int p = 0; p < k + 10; p++) {

                for (int q = 0; q < WIDTH; q++) {

                    child->recField[p][q] = root->recField[p][q];

                }

            }

            c\_score += AddBlockToField(child->recField, nextBlock[root->level], i, k, j);

            c\_score += DeleteLine(child->recField);

            child->acc\_score = c\_score;

            child->level = root->level + 1;

            //

            c\_score += 20\*k;

            //

            int space\_num = 0;

            for (int x = k; x < k+2; x++) {

                for (int y = -1; y < 13; y++) {

                    if (child->recField[x][y] == 0)

                        space\_num++;

                }

            }

            c\_score -= space\_num;

            //

            if ( (child->level < max\_level) && (c\_score > 40) )

                c\_score += modified\_recommend(child);

            if (c\_score > max) {

                max = c\_score;

                if (root->level == 0) {

                    recBlockX = j;

                    recBlockY = k;

                    recBlockR = i;

                }

            }

        }

    }

    return max;

}

<코드 설명>

전반적으로 recommend 함수를 그대로 차용하였고, 중간중간 효율성을 높이기 위한 장치를 설치하였다. recommend 함수에서 새로 추가한 부분만 설명하겠다.

(1). 우선 memory size를 줄이기 위해 child의 recField를 root의 recField와 같게 하는 부분에서 모든 높이를 고려하지 않고, 현재 y좌표(k)부터 10칸 아래까지만 고려하였다. 사실 10칸이면 필드 중간 아래에 블록들이 쌓일 때는 거의 영향이 없지만, 많이 쌓여서 필드 중간을 넘어가면 그때부터는 충분히 memory size를 줄이는 데 도움이 될 것이다. 10을 비롯해서 여러 상수를 대입해 보았지만 너무 작게 되면 금방 블록들이 쌓여서 gameover가 일어남을 볼 수 있었다. 게임이 끝나지 않는 것이 가장 큰 목표이기 때문에 적절하게 넉넉한 값을 설정하였다.

(2). running time을 줄이기 위해 pruning을 시행하였다. 함수 내에서 노드가 가지는 점수(c\_score)가 40 이상일 때에만 recursive하게 함수를 호출하였다. 이 말은 점수가 40 미만인 노드는 더 이상 child 노드를 가지지 않는다는 뜻이기 때문에 시간적으로 효율성을 높일 수 있다. 하지만 이 40이라는 숫자도 아래 c\_score의 계산 방법이 나오게 되면 아주 낮은 값임을 알 수 있는데, 이미 언급했듯이 이 게임의 목표는 죽지 않는 것이기 때문에 여러 번의 시행 끝에 40으로 설정하였다.

(3). c\_score의 계산 방법에 약간 변화를 주었다. 원래 recommend와 마찬가지로 당연히 AddBlockToField와 DeleteLind 함수를 통한 점수는 추가하였다. 우선, 블록들이 최대한 아래에 많이 쌓이도록 하기 위해 블록의 y좌표(k)에 20을 곱한 값을 c\_score에 추가해 주었다. 이렇게 되면 여러 경우의 수 중 상대적으로 블록을 아래에 쌓을 수 있는 노드들이 선택될 가능성이 높아진다. 다음으로, 블록들 사이에 있는 빈칸의 개수를 세어, 그 개수만큼 c\_score에서 빼주었다. 이를 통해 빈 칸을 최대한 적게 만들 수 있는 위치가 선택되도록 하였다. 이 과정에서, 처음엔 쌓여있는 블록의 모든 위치를 확인하였지만, memory size나 running time에 너무 소모가 클 것으로 예상되어, 현재 y좌표(k)를 기준으로 두 줄 아래까지만 탐색하도록 하였다. 실질적으로 현재 블록이 영향을 줄 수 있는 위치는 1~4줄 아래까지이기 때문에, 두 줄 정도로만 설정해도 큰 문제가 없을 것이라고 판단하였다.

1 - 7. 시간복잡도, 공간복잡도의 차이.

시간복잡도. 우선 worst case를 생각하기 때문의 block의 회전수는 4로 고정하겠다. recommend 함수의 경우, for loop이 두 번 돌기 때문에(rotation, WIDTH) 우선 4 \* WIDTH. loop 안에서 CheckToMove함수의 시간은 block\_height \* block\_width 이지만 child의 recField를 초기화할 때 그보다 훨씬 큰 HEIGHT \* WIDTH가 걸린다. 이 모든 걸 recursive하게 VISIBLE\_BLOCKS만큼 실행하게 되면 (4 + WIDTH^2 \* HEIGHT)^VISIBLE\_BLOCKS 정도로 생각하면 된다. modified\_recommend도 거의 비슷하지만, c\_score에 따라 노드의 child가 생기지 않을 수도 있기 때문에 시간적으로 recommend 함수보다는 효율이 좋은 것을 확인할 수 있다.

공간복잡도. recommend 함수가 한 번 실행될 때마다 child 노드는 최대 (4 \* WIDTH)^VISIBLE\_BLOCKS 만큼 생길 수 있다. 노드 안에 field의 메모리까지 생각하면 (4 \* WIDTH^2 \* HEIGHT)^VISIBLE\_BLOCKS 정도로 memory\_size를 볼 수 있다. modified\_recommend에서 이 size를 줄일 수 있는 점을 확인하면, 우선 child 노드의 개수가 pruning으로 인해 줄어든다. 또한, field의 정보도 k부터 최대 10까지만 받아오기 때문에 memory를 줄이는 데 도움이 된다.

**2.**

1번 문항에 대한 답변 과정에 드문드문 향상된 점과 그렇지 않은 점에 대해 적혀있기 때문에 최대한 요약하겠다.

우선 pruning 과정으로 인해 시간을 줄일 수 있었다. 하지만 pruning을 통해 삭제하는 노드의 개수가 아주 적기 때문에 시간 면에서 엄청난 효율을 얻었다고 할 수는 없다.

다음으로 data simplication을 통해 memory size를 줄일 수 있었다. 다만 이 부분도 블록의 높이가 높은 상황에서만 발생하는 부분이기 때문에 마찬가지로 엄청난 효율을 얻진 못했다.

하지만 두 과정을 철저하게 학습하였기 때문에 기존 recommend 함수보다는 분명 소모되는 부분을 줄였다고 할 수 있다.

마지막으로, 점수를 높이는 데 가장 큰 노력을 다했다. y좌표에 가중치를 줌으로써 블록이 최대한 아래로 쌓이게 만들었고, 빈칸의 개수에 음의 가중치를 줌으로써 블록과 블록 사이의 공간이 최대한 없도록 만들었다. 효율성 측면으로만 보자면, score/time과 score/memory가 아주 급격하게 향상됨을 확인할 수 있었다. 기본적으로 score가 20만점을 넘어가고, 60만점 이상이 되어 강제 종료시킨 경우도 있었다.

물론 점수를 높이기 위한 시스템을 모든 노드를 고려하는 Tree구조에 적용했다면 훨씬 더 높은 점수를 얻을 수 있었을 것이다. 하지만 시간과 공간을 줄이는 것도 과제의 일부였기 때문에 이를 위한 노력도 학습하고 실제로 구현해 보았다.

**3.**

3주간 테트리스 프로젝트를 진행하면서 여러모로 공부하는데 도움이 많이 되었다. 이렇게 길고 많은 함수를 가진 코드를 한번에 구현해본 적이 처음이었기 때문에 처음에는 힘든 부분이 많았지만, 점차 익숙해지며 어떻게 하면 더 좋은 코드를 만들어낼 수 있을까 고민하는 습관도 가지게 되었다. 또한 마지막 주차에서 높은 점수를 주기 위한 가중치를 생각해낸 부분은 유고결석 대체 보고서로 제출한 ‘유전 알고리즘’을 공부한 점이 아주 큰 도움이 되었다. 조교님께서 공부하는 데 도움이 될 것이라고 추천하신 유전 알고리즘에 대한 보고서를 작성하면서, 테트리스 게임에 적용할 수 있는 가중치들을 찾아보았고, 적용하는 방법은 약간 달랐지만 이번 Tree 구조의 추천 시스템에서도 직접 구현해보았다는 점에서 아주 좋은 경험을 했다고 생각한다. 처음으로 해본 대형(?) 프로젝트였고, 정말 많은 노력을 다 했던 실습, 과제였기 때문에 아주 오래 기억에 남을 것 같다.