10주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20231515 이름: 김다은

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

**1-1. 실습 시간에 작성한 프로그램**

RecNode 구조체를 초대로 트리구조를 구현하였다. RecNode에는 현재 lv(depth)와 score, field 정보, child RecNode를 가리키는 포인터배열, 추천된 위치정보와 회전정보가 들어 있다.

typedef struct \_RecNode{

    int lv,score; *// lv: tree의  level(depth) 나타냄*

    char f[HEIGHT][WIDTH];

    struct \_RecNode \*c[CHILDREN\_MAX]; *// child를 가리키는 pointer*

    int recBlockX, recBlockY, recBlockRotate; *// 추천된 위치 및 회전수*

} RecNode;

너비우선탐색(BFS)를 이용하여 최선의 블록을 찾아내는 프로그램을 구현하였다. recommend함수가 호출되었을 때, 먼저 child RecNode에 메모리 동적할당을 한다.

if(root->lv < VISIBLE\_BLOCKS) {

        for(int i=0; i<CHILDREN\_MAX; i++) {

            root->c[i] = (RecNode\*)malloc(sizeof(RecNode));

            root->c[i]->lv = root->lv + 1;

            root->c[i]->score = root->score;

        }

    }

이후 블록의 모든 회전 경우에 대해 child의 경우를 계산한다.

*// 각 rotation의 경우에 대해 child의 경우들 계산*

    for(int r=0; r<NUM\_OF\_ROTATE; r++) {

        for(int j=0; j<WIDTH; j++) {

            if(child\_idx >= CHILDREN\_MAX) break;

*// 블럭을 놓는 것이 불가능한 경우*

            if(CheckToMove(root->f, nextBlock[root->lv], r, 0, j)==0) {

                root->c[child\_idx++]->score = -1;

                continue;

            }

*// child에 field 복사*

            memcpy(root->c[child\_idx]->f, root->f, sizeof(field));

            int i;

            for(i=0; i<HEIGHT; i++) {

*// x = j, y = i+1 자리에 블럭을 놓을 수 없을 때까지 i 증가*

                if(CheckToMove(root->c[child\_idx]->f, nextBlock[root->lv], r, i+1, j)==0) break;

            }

*// 위치 저장*

            root->c[child\_idx]->recBlockRotate = r;

            root->c[child\_idx]->recBlockX = j;

            root->c[child\_idx]->recBlockY = i;

*// 점수 계산*

            root->c[child\_idx]->score += AddBlockToField(root->c[child\_idx]->f, nextBlock[root->lv], r, i, j);

            root->c[child\_idx]->score += DeleteLine(root->c[child\_idx]->f);

*// 다음 child node를 계산하도록 child\_idx 1 증가*

            child\_idx++;

        }

    }

아직 목표 depth에 도달하지 않은 경우, recommend 함수를 다시 한번 호출한다.

*// 아직 탐색 목표 depth에 도달하지 않은 경우*

    if(root->lv < VISIBLE\_BLOCKS-1) {

        for(int i=0; i<CHILDREN\_MAX; i++) {

*// child node 존재?? 걍 0점 제외??*

            root->c[i]->score += recommend(root->c[i]);

        }

    }

최상위 노드인 경우 최적의 블록을 두는 경우를 도출해야 한다. 이때, 점수가 더 높은 경우를 선택한다. 만약 점수가 같은 경우들이 존재한다면, 블록이 설치되는 높이(recY)가 낮은 경우를 선택한다. 이 부분이 최선의 해를 구하는 기준이 되므로 추천 프로그램에서 제일 중요한 부분이라고 할 수 있겠다.

    for(int i=0; i<CHILDREN\_MAX; i++) {

        if(maxscore <= root->c[i]->score) {

            if(root->lv == 0) { *// 최상위 노드인 경우*

*// 블록이 더 아래에 놓이거나 score가 더 높은 경우*

                if(maxscore < root->c[i]->score || (maxscore == root->c[i]->score && recY < root->c[i]->recBlockY))

{

                    recX = root->c[i]->recBlockX;

                    recY = root->c[i]->recBlockY;

                    recRotate = root->c[i]->recBlockRotate;

                }

            }

            maxscore=root->c[i]->score;

        }

*// 할당한 메모리 해제*

        free(root->c[i]);

    }

    return maxscore;

<시간복잡도>

편의를 위해 한 child들을 탐색하는데 소모되는 비용을 CHILD 라고 표현하겠다. 이때 시간복잡도는 (O(CHILD)\*CHILDREN\_MAX)^VISIBLE\_BLOCKS가 된다. nextBlock를 많이 고려하게 될수록 연산 시간은 급격하게 증가하게 된다는 것을 확인할 수 있다.

<공간복잡도>

편의를 위해 한 child들을 탐색하는데 소모되는 비용을 CHILD 라고 표현하겠다. 이때 공간복잡도는 O(CHILD)\*CHILDREN\_MAX)^VISIBLE\_BLOCKS가 된다. 마찬가지로 nextBlock를 많이 고려하게 될수록 메모리 사용량이 급격하게 증가하게 된다는 것을 확인할 수 있다.

1-2. 과제로 작성한 프로그램(업그레이드 된 프로그램)

**(1) 데이터 단순화**

실습시간에 구현한 recommend함수의 경우 field에 추가 가능한 블록의 경우를 찾기 위해 모든 좌표에 대해 CheckToMove 함수를 호출한다. 이는 매우 비효율적이다.

시간을 절약하기 위해 RecNode 구조체에 minifield라는 1차원 배열을 추가하였다.

typedef struct \_RecNode{

    int lv,score; *// lv: tree의  level(depth) 나타냄*

    char f[HEIGHT][WIDTH];

    char minifield[WIDTH];

    struct \_RecNode \*c[CHILDREN\_MAX]; *// child를 가리키는 pointer*

    int recBlockX, recBlockY, recBlockRotate; *// 추천된 위치 및 회전수*

} RecNode;

minifield는 index값으로 width 값을 갖는다. minifield[width]는 해당 width의 최상위에 있는 블록의 높이가 저장되어 있다.

먼저, modified\_recommend 함수가 호출되기 직전에 다음과 같이 field\_to\_minifield 함수를 호출한다.

    RecNode root;

    root.lv = 0;

    root.score = 0;

    memcpy(root.f, field, sizeof(field));

    field\_to\_minifield(root.f, &root);

    modified\_recommend(&root);

field\_to\_minifield함수는 field 값을 토대로 minifield를 생성한다.

void field\_to\_minifield(char f[HEIGHT][WIDTH], RecNode \*root) {

~

    for(int j=0; j<WIDTH; j++) {

        int i=0;

        for(i=0; i<HEIGHT && f[i][j]==0; i++);

        root->minifield[j] = i;

    }

}

이후 modified\_recommend 함수 안에는 현재 블록의 모든 회전의 경우에 대해 블록을 어디에 추가할 수 있는지 고려하게 된다. 이때, 다음과 같이 i를 minifield[j]-1의 값으로 초기 설정한다. 이를 통해 전에는 0~HEIGHT까지 순환하며 CheckToMove 함수를 호출하는 과정을 생략할 수 있다.

for(int r=0; r<rotatemax; r++) {

        for(int j=-1; j<WIDTH; j++) {

            ~

            char flag = 1;

            int i;

            if(j<0) i = 21;

            else i = root->c[child\_idx]->minifield[j] - 1;

            for(int tmpi = 0; i-tmpi>=0; tmpi++) {

                if(recommend\_CheckToMove(root->c[child\_idx]->f, nextBlock[root->lv], r, i-tmpi, j, root->c[child\_idx]->minifield)==1) {

                    flag = 0;

                    i = i-tmpi;

                    break;

                }

            }

            if(flag==1) continue;

}

}

이때 recommend\_CheckToMove 함수를 살펴보겠다.

int recommend\_CheckToMove(char f[HEIGHT][WIDTH], int currentBlock, int blockRotate, int blockY, int blockX, char minif[WIDTH]) {

    for(int i=0; i<BLOCK\_HEIGHT; i++) {

        for(int j=0; j<BLOCK\_WIDTH; j++) {

            if(block[currentBlock][blockRotate][i][j] == 1) {

*// 블록이 게임판 밖에 위치하는 경우 또는 이미 블럭이 존재하는 경우*

                if(i+blockY<0 || HEIGHT <= i+blockY || j+blockX<0 || WIDTH <= j+blockX || f[i+blockY][j+blockX] == 1) {

                    return 0;

                }

*// 블록이 사이에 존재하는 경우*

                if(minif[blockX+j] <= blockY+i) return 0;

            }

        }

    }

    return 1;

}

기존 CheckToMove 함수를 사용할 경우, 다음 사진과 같이 불가능한 위치에 블록을 추천하는 문제가 발생한다.

스크린샷, 텍스트, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명

이는 다양한 블록을 4\*4로 표현하다 보니 4\*4 중 비어 있는 줄이 발생했기 때문이다. 이 문제를 해결하기 위해 if(minif[blockX+j] <= blockY+i) return 0; 를 추가해 주었다.

**(2) Pruning Tree**

최선의 경우가 될 확률이 적은 노드를 탐색에서 제외하여 시간을 절약하는 방법이다.

modified\_recommend 함수에서 다음과 같이 변수를 선언했다.

int sumscore = 0; *// child 경우의 score 합*

int childcnt = 0; *// 유효한 child의 갯수*

이후 유효한 child의 score를 sumscore에 더하고 childcnt를 +1 해 주었다.

sumscore += root->c[child\_idx]->score;

childcnt++;

이후 score 평균(sumscore/childcnt)를 구하여, 해당 child의 score가 평균 이상인 경우에만 modified\_recommend 함수를 호출하여 탐색을 이어가도록 했다.

if(root->lv < VISIBLE\_BLOCKS-1) {

        for(int i=0; i<CHILDREN\_MAX; i++) {

*// score가 child들의 평균 score보다 큰 경우*

            if(childcnt == 0) return 0;

            if(root->c[i]->score >= sumscore/childcnt) {

                root->c[i]->score += modified\_recommend(root->c[i]);

            }

        }

}

<시간복잡도>

편의를 위해 한 child들을 탐색하는데 소모되는 비용을 CHILD 라고 표현하겠다. 이때 시간복잡도는 (O(CHILD)\*CHILDREN\_MAX)^VISIBLE\_BLOCKS가 된다. nextBlock를 많이 고려하게 될수록 연산 시간은 급격하게 증가하게 된다는 것을 확인할 수 있다.

<공간복잡도>

편의를 위해 한 child들을 탐색하는데 소모되는 비용을 CHILD 라고 표현하겠다. 이때 공간복잡도는 O(CHILD)\*CHILDREN\_MAX)^VISIBLE\_BLOCKS가 된다. 마찬가지로 nextBlock를 많이 고려하게 될수록 메모리 사용량이 급격하게 증가하게 된다는 것을 확인할 수 있다.

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

데이터 단순화(field를 1차원 minifield로 압축하여 표현)은 2차원 배열을 순환하며 블록의 이동 가능 여부를 판단하는 대신, 빠르게 블록의 최적 위치를 찾도록 해주어 시간을 절약하는 역할을 한다. 다만 minifield라는 새로운 변수를 선언하고 관리해야 하므로 추가적인 메모리가 필요하다는 단점이 존재한다.

pruning tree는 child들의 평균 점수를 구하고, 평균보다 적은 점수를 가진 child는 고려하지 않는다. 이를 통해 연산시간을 효과적으로 단축할 수 있다. 다만 현재 관점에서는 평균보다 점수가 낮아 고려되지 않았지만, 장기적으로 보았을 때는 더 많은 점수를 얻게 해줄 수 있는 경우가 존재한다. 하지만 이를 고려하지 못한다는 점에서 항상 최적의 해를 보장하지 못한다는 단점을 가진다.

이러한 단점에도 불구하고 필자가 데이터 단순화와 pruning tree 기법을 사용한 이유는 다음과 같다. 게임은 ‘실시간’ 진행이 핵심이다. 게임 연산에 시간이 오래 걸려 버퍼링이 생기는 것은 치명적인 문제이다. 따라서 메모리 사용량을 조금 올리고, 완벽한 최적의 해 대신 연산 시간을 줄이는 것이 합리적이라고 판단했다.

**3. 테트리스 프로젝트 3주 과정을 통해 습득한 내용이나 느낀 점을 기술하시오.**

코딩 과정에서 segment error가 많이 발생했다. segment error의 원인을 찾기 위해 자연스럽게 gdb 사용법을 익히게 되었는데, 이제는 breakpoint를 설정하고 변수의 값을 확인하거나 대입해보면서 잘못된 부분을 쉽게 찾을 수 있게 되었다.

프로그램에서 공간복잡도와 시간복잡도를 어떻게 분배할 것인지가 중요하다고 느꼈다. 효율적인 알고리즘을 통해 최대한 시간복잡도를 줄였다고 가정했을 때, 프로그램의 목적에 맞춰서 공간복잡도를 조금 희생해서 연산 시간을 단축할 것인지, 혹은 시간복잡도를 조금 희생해서 메모리 사용량을 절약할 것인지 잘 판단하는 것이 중요한 것 같다. 다시 말해 프로그램의 목적에 대해서도 잘 파악하는 것이 중요하다고 느꼈다.