최종 보고서

전공: 경영학과 학년: 4학년 학번: 20190808 이름: 방지혁

작성일 : 2025.12.18

1. 프로젝트 목표 및 실험 환경에 대한 설명

본 프로젝트는 수업에서 실습한 기존의 테트리스 추천 알고리즘을 개선하는 것이며 변경 사항은 다음과 같이 요약할 수 있다.

- Beam Search 알고리즘을 통한 최적화
- 필드 저장 방식 변환
- 필드 평가 함수 개선: 구멍 개수 세기, 울퉁불퉁 계산, 높이 차이 계산, 골짜기 계산, 채운 라인 계산

개발환경에 대해 설명하자면 macOS 환경에서 개발했으며, 컴파일러는 gcc이다. 라이브러리는 기존과 똑같이 ncurses가 필요하다. gcc tetris.c -lncurses -o tetris 해당 명령어로 컴파일하여 ./tetris로 실행할 수 있다.

기존에 비해 알고리즘은 개선할 수 있었고 가장자리에 깊은 골짜기를 생성하는 경향은 보고서를 작성하면서 해결했다.

2. 각 변수에 대한 설명

1) 전역 변수

게임 필드 관련

char field[HEIGHT][WIDTH];

현재 필드 상태를 저장하는 변수로 0이면 비어 있고 1이면 해당 위치에 블록이 존재한다.

블록 정보

Int nextBlock[BLOCK_NUM];

현재 블록과 다음 2개 블록의 ID를 저장하는 변수이다.

현재 블록 상태 관련

Int blockRotate; - 회전 수 저장

Int blockY; - y 좌표 저장

Int blockX; - x 좌표 저장

게임 상태

Int gameover; - 게임 종료 여부 저장, 0이면 아직 진행 중, 1이면 게임 종료된 상태

Int score; - 현재 게임 점수 저장 변수

Int timed_out; - 타이머 타임아웃 플래그 변수

랭킹 관련

Node *head; - 내림차순으로 랭킹 정보를 저장하는 링크드 리스트의 헤드 포인터

Int score_number; - 랭킹 정보에 등록된 총 랭킹의 수

추천 알고리즘 관련

RecNode* recRoot; - 추천알고리즘을 위한 노드들로 이루어진 트리의 루트 포인터

Int recommendR; - 추천 회전 횟수

Int recommendY; - 추천 y 좌표

Int recommendX; - 추천 x 좌표

2) 구조체

추천 정보 저장 노드

```
typedef struct _RecNode {
   int level; // 추천 트리의 level(or depth)
   int accumulatedScore; // 누적된 점수
   char recField[HEIGHT][WIDTH]; // 추천된 블록의 위치와 회전 수를 고려해서 놓을 때 필드 상태
   // tree에서 children 노드들을 가리키는 포인터 배열, child 수만큼 동적 할당
   struct _RecNode **child;
   // 블록 위치 정보
   int blockX;
   int blockY;
   int blockRotate;
} RecNode;
```

구조체 변수 설명은 표에 있는 주석 참조

랭킹 정보 저장 노드

```
typedef struct Node {
   char name[NAMELEN]; // 플레이어 이름 참조
   int score; // 플레이어 점수 저장
   struct Node* next; // 다음 노드를 가리키는 포인터
} Node;
```

구조체 변수 설명은 표에 있는 주석 참조

3) 상수

WIDTH = 10: 필드 넓이

HEIGHT = 22: 필드 높이

BLOCK_NUM = 3: 미리 볼 수 있는 블록 개수

NAMELEN = 16 : 랭킹 정보에 등록된 이름의 최대 길이

BEAM_WIDTH = 3 : 추천 알고리즘에서 beam search를 하면서 유지 및 고려하는 후보의 개수

CHILDREN_MAX = 36 : 추천 노드의 최대 자식 개수

4) 지역 변수 – 주요 함수만

evaluateField 함수

int holes : 구멍 개수

int heightColumns[WIDTH] : 각 열의 높이를 저장하는 배열

int totalHeight: 모든 열의 높이들의 합

int bumpiness: 인접 열 간 높이 차이의 합

int wells: 골짜기 패널티 점수

int maxHeight: 가장 높은 열의 높이

int minHeight: 가장 낮은 열의 높이

modified_recommend 함수

int currentLevel: 현재 트리 깊이

int maxScore: 최대 점수

int currentBlockID: 현재 평가 중인 블록 ID

RecNode* candidates[CHILDREN_MAX]: 후보 노드 배열

int candidateCnt: 실제 생성된 후보 개수

int beamWidth: 선택할 상위 노드의 개수

3. 각 함수에 대한 설명

int main();

ncurses 및 난수 생성기를 초기화 하고 rank.txt를 로드하고 메인 메뉴 루프를 실행한다. 종료시 랭킹을 다시 저장하고 메모리를 해제한다.

void InitTetris();

게임 시작 시 모든 변수와 상태를 초기화한다. Play() 및 recommendedPlay()에서 호출된다.

Int CheckToMove();

필드 경계를 벗어나는지, 다른 블록과 겹치는지 확인하여 블록이 해당위치로 이동 가능하지 확인한다.

Int AddBlockToField();

블록을 필드에 추가하고 바닥이나 다른 블록에 닿았을 경우 점수를 반환한다.

Int DeleteLine();

완성된 라인을 삭제하고 점수를 반환한다.

Void BlockDown();

1초마다 자동 호출되며 블록을 한 칸씩 아래로 내린다. 다음 블록을 준비하고 추천 트리를 재생성한다.

추천 알고리즘 함수

Int evaluateField();

Holes, wells, bumpiness, totalHeight, heightDifference, heightPenalty 같은 여러 요소를 이용해서 필드 상태를 평가하여 페널티를 계산한다.

Int modifiedRecommend();

Beam search 알고리즘으로 최적의 배치를 찾는다. 설명은 4번 항목에서 추가적으로 하겠다.

Int compareRecNodes();

modifiedRecommend함수에서 후보군들을 qsort로 내림차순 정렬할 때 필요한 비교 함수이다.

Void freeRecNodes();

추천 트리의 모든 노드에 대해 메모리 해제한다.

Void play();

수동 플레이 모드 함수이다. 게임 초기화를 하고 타이머를 시작하고 사용자 키 입력에 대해 처리한다. 게임 오버시 랭킹에 등록한다.

Void recommendedPlay();

자동 플레이 모드 함수이다. 게임 초기화를 하고 추천 위치를 가져와서 블록을 이동시키고 고정한다.

Void createRankList();

Rank.txt 파일에서 랭킹 정보를 읽는다.

Void writeRankFile();

현재 랭킹 정보를 txt파일에 저장한다.

Void newRank();

게임 종료 후 새로운 점수를 등록하고 리스트에 삽입하여 저장한다.

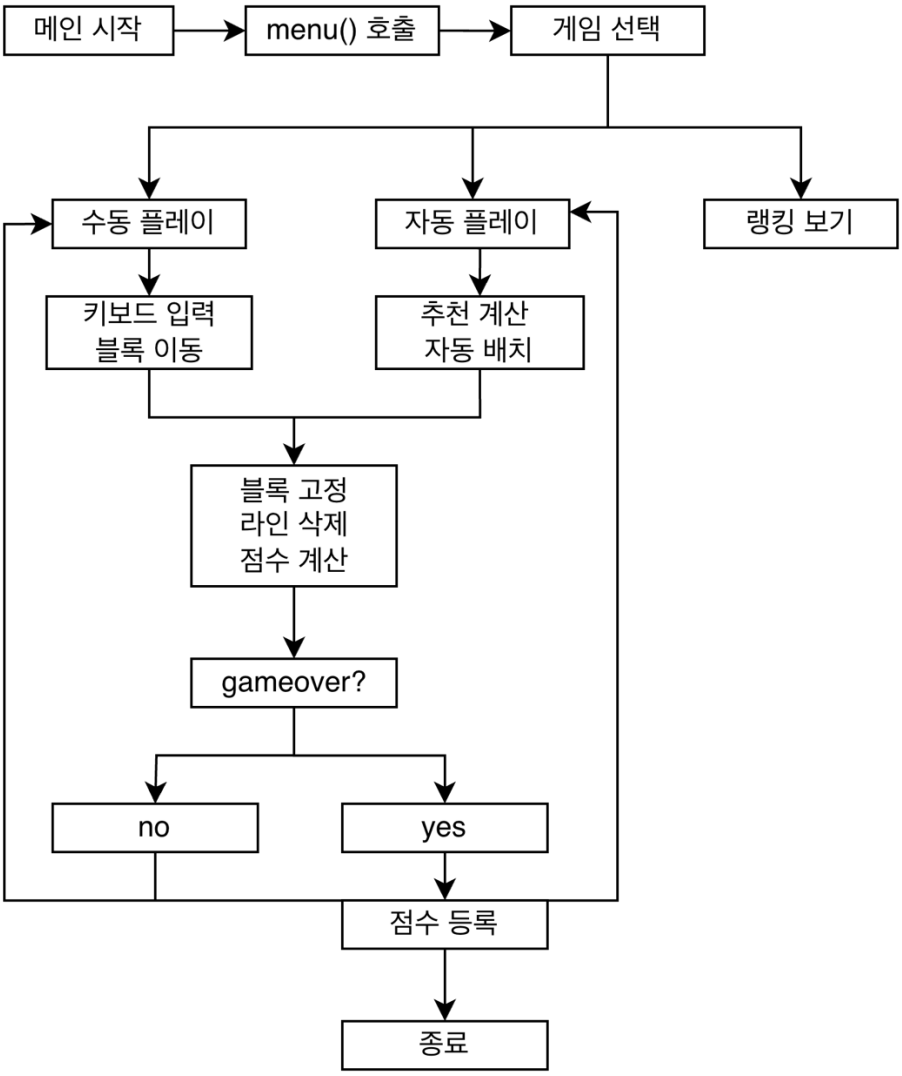
Void insertNode();

노드를 랭킹 리스트에 오름차순으로 삽입한다.

Void rank();

랭킹을 조회하거나 검색하거나 삭제할 수 있는 메뉴를 제공한다.

4. 프로젝트 전체 플로우 차트 + 자료구조 및 알고리즘 + 시간/공간 복잡도



Beam Search

알고리즘

너비 우선 탐색의 변형이라고 볼 수 있다. 각 레벨에서 모든 후보군을 탐색하는 것이 아니라 가장 유망한 BEAM_WIDTH개만큼 자식 노드를 유지하여 탐색하는 기법이다.

1. 현재 블록을 고려하여 가능한 모든 배치 생성한다.

2. 각 배치마다 블록을 떨어뜨려 블록을 추가하고 라인 삭제를 하고 필드 평가함수를 사용하여 점수를 계산한다.

3. 점수 순으로 정렬한다.

4. 상위 N개만큼만 선택한다.

5. 해당 남겨진 배치들에 대하여 재귀적으로 탐색한다.

6. 최종적으로 가장 높은 점수인 경로를 선택한다.

이 알고리즘에 대해 분석하자면 기존의 완전 탐색 알고리즘보다 효율적이라고 볼 수 있다. 후보군을 줄여 메모리 및 시간을 절약하기 때문이다. 그러나 BEAM_WIDTH에 따라 성능 차이가 발생할 수 있고 가장 최적의 해가 보장되지는 않는다.

자료구조

다진 트리: 각 노드가 최대 BEAM_WIDTH 만큼의 자식을 가지는 트리이다. 각 노드는 앞서 서술한 RecNode 구조체로 이루어져 있다.

동적 배열: candidates[CHILDREN_MAX] 배열로 후보 노드를 qsort로 정렬하여 상위 N개만 선택하기 전 임시 저장하는 배열이다.

시간복잡도

Beam search의 시간 복잡도는 O(BEAM_WIDTH^탐색 깊이 * 최대회전횟수 * 필드 넓이 * 필드 높이)로 표현될 수 있다. 각 레벨에서 모든 블록 배치를 생성하고 떨어뜨리고 필드에 추가하고 라인을 삭제하고 필드를 평가하고 이후 정렬하여 상위 BEAM_WIDTH만큼만 선택하기 때문이다.

공간복잡도

트리에 저장되는 총 노드의 개수는 1 + B + B^2 + .. B^D 이기 때문에 총 (B^(D+1) - 1)/(B-1)라고 볼 수 있다.

필드 평가

알고리즘

현재 필드 상태가 좋은 지 나쁜 지 계산한다. 페널티가 작을수록 좋은 배치이다.

평가 요소로는 구멍, 골짜기(양 옆의 높이가 높고 가운데가 낮은 지 평가), 울퉁불퉁한 정도(인접할 열의 높이 차이의 합), 전체 높이, 최고 높이와 최저 높이 간의 차이, 특정 높이 시 페널티 부여가 있다. 이러한 점수들을 모두 더해 최종적으로 페널티를 계산한다.

자료구조

각 열의 높이를 저장하는 heightColums[WIDTH] 배열, 그 외에 앞서 서술한 평가 지표를 저장하는 int형 지역변수들이 있다.

시간복잡도

필드 평가 함수는 앞서 언급한 평가 지표들을 계산하기 위해 모든 칸을 순회하기에 O(WIDTH * HEIGHT) 만큼 시간 복잡도가 소요된다. 골짜기 계산은 최악의 경우 각 열에서 양 옆의 더 높은 기둥을 찾아야 하기에

O(WIDTH * WIDTH)만큼 소요된다. 그렇기에 전체 시간 복잡도는 O(WIDTH * HEIGHT + WIDTH^2)라고 볼 수 있다.

공간복잡도

앞서 언급한 각 열의 높이를 저장하는 배열 그리고 int형 지역변수들이 필요하다. Int형 변수들은 8개(8 * 4bytes), 배열은 10 * 4byte이기에 총 72bytes가 필요하다. 전체 공간 복잡도는 O(WIDTH)라고 볼 수 있다.

5. 창의적 구현에 대한 설명

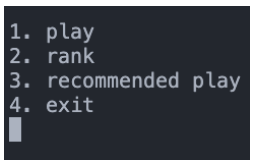
본 프로젝트에서 창의적인 구현은 beam search 알고리즘과 필드 평가 함수의 개선이다.

기존에는 그냥 단순하게 모든 경우를 탐색하거나 점수와 상관없이 노드 개수를 제한하는 방식을 뒀었다. 그러나 이를 개선하기 위해 탐색 후 점수로 정렬을 하고 상위 N개의 노드만 재귀적으로 탐색하는 beam search를 도입하였다.

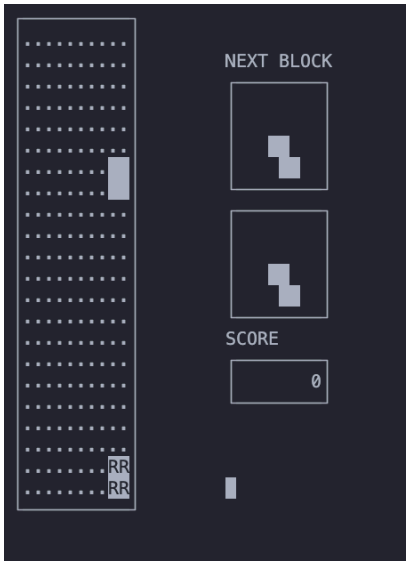
또한 기존에는 단순히 열 높이만 저장하였었지만, 더 자세한 평가를 위해 필드 상태를 정확히 보존하도록 바꾸었다. 이 때문에 메모리 사용량은 증가하지만 더 정확한 판단이 가능해졌다.

또한 평가 함수에 있어 창의성을 구현했다고 생각한다. 단순히 높이를 고려하는 것이 아니라 서로 다른 6가지의 특성을 고려하였다. 가장 중요하게 생각하는 것은 골짜기를 탐지하는 것을 넣고 계산식도 다르게 하여 깊으면 깊을수록 더 큰 불이익을 주었다. 그리고 높이 페널티를 통해 게임오버까지 가는 상황을 최대한 지연시키고자 하였다.

6. 프로젝트 실행 결과 캡쳐

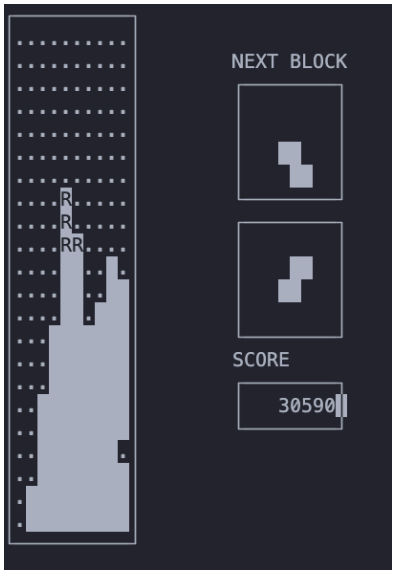


초기화면



1. play 화면

2. rank



3. recommended play

7. 느낀 점 및 개선사항

본 프로젝트를 진행하면서 이론과 실제 구현과의 간극을 느꼈다.

Beam search는 비교적 간단한 개념일 수도 있겠지만 가중치를 결정하는 과정은 실질적으로 경험적 접근이 필요한 부분이었다. 또한, 디버깅을 하면서 예를 들어 추천플레이 함수에서 블록이 공중에 떠 있는 버그가 발생한 적이 있다. 이는 blockY를 recommendY로 직접 설정하면서 생긴 문제로 이를 위해 -1로 초기화하고 자연스럽게 떨어뜨리는 방식으로 해결했다.

기존보다 게임이 오래 지속되는 측면에서 만족스러웠지만 가장 큰 문제는 여전히 한쪽으로 치우쳐 쌓는 경향이 있다는 것이다. 여러 가중치를 조절해봤지만 이 문제는 아직 해결되지 않은 상태이다.

그러나 이 보고서를 작성하면서 너무 답답해서 열심히 찾아본 결과

```
// 모든 가능한 회전에 대해 시도
for (int rotate = 0; rotate < maxRotations[currentBlockID]; rotate++) {
  // 블록을 필드의 모든 x좌표에 대해 시도
  for (int x = -3; x < WIDTH; x++) {
    int y = 0; // 맨 위에서부터 시작
    // 블록이 해당 위치에 놓일 수 있는지 확인
    if (!CheckToMove(root->recField, currentBlockID, rotate, y, x)) {
      continue;  // 해당 위치에 놓을 수 없으면 다음 x좌표로
    }
    while (CheckToMove(root->recField, currentBlockID, rotate, y + 1, x)) {
      y++; // 블록이 더 내려갈 수 있을 때까지 내림
    }
```

기존에 어떤 블록 모양은 -3까지 체크해도 실제로 놓일 수 있고 checktoMove로 최종적으로 확인하면 되는데 그 부분을 고려하지 않았다는 점을 발견했다. 그리고 이를 개선했다. 또한, 블록 종류별 차별화 전략을 쓰면 더 높은 점수를 쓸 수 있지 않을까 생각해봤다.