

## 1. 실습 시간에 작성한 프로그램의 알고리즘과 자료구조 및 시간/공간 복잡도

다음과 같이 선언된 구조체를 선언해 트리를 구성하는 노드로 사용했고, root 노드를 전역변수로 선언했다.

```
//week8 recommendation
typedef struct _RecNode{
    int level; //tree level
    int accumulatedScore; //accumulated score
    char recField[HEIGHT][WIDTH]; //field state
    struct _RecNode *child[CHILDREN_MAX]; //child nodes
    int recommendY, recommendX, recommendR; /* 추천 블록 배치 정보. 차례대로 Y 좌표, X 좌표, 회전 */
} RecNode;
RecNode *recRoot; /* Root node of the recommendation tree. */
```

recommend() 함수는 알고 있는 미래의 블록에 대해 모든 회전수, x좌표를 고려하여 모든 상황에 대해 score를 계산하고 가장 큰 score를 만드는 현재로서 최선의 선택을 한다. VISIBLE\_BLOCKS보다 tree의 level이 작은 동안 재귀를 반복한다.

```
int recommend(RecNode *root){
    int max=0; //(accumulated) maximum score considering the recommended placement of blocks

    int n = 0;
    int x, y, rotate;
    int score;
    RecNode **child = root->child;

    //For each rotation of block
    for (rotate = 0; rotate < NUM_OF_ROTATE; rotate++) {
        //For each position
        for (x = -3; x < WIDTH+3; x++) {
            //possible to put the block
            if (CheckToMove(root->recField, nextBlock[root->level], rotate, 0, x)) {
                score = 0;

                //Make a new field where block is put
                //Store all the information
                child[n] = (RecNode*)malloc(sizeof(RecNode));
                CopyField(child[n]->recField, root->recField);
                child[n]->level = root->level + 1;

                y = GetY(child[n]->recField, 0, x, nextBlock[root->level], rotate);
                score += AddBlockToField(child[n]->recField, nextBlock[root->level], rotate,
y, x);

                score += DeleteLine(child[n]->recField);

                //if (current_level < MAX_LEVEL)
                if (child[n]->level < VISIBLE_BLOCKS) {
                    //calculate current score and recursive call
                    score += recommend(child[n]);
                }
                //if (accumulated maximum score < score)
                if (max < score) {
                    //Update the information of the block
                    //with the accumulated maximum score
                    max = score;
                    root->recommendX = x;
                    root->recommendY = y;
                    root->recommendR = rotate;
                    root->accumulatedScore = max;
                }
            }
        }
    }
}
```

```

    }
    n++;
  }
}

return max;
}

```

recommend() 함수 내에서 사용된 GetY() 함수는 블록이 놓일 수 있는 y좌표를 계산하는 함수, CopyField는 parent 노드의 필드 정보를 child 노드의 필드 정보에 복사해 저장하는 함수이다.

```

int GetY(char recField[HEIGHT][WIDTH], int y, int x, int blockID, int blockRotate) {
    do {
        y++;
    } while(CheckToMove(recField, blockID, blockRotate, y, x));
    return (y-1);
}

void CopyField(char childField[HEIGHT][WIDTH], char rootField[HEIGHT][WIDTH]) {
    int i, j;
    for(j=0; j<HEIGHT; j++)
        for(i=0; i<WIDTH; i++)
            childField[j][i]=rootField[j][i];
}

```

시간 복잡도: 한 번 recommend()가 호출될 때 for문에서 NUM\_OF\_ROTATE \* WIDTH의 반복이 일어나며, child 노드들에 대해 재귀가 일어난다. 따라서 최악의 경우 시간복잡도는 다음과 같다.

$$O((\text{NUM\_OF\_ROTATE} * \text{WIDTH})^{\text{VISIBLE\_BLOCKS}})$$

공간 복잡도: 각 노드들에 대해 필드의 정보 등이 저장되므로 최악의 경우 다음과 같다.

$$O((\text{NUM\_OF\_ROTATE} * \text{WIDTH})^{\text{VISIBLE\_BLOCKS}} * \text{HEIGHT} * \text{WIDTH})$$

## 2. 모든 경우를 고려하는 tree 구조와 비교해 향상된 점 & 향상되지 않은 점

modified\_recommend() 함수에서는 이중 for문의 반복 횟수를 줄이고자 tetris.h에 블록의 회전수와 놓일 수 있는 x좌표를 블록의 모양에 따라 별도로 저장해주었다.

```

int rotateNum[NUM_OF_SHAPE] = {2, 4, 4, 4, 1, 2, 2};

int XRange[NUM_OF_SHAPE][NUM_OF_ROTATE][2] = {
    { /*[0][][]*/
        { 0, 6}, //
        {-1, 8}, //
        { 0, 6},
        {-1, 8}
    },
    { /*[1][][]*/
        {-1, 6}, //
        {-2, 6}, //
        {-1, 6}, //
        {-1, 7} //
    },
    { /*[2][][]*/
        {-1, 6}, //
        {-2, 6}, //
        {-1, 6}, //
        {-1, 7} //
    },
}

```

```

{ /*[3][][]*/
  { 0, 7}, //
  { 0, 8}, //
  { 0, 7}, //
  {-1, 7} //
},
{ /*[4][][]*/
  {-1, 7}, //
  {-1, 7},
  {-1, 7},
  {-1, 7}
},
{ /*[5][][]*/
  {-1, 6}, //
  {-1, 7}, //
  {-1, 6},
  {-1, 7}
},
{ /*[6][][]*/
  {-1, 6}, //
  {-1, 7}, //
  {-1, 6},
  {-1, 7}
}
};

```

x좌표를 modified\_recommend() 함수 내에서 활용하기 위해 getXRange() 함수를 작성해 활용했다.

```

void getXRange(int blockID, int rotate, int *left, int *right) {
  *left = XRange[blockID][rotate][0];
  *right = XRange[blockID][rotate][1];
}

```

```

int modified_recommend(RecNode *root){
  int max=0; //(accumulated) maximum score considering the recommended placement of blocks

  int n = 0;
  int x, y, rotate;
  int score;
  RecNode **child = root->child;

  int rotMax, Xmin, Xmax;

  //For each rotation of block
  rotMax = rotateNum[nextBlock[root->level]]; //modified
  for (rotate = 0; rotate < rotMax; rotate++) { //modified
    //For each position possible to put the block
    getXRange(nextBlock[root->level], rotate, &Xmin, &Xmax); //modified
    for (x = Xmin; x <= Xmax; x++) { //modified

      if (CheckToMove(root->recField, nextBlock[root->level], rotate, 0, x)) {
        score = 0;

        //Make a new field where block is put
        //Store all the information
        child[n] = (RecNode*)malloc(sizeof(RecNode));
        CopyField(child[n]->recField, root->recField);
        child[n]->level = root->level + 1;

        y = GetY(child[n]->recField, 0, x, nextBlock[root->level], rotate);
        score += AddBlockToField(child[n]->recField, nextBlock[root->level], rotate,
y, x);

        score += DeleteLine(child[n]->recField);
        score += y * HEIGHT; //modified

        /* modified */
        //close to the root : NOT prune
        if (child[n]->level <= VISIBLE_BLOCKS/2) {
          score += modified_recommend(child[n]); //recursive call
        }
      }
    }
  }
}

```

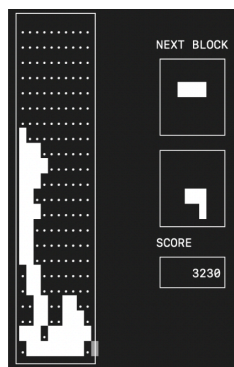
```

//pruning
else if (child[n]->level < VISIBLE_BLOCKS) {
    if (score >= VISIBLE_BLOCKS * VISIBLE_BLOCKS * VISIBLE_BLOCKS * 10) {
//pruning!!
//
        fprintf(testfp, "%d\n", score);
        score += modified_recommend(child[n]); //recursive call
    }
}
/* modified end */

if (max < score) {
    //Update the information of the block
    //with the accumulated maximum score
    max = score;
    root->recommendX = x;
    root->recommendY = y;
    root->recommendR = rotate;
    root->accumulatedScore = max;
}
n++;
}
}
return max;
}

```

modified\_recommend()에서는 pruning을 통해 일정 child 노드들만 만들도록 했다. 각 상황에 따라 트리를 정렬하여 best case만 택하는 경우가 있지만, 결국 모든 경우의 수를 계산해야 하고 정렬에 시간이 걸리는 점을 고려해 특정 점수 이상인 경우에만 트리를 확장하는 방법을 택했다. 또한 기존 추천 시스템의 경우 블록이 지속적으로 왼쪽에 쌓이며 기둥을 만드는 경우를 자주 발견했다. 이러한 현상이 나타나면 GameOver가 되기 쉽기 때문에 이를 해결하고자 했다.

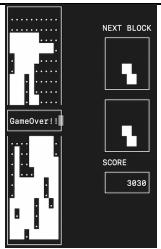
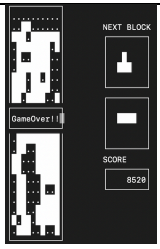
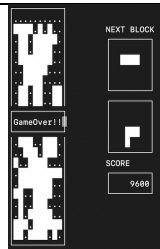
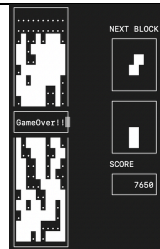
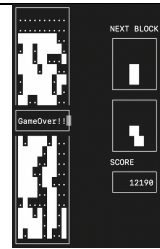


왼쪽에 기둥을 쌓는 현상

이는 점수가  $100 * (\text{지워진 라인의 개수}) * (\text{지워진 라인의 개수}) + 10 * (\text{블록이 닿은 면적})$  으로 계산되기 때문에 (지워진 라인의 개수)를 극대화하고 점수를 내지 못 할 경우 (블록이 닿은 면적) 점수를 얻기 위해 이러한 방식으로 추천한 것이라고 가설을 세우고, 블록을 아래쪽에 쌓는 것을 더 추천할 수 있도록 y좌표가 클수록(아래쪽일수록) 점수를 올려주는 방식을 택했다. 이를 통해 왼쪽에 기둥을 쌓는 현상이 확연히 개선되었다.

위와 같은 pruning과 y 값에 따른 잠정적 score 증가를 통해 효율성을 늘리는 효과를 가져올 수 있었다. 모든 경우를 고려하는 기존의 추천 시스템과 달리, 블록 모양에 따라 회전수와 가능한 x좌표를 고려해줌으로써 계산 횟수를 줄일 수 있었고, pruning을 통해 시간 복잡도 개선이 가능했다. 트리의 노드 개수를 줄임으로써 공간 복잡도 개선이 있었다고 예상이 되지만, 필드 정보를 온전히 모두 저장하는 방식은 그대로 유지했기 때문에 개선의 여지가 남아있다. 또한 pruning의 한계인 삭제된 경로 중에서 그 다음 노드까지 고려할 때 가장 큰 누적 점수를 갖게 되는 경우가 발생할 수 있다는 점을 그대로 한계로 가진다.

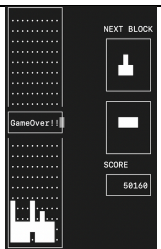
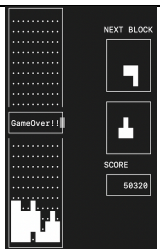
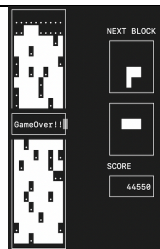
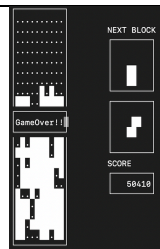
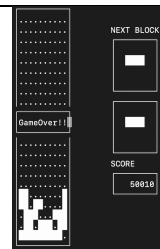
(초기) recommend() 함수를 통한 추천 시스템 분석

					
점수	3030	8520	9600	7650	12190
시간	124	224	187	163	256
점수/시간	24.4	38.0	51.3	46.9	47.6

점수/시간 평균: 41.6

recommend\_modified() 함수를 통한 추천 시스템 분석

(쉽게 게임오버되지 않아 50,000점을 초과한 경우 종료되도록 했다.)

					
점수	50160	50320	44550	50410	50010
시간	723	861	772	692	683
점수/시간	69.4	58.4	57.7	72.8	73.2

점수/시간 평균: 66.3

효율성을 점수/시간으로 평가하였을 경우 효율성이 개선되었다.

### 3. 기타 구현

```
int main(){
    . . .
    while(!exit){
        clear();
        switch(menu()){
            case MENU_PLAY: play(); break;
            case MENU_RANK: rank(); break; //week7
            case MENU_REC_PLAY: recommendedPlay(); break; //week8
            case MENU_EXIT: exit=1; break;
            default: break;
        }
    }
    . . .
}
```

menu에 MENU\_REC\_PLAY를 추가해 자동적으로 추천 시스템에 따라 블록을 놓으며 플레이되는 메뉴를 추가했다.

recommendedPlay() 함수는 Play() 함수를 참고해 작성했고, BlockDown() 함수 대신 RecBlockDown() 함수를 호출해 블록이 서서히 내려오는 것이 아니라 추천 위치에 놓이도록 했다. 또한 사용자에게 받을 수 있는 커맨드는 Q/q로 제한했다.

```
void recommendedPlay(){
#ifdef TESTTIME
    time_t start;
    time_t stop;
    double duration;
    start = time(NULL);
#endif
    int command;
    clear();
    act.sa_handler = RecBlockDown;
    sigaction(SIGALRM,&act,&oact);
    InitTetris();
    do{
        if(timed_out==0){
            alarm(1);
            timed_out=1;
        }

        command = GetCommand();
        if(RecProcessCommand(command)==QUIT){
            alarm(0);
            DrawBox(HEIGHT/2-1,WIDTH/2-5,1,10);
            move(HEIGHT/2,WIDTH/2-4);
            printf("Good-bye!!");
            refresh();
            getch();

            return;
        }
    }while(!gameOver);

    alarm(0);
    getch();
    DrawBox(HEIGHT/2-1,WIDTH/2-5,1,10);
    move(HEIGHT/2,WIDTH/2-4);
    printf("GameOver!!");
    refresh();
    getch();
    // newRank(score);
#ifdef TESTTIME
    stop = time(NULL);
    duration = (double)difftime(stop, start);
    FILE* fp;
    fp = fopen("time_test.txt", "w");
    printf(fp, "score:%d, time:%f\n", score, duration);
#endif
}
```

```
fclose(fp);  
#endif  
}
```

#### 4. 본 실험 및 숙제를 통해 습득한 내용 및 느낀점

재귀 함수를 통해 트리를 만들며 높은 점수를 만들 수 있도록 추천하는 알고리즘에 대해 공부할 수 있었다. 호출 stack이 쌓이다 보니 segmentation fault 발생에 주의해야 했다. 또한 모든 경우의 수를 고려하는 방식을 개선하기 위한 방법을 생각해 볼 수 있었고 pruning을 구현하는 방법을 공부할 수 있었다. Pruning 시에는 모든 경우의 수를 고려하는 것이 아니며, 수행 속도를 줄이기 위해 가지 치기를 많이 하면 할 수록 더 높은 score를 얻는 방법을 놓칠 수도 있으므로 적절한 균형점을 찾아야 한다는 것을 배울 수 있었다.