

# CSE3013 (컴퓨터공학 설계 및 실험 I) 테트리스 프로젝트

서강대학교 컴퓨터공학과 박수현 (20181634)

제 2분반; 담당교수: 서강대학교 컴퓨터공학과 장형수

## 1 설계 문제 및 목표

ncurses 라이브러리가 제공하는 리눅스 터미널 상에서의 GUI를 이용하여 테트리스 게임 프로그램을 제작하고 여기에 랭킹 시스템과 블록 배치 추천 기능을 추가 구현한다. 테트리스 게임은, 블록을 90도씩 반시계 방향으로 회전하거나 세 방향(좌, 우, 하)으로 움직여 필드에 쌓으면, 빙름없이 채워진 줄은 지워지고 그에 따른 스코어를 얻어 결과적으로 가장 높은 최종 스코어를 얻는 것이 목적이다.

프로그램을 실행하면 사용자는 메뉴를 선택할 수 있다. 1번을 선택하면 테트리스 게임을 플레이할 수 있고, 2번을 선택하면 랭킹 정보를 확인할 수 있고, 3번을 선택하면 블록 배치 추천 기능을 따라 플레이되는 모드로 테트리스 게임이 실행되고, 마지막으로 4번을 선택하면 프로그램이 종료된다.

구현 1주차에는 기본적인 게임의 로직을 구현한다. 구현 2주차에는 게임에 랭킹 시스템을 추가한다. 구현 3주차에는 블록을 놓는 위치를 컴퓨터가 추천해 주는 추천 시스템을 제작하며, 이를 통해 자동 플레이를 구현한다.

## 2 요구사항

### 2.1 설계 목표 설정

구현 1주차에는 테트리스 로직을 구현한다. 기본적인 테트리스 로직 구현을 위해 다음과 같은 항목들의 구현이 필요하다.

- 블록이 필드의 특정 위치에 놓여질 수 있는지 판단하는 로직
- 시간이 지남에 따라 블록을 움직이고, 떨어뜨리고, 필드를 업데이트하는 로직
- 현재 블록의 그림자 위치를 계산하고 표시하는 로직
- 점수를 계산하는 로직

구현 2주차에는 랭킹 시스템을 구현한다. 랭킹 시스템에는 다음과 같은 항목들의 구현이 필요하다.

- 랭킹 정보를 파일에 저장하고 읽어들이는 로직

- 새로운 랭킹 정보를 삽입하는 로직
- 존재하는 랭킹 정보를 삭제하는 로직
- 특정 범위 내의 랭킹을 쿼리하는 로직
- 특정 이름을 갖는 플레이어의 랭킹 정보를 쿼리하는 로직

구현 3주차에는 추천 시스템을 구현한다. 추천 시스템에는 다음과 같은 항목들의 구현이 필요하다.

- 추천 위치를 계산하는 로직
- 컴퓨터가 자동으로 게임을 진행하는 로직

추가로 추천 알고리즘을 개선하는 프로그램을 개발한다.

## 2.2 합성

**이진 탐색 트리** binary search tree 이진 탐색 트리는 이진 트리의 일종으로서, 각 노드의 왼쪽 서브트리는 해당 노드의 값보다 작은 노드들만, 오른쪽 서브트리는 해당 노드의 값보다 큰 노드들만을 포함하고 있는 트리를 말한다.<sup>1, pp. 286-287</sup> 이진 탐색 트리에서 원소 삽입과 삭제에 걸리는 시간은 일반적으로  $\mathcal{O}(\log n)$ 이다. 또한 트리를 중위 순회( $\mathcal{O}(n)$ )할 경우 항상 정렬된 상태로 순회하게 되는 특징이 있다. 따라서 랭킹을 정렬된 상태로 조회해야 하는 쿼리들을 수행하는 데 효율적이므로, 2주차 구현에 이진 탐색 트리 활용될 수 있다.

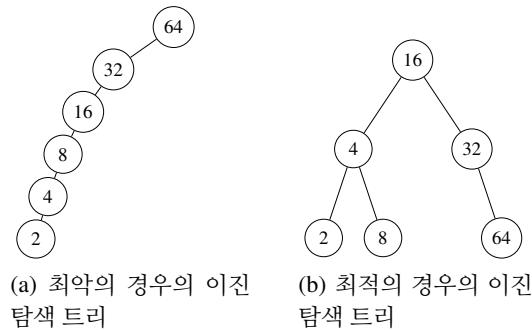


Fig. 1: 최악과 최적의 경우의 이진 탐색 트리

하지만 Figure 1a의 경우처럼 이진 탐색 트리의 균형이 맞지 않을 경우 최대  $n$ 개의 노드를 순회해야 하므로 삽입/삭제 연산의 시간 복잡도는 링크드 리스트와 같아지게 된다. 여기에서 자가 균형 이진 탐색 트리 self-balancing binary search tree 를 도입한다면 트리의 높이는 항상  $\lceil \log_2 n \rceil$ 가 되고, 최대  $\lceil \log_2 n \rceil$ 개의 노드만 순회하게 되므로 평균  $\mathcal{O}(\log n)$ 의 시간으로 원소를 삽입 및 삭제할 수 있게 된다.

자가 균형 이진 탐색 트리에는 AVL 트리, red-black 트리 등이 존재한다. C++ STL의 정렬 리스트 자료구조인 `std::set`는 일반적으로 red-black 트리로 구현되어 있으므로<sup>2</sup> 여기에서도 red-black 트리 (이하 RB 트리)를 이용한다.

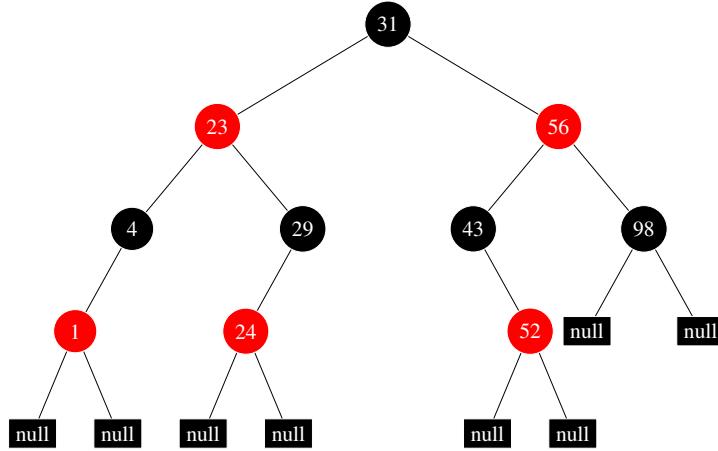


Fig. 2: Red-black 트리의 예시

RB 트리는 이진 탐색 트리에 대해 다음과 같은 성질들이 추가로 적용되는 트리이다.<sup>1</sup>, pp. 308-309

- 노드는 null일 수 있다.
- 각 노드의 색은 빨강, 검정 중 하나이다. 루트 노드나 null 노드는 항상 검정색이다.
- 어떤 노드가 빨강색일 경우 자식 노드는 모두 검정색이다.
- 임의의 노드에서 리프 노드까지의 모든 경로에는 같은 수의 검정 노드가 있다.

루트에서 리프 노드까지의 최단 경로가 모두 검정 노드로만 구성되어 있다고 했을 때, 최장 경로는 검정 노드와 빨강 노드가 번갈아 나오는 경로가 된다. 임의의 노드에서 리프 노드까지의 모든 경로에는 같은 수의 검정 노드가 있다고 했고, 빨강 노드의 자식은 무조건 검정 노드이므로 모든 경로에 대해 최장 경로의 거리는 최단 경로의 두 배 이상이 될 수 없게 된다.

**결정 트리**<sup>decision tree</sup> 결정 트리는 의사 결정 규칙과 결과를 트리 구조로 도식화한 트리이다. 결정 트리로 미래의 가능한 보드 상태들과 그 때의 결과를 저장하고 관리하면 볼 수 있는 미래의 상태 공간을 쉽게 탐색할 수 있으므로 이를 이용해 3주차의 추천 알고리즘을 구현할 수 있다.

**가지치기**<sup>pruning</sup> 가능한 모든 상태에 대한 완전 결정 트리는 그 구성과 탐색에 많은 시간과 공간을 필요로 하므로, 가능성이 떨어지는 노드들을 삭제하는 방법으로 가지치기를 할 수 있다.

Figure 3의 경우 각 노드마다 10개의 자식 노드가 있는 트리이다.  $n$ 개의 레벨에 대해 이 트리의 공간 복잡도는  $\mathcal{O}(10^n)$ 이다. 하지만 각 레벨마다 회색으로 표시된 노드들은 삭제하고 최대 4개의 노드만 남긴다면 공간 복잡도는  $\mathcal{O}(n)$ 이 되는 것을 확인할 수 있다.

가지치기의 구체적인 방법은 너비 우선 탐색과 쿼드 정렬 항목에 기술한다.

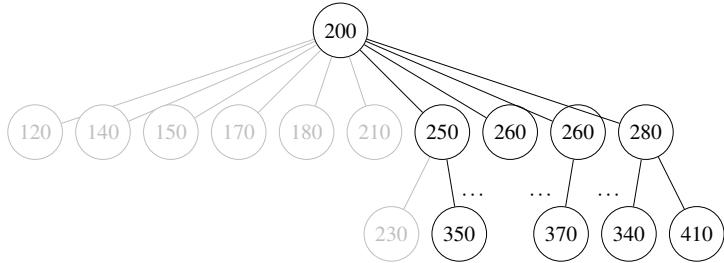


Fig. 3: 결정 트리에서 레벨당 최대 4개의 노드만 남기고 가지치기한 모습

**너비 우선 탐색breadth-first search** 너비 우선 탐색이란 그래프에서 임의의 노드에서 탐색을 시작해 가장 멀리 떨어져 있는 정점을 나중에 탐색하는 알고리즘이다.

트리에서 루트 노드를 레벨 0으로 두고, 레벨  $n$ 인 노드의 자깃 노드의 레벨을  $n+1$ 으로 둔다면 너비 우선 탐색을 사용할 경우 탐색 순서는 레벨이 작은 정점에서 높은 정점 순으로 진행되게 된다. 추천 알고리즘의 경우 현재 상태를 전부 탐색한 후 다음 블록이 놓이고 난 상태를 전부 탐색하고, 그 후에 그 다음 블록이 놓인 상태를 전부 탐색하는 식으로 동작하게 되는데 결정 트리의 각 레벨마다 노드 개수의 상한선을 유지하면서 동적으로 트리를 생성하려면 너비 우선 탐색법이 효율적이다.

결정 트리의 한 레벨을 탐색하고, 노드들을 효율적인 결과 순으로 정렬해 상한선만큼만 남기고 가지치기한 후, 남은 노드들에 대해서만 다음 레벨을 생성하고 탐색하는 식으로 결정 트리의 시공간 복잡도를 지수 복잡도에서 선형 복잡도로 개선할 수 있다.

**퀵 정렬quicksort** 가지치기를 할 때 상위 몇 개의 값을 가지는 노드를 골라내기 위해서는 정렬을 하는 것이 효율적이다. 퀵 정렬은 분할 정복법divide and conquer method을 이용한 정렬 알고리즘<sup>1, p. 181</sup>이다.

퀵 정렬을 하기 위해서는 우선 어떤 리스트에서 임의의 원소를 골라 피벗으로 설정하고, 피벗 앞에는 피벗보다 큰 값을, 뒤에는 피벗보다 작은 값을 놓는다. 그리고 피벗을 기준으로 왼쪽을 하나의 리스트로, 오른쪽을 하나의 리스트로 보고 각각의 리스트에 대해 재귀적으로 퀵 정렬을 수행한다.

버블 정렬이나 삽입 정렬은  $n$ 개의 원소를 정렬하는 데에  $\mathcal{O}(n^2)$ 의 시간이 걸리는 데 반해 퀵 정렬은 평균적으로  $\mathcal{O}(n \log n)$ 의 시간이 걸리는 효율적인 알고리즈다.<sup>1, p. 181</sup>

**발견법heuristic적 알고리즘** 완전 최적해를 찾는 것은 시공간적 한계에 의해 무리가 있으므로 현재의 여러 상황을 고려해 최적해일 확률이 높은 쪽으로 결정하는 알고리즘을 사용할 수 있다.

*Classic Tetris World Championship(CTWC)*은 여러 플레이어가 NES<sup>Nintendo Entertainment System</sup> 테트리스에서 게임 오버 직전까지 달성한 점수로 겨루는 토너먼트 대회이다. CTWC 결승 플레이어들은 왼쪽 9줄에는 블럭을 최대한 구멍 없이 쌓고, 남겨 둔 오른쪽 1줄에 I 블록을 끼워넣음으로서 4라인 클리어를 최대화시키는 전략을 사용하는 것을 확인할 수 있다.<sup>4</sup>

따라서 점수를 통해 추천 위치를 판단하는 것보다는 보드 상태에 대해 여러 기준을 두고, 이를 바탕으로 추천하는 것이 바람직함을 알 수 있다. 4라인 클리어를 최대화하기 위해서 다음과 같은 휴리스틱을 설정할 수 있다.

- 현재 4라인 클리어가 가능한가의 여부
- 현재 상태에서 클리어되는 줄 수

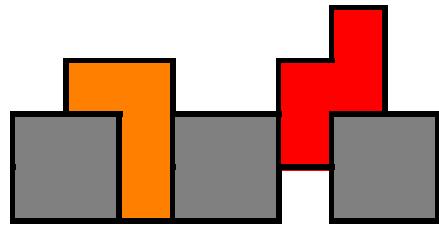


Fig. 4: 깊이 2의 골짜기

블록을 안정적으로 배치하려면 현재 필드에 구멍이 적어야 하며 높이가 평탄해야 하고, 비교적 낮은 위치에 블록을 놓는 게 바람직하다.

특히 이웃한 행들의 높이보다 낮은 높이를 갖는 행을 ‘골짜기’라고 정의한다면, Figure 4에서 보이듯이 높이 2 이상의 골짜기에는 L, J 블록만, 높이 3 이상의 골짜기에는 I 블록만 놓아야 필드 중간에 구멍이 생기지 않는다. 필드 중간에 구멍이 생기면 비효율적이므로 너무 높은 골짜기가 생기는 것은 가급적 피해야 좋을 것이다.

따라서 블록의 안정적 배치를 위해 다음과 같은 휴리스틱들을 추가로 설정할 수 있다.

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>- 현재 블록이 놓인 y 좌표</li> <li>- 모든 행의 높이의 합</li> <li>- 모든 행의 이웃한 행과의 높이의 차이의 합</li> <li>- 가장 높은 행과 가장 낮은 행의 높이의 차이</li> <li>- 가장 낮은 행의 높이</li> <li>- 구멍의 수</li> <li>- 구멍에서 바닥까지의 거리의 합</li> </ul> | <ul style="list-style-type: none"> <li>- 구멍에서 표면까지의 거리의 합</li> <li>- 가장 낮은 구멍의 y 좌표</li> <li>- 가장 높은 구멍의 y 좌표</li> <li>- 골짜기의 높이의 합</li> <li>- 높이 3 이상의 골짜기의 수</li> <li>- 모든 열에 대해 이웃하게 연결되어 있는 셀 요소들의 개수</li> </ul> |
|--|--|

높이가 낮을 때는 4라인 클리어를 하기 위해 스택을 쌓는 것이 효율적이지만, 높이가 높은데 I 블록이 안 나온다면 게임오버를 당할 확률이 높아진다. 이를 막기 위해 다음과 같은 휴리스틱을 추가로 설정한다.

- 현재 필드의 총 셀의 개수
- 현재 필드의 모든 셀의 높이의 합
- y 좌표가 10 이하인 모든 셀의 높이의 합

**담금질 기법 simulated annealing** 담금질 기법은 전역 최적화 문제에 대한 확률적 메타 알고리즘이다. ‘담금질 기법’이라는 이름의 유래는 금속 공학의 풀링 annealing의 오역에서 비롯되었는데, 풀링은 금속 재료를 가열한 다음 조금씩 냉각해 결정을 성장시켜 그 결함을 줄이는 작업이다. 열에 의해서 원자는 초기의 위치로부터 멀어져 에너지가 더욱 높은 상태로 추이되며, 금속을 천천히 냉각함으로써 원자는 초기 상태보다 내부 에너지가 한 층 더 극소인 상태를 얻을 가능성이 많아진다. 담금질 기법은 이런 현상과 비슷한 원리로 큰 탐색 공간에서 어떤 함수의 전역 최적해 global optima에 대한 근사를 진행하는 과정이다.<sup>3</sup>

예를 들어 어떤 벡터  $\mathbf{v} = \langle a, b \rangle$ 에 대해 보드의 점수가  $f(\mathbf{v}) = a \times (\text{줄 클리어 수}) - b \times (\text{문힌 구멍 수})$ 로 계산된다면, 기준으로 삼았을 때 가장 좋은 결과를 주는  $\mathbf{v}$ 를 찾는 것은 쉬운 일이 아니다. 여러 휴리스틱을 사용해 벡터의 차원이 늘어난다면 더 그러하다. 실제로 이 프로젝트의 추천 시스템은 18개의 휴리스틱을 사용하고 있어, 18차원 계수 벡터 형태의 최적해 산출이 필요하다. 이 기법은 해를 반복해 개선함으로써, 현재의 해 근방에 있는 해를 임의로 찾아가는 방법으로 최적인  $\mathbf{v}$ 의 근사값을 구할 수 있다.

‘에너지’를 일종의 비용 함수<sup>cost function</sup>으로 정의하자. 임의의 온도  $T$ 를 설정하고, 현재 상태의 에너지  $e_1$ , 현재 상태 주변의 계수들을 이용해 계산한 랜덤 상태의 에너지  $e_2$ 를 비교하여 확률  $p = \exp\left(\frac{e_1 - e_2}{kT}\right)$ 을 얻는다. 이 때  $k$ 는 상수이다. 여기서 0과 1 사이에서 얻은 무작위의 실수  $r$ 에 대해,  $p > r$  면 현재 상태를 새 상태로 바꾼다. 마지막으로  $k$ 에 온도 감률을 곱한다. 이 과정을  $T$ 가 임계온도 이상일 경우 계속 반복한다.

만약  $e_1 \geq e_2$ 라면 새로운 상태가 기존 상태보다 에너지가 낮음을 의미하므로 새로운 상태의 계수를 사용하는 것이 이득이다. 한편  $e_1 > e_2$ 일 경우 에너지가 더 높아지는데, 이 상태는 탐욕법<sup>greedy algorithm</sup>을 쓸 경우 무조건 무시되는 상태이지만, 에너지가 높은 상태를 거쳐야만 다시 작은 상태가 나오는 경우도 무시할 수 없으므로 위에서 계산한  $p$ 의 확률로 에너지가 높은 경우에도 탐색을 진행한다.

### 2.3 분석

**1주차:** 블록이 필드의 특정 위치에 놓여질 수 있는지 판단하는 로직 현재 블록에 대응하는 필드의 모든 칸에 대해, 하나 이상의 칸이 차 있을 경우 블록은 현재 위치에 놓일 수 없다. 따라서 이를 구현한다. 시간 복잡도는  $\mathcal{O}(1)$ 이다.

```
CHECKTOMOVE( $f, currentBlock, blockRotate, blockY, blockX$ )
1   for  $i = 0$  to 3
2     for  $j = 0$  to 3
3       if  $block[currentBlock][blockRotate][i][j] == 1$ 
4          $x = blockX + j$ 
5          $y = blockY + i$ 
6         if  $(0 \leq x < WIDTH \text{ and } 0 \leq y < HEIGHT) \neq \text{TRUE}$ 
7           return FALSE
8         if  $f[y][x] == 1$ 
9           return FALSE
10    return TRUE
```

**1주차:** 시간이 지남에 따라 블록을 움직이고, 떨어뜨리고, 필드를 업데이트하는 로직 블록을 떨어뜨리고 필드를 업데이트하는 과정은 다음과 같이 세분화된다.

- 블록을 움직이거나 떨어뜨릴 수 있는지 확인한다. - BLOCKDOWN, CHECKTOMOVE
- 블록을 움직이거나 떨어뜨리는 데 성공했을 경우 필드와 현재 블록을 다시 그린다. - DRAWCHANGE

- 블록을 더 떨어뜨릴 수 없을 경우 블록을 필드에 고정시킨다. - ADDBLOCKTOFIELD
- 블록이 새로 고정되었을 경우 필드에서 완성된 줄이 있는지 확인하고 줄을 지운다. - DELETELINE

필드와 현재 블록을 다시 그리는 것은 필드를 먼저 그려 이전 블록을 덮어씌우고 난 뒤 현재 블록을 현재 위치에 새로 그리는 것과 같다. DRAWFIELD와 DRAWBLOCK의 시간 복잡도가  $\mathcal{O}(WIDTH \times HEIGHT)$  이므로 전체 시간 복잡도도  $\mathcal{O}(WIDTH \times HEIGHT)$ 이다.

DRAWCHANGE(*f, command, currentBlock, blockRotate, blockY, blockX*)

```

1 // Erase current block
2 DRAWFIELD()
3 // Draw new block
4 DRAWBLOCK(blockY, blockX, currentBlock, blockRotate, ' ')

```

블록을 떨어뜨릴 수 있을 경우 떨어뜨리고, 불가능할 경우 블럭을 필드에 고정시킨다. 이 과정에서 새로운 다음 블럭을 만들고 점수도 업데이트한다. CHECKTOMOVE, DRAWCHANGE, ADDBLOCKTOFIELD, DELETELINE, DRAWNEXTBLOCK, DRAWFIELD, PRINTSCORE가 호출되는데 이 중 최악의 시간 복잡도를 보이는 메서드는 amortized  $\mathcal{O}(WIDTH \times HEIGHT^2)$ 의 시간을 사용하는 DELETELINE이므로, 전체 시간 복잡도도  $\mathcal{O}(WIDTH \times HEIGHT^2)$ 이다.

BLOCKDOWN(*sig*)

```

1 if CHECKTOMOVE(f, currentBlock, blockRotate, blockY + 1, blockX)
2     blockY = blockY + 1
3     DRAWCHANGE(f, command, currentBlock, blockRotate, blockY, blockX)
4 else
5     if blockY == 1
6         gameOver = TRUE
7     else
8         score = score + ADDBLOCKTOFIELD(f, currentBlock, blockRotate, blockY, blockX)
9         score = score + DELETELINE(f)
10        // Generate new block
11        nextBlock[0] = nextBlock[1]
12        nextBlock[1] = (Random integer in 0 .. 6)
13        DRAWNEXTBLOCK(nextBlock)
14        blockY = -1, blockX = (Center of field)
15        DRAWFIELD()
16        PRINTSCORE(score)

```

블럭을 필드에 고정시키려면 현재 블록에 대응하는 필드의 모든 칸에 대해 필드를 채워 주면 된다. 여기에서 인접 칸 수에 따른 점수를 계산해 반환한다. 시간 복잡도는  $\mathcal{O}(1)$ 이다.

```

ADDBLOCKTOFIELD( $f, currentBlock, blockRotate, blockY, blockX$ )
1  $adjacentCount = 0$ 
2 for  $i = 0$  to 3
3   for  $j = 0$  to 3
4     if  $block[currentBlock][blockRotate][i][j] == 1$ 
5        $ny = blockY + i$ 
6        $nx = blockX + j$ 
7       if  $0 < ny \leq HEIGHT$  and  $0 < nx \leq WIDTH$ 
8          $f[blockY + i][blockX + j] = 1$ 
9         if  $ny == HEIGHT - 1$  or  $f[ny + 1][nx] \neq 0$ 
10           $adjacentCount = adjacentCount + 1$ 
11 return  $adjacentCount \times 10$ 

```

꽉 찬 줄이 있는지 체크하고, 있을 경우 줄을 지우고 스코어를 증가시킨다. 완성된 줄은 y좌표마다 모든 x좌표의 칸이 차 있는가로 검사하며, 아래 칸부터 위 칸 방향으로 진행하면서 줄이 지워질 경우 위에 있는 칸들을 한 칸씩 아래로 내린다. 여기에서 지워진 줄 수에 따른 점수를 계산해 반환한다. 모든 y좌표마다 x좌표를 확인하는 데  $\mathcal{O}(WIDTH \times HEIGHT)$ 의 시간이 걸리고 만약 줄이 지워진다면 위의 블럭을 아래로 내리는 데 지워진 줄마다  $\mathcal{O}(WIDTH \times HEIGHT)$ 의 시간을 추가로 사용하므로 만약 지워지는 줄 수를  $t$ 라 하면 루프 횟수는  $(WIDTH + t \times WIDTH \times HEIGHT) \times HEIGHT = WIDTH \times HEIGHT \times (1 + t \times HEIGHT)$ 이다.

테트리스 게임의 특성상 블록당 4칸의 공간을 차지하므로  $n$ 블록이 떨어졌을 때 지워지는 줄 수의 상한은  $2.5n$ 이다. 따라서 분할상환분석으로 알고리즘을 분석하면  $\mathcal{T}(n) = WIDTH \times HEIGHT \times (1 + 2.5n \times HEIGHT)$ 이므로 전체 시간 복잡도는 amortized  $\mathcal{O}(WIDTH \times HEIGHT^2)$ 이다.

```

DELETELINE( $f$ )
1  $erased = 0$ 
2 for  $i = 0$  to  $HEIGHT - 1$ 
3    $flag = \text{TRUE}$ 
4   for  $j = 0$  to  $WIDTH - 1$ 
5     if  $f[i][j] == 0$ 
6        $flag = \text{FALSE}$ 
7     if  $flag == \text{TRUE}$ 
8        $erased = erased + 1$ 
9     for  $y = i$  downto 1
10      for  $x = 0$  to  $WIDTH - 1$ 
11         $f[y][x] = f[y - 1][x]$ 
12      for  $x = 0$  to  $WIDTH - 1$ 
13         $f[0][x] = 0$ 
14       $i = i - 1$ 
15 return  $100 \times erased^2$ 

```

**1주차:** 현재 블록의 그림자 위치를 계산하고 표시하는 로직 그림자를 그리기 위해 현재 블록을 얼마 만큼 아래로 내릴 수 있는지 CHECKTOMOVE를 이용해 판정한다. 시간 복잡도는  $O(HEIGHT)$ 이다.

GHOSTY( $y, x, blockID, blockRotate$ )

```

1 while CHECKTOMOVE( $field, nextBlock[0], blockRotate, y + 1, x$ )
2      $y = y + 1$ 
3 return  $y$ 
```

**1주차:** 점수를 계산하는 로직 ADDBLOCKTOFIELD과 DELETELINE에서 계산된 점수는 BLOCKDOWN에서 더해진다.

**2주차:** 랭킹 정보를 파일에 저장하고 읽어들이는 로직 C 표준 라이브러리 stdio.h의 입출력 함수 fprintf와 fscanf를 사용해 랭킹 파일 I/O를 수행한다. 삽입, 삭제 연산을 다루기 전에 우선 RB 트리의 성질들을 만족하게 하기 위해 연결 구조를 바꾸는 연산들을 정의한다. 모두  $\mathcal{O}(1)$  연산들이다.

BST-LEFT-ROTATE: 노드  $x$ 의 위치로  $x$ 의 오른쪽 자식을 옮긴다.

BST-LEFT-ROTATE( $x$ )

```

1 if  $x$  has left child
2     return
3      $y = x.right$ 
4      $x.size = x.left.size + y.left.size + 1$ 
5      $y.size = x.size + y.right.size + 1$ 
6      $x.right = y.left$ 
7     if  $y.left \neq \text{NULL}$ 
8          $y.left.parent = x$ 
9      $y.parent = y.parent$ 
10    if  $x.parent == \text{NULL}$ 
11         $root = y$ 
12    elseif  $x == x.parent.left$ 
13         $x.parent.left = y$ 
14    else
15         $x.parent.right = y$ 
16     $y.left = x$ 
17     $x.parent = y$ 
```

BST-RIGHT-ROTATE: 노드  $x$ 의 위치로  $x$ 의 왼쪽 자식을 옮긴다. BST-LEFT-ROTATE의 의사 코드에서  $left$ 와  $right$ 을 바꾸면 된다.

BST-TRANSPLANT:  $u$ 의 서브트리를  $v$ 의 서브트리로 대체한다.

```
BST-TRANSPLANT( $u, v$ )
1 if  $u.parent == \text{NULL}$ 
2      $root = v$ 
3 elseif  $u == u.parent.left$ 
4      $u.parent.left = v$ 
5 else
6      $u.parent.right = v$ 
7  $v.parent = u.parent$ 
```

$i$ 번째 순위의 노드를 가져오는 연산을 추가로 정의한다. DFS와 비슷한 식으로 탐색하며, 최대  $\lceil \log_2 n \rceil$  개의 노드만 순회하므로 시간 복잡도는  $\mathcal{O}(\log n)$ 이다.

BST-GET:  $i$ 번째 순위의 노드를 가져온다.

```
BST-GET( $u, i$ )
1 if  $i < u.left.size$ 
2     return BST-GET( $u.left, i$ )
3 elseif  $i > u.left.size$ 
4     return BST-GET( $u.right, i - u.left.size - 1$ )
5 else
6     return  $u$ 
```

**2주차: 새로운 랭킹 정보를 삽입하는 로직** 새 데이터를 갖는 빨강색 노드를 맨 밑에 삽입하고 트리의 성질에 맞도록 노드들을 재배치한다. BST-INSERT에서 탐색 과정에 최악의 경우 트리의 높이만큼 탐색하고, 또 BST-INSERT-REVALIDATE에서 최악의 경우 루트로 두 칸씩 되돌아면서  $\frac{\text{트리의 높이}}{2}$  만큼 탐색하므로 시간 복잡도는  $\mathcal{O}(\log n)$ 이다.

BST-INSERT( $u$ )

```

1   $y = \text{NULL}$ 
2   $x = \text{root}$ 
3   $u.\text{size} = 1$ 
4  while  $x \neq \text{NULL}$ 
5       $x.\text{size} = x.\text{size} + 1$ 
6       $y = x$ 
7      if  $u.\text{key} < x.\text{key}$ 
8           $x = x.\text{left}$ 
9      else
10          $x = x.\text{right}$ 
11   $u.\text{parent} = y$ 
12  if  $y == \text{NULL}$ 
13       $\text{root} = u$ 
14  elseif  $u.\text{key} < y.\text{key}$ 
15       $y.\text{left} = u$ 
16  else
17       $y.\text{right} = u$ 
18   $u.\text{left} = \text{NULL}$ ,  $u.\text{right} = \text{NULL}$ ,  $u.\text{color} = \text{RED}$ 
19  BST-INSERT-REVALIDATE( $u$ )
20   $\text{size} = \text{size} + 1$ 
```

BST-INSERT-REVALIDATE( $u$ )

```

1  while  $u.\text{color} == \text{RED}$ 
2      if  $u.\text{parent} == u.\text{parent}.\text{parent}.\text{right}$ 
3           $v = u.\text{parent}.\text{parent}.\text{right}$ 
4          if  $v.\text{color} == \text{RED}$ 
5               $u.\text{parent}.\text{color} = \text{BLACK}$ 
6               $v.\text{color} = \text{BLACK}$ 
7               $u.\text{parent}.\text{parent}.\text{color} = \text{RED}$ 
8               $u = u.\text{parent}.\text{parent}$ 
9          else
10             if  $u == u.\text{parent}.\text{right}$ 
11                  $u = u.\text{parent}$ 
12                 BST-LEFT-ROTATE( $u$ )
13                  $u.\text{parent}.\text{color} = \text{BLACK}$ 
14                  $u.\text{parent}.\text{parent}.\text{color} = \text{RED}$ 
15                 BST-RIGHT-ROTATE( $u.\text{parent}.\text{parent}$ )
16     else
17         Do the same with left and right exchanged
18   $\text{root}.\text{color} = \text{BLACK}$ 
```

**2주차: 존재하는 랭킹 정보를 삭제하는 로직** BST-GET를 이용해 노드를 가져오고 일반적인 트리에서 원소를 삭제하듯이 삭제한다. BST-DELETE 탐색 과정은 BST-GET과 같은 시간이 걸리고, BST-DELETE-REVALIDATE에서 최악의 경우 마찬가지로 루트로 두 칸씩 되돌아면서  $\frac{\text{트리의 높이}}{2}$  만큼 탐색하므로 시간 복잡도는  $\mathcal{O}(\log n)$ 이다.

```

BST-DELETE(i)
1  u = BST-GET(root, i)
2  y = u
3  originalColor = y.color
4  if u.left == NULL
5      x = u.right
6      BST-TRANSPLANT(u, u.right)
7  elseif u.right == NULL
8      x = u.left
9      BST-TRANSPLANT(u, u.left)
10 else
11     y = (node with minimum index in u.right)
12     x = y.right
13     if y.parent == u
14         x.parent = y
15     else
16         BST-TRANSPLANT(y, y.right)
17         y.right = u.right
18         y.right.parent = y
19     BST-TRANSPLANT(u, y)
20     y.left = u.left, y.parent = y, y.color = u.color
21     if originalColor == BLACK
22         BST-DELETE-REVALIDATE(x)

```

```

BST-DELETE-REVALIDATE(x)
1  while  $x \neq \text{root}$  and  $x.\text{color} == \text{BLACK}$ 
2      if  $x == x.\text{parent}.left$ 
3           $w = x.\text{parent}.right$ 
4          if  $w.\text{color} == \text{RED}$ 
5               $w.\text{color} = \text{BLACK}, w.\text{parent}.\text{color} = \text{RED}$ 
6              BST-LEFT-ROTATE( $x.\text{parent}$ )
7               $w = x.\text{parent}.right$ 
8          if  $w.\text{left}.\text{color} == \text{BLACK}$  and  $w.\text{right}.\text{color} == \text{BLACK}$ 
9               $w.\text{color} = \text{RED}$ 
10              $x = x.\text{parent}$ 
11         else
12             if  $w.\text{right}.\text{color} == \text{BLACK}$ 
13                  $w.\text{left}.\text{color} = \text{BLACK}, w.\text{color} = \text{RED}$ 
14                 BST-RIGHT-ROTATE( $w$ )
15                  $w = x.\text{parent}.right$ 
16                  $w.\text{color} = w.\text{parent}.\text{color}$ 
17                  $x.\text{parent}.\text{color} = \text{BLACK}, w.\text{right}.\text{color} = \text{BLACK}$ 
18                 BST-LEFT-ROTATE( $x.\text{parent}$ )
19                  $x = \text{root}$ 
20     else
21         Do the same with left and right exchanged
22      $x.\text{color} = \text{BLACK}$ 

```

**2주차:** 특정 범위 내의 랭킹을 쿼리하는 로직  $l$ 번째부터  $r$ 번째까지의 원소를 순회하도록 한다. 중위 순회한 결과값의  $l$ 부터  $r$ 번째 원소를 반환하면 된다. 시간 복잡도는  $\mathcal{O}(r)$ 이다. 다만 모든  $l$ 부터  $r$  번째 인덱스마다  $\mathcal{O}(\log n)$ 의 BST-GET을 호출하는 경우 시간 복잡도는  $\mathcal{O}((r-l)\log n)$ 이므로  $r$ 보다  $(r-l+1)\log n$ 이 현저히 작을 경우 이와 같은 방법으로 최적화할 수 있어 보인다. 아래 의사 코드는 최적화가 이루어져 있지는 않다.

```

BST-QUERY( $l, r$ )
1  if ( $1 \leq l \leq r \leq size$ )  $\neq$  TRUE
2      error Array index out of range
3  Allocate new stack  $s$ 
4   $curr = root$ 
5  while  $curr \neq NULL$ 
6      Push  $curr$  to  $s$ 
7       $curr = curr.left$ 
8  for  $i = 1$  to  $r$ 
9      if  $l \leq i$ 
10      $curr =$  pop element from  $s$ 
11     Add  $curr$  to query result
12  else
13      Pop element from  $s$ 
14       $curr = curr.right$ 
15      while  $curr \neq NULL$ 
16      Push  $curr$  to  $s$ 
17       $curr = curr.left$ 

```

**2주차: 특정 이름의 랭킹을 쿼리하는 로직** 모든 원소를 순회하면서 이름이 입력과 같은 원소들을 찾는다. 시간 복잡도는 완전 탐색에  $\mathcal{O}(n)$ , 길이  $l$ 의 이름에 대해 이름 비교에  $\mathcal{O}(l)$ 이 걸리므로 전체 시간 복잡도는  $\mathcal{O}(nl)$ 이다.

BST-QUERYBYNAME( $name$ )

```

1   $query\_entries = BST\text{-}QUERY(1, size of list)$ 
2  for  $i = 1$  to size of list
3      if  $name == list[i]$ 
4          Print  $list[i]$ 

```

**3주차: 추천 위치를 계산하는 로직** 추천 위치를 계산하는 로직은 다음과 같이 세분화된다.

- 결정 트리를 만드는 로직
- 보드 상태 점수를 계산하는 로직
- 결정 트리를 최적화하는 로직

현재 상태를 루트라고 하면, 다음  $n$ 개 블록에 대한 높이  $n+1$ 의 결정 트리를 만들 수 있다. 결정 트리의 각 노드에는 보드 상태와 점수가 포함된다.

공간 복잡도의 개선을 위해 탐색과 가지치기를 진행하면서 결정 트리를 만들어나간다. 가지치기를 하기 위해서는 노드를 정렬할 필요가 있다. 아래는  $n$ 개의 노드를 amortized  $\mathcal{O}(n \log n)$ 의 시간에 정렬하는 메서드이다. Swap의 의사 코드는 trivial하므로 기술하지 않는다.

```

QUICKSORT(array, start, end)
1  if start  $\geq$  end
2    return
3  // Set pivot to first element
4  i = start, j = end, p = start
5  while i < j
6    while array[i]  $\leq$  array[p] and i < end
7      i = i + 1
8    while array[j] > array[p]
9      j = j + 1
10   if i < j
11     Swap array[i] and array[j]
12 Swap array[p] and array[j]
13 QUICKSORT(array, start, j - 1)
14 QUICKSORT(array, j + 1, end)

```

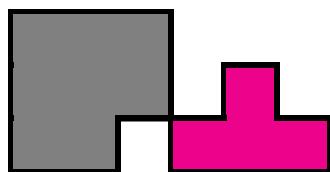


Fig. 5: 비효율적으로 배치된 T 블록

결정 트리를 만들 때, 블록을 놓을 수 있는 위치를 모두 계산해야 한다. 단순히 각 x좌표마다 블록이 놓일 수 있는 최대 y좌표를 계산하는 것은 Figure 5와 같은 상황에서 T 블록을 왼쪽으로 끼워넣어 효율적인 상황을 만들 수 있음에도 불구하고 끼워넣지 않는 비효율적인 상황만을 탐색하게 되므로, 이런 상황과 같이 조각을 밀어넣어 필드를 더 효율적으로 만들 수 있는 경우도 놓치지 않고 탐색하기 위해 BFS를 사용한다.

```

FINDAVAILABLESPOTS(field, nextBlock)
1   Allocate new queue q, available_spots
2   for rot = 0 to 4
3       for x = -4 to WIDTH + 4
4           if CHECKTOMOVE(field, nextBlock, rot, 1, x) == TRUE
5               Emplace (5, x + 4, rot) to q
6   while q.size > 0
7       (y + 4, x + 4, r) = q.front
8       Pop q
9       for all neighboring cell (ny, nx) to (y, x)
10      for current rotation state and next rotation state nr
11          if state (ny, nx, nr) not checked
12              if CHECKTOMOVE(field, nextBlock, nr, ny, nx) == TRUE
13                  if ny == y + 1 and nr == r
14                      Emplace (y + 4, x + 4, r) to available_spots
15                  else
16                      Emplace (ny + 4, nx + 4, nr) to q
17
18  return available_spots

```

위의 메서드는 블록을 필드 또는 바닥에 닿게 하면서 사용자가 키보드를 조작해 블록을 놓을 수 있는 모든 위치를 BFS로 찾아 준다. 전체 필드 공간을 3-tuple을 이용한 BFS로 탐색하면서 CHECKTOMOVE를 호출하므로 시간 복잡도는  $\mathcal{O}(WIDTH \times HEIGHT)$ 인데, 이는 각 *x*좌표마다 블록이 놓일 수 있는 최대 *y*좌표를 계산하는 시간 복잡도와 동일하다.

결정 트리를 만드는 것과 최적화하는 것은 동시에 이루어진다. RECOMMEND-BFS는 한 레벨의 노드들의 리스트를 받아 다음 노드들을 만든 후, 정렬하고, 상위 몇 개 노드들을 다시 RECOMMEND-BFS로 보내 주는 재귀적 메서드이다.

이 메서드는 끝 레벨이 아닐 경우 매번 호출될 때마다 *nodes*의 크기만큼 FINDAVAILABLESPOTS를 호출하며  $(nodes \times children)$ 의 크기만큼 BOARD-SCORE를 호출한다. *nodes*는 위의 레벨에서 가지치기 상한선 개만큼 들어올 수 있으며 FINDAVAILABLESPOTS의 공간 복잡도는  $\mathcal{O}(WIDTH \times HEIGHT)$ 이므로 *children*의 공간 복잡도도 그러하고, BOARD-SCORE의 시간 복잡도는  $\mathcal{O}(WIDTH \times HEIGHT^2)$ 이므로 노드 하나에 대해 *child*s를 만드는 시간은  $\mathcal{O}(WIDTH^2 \times HEIGHT^3)$ 이다.

한편 가지치기 상한선을 *m*이라 하면, 생길 수 있는 *child*s의 개수는 *node*마다  $\mathcal{O}(WIDTH \times HEIGHT)$  개씩이고 이후에 amortized  $\mathcal{O}(n \log n)$ 의 시간에 정렬되므로, 이를 수행하는 데 걸리는 시간은

$$\mathcal{O}(m \times WIDTH \times HEIGHT \times \log(m \times WIDTH \times HEIGHT))$$

이다. 따라서 이 메서드가 한 번 호출되려면

$$\mathcal{O}(m \times WIDTH^2 \times HEIGHT^3 + m \times WIDTH \times HEIGHT \times \log(m \times WIDTH \times HEIGHT))$$

의 시간이 필요하며, 이는  $\mathcal{O}(m \times WIDTH^2 \times HEIGHT^3)$ 과 같다.

이 작업을 트리의 높이만큼 재귀적으로 반복하므로 다음 블록의 개수를  $k$ 라 하면 전체 시간 복잡도는

$$\mathcal{O}(k \times m \times \text{WIDTH}^2 \times \text{HEIGHT}^3)$$

이다.

#### RECOMMEND-BFS( $level, nodes$ )

```

1  if  $level ==$  number of future blocks
2       $max\_score = -\infty$ 
3       $max\_node = \text{NULL}$ 
4      for  $node$  in  $nodes$ 
5          if  $max\_score > node.score$ 
6               $max\_score = node.score$ 
7               $max\_node = node$ 
8      if  $max\_node == \text{NULL}$ 
9          return  $\text{NULL}$ 
10     while  $max\_node.parent \neq \text{NULL}$  and  $max\_node.parent.parent \neq \text{NULL}$ 
11          $max\_node = max\_node.parent$ 
12     return  $max\_node$ 
13 Allocate new list  $childs$ 
14 for  $node$  in  $nodes$ 
15      $children = \text{FINDAVAILABLESPOTS}(node.field, nextBlock[level])$ 
16     while  $children.size > 0$ 
17         Allocate new node  $newChild$ 
18         Copy  $node.field$  to  $newChild.field$ 
19          $newChild.curBlockID = nextBlock[level]$ 
20          $(y+4, x+4, r) = children.front$ 
21          $newChild.recBlockX = x, newChild.recBlockY = y, newChild.recBlockRotate = r$ 
22         Pop  $children$ 
23          $board\_score = \text{BOARD-SCORE}(newChild.field, nextBlock[level])$ 
24          $newChild.parent = node, newChild.score = board\_score$ 
25         Add  $newChild$  to  $childs$ 
26     QUICKSORT( $childs, 0, \text{count of } childs - 1$ )
27      $pruned\_size = \min(\text{count of } childs, \text{tree prune limit})$ 
28     Allocate new list  $pruned$ 
29     for  $i = 0$  to  $pruned\_size$ 
30         Add  $childs[i]$  to  $pruned$ 
31     return RECOMMEND-BFS( $level + 1, pruned$ )

```

보드 점수 계산은 2.2에서 언급한 18개의 휴리스틱들로 이루어진다. 보드 점수  $c$ 는 각각 산출된 18개의 휴리스틱 값 벡터  $\mathbf{h}$ 에 18차원 계수 벡터  $\mathbf{x}$ 를 내적해 구할 수 있다. 식으로 나타낼 경우

$$c = \mathbf{h} \cdot \mathbf{x}$$

이다. 각각의 휴리스틱을 계산하는 방법은 간단하므로 생략한다.

**BOARD-SCORE**(*field, nextBlock, rot, blockY, blockX*)

- 1 Calculate heuristic value vector **h**
- 2 **return** **h** · **x**

이 메서드는 줄을 지우는 부분에서 DELETILINE과 같은  $\mathcal{O}(WIDTH \times HEIGHT^2)$  번의 동작을 하며, 다른 휴리스틱을 구하는 부분에서는 최대  $\mathcal{O}(WIDTH \times HEIGHT)$  번의 루프를 돈다. 따라서 전체 시간 복잡도는  $\mathcal{O}(WIDTH \times HEIGHT^2)$  이다.

**3주차: 컴퓨터가 자동으로 게임을 진행하는 로직** 자동 플레이 플래그를 만들고, 플래그가 설정되어 있을 경우 프로그램의 실행 동작을 바꾸도록 한다. 가령 자동 플레이 플래그가 설정되어 있고 현재 추천 위치에 블록을 놓을 수 있을 경우 BLOCKDOWN에서 블록을 추천 위치에 고정시며, 사용자의 입력을 모두 무시한다.

**3주차: 추천 알고리즘을 개선하는 프로그램** 추천 알고리즘을 효과적으로 개선하기 위해 2.2에서 소개한 담금질 기법을 사용한다.  $T = 1000$ ,  $k = 0.01$ 로 설정하며  $T < 10^{-3}$ 일 경우 충분히 최적해에 수렴했다고 보고 훈련을 종료한다. 비용 함수는 위에서 계산한 보드 점수이다.

프로그램을 실행하고 게임 오버가 되기를 기다린 뒤 결과를 직접 수집하는 것은 효율적인 방법이 아니다. 보유하고 있는 장비들은 다수의 CPU 코어를 사용할 수 있는 경우가 많은데, 프로그램을 동시에 실행할 수 있다면 컴퓨터의 자원을 최대한 활용할 수 있어 좋을 것이다. 또한 결과가 나오기까지 직접 기다리지 않고 추천 알고리즘을 개선하는 프로그램이 결과를 수집해서 통계까지 산출해 준다면 이상적일 것이다.

이를 위해 **JNI**<sup>Java Native Interface</sup>를 사용한다. JNI는 **JVM**<sup>Java Virtual Machine</sup> 위에서 실행되는 Java 프로그램이 네이티브 라이브러리를 참조할 수 있게 만든 만들어진 인터페이스이다.

C에서 멀티스레딩을 구현하는 것은 어렵지만 이미 존재하는 C 코드를 JNI가 사용할 수 있는 라이브러리로 포팅하는 것은 비교적 쉬우므로, 우선 ncurses.h에 의존하는 기능을 전부 지우고 main 함수를 계수 벡터 **x**를 받아 추천 알고리즘을 시뮬레이션하게 만든다. 그리고 게임 오버 직전에 떨어뜨린 블록 수, 점수, 블록 당 점수 등의 정보를 반환해 Java 프로그램에서 이를 받아 **x** 값에 반영하도록 한다.

Java는 여러 스레드<sup>thread</sup>를 생성해 동시에 여러 CPU 코어를 사용하기 쉽게 만들어진 언어이므로 이와 같은 프로그램을 효율적으로 작성할 수 있다. 또한 Java는 플랫폼 독립적이기 때문에 macOS에서 빌드한 .jar 파일을 Linux에서 그대로 실행시킬 수 있다. 다만, JNI에서 사용되는 네이티브 라이브러리는 macOS에서는 .dylib, Linux에서는 .so와 같은 식으로 플랫폼 종속적이므로 따로 빌드해 줘야 한다.

## 2.4 제작

2.3에서 설명한 이론과 실제 코드를 비교하고 분석한다.

**테트리스 게임 클라이언트** 테트리스 게임 클라이언트는 다음과 같은 파일로 구성된다.

파일 이름	기능
tetris.c, tetris.h	핵심적인 테트리스 게임 로직
ordered_list.c, ordered_list.h	랭킹 정보를 담는 노드와 정렬된 리스트의 RB 트리 구현 및 관련된 알고리즘
decision_tree.c, decision_tree.h	가능한 미래 상태들을 저장하는 노드와 결정 트리의 구현 및 관련된 알고리즘
queue.c, queue.h	가능한 조각 놓는 위치를 판정하기 위해 BFS에서 사용하는 큐와 가능한 상태를 나타내는 3-tuple의 구현 및 관련된 알고리즘
makefile	프로젝트의 빌드 정보

**테트리스 게임 클라이언트 1주차** 1주차에 구현된 함수들은 다음과 같다.

**int CheckToMove(char f[HEIGHT][WIDTH], int currentBlock, int blockRotate, int blockY, int blockX)** : 주어진 위치에 블록을 놓을 수 있는지 확인한다. 구현된 코드는 설계한 의사 코드를 그대로 구현하였다.

**void DrawChange(char f[HEIGHT][WIDTH], int command, int currentBlock, int blockRotate, int blockY, int blockX)** : 필드에서 바뀐 부분을 화면에 업데이트한다. 구현된 코드는 설계한 의사 코드를 그대로 구현하였다.

**void BlockDown(int sig)** : 블럭이 일정 시간마다 내려갈 수 있도록 한다.

3주차의 자동 플레이에 대응하기 위해 flag\_autoplay가 설정되어 있을 경우 현재 블록을 무조건 추천 블록 위치에 가져다 놓는 루틴이 추가되어 있다(줄 425–429). 또한 추천 알고리즘을 강도 있게 훈련하기 위해 디버그 옵션 DEBUG\_GARBAGE\_MODE이 설정되어 있으면 매 15개의 블록을 떨어뜨릴 때마다 한 칸만 비어 있는 쓰레기 열을 밑에서 추가하도록 수정하였다(줄 443–457).

다음 블록의 개수가 2개 이상으로 설정되어 있더라도 그만큼 새로운 블록을 생성하도록 수정되었으며(줄 459–462), 현재 블록이 고정되는 순간 다음 블록에 대한 추천 위치를 계산하도록 수정하였다(줄 470–471). 마지막으로 전술한 바와 같이 테트리스 추천 알고리즘의 목표는 4라인 클리어를 최대화하는 것이므로, 이를 현재 추천 알고리즘이 잘 수행하고 있는지 확인하기 위해 DEBUG가 설정되어 있으면 현재 게임에 대해 각 라인 클리어 별 빈도 수를 확인할 수 있는 통계를 표시하는 기능을 추가하였다(줄 476–501).

**int AddBlockToField(char f[HEIGHT][WIDTH], int currentBlock, int blockRotate, int blockY, int blockX)** : 필드의 특정 위치에 블록을 고정시킨다. 인접한 칸 수에 비례해 계산된 점수를 반환한다. 구현된 코드는 설계한 의사 코드를 그대로 구현하였다.

**int DeleteLine(char f[HEIGHT][WIDTH])** : 필드에서 완성된 줄이 있으면 지우고 점수를 계산해 반환한다. DEBUG가 설정되어 있으면 지워진 줄 수에 대한 통계를 계산하기 위해 통계 값을 증가시키는 코드가 추가되었고(줄 550–555), 이외의 부분은 설계한 의사 코드를 그대로 구현하였다.

**int GhostY(int y, int x, int blockID, int blockRotate)** : 주어진 위치에서 그림자의 y 좌표를 계산한다. 구현된 코드는 설계한 의사 코드를 그대로 구현하였다.

테트리스 게임 클라이언트 2주차 2주차에 구현된 함수들은 이름으로 쿼리하는 부분을 제외하고 모두 의사 코드를 그대로 구현하였다.

이름으로 쿼리하는 부분은 따로 함수가 만들어지지 않았고, tetris.c의 줄 650–672에 구현되었다. 의사 코드에 추가해 flag를 플래그로 활용하여 검색한 결과가 없을 경우 검색 실패 메시지를 표시하도록 수정되었다.

테트리스 게임 클라이언트 3주차 3주차에 구현된 함수들은 다음과 같다.

**void decision\_tree\_node\_quicksort(decision\_tree\_node \*array, int start, int end)** : 결정 트리 노드 배열을 정렬한다. 점수를 2-tuple로 관리하면서, 점수 비교를 위해 tuple의 두 값을 더해 비교하는 점이 이론과 달라졌다(줄 39–40, 43–44).

가능한 위치들을 찾는 함수는 따로 구현되지 않았고, tetris.c의 줄 17–19와 951–1010에 구현되었다.

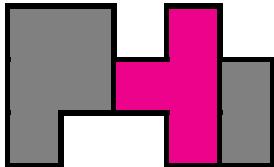


Fig. 6: 회전 불가능한 T

이론을 구체화해 visit 여부를 체크하는 3차원 배열을 만들어 방문 여부를 체크할 수 있도록 했다. visit을 새로 만드는 것은 시간이 걸릴 것으로 예상해, 전역 변수로 두고 새로 탐색할 필요가 있을 때마다 memset으로 초기화하였다. 인접한 칸을 체크하는 부분은 dx, dy 배열을 만들어 for 루프로 쉽게 체크할 수 있도록 하였다.

이론에서 틀린 부분이 있었는데, Figure 6의 경우 T 블록을 회전해 왼쪽 안으로 밀어넣는 것이 불가능하다. 이론에서는 이동과 회전을 동시에 고려해, 이런 상태에서 T 블록을 회전해 왼쪽 안으로 밀어넣는 수를 추천하는 상황이 발생했다. 이 오류는 이동과 회전을 따로 생각하게 코드를 수정하면서 해결되었다(줄 971–1009).

또한 회전 시 블록이 놓일 수 있는 위치를 판별하는 로직을 변경하였다(줄 1004–1008). 만약 회전을 했는데 필드에 놓일 수 없다면 회전된 상태는 판별하지 않는 것이 바람직하다.

이외에도 3-tuple을 사용하는 등 자료구조 상의 약간의 구체화가 있었으나 알고리즘 자체는 설계한 의사 코드에서 크게 변하지 않았다.

**decision\_tree\_node recommend\_bfs(int level, decision\_tree\_node \*parents, int parent\_count)** : 블록의 최적 추천 위치를 찾기 위해 너비 우선 탐색을 수행한다. 대부분이 의사 코드 그대로 구현되었으나, 구체화된 점들이 일부 존재한다.

max\_score는  $\infty$ 가 아닌, 임의의 큰 수 987654321로 정의되었다. childs는 동적 배열로 정의되었다. 가능한 위치들을 찾는 함수는 구현하는 대신 recommend\_bfs 함수 내부에 코드로 포함되었다. 또한 필드 점수가 2-tuple 구조를 사용하도록 하였는데, boardScore와 propagatedScore를 따로 두어 boardScore의 경우 현재 노드에만 반영하지만 propagatedScore의 경우 자식 노드에 누적되도록 하였다. 이유는 후술한다.

`score_pair boardScore(char f[HEIGHT][WIDTH], int nextBlock, int rot, int blockY, int blockX)` : 필드의 점수를 계산한다. 실제 구현은 다음과 같다. 휴리스틱을 계산하는 부분은 비슷한 루프를 한 루프로 묶은 구현 상의 차이(줄 832–866)를 제외하고는 의사 코드와 같다.

여기에서 가장 중요한 부분은 `boardScore`와 `propagatedScore`가 따로 계산된다는 점인데, `boardScore`는 현재 보드 상황에 매긴 점수인 반면 `propagatedScore`는 현재 동작에 매긴 점수이다.

만약 다음 블록 3개의 조합이 차례로 3줄, 4줄, 1줄을 지울 수 있는 경우와 차례로 1줄, 1줄, 2줄을 지울 수 있는 경우가 있다고 생각하자. 현재 동작에 대한 점수가 누적되지 않는다면 추천 알고리즘은 최종 노드에서 할 수 있는 최선의 선택인 차례로 1줄, 1줄, 2줄을 지우는 경우를 택할 것이다. 하지만 차례로 3줄, 4줄, 1줄을 지우는 경우가 더 효율적이다. 이런 상황을 방지하기 위해 누적되는 점수의 개념을 따로 도입했다.

**테트리스 훈련 프로그램** 테트리스 훈련 프로그램은 JVM 언어인 Kotlin(.kt)과 Java(.java)로 구성되어 있다. 빌드를 위해 Gradle을 사용했으며, `build.gradle`은 Gradle 빌드 스크립트이다. 전체 프로젝트는 다음과 같은 파일로 구성된다.

파일 이름	기능
<code>com.shiftph.tetris.ai.main.kt</code>	훈련 진행
<code>com.shiftph.tetris.ai.constraints.kt</code>	계수 벡터에 대한 데이터 클래스 정의
<code>com.shiftph.tetris.ai.result.kt</code>	실행 결과에 대한 데이터 클래스 정의
<code>com.shiftph.tetris.ai.TetrisJNI.java</code>	네이티브 라이브러리로 포팅한 테트리스 프로그램과 Java 훈련 프로그램 사이의 JNI 인터페이스
<code>libtetris.so</code>	Linux x86_64에서 빌드한, JNI로 포팅된 테트리스 프로그램의 네이티브 라이브러리
<code>libtetris.dylib</code>	macOS x86_64에서 빌드한, JNI로 포팅된 테트리스 프로그램의 네이티브 라이브러리
<code>build.gradle</code>	프로젝트의 빌드 정보

`javac -h . TetrisJNI.java`를 통해 C 프로그램을 JNI로 포팅하기 위한 헤더 파일 템플릿을 생성할 수 있다.<sup>5</sup> 생성된 헤더 파일에는 다음과 같은 코드가 있는데, 이를 구현한다.

```
JNIEXPORT jstring JNICALL Java_com_shiftph_tetris_lai_TetrisJNI_run
(JNIEnv *, jobject, jobjectArray);
```

이 형식들은 JDK의 `jni.h`에 정의된 형식들이다. `jobjectArray`가 `String[] constraints`를 의미 한다.

ncurses 기능을 전부 제거한 C 파일에는 다음과 같이 구현한다.

```
JNIEXPORT jstring JNICALL Java_com_shiftph_tetris_lai_TetrisJNI_run
(JNIEnv *env, jobject object, jobjectArray stringArray) {
```

```

for (int i = 0; i < 18; i++) {
    feature_weights[i] = strtold((*env)>GetStringUTFChars(env, (jstring)
        →  ((*env)>GetObjectArrayElement(env, stringArray, 0)), 0), NULL);
}
srand((unsigned int)time(NULL));
recommendedPlay();
char buffer[1000];
sprintf(buffer, "%d %d %lf\n\n", score, droppedBlocks, (double) score /
    →  droppedBlocks);
return (*env)>NewStringUTF(env, buffer);
}

```

이 함수는 Java 문자열의 배열을 받아 C 문자열로 해석한 뒤에 이를 long double로 해석하고, 이를 가중치 벡터로 적용하여 프로그램을 실행한 뒤 실행이 끝나면 점수와 게임 오버 직전까지 떨어뜨린 블록 수와 블록 당 점수를 문자열로 반환한다.

최적의 값을 찾는 알고리즘은 다음과 같다. 현재까지 최적의 벡터를 평균으로 하고 분산을 50으로 하는 가우시안 분포에서 주변의 벡터를 하나 고른다. 이 벡터를 바탕으로 프로그램을 16번 실행하고 얻은 결과의 12.5% 절사평균을 구한다. 마지막으로 설계한 대로 확률  $p$ 를 산출해, 이 확률에 따라 값을 계속 업데이트하면서 진행한다. 0으로 초기값을 설정하면 훈련이 오래 걸릴 것을 우려해, 시행착오를 통해 얻은 다음 벡터를 초기값으로 설정하였다. 단, 4줄을 동시에 없애는 휴리스틱에 대한 가중치는 항상  $-10^9$ 로 고정시켰다.

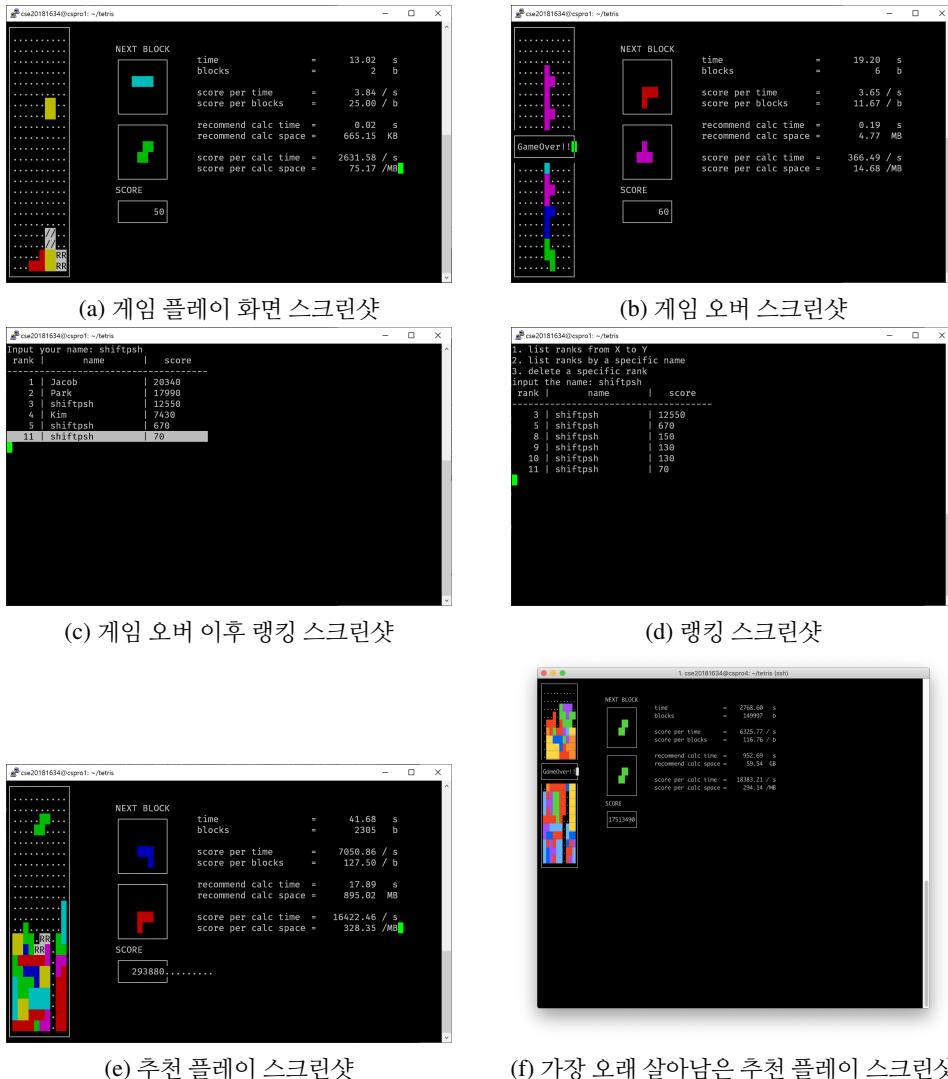
$$\mathbf{x}_0 = 100 \langle 25, 150, -45, 25, -100, -180, \\ 160, 190, -24, 25, 30, 60, \\ 80, -240, 20, 96, 75 \rangle$$

만들어진 바이너리를 AWS EC2 인스턴스에서 실행시켜 봤으나, 문제가 발생하였다. ThreadPool을 이용해 여러 스레드를 만들어 CPU의 자원을 최대한 활용하려 했으나, 어째서인지 단일 스레드로 높은 점수를 내는 벡터가 다중 스레드에서는 20블록도 넘기지 못하고 게임 오버를 당했다. 정확한 원인을 찾지 못해 결국 단일 스레드만을 활용하도록 프로그램을 수정해야 했다.

둘째는 프로그램의 실행 시간이 너무 오래 걸린다는 것이었다. 초기 벡터의 경우만 하더라도 약 5,000블록을 진행하고도 게임 오버가 되지 않는 경우가 있었다. 이 문제는 훈련 목표에서 게임 오버 직전까지 떨어뜨린 블록 수를 빼고 순수히 블록 당 얻은 점수로만 설정함으로써 해결하였다. 그럼에도 불구하고 프로젝트를 진행할 수 있는 시간이 한정되어 있어서 훈련 과정을 제대로 진행하지 못했다. 마지막으로 얻은 최적해는 다음과 같았다.

$$\mathbf{x} = \langle 2495, 14835, -4424, 2510, -10355, -18376, \\ 16388, 18910, -2315, 2528, 3160, 6203, \\ 7942, -23070, 2111, 9615, 7624 \rangle$$

## 2.5 시험



## 2.6 평가

Figure 7a은 보통 플레이 모드의 스크린샷이다. 블록 조작, 스코어 가산, 그림자, 추천 기능, 다음 블록을 생성하고 보여주는 기능 등 핵심적인 기능들이 모두 잘 동작하였다.

Figure 7b은 게임 오버 스크린샷이다. 게임 오버 조건을 만족하였을 때 정상적으로 게임이 종료되었다.

Figure 7c은 게임 오버 이후 랭킹을 입력하면 보이는 화면의 스크린샷이다. 랭킹 정보를 문제 없이 읽고 쓸 수 있었다.

Figure 7d은 랭킹 기능에서 이름을 기준으로 퀴리한 결과의 스크린샷이다. 랭킹 범위로 검색, 랭킹 삭제, 이름으로 퀴리 모두 정상적으로 작동하였다.

마지막으로 Figure 7e은 추천 플레이 기능의 스크린샷이다. 추천 플레이 기능은 미래 블록 3개를 고려할 때 초당 약 60개의 블록을 떨어뜨리도록 수정되었다.

추천 플레이 기능은 잘 동작하였으나, 점수 표시 칸 오른쪽에 알 수 없는 잔상이 남았다. 오른쪽의 통계를 그리느라 남은 잔상으로 추정되나 정확한 원인을 규명할 수 없어서 해결하지 못했다.

Figure 7f는 가장 오래 살아남은 추천 플레이의 결과 스크린샷이다. 3블록 앞을 볼 때 가장 오래 살아남았을 경우 149,997블록을 떨어뜨리고 게임오버되었으며, 점수는 17,513,490점이었다. 초당 54개의 블록을 떨어뜨렸고 총 실행 시간은 46.1분이었다.

추천 알고리즘의 복잡도와 효율성을 검증하기 위해 다음과 같은 평가를 수행하였다.

- 다른 조건은 그대로 두고, 미래에 보는 블록 수에 따라 시뮬레이션을 12번 진행해 추천 알고리즘의 누적 계산 시간과 누적 사용 공간, 점수의 평균과 표준편차를 각각 구한다.
- 위 평가를 트리 가지치기를 하지 않고 진행한다.
- 다른 조건은 그대로 두고, 트리 가지치기 이후에 남기는 노드의 상한에 따라 시뮬레이션을 12번 진행해 추천 알고리즘의 누적 계산 시간과 누적 사용 공간, 점수의 평균과 표준편차를 각각 구한다.

단, 프로그램이 쉽게 게임 오버되지 않는 점을 고려해 1,000번째 블록을 떨어뜨리는 시점에서 강제로 게임이 종료되도록 한다. 또한 1,000번째 블록 이전에서 게임 오버를 당할 경우 1,000블록 당 시간, 공간, 점수를 계산한다.

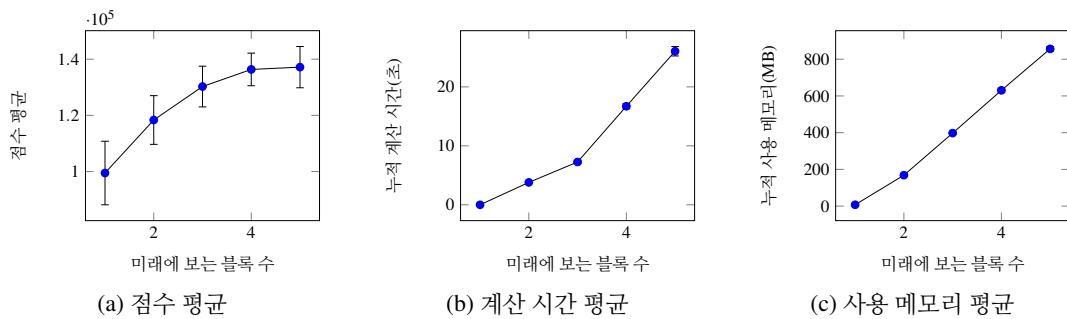
1번째 평가의 결과는 다음과 같았다. 남기는 노드의 상한은 모든 경우에서 32였다.

미래에 보는 블록 수	1*	2	3	4	5
점수 평균	99,470	118,352	130,272	136,363	137,178
점수 표준편차	11,317	8,675	7,253	5,806	7,357
누적 계산 시간 평균 (초)	< 0.01	3.80	7.26	16.71	26.04
누적 계산 시간 표준편차 (초)	< 0.01	0.14	0.25	0.43	0.80
1초당 점수	$\infty$	31,145	17,944	8,161	5,268
누적 사용 메모리 평균 (MB)	7.29	168.31	397.50	630.41	856.28
누적 사용 메모리 표준편차 (MB)	0.22	3.82	4.97	12.44	12.78
1MB당 점수	13,645	703	327	216	160

\* 미래에 1블록만 보는 경우는 평균적으로 456번째 블록이 내려온 직후 게임 오버 당했다. 나머지 경우에는 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않았다.

1초당 점수와 1MB당 점수는 미래에 보는 블록 수가 증가할수록 오히려 감소했지만, 미래에 1블록만 보는 경우에서 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않은 경우가 없었던 것으로 미루어 보아 미래에 많은 블록을 볼수록 안정성이 높아진다고 추측할 수 있다. 따라서 안정성과 효율의 균형이 잘 맞는 블록 수를 선택해야 할 것이다.

그래프는 다음과 같았다.



점수는 미래에 보는 블록 수가 증가할수록 어떤 값에 수렴하는 형태를 보였으며, 계산 시간은 1-3 블록, 3-5블록은 각각 선형의 경향성을 보이고 있다. 3블록을 기점으로 누적 계산 시간이 꺾였는데, 요인은 잊은 캐시 미스 혹은 소팅 시간의 증가라고 추측된다. 사용 메모리는 선형의 경향성을 보이고 있다. 이는 가지치기를 했을 때 공간 복잡도가 미래에 보는 블록의 수에 비례함을 뒷받침한다.

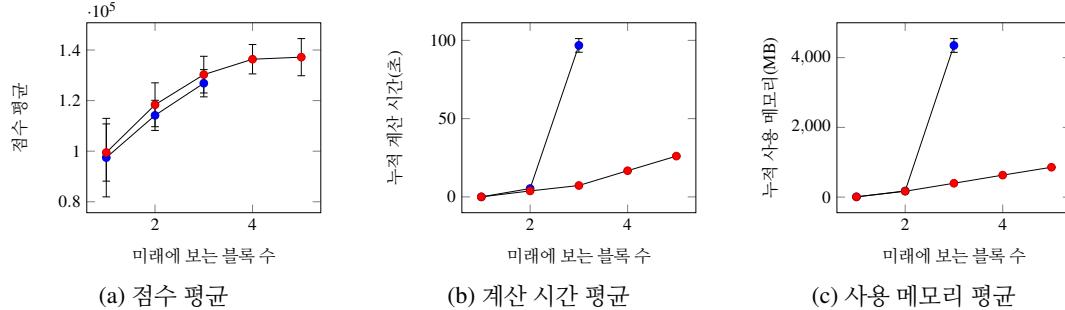
2번째 평가의 결과는 다음과 같았다.

미래에 보는 블록 수	1*	2	3	4**	5**
점수 평균	97,456	114,163	126,863	-	-
점수 표준편차	15,523	5,958	5,383	-	-
누적 계산 시간 평균 (초)	< 0.01	5.33	96.78	-	-
누적 계산 시간 표준편차 (초)	< 0.01	0.22	4.36	-	-
1초당 점수	$\infty$	21,479	1,311	-	-
누적 사용 메모리 평균 (MB)	7.40	176.72	4349.17	-	-
누적 사용 메모리 표준편차 (MB)	0.17	3.52	197.27	-	-
1MB당 점수	13,170	646	29	-	-

\* 미래에 1블록만 보는 경우는 평균적으로 477번째 블록이 내려온 직후 게임 오버 당했다. 나머지 경우에는 1,000번째 블록이 내려올 때까지 게임 오버를 당하지 않았다.

\*\* 미래에 4블록, 5블록을 보는 경우는 1,000번째 블록이 떨어질 때까지의 시간이 너무 오래 걸려 테스트하지 못했다.

가지치기를 하지 않을 경우 사용하는 시간과 공간이 기하급수적으로 늘어남을 알 수 있다. 그래프는 다음과 같았다. 붉은색 점은 가지치기를 했을 때의 결과이다.



점수 평균에서는 가지치기를 했을 때의 결과가 약간 더 좋았으나 의미 있는 차이가 보이지 않았다. 또한 가지치기를 하지 않았을 때 미래에 3블록을 보는 경우 기하급수적으로 계산 시간과 사용 메모리가 늘어났음을 확인할 수 있다. 종합해 보면 가지치기는 시간, 공간적 측면 모두에서 결정 트리를 효율적으로 사용할 수 있게 해 주는 수단이라는 것을 확인할 수 있다.

마지막으로 2번째 평가의 결과는 다음과 같았다. 미래에 보는 블록의 수는 3개로 같았다.

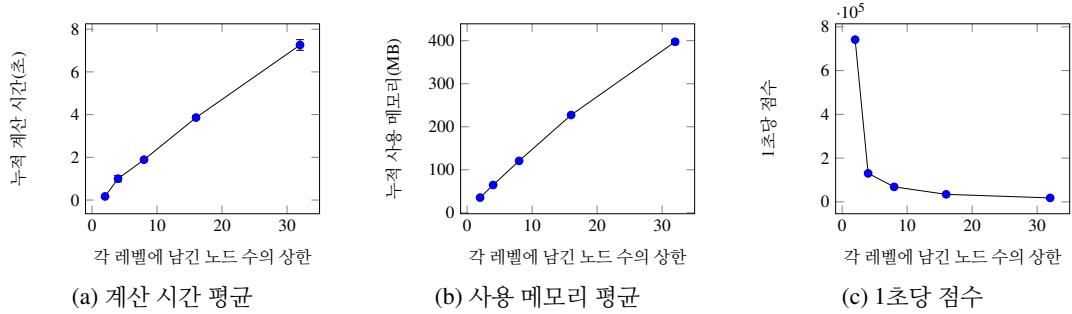
각 레벨에 남긴 노드 수의 상한	2*	4	8	16	32
점수 평균	126,047	130,127	129,551	132,435	130,272
점수 표준편차	7,210	4,449	5,871	8,672	7,253
누적 계산 시간 평균 (초)	0.17	1.00	1.89	3.86	7.26
누적 계산 시간 표준편차 (초)	0.07	0.15	0.07	0.08	0.25
1초당 점수	741,453	130,127	68,546	34,310	17,944
누적 사용 메모리 평균 (MB)	35.32	64.68	120.70	227.33	397.50
누적 사용 메모리 표준편차 (MB)	0.67	0.97	1.64	3.06	4.97
1MB당 점수	3,569	2,012	1,073	583	324

\* 게임 오버를 당하는 경우는 제외하였다.

의외로 노드 수의 상한과 점수 평균에는 상관관계가 없음을 확인할 수 있다. 다만, 노드를 2개 남겼을 때에는 1,000번째 블록이 떨어지기 전에 게임 오버를 당하는 경우가 있었다. 이 요인도 안정성과 효율의 균형이 맞는 값을 잘 선택해야 할 것이다.

이 평가를 하기 직전까지는 프로젝트에서 레벨마다 확인하는 노드의 상한이 32개였는데, 평가 이후 5개로 감소시켰다. 그래프는 다음과 같았다.

누적 계산 시간 평균과 누적 사용 메모리 평균은 선형의 경향성을 보였다. 이는 추천 알고리즘의 공간 복잡도가 가지치기 상한선에 비례함을 뒷받침한다. 또한 노드 수의 상한과 1초당 점수는 반비례



하는 경향을 보였다. 이는 남긴 노드 수와 관계없이 얻은 점수 평균이 거의 일정한 것에서도 확인할 수 있다.

## 2.7 환경

학생들이 리눅스 서버를 접속하여 프로젝트를 진행하므로 해당 서버에 접속할 수 있는 데스크탑과 ssh 접속 프로그램을 제공한다. 접속하는 리눅스 서버에 각 학생들에게 하위 계정을 발급하여 할당받는 용량에 한하여 자유롭게 이를 이용하여 프로젝트를 진행할 수 있는 환경을 제공한다.

## 2.8 미학

블록의 각 칸이 모양에 따라 색칠되도록 프로그램이 확장되었다. 이외의 부분에 대해서는 매 주차 구현은 기존 인터페이스를 최대한 유지하도록 개발되었다.

## 2.9 보건 및 안정

게임플레이 중, 명령어와 관련되지 않은 키는 전부 무시하므로 이론적으로는 오동작이 생길 수 없다. 또한 자동 플레이 중에는 플레이어의 모든 명령을 무시하므로 오동작이 생길 수 없다.

랭킹 파일을 사용자가 임의로 수정할 경우 오동작이 일어날 수 있다.

## 3 기타

### 3.1 환경 구성

이 프로젝트를 제작하는 데 사용된 시스템은 총 4종류이며, 각각 cspro4.sogang.ac.kr 서버(이하 cspro), 개인 소유의 iMac(이하 shiftsh-imac), 개인 소유의 Windows 기반 랩탑(이하 shiftsh-xps),

개인 소유의 AWS EC2 인스턴스(이하 shift-computing)이다. 프로젝트는 cspro와 shiftsh-imac, shift-computing에서 각각 빌드되었다. 프로그램은 cspro에서 디버깅되었다. 프로그램의 소스 코드는 shiftsh-xps와 shiftsh-imac에서 작성되었다. 프로그램의 훈련 프로그램은 shiftsh-imac에서 디버깅되고 빌드되었다. 프로그램의 훈련은 shift-computing에서 진행되었다.

ncurses.h의 경우 shiftsh-imac에서는 Homebrew 2.1.3를 통해 설치된 버전을 참조하였으며, 다른 환경의 경우 /usr/include/ncurses.h에 선언되어 있는 버전 정보를 참조하였다.

각각 시스템의 환경 구성은 다음과 같다.

### cspro

항목	값
운영 체제	Ubuntu 16.04.2 LTS
커널	Linux 4.4.0-141-generic
아키텍쳐	x86_64
gcc	gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10)
gdb	GNU gdb (Ubuntu 7.11.1-0ubuntu1 16.5) 7.11.1
make	GNU Make 4.1 Built for x86_64-pc-linux-gnu
ncurses.h	NCURSES_VERSION = “6.0”, NCURSES_VERSION_PATCH = 20160213

### shiftsh-imac

항목	값
운영 체제	macOS 10.14.4 (18E226)
커널	Darwin 18.5.0
아키텍쳐	x86_64
C IDE	Visual Studio Code 1.34.0(1.34.0)
gcc	Apple LLVM version 10.0.1 (clang-1001.0.46.4)
make	GNU Make 3.81 built for i386-apple-darwin11.3.0
ncurses.h	stable 6.1 (bottled) [keg-only]
Java/Kotlin IDE	IntelliJ IDEA 2018.3.4 (Community Edition)
JRE	Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
JDK	javac 1.8.0_191
JVM	OpenJDK 64-Bit Server VM by JetBrains s.r.o
Kotlin	1.3.30
Gradle	4.10

### shiftsh-xps

항목	값
운영 체제	Windows Home 10.0.17763.503
아키텍처	x86-64
C IDE	Visual Studio Code 1.34.0(1.34.0)

### shift-computing

항목	값
인스턴스 유형	m5d.24xlarge
지역	ap-northeast-2c
운영 체제	Ubuntu 18.04.1 LTS
커널	Linux 4.15.0-1039-aws
아키텍처	x86_64
gcc	gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
make	GNU Make 4.1 Built for x86_64-pc-linux-gnu
ncurses.h	NCURSES_VERSION = "6.1", NCURSES_VERSION_PATCH = 20180127
JRE	OpenJDK Runtime Environment (build 11.0.3+7-Ubuntu-1ubuntu218.04.1)
JVM	OpenJDK 64-Bit Server VM

## 3.2 참고 사항

프로그램이 터미널의 색상 모드를 판단하는 과정에서 다음과 같은 터미널에서 정상 작동함을 확인하였다.

- PuTTY 0.71 x64 on Windows Home 10.0.17763.503: 8색 모드로 정상 작동하였다.
- ssh with git bash 4.4.23(1)-release on Windows Home 10.0.17763.503: 8색 모드로 정상 작동하였다.
- Terminal on macOS 10.14.4 (18E226) with Darwin 18.5.0 kernel: 256색 모드로 정상 작동하였다.
- iTerm2 Build 3.2.8 on macOS 10.14.4 (18E226) with Darwin 18.5.0 kernel: 256색 모드로 정상 작동하였다.

다음과 같은 터미널에서는 색상이 정상적으로 표시되지 않았다.

- ssh with bash 4.4.19(1)-release on Ubuntu 18.04.1 LTS, Windows Subsystem for Linux on Windows Home 10.0.17763.503: 색상이 정상적으로 출력되지 않았다.

이외의 터미널에서는 테스트해 보지 않았으므로 실행에 주의가 필요하다.

RB 트리에 관한 의사 코드는 *Introduction to Algorithms* (3rd ed)<sup>1</sup>를 참조하였다.

JNI 구현에 대한 코드는 ”Java Native Interface (JNI) - Java Programming Tutorial”<sup>5</sup>를 참조하였다.

### 3.3 팀 구성

서강대학교 컴퓨터공학과 박수현 (20181634) 100% 기여. 개인 프로젝트로 진행하였다.

### 3.4 수행기간

2019년 4월 29일부터 2019년 5월 27일까지.

## 참고문헌

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (3rd ed). Cambridge, MA: The MIT Press, 2009.
- [2] “libstdc++: stl\_set.h Source File”, gcc.gnu.org. [Online]. Available: [https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a01064\\_source.html](https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a01064_source.html). [Accessed: 25- May- 2019].
- [3] A. Khachaturyan, S. Semenovsovskaya and B. Vainshtein, “The Thermodynamic Approach to the Structure Analysis of Crystals”. *Acta Crystallographica*, vol. 37 (A37), pp. 742–754, 1981.
- [4] Classic Tetris, *16 Y/O UNDERDOG vs. 7-TIME CHAMP - Classic Tetris World Championship 2018 Final Round*. youtube.com. [Online]. Available: [https://www.youtube.com/watch?v=L\\_UPHsGR6fM](https://www.youtube.com/watch?v=L_UPHsGR6fM). [Accessed: 19- May- 2019].
- [5] ”Java Native Interface (JNI) - Java Programming Tutorial”, www3.ntu.edu.sg, [Online]. Available: <https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html>. [Accessed: 26- May- 2019].