10주차 결과보고서

전공: 신문방송학과 학년: 3학년 학번: 20191150 이름: 전현길

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

구조체 _RecNode

```
typedef struct _RecNode {
  int lv, score;
  int x, y, r;
  char f[HEIGHT][WIDTH];
  struct _RecNode *next;
} RecNode;

const int blockRotateShape[] = {2, 4, 4, 4, 1, 2, 2}; // 블록별로 가능한 형태
RecNode *recRoot = NULL; // 추천 노드의 헤드 포인터
```

int recommend(RecNode *root)
modifiedRecommend(RecNode *root)와 다르게, 모든 경우의 수를 고려해 추천 트리를 구성하는 함수이다. 현재 블록을 포함한 VISIBLE_BLOCK개의 블록을 고려해서 모든 play sequence에 대한 경우의 수를 센다. recommend()의 작업은 다음과 같으며, ppt에 제시된 것과 크게 다르지 않다. 1. 블록의 가능한 형태, 놓일 수 있는 위치에 따라 RecNode 생성 2. 현재 lv이 최대 lv보다 작으면, 누적 score 계산을 위해 재귀적 함수 호출 3. 고려하는 lv과 최대 lv이 같으면, accumulatedScore에 점수 저장 4. 최대 누적 점수, 현재 얻어진 점수를 비교해서 블록의 회전수, 위치 갱신 단, 점수를 계산하는 방법을 바꾸었다. 필드의 상태를 확인하고 필드의 블록 아래에 빈 공간이 있을 경우, 필드에 블록이 높이까지 세워져 있을 경우 감점했다. 해당 감산식을 도입하자 훨씬 효율적으로 블록을 쌓게 되었지만, 비효율적으로 세로로 길게 쌓으려는 경향 역시 강해져, 가로로 낮게 블록을 쌓는 것에 가중치를 주기 위해 AddBlockToField의 가산값을 강화했다. 대체적인 **시간 복잡도는** ppt에서도 제시되었듯이 **O(34^VISIBLE_BLOCK)**이다. 하지만 VISIBLE_BLOCK마다 어느 정도의 연산이 이뤄지는지 파악하기 위해선 앞에 붙는 상수를 파악할 필요가 있다. 아래의 코드를 확인하면, 34라는 숫자는 WIDTH * BLOCK_ROTATE의 결과로 얻을 수 있는 숫자임을 알 수 있다. 그렇다면 34개의 각 노드마다 얼마나 연산이 이뤄지는지가 중요해진다. 먼저, 필드를 복사하는 데에 WIDTH * HEIGHT로 220회, CheckToMove() 함수의 경우 필드가 비어 있는 최악의 경우를 고려하면 BLOCK_HEIGHT * BLOCK_WIDTH * FIELD_WIDTH * FIELD_HEIGHT의 연산이 발생하므로 3,520회의 연산이 발생한다. 다음으로 필드에 따른 가중치를 계산하는 데에 다시 WIDTH * HEIGHT, 220회의 연산이 발생하며, AddBlockToField에서 16회, deleteLine에서 FIELD * HEIGHT^2회의 연산이 발생하지만 실질적으로 줄이 4개 이상 한꺼번에 삭제될 일은 없으므로 FIELD * HEIGHT * 4를 적용하면 880회의 연산이 최악의 경우 발생할 수 있다. 따라서 **각 노드별로 발생하는 연산 회수는 최악의 경우 220 + 3,520 + 220 + 16 + 880, 4,856회**이며, 상수 회수이더라도 VISIBLE_BLOCK이 크지 않은 상황에서 무시할 수 없는 수준의 연산이 발생한다. 공간 복잡도의 경우 34개의 자식 노드를 저장하는 대신 매번 동적 할당하고 해제하는 것을 반복하기 때문에 논리적으로는 각 RecNode 크기 * VISIBLE_BLOCK만큼의 메모리가 소모된다. 따라서 O(VISIBLE_BLOCK)이다.

```c
int recommend(RecNode *root) {
  int accumulatedScore = 0; // 최대 누적 점수를 저장하는 변수

  // rr, rx, ry → rec_rotate, rec_x, rec_y: 추천 블록의 회전수, 위치
  for (int rr = 0; rr < blockRotateShape[nextBlock[root->lv]]; rr++) { // 회전수
    for (int rx = -2; rx < 13; rx++) {
      if (!CheckToMove(root->f, nextBlock[root->lv], rr, -2, rx)) continue;

      RecNode *curNode = (RecNode *)malloc(sizeof(RecNode));
      curNode->lv = root->lv + 1;

      // root 노드의 필드 복사
      for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
          curNode->f[i][j] = root->f[i][j];

      int ry = -3;
      while (CheckToMove(curNode->f, nextBlock[root->lv], rr, ry + 1, rx)) ry++;

      curNode->score = 10000; // curNode->score가 음수가 될 수 있으므로 10000 초기화
      curNode->score += 5 * AddBlockToField(curNode->f, nextBlock[root->lv], rr, ry, rx);
      curNode->score += DeleteLine(curNode->f);

      for (int i = 0; i < WIDTH; i++)
        for (int j = 0; j < HEIGHT; j++)
          if (curNode->f[j][i] == 1)
            while (j < HEIGHT) {
              j++;
              if (curNode->f[j][i] == 0) curNode->score -= 50;
              curNode->score -= 10 * (HEIGHT - j); // 블록이 높을수록 점수 감소율 ↑
            }

      // 현재 lv이 최대 lv보다 작으면 누적 score 계산을 위해 재귀적 함수 호출
      if (curNode->lv < VISIBLE_BLOCKS)
        curNode->score += recommend(curNode);

      // 현재 블록의 점수가 가장 높다면 추천 블록의 위치 갱신
      if (curNode->score >= accumulatedScore) {
        accumulatedScore = curNode->score;
        recommendX = rx;
        recommendY = ry;
        recommendR = rr;
      }

      free(curNode);
      curNode = NULL;
    }
  }

  return accumulatedScore;
}
```

int modifiedRecommend(RecNode *root)

modifiedRecommend() 함수는 recommend() 함수와 달리 매 VISIBLE_BLOCK의 BLOCK_ROTATE마다 고득점을 받은 상위 PRUNING_LIMIT개의 블록을 뽑는 **가지치기를 수행**한다. 따라서 시간 복잡도는 **O((PRUNING_LIMIT*BLOCK_ROTATE)^VISIBLE_BLOCK)**이며, 공간 복잡도 역시 **O((PRUNING_LIMIT*BLOCK_ROTATE)^VISIBLE_BLOCK)**이다. 위의 recommend()와 달리 3개의 노드를 저장해야 하기 때문에 공간 복잡도는 이전보다 증가하게 되지만, 시간 복잡도의 개선에 비해 감수할 만한 증가로 보인다.

modifiedRecommend() 함수는 다음과 같이 가지치기를 수행한다. recommend() 함수에서도 수행되는 작업에 대해서는 구체적으로 설명하지 않는다.

1. 각 BLOCK_ROTATE마다 tNodes[10], nc를 선언한다. tNodes[]는 노드를 저장할 배열이며, nc는 노드의 현재 수를 저장하는 변수이다.
2. recommend()와 동일하게 각 노드에 대해 점수를 계산한다.
3. qsort(tNodes, 10, sizeof(RecNode), compare) 함수로 노드를 점수 기준 **내림차순 정렬**한다.
4. 공간 복잡도를 줄이기 위해, 앞의 3개 노드만 뽑아서 링크드 리스트 배열을 생성한다.
5. {} 밖으로 나가며 tNodes[] 배열이 차지하는 메모리를 해제한다.
6. 링크드 리스트를 따라가며 modifiedRecommend() 함수를 재귀 호출한다.
7. 최대 누적 점수, 현재 얻어진 점수를 비교하여 블록의 회전수, 위치를 갱신한다.

```
int modifiedRecommend(RecNode *root) {
  int accumulatedScore = 0; // 최대 누적 점수를 저장하는 변수

  // rr, rx, ry → rec_rotate, rec_x, rec_y: 추천 블록의 회전수, 위치
  for (int rr = 0; rr < blockRotateShape[nextBlock[root->lv]]; rr++) {

    RecNode* tempRecRoot = NULL;

    {

    RecNode tNodes[10];  // temp_nodes
    int nc = 0;          // node_count

    for (int rx = -2; rx < 13; rx++) {
      if (!CheckToMove(root->f, nextBlock[root->lv], rr, -2, rx))
        continue;

      // root 노드의 필드 복사
      for (int i = 0; i < HEIGHT; i++)
        for (int j = 0; j < WIDTH; j++)
          tNodes[nc].f[i][j] = root->f[i][j];

      int ry = -3;
      while (CheckToMove(tNodes[nc].f, nextBlock[root->lv], rr, ry + 1, rx))
        ry++;

      // 결정된 블록의 위치 복사
      tNodes[nc].x = rx, tNodes[nc].y = ry, tNodes[nc].r = rr;

      tNodes[nc].score = 1e7; // curNode->score가 음수가 될 수 있으므로 1 초기화
      tNodes[nc].score += 5 * AddBlockToField(tNodes[nc].f, nextBlock[root->lv], rr, ry, rx);
      tNodes[nc].score += DeleteLine(tNodes[nc].f);
```

```
  for (int i = 0; i < WIDTH; i++)
    for (int j = 0; j < HEIGHT; j++)
      if (tNodes[nc].f[j][i] == 1)
        while (j < HEIGHT) {
          j++;
          if (tNodes[nc].f[j][i] == 0) tNodes[nc].score -= 50;
          tNodes[nc].score -= 10 * (HEIGHT - j); // 블록이 높을수록 점수 감소율 ↑
        }

  nc++;
}

// 가지치기를 위해 점수 기준으로 정렬
qsort(tNodes, 10, sizeof(RecNode), compare);

// 최대 PRUNING_LIMIT 개만큼 가지치기

RecNode *prevNode = NULL;
for (int p = 0; p < PRUNING_LIMIT; p++) {
  // curNode를 동적 할당하고 값 복사
  RecNode *curNode = (RecNode*)malloc(sizeof(RecNode));
  for (int i = 0; i < HEIGHT; i++)
    for (int j = 0; j < WIDTH; j++)
      curNode->f[i][j] = tNodes[p].f[i][j];
  curNode->lv = root->lv + 1;
  curNode->score = tNodes[p].score;
  curNode->next = NULL;
  curNode->x = tNodes[p].x, curNode->y = tNodes[p].y, curNode->r = tNodes[p].r;

  // 연결
  if (prevNode != NULL && p < PRUNING_LIMIT - 1)
    prevNode->next = curNode;
  if (prevNode == NULL)
    tempRecRoot = curNode;
  prevNode = curNode;
}

} // 메모리 관리를 위해 가지를 친 후 tNodes 배열 해제
```

```
    // 재귀 수행
    RecNode* curNode = tempRecRoot;
    while (curNode != NULL) {
      // 현재 lv이 최대 lv보다 작으면 누적 score 계산을 위해 재귀적 함수 호출
      if (curNode->lv < VISIBLE_BLOCKS)
        curNode->score += modifiedRecommend(curNode);
      // 현재 블록의 점수가 가장 높다면 추천 블록의 위치 갱신
      if (curNode->score >= accumulatedScore) {
        accumulatedScore = curNode->score;
        recommendX = curNode->x;
        recommendY = curNode->y;
        recommendR = curNode->r;
      }

      RecNode *delNode = curNode;
      curNode = curNode->next;

      free(delNode);
    }
  }

  return accumulatedScore;
}
```

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

모든 경우를 고려하는 가장 기초적인 tree 구조와 비교했을 때, 시간 복잡도, 공간 복잡도가 각각 O(34^n)에서 PRUNING_LIMIT 상수를 3으로 잡았을 때 O(12^n) 수준으로 향상되었지만 모든 경우를 고려하지 않은 만큼 해당 경우가 최적해임을 보장하지 못하게 되었다. 뿐만 아니라 단순히 점수를 통해서 비교하는 대신 필드를 바탕으로 가중치를 계산했는데, 이것 역시 절대적인 기준이 아닌 모델의 주관적인 기준이므로 모델의 작동이 최적해임을 보장하기 어렵게 만드는 요소이다.

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

segmentation fault 오류, core dumped 오류와 부딪치면서 이전에 rankRoot 노드를 해제하지 않는 코드를 작성한 것을 디버깅하거나, 추천 노드의 root 노드의 할당, 해제의 위치를 생각하는 등 메모리 관리의 중요성을 체감했다. 평소에 작성하던 것보다 긴 코드를 작성하면서 경계값 초과 오류, NULL 포인터 접근 오류, 이미 해제된 메모리에 대한 접근 등 다양한 버그들

을 마주할 수 있었다. 대부분이 전체 프로그램이 어떻게 작동하는지에 대한 감각이 없는 상태에서 주먹구구식으로 짰던 것이 원인이였다.

이전에는 그리 와닿지 않았던 메모리 관리의 중요성, 디버깅이 용이한 코드의 중요성 등을 체감하는 기회가 되었으며, 추천 노드의 recommend() 함수를 작성하는 과정에서 이전에 알고리즘을 공부할 때 직관적으로 잘 이해되지 않았던 개념이었던 백트래킹을 이해할 수 있었다.