2020년도 2학기 컴퓨터공학설계및실험Ⅰ

9주차 테트리스 프로그램 3주차 결과 보고서

20161663 허재성

1. 실습 목적

테트리스 게임을 플레이하는 사용자를 위해 어떤 위치에 블록을 놓으면 높은 점수를 받을 수 있는지를 추천하는 추천 시스템을 구현한다. 추천된 블록은 1주차 숙제의 그림자 기능을 이용해서 화면 상에 나타낸다. 이 과정을 통해 tree와 포인터(pointer)에 대한 이해를 높일 수 있다.

2. 실습 구현 내용

헤더 파일 tetris.h에 다음 매크로를 추가한다.

|  |
| --- |
| #define VISIBLE\_BLOCKS 3 |

VISIBLE\_BLOCKS은 recommend 시 생성될 트리의 깊이(depth)이다. 루트 노드의 depth를 0으로 한다. depth 1은 테트리스 게임 시 필드에 보이는 현재 블록을 고려한 노드의 깊이이다. depth 2부터 다음 블록을 고려한 노드의 깊이이다. VISIBLE\_BLOCKS가 3일 경우 depth는 3까지 존재하며 현재 블록 1개와 앞으로 나올 블록 2개를 고려한다.

recommend를 위한 트리의 노드 구조체 RecNode를 다음과 같이 정의한다. 블록의 회전 상태 r, 해당 회전 상태 r에서 최종적으로 필드에 놓였을 때의 x좌표를 고려하여 (r, x)마다 한 개의 노드가 만들어진다.

|  |
| --- |
| typedef struct \_RecNode {  int lv, score;  char f[HEIGHT][WIDTH];  struct \_RecNode \*c[CHILDREN\_MAX];  } RecNode; |

lv는 노드의 트리에서의 깊이(depth)이다. 루트 노드의 lv=0을 기준으로 N번째 블록의 lv는 N이 된다. score는 부모 노드의 점수에 해당 블록이 회전 상태 r일 때 필드의 x좌표 x에 놓였을 때 얻을 점수를 더한 값이다. 얻은 점수는 블록이 필드의 이미 쌓인 블록과 맞닿은 면적과 블록이 내려와 필드 상에 지운 줄을 고려하여 계산된다 루트 노드일 경우 score는 0이다..

2차원 char형 배열 f는 현재 블록이 놓일 필드 정보를 저장한다. 부모 노드의 필드의 정보를 복사하여 가지고 있다가 회전 상태 r, x좌표 x에 놓였을 때 f를 업데이트한다.

\_RecNode 포인터 배열 c는 현재 노드의 자식 노드의 주소를 저장하는 배열이다. 블록이 회전했을 때 모양이 같아도 다른 경우로 생각하면 블록 4(정사각형 모양 블록)의 위치는 회전 상태 0, 1, 2, 3에 대하여 각각 x좌표가 -1부터 7까지 총 9개 존재 가능하다. 따라서 블록 4의 경우 최대 36개의 위치가 가능하다. (r, x)의 개수 = 4 \* 9 = 36 개 따라서 중복을 허용할 경우 자식 노드는 최대 36개가 생성될 수 있다. CHILDREN\_MAX는 매크로로 36으로 정의되어 있으며 c에는 최대 36개의 자식 노드의 주소가 저장될 수 있다.

tetris.h 파일에 recommend 트리의 루트 노드 역할을 할 recRoot를 전역 변수로 선언한다.

|  |
| --- |
| RecNode\* recRoot; |

recommend 트리를 만들기 위해서는 RecNode 구조체를 할당하고 해제할 수 있어야 한다. 이를위해 추가한 관련 함수는 다음과 같다.

(1) RecNode\* makeRecNode()

recommend 트리의 노드를 동적 할당하는 함수이다. malloc 함수를 이용하여 RecNode 구조체를 동적 할당하고 RecNode 포인터 node가 해당 구조체를 가리킨다. 해당 구조체의 lv과 score을 모두 0으로 초기화하고, 아직 자식 노드를 가지지 않으므로 배열 c의 모든 값을 NULL로 초기화한 후 node를 반환한다. CHILDREN\_MAX의 값이 고정되어 있으므로 시간 복잡도는 O(1)이다. 공간 복잡도는 각 노드마다 필드 정보를 가지고 있으므로 O(HEIGHT\*WIDTH)에 추가적으로 자식 노드를 저장할 배열을 가지고 있으므로 O(CHILDREN\_MAX)이다. HEIGHT, WIDTH, CHILDREN\_MAX 모두VISIBLE\_BLOCKS와는 관계 없는 상수이므로 O(1)이다.

input : 없음

output : 동적 할당된 RecNode 구조체를 가리키는 포인터 변수 node

(2) void deleteRecNode(RecNode\*\* del)

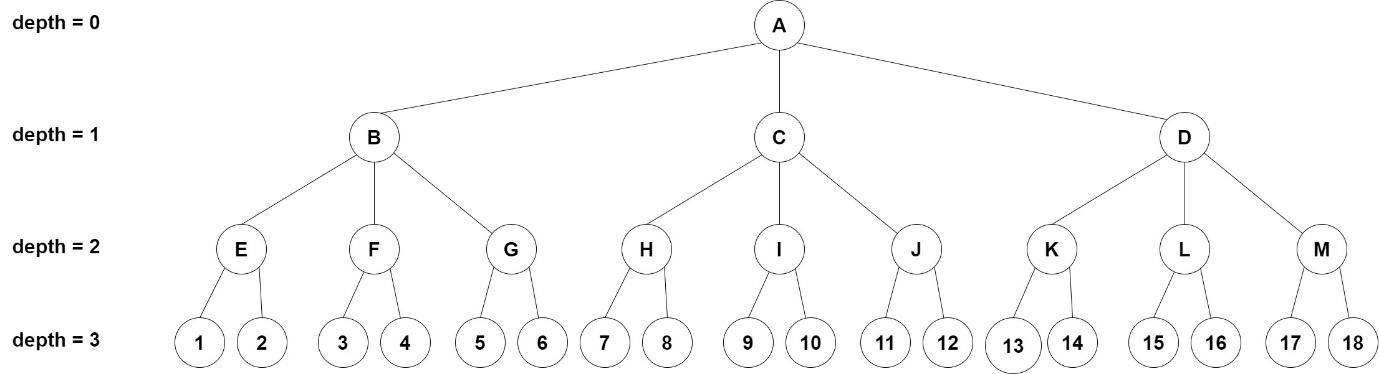
동적 할당된 RecNode로 이루어진 트리를 해제하는 함수이다. 트리의 루트 노드의 이중 포인러를 입력으로 받는다. 루트 노드의 자식 노드들을 재귀적으로 해제하고 루트 노드를 해제한다.

루트 노드의 자식 노드(서브 트리)에 대하여 재귀적으로 deleteRecNode 함수를 호출한다. 만약 대상으로 호출한 노드가 단말 노드(leaf node)일 경우 해당 노드만 해제한다. 반대로 자식 노드가 있을 경우 각 자식 노드에 대하여 재귀적으로 deleteRecNode를 호출한다.

해제하려는 노드를 가리키는 포인터가 자식 노드가 없는 단말 노드일 경우, free 함수를 호출해서 해당 노드를 해제하고 해당 노드를 가리키는 포인터를 NULL로 설정한다.

노드가 단말 노드인지 확인하기 위해서 각 노드에 저장된 lv 변수를 VISIBLE\_BLOCKS와 비교한다. lv(depth에 해당)이 VISIBLE\_BLOCKS보다 작은 경우 깊이가 최대가 아니므로 자식 노드가 있을 수 있다. 실제로 실습에서 사용한 트리는 노드의 depth가 VISIBLE\_BLOCKS보다 작은 경우 모두 자식 노드를 가지고 있다. 반대로 lv이 VISIBLE\_BLOCKS일 경우, 해당 노드는 반드시 단말 노드이다.

다음은 트리를 해제하는 방법의 예시이다.



VISIBLE\_BLOCKS가 3일 때, 깊이가 3인 트리를 해제한다. 알파벳은 단말 노드가 아닌 노드, 숫자는 단말 노드를 가리키는 포인터이다. 해당 트리를 해제하기 위해 다음과 같이 함수를 호출한다.

|  |
| --- |
| deleteRecNode(&A); |

A 노드를 해제하기 위해 A 노드의 lv을 확인한다. lv이 0이므로 해당 노드는 단말 노드가 아니다. 따라서 자식 노드 B, C, D에 대하여 재귀적으로 deleteRecNode 함수를 호출한다.(재귀적으로 B, C, D를 루트 노드로 하는 서브 트리를 해제한다.)

B 노드를 해제하기 위해 B 노드의 lv을 확인 후 lv이 1이므로 단말 노드가 아님을 확인한다. 자식 노드 E, F, G에 대하여 재귀적으로 deleteRecNode 함수를 호출한다.

E 노드를 해제하기 위해 E 노드의 lv을 확인 후 lv이 2이므로 단말 노드가 아님을 확인한다. 자식 노드 1, 2에 대하여 재귀적으로 deleteRecNode 함수를 호출한다.

1 노드를 해제하기 위해 1 노드의 lv을 확인한다. lv이 3으로 단말 노드이므로 해제한다. 마찬가지로 2 노드도 해제한다.

E 노드의 자식 노드를 모두 해제했으므로 E 노드를 해제한다. 이와 같은 방법으로 F, G 노드와 그 자식 노드들을 모두 해제한다. B의 자식 노드 E, F, G 노드가 모두 해제되었으므로 B 노드를 해제한다. 이와 같은 방법으로 A의 자식 노드 C, D를 모두 해제하고 마지막으로 A 노드를 해제한다.

위의 예시에서 노드의 해제 순서는 다음과 같다.

1->2->E->3->4->F->5->6->G->B->7->8->H->9->10->I->11->12->J->C->13->14->K->15->16->L->17->18->D->A

이진 트리의 순회 방법 중 LRV의 방법으로 순회하는 후위 순회(Postorder Traversal)의 방법과 유사하다.

모든 노드를 순회해서 삭제해야 하므로 시간 복잡도는 O(트리의 노드의 개수)이다. 후술할 recommend 함수에 의하여 34진 트리를 가정했으므로 VISIBLE\_BLOCKS가 N일 때 시간 복잡도는 O(34N)이다.

input : 해제할 트리의 루트 노드의 이중 포인터 RecNode\*\* del

output : 없음

실습에서 구현해야 하는 함수는 다음과 같다.

(1) void DrawRecommend(int y, int x, int blockID, int blockRotate)

현재 블록이 떨어지도록 추천되는 위치에 그림자를 그리는 함수이다. 기존의 그림자와 구분되게 문자 R로 표시된 그림자를 그린다. DrawBlock 함수에 그릴 문자 ‘R’을 넘겨줘서 호출하여 추천 위치를 그린다. DrawBlock 함수를 호출하는 역할만 하므로 시간 복잡도, 공간 복잡도는 O(1)이다.

input : 추천된 위치의 좌표 y, x, 블록 번호(0~6), 추천된 블록의 회전 상태(0~3)

output : 없음

(2) void recommendStart()

현재 블록의 추천 위치를 찾아주는 recommend 함수를 위한 사전 처리를 하는 함수이다. 블록의 위치 추천을 위한 트리를 만들기 위해 루트 노드 recRoot를 동적 할당하고 recRoot의 필드 정보에 테트리스 게임의 필드 정보를 복사한다. 사전 처리를 마치고 recommend 함수에 루트 노드 recRoot를 넘겨줘서 호출한다. recommend 함수를 호출하는 것을 제외하면 필드 정보를 복사하므로 시간 복잡도는 O(HEIGHT\*WIDTH)로 O(1)이고, 추가적인 공간 소모는 없으므로 공간 복잡도는 O(1)이다.

input : 없음

output : 없음

(3) int recommend(RecNode\* root)

VISIBLE\_BLOCKS만큼의 깊이를 가지는 트리를 생성한다. 루트 노드 root의 lv=0을 기준으로 한다. 블록을 고려하여 블록의 가능한 회전 수, 각 회전수에 대하여 필드 상에 존재 가능한 x 좌표 범위를 계산한다. 이 때 최대한 경우를 줄이기 위해 회전하여 모양이 같은 경우는 하나의 경우로 간주하여 중복을 허락하지 않는다. 아래 표를 기반으로 회전 범위와 x 좌표 범위를 결정한다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **회전 상태 0** | **회전 상태 1** | **회전 상태 2** | **회전 상태 3** |
| **블록 0** | 0 ≤ x ≤ 6 | -1 ≤ x ≤ 8 |  |  |
| **블록 1** | -1 ≤ x ≤ 6 | -2 ≤ x ≤ 6 | -1 ≤ x ≤ 6 | -1 ≤ x ≤ 7 |
| **블록 2** | -1 ≤ x ≤ 6 | -2 ≤ x ≤ 6 | -1 ≤ x ≤ 6 | -1 ≤ x ≤ 6 |
| **블록 3** | 0 ≤ x ≤ 7 | 0 ≤ x ≤ 8 | 0 ≤ x ≤ 7 | -1 ≤ x ≤ 7 |
| **블록 4** | -1 ≤ x ≤ 7 |  |  |  |
| **블록 5** | -1 ≤ x ≤ 6 | -1 ≤ x ≤ 7 |  |  |
| **블록 6** | -1 ≤ x ≤ 6 | -1 ≤ x ≤ 7 |  |  |

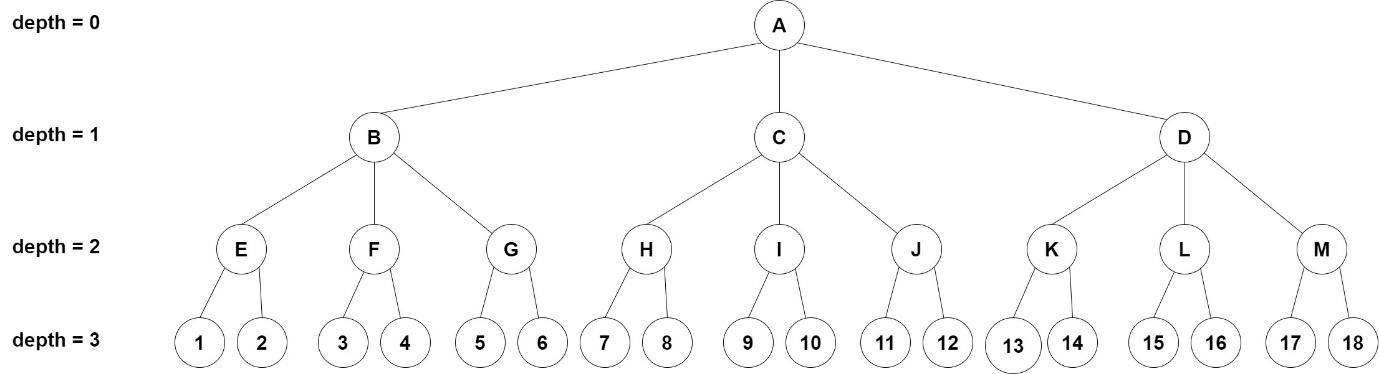
블록 0, 5, 6의 경우 회전 상태는 0, 1이 존재하며 각각의 경우 x 좌표의 개수는 8개, 9개로 총 17개이다. 블록 1, 2, 3의 경우 회전 상태 0, 1, 2, 3이 모두 다르므로 회전 상태는 0, 1, 2, 3이 존재하고 블록 1, 2의 경우 각각의 회전 상태에 대한 x 좌표의 개수는 8, 9, 8, 9로 총 34개, 블록 3의 경우도 마찬가지로 각각의 회전 상태에 대한 x 좌표의 개수는 8, 9, 8, 9로 총 34개이다. 블록 4의 경우 회전 상태는 0만 존재하며 해당 경우 x 좌표의 개수는 9개이다. 위의 경우를 종합해 보면 블록에 따라 생길 수 있는 최대 노드의 개수는 34개이므로 해당 트리는 34진 트리가 된다. 중복을 허용했을 경우 36진 트리가 만들어 졌는데 중복을 허용하지 않을 경우 34진 트리가 만들어진다.

회전 범위를 0~rend로 정하고, 존재 가능한 x 좌표의 범위를 xstart, xend로 정한다. 블록의 회전 상태, y 좌표, x 좌표를 rpos, ypos, xpos라 할 때 rpos의 범위는 0 이상 rend 이하, xpos의 범위는 xstart 이상 xend 이하이다. 각각의 rpos, xpos에 대하여 CheckToMove 함수를 호출하여 블록이 놓일 위치의 y 좌표 ypos를 찾는다. 만약 ypos가 -1일 경우 해당 rpos, xpos로는 블록을 놓을 수 없다는 의미이므로 다음 x 좌표로 넘어간다. 만약 놓을 수 있을 경우 자식 노드 child를 makeRecNode 함수를 호출하여 할당한다. child의 lv 변수를 부모(root) 노드의 lv 변수의 값에 1을 추가하고 score 변수를 부모 노드의 score 변수로 설정한다. child 노드의 필드 정보에 부모 노드의 필드 정보를 복사한다. child의 점수에 블록을 놓았을 때 얻을 수 있는 점수를 추가 후, child를 root의 자식으로 놓는다.

만약 child의 lv이 VISIBLE\_BLOCKS보다 작다면 해당 child는 단말 노드가 아니므로 자식 노드를 갖게 된다. 따라서 해당 child를 대상으로 recommend를 재귀적으로 호출한다.

child의 lv이 VISIBLE\_BLOCKS라면 child의 score가 VISIBLE\_BLOCKS 개의 블록을 고려했을 떄 얻을 수 있는 점수다. 해당 점수를 temp 변수에 저장한다.

만약 temp 변수가 기존까지 알려진 최댓값보다 크다면 최댓값을 갱신한다. 모든 경우에 대한 비교가 끝난 후 최댓값일 때의 회전 상태 r, y 좌표, x 좌표를 추천 위치로 설정한다.



위의 설명을 간단한 예로 보충한다. 위의 그림에서 A 노드를 대상으로 recommend 함수를 호출하면 A 노드는 루트 노드이므로 lv이 0이다. 첫 번째 블록의 경우의 수에 대한 노드를 만들어야 한다. 첫 번째 블록의 경우의 수가 3가지일 경우 먼저 B를 만든다. B를 만든 후 B에 점수를 추가한다. B의 lv은 1로 VISIBLE\_BLOCKS 3보다 작으므로 두 번째 블록에 대한 노드를 만들어야 한다. B에 대해서 recommend 함수를 호출한다. 두 번째 블록의 경우의 수가 3이므로 먼저 E를 만든다. E를 만든 후 E에 점수를 추가한다. E의 lv은 2로 VISIBLE\_BLOCKS 3보다 작으므로 세 번째 블록에 대한 노드를 만든다. E에 대해서 recommend 함수를 호출한다. 세 번째 블록의 경우의 수가 2이므로 먼저 1을 만든 후 1에 점수를 추가한다. 1의 lv은 3이므로 단말 노드이다. 단말 노드의 점수가 VISIBLE\_BLOCK 개의 블록을 모두 놓았을 때의 점수임을 알 수 있다.

E에서 1을 만든 후, 2를 만든다. B에서 서브 트리 E가 만들어 졌으므로 F에 대하여 recommend 함수를 호출하여 서브 트리를 만들고 G에 대하여 recommend 함수를 호출하여 서브 트리 B가 완성된다. 같은 방법으로 서브 트리 C, D를 만든다. 서브 트리를 만드는 과정에서 모든 단말 노드의 점수들을 비교한다. 최대의 경우, 단말 노드에 해당하는 회전 상태, y 좌표 x 좌표가 바로 추천 위치이다.

위의 그림에서 노드의 생성 순서는 다음과 같다.

(루트 노드 A)->B->E->1->2->F->3->4->G->5->6->C->H->7->8->I->9->10->J->11->12->D->K->13->14->L->15->16->M->17->18

모든 단말 노드를 생성 및 확인 후 호출 스택에 쌓인 recommend 함수가 종료되어 A를 대상으로 호출된 recommend 함수만 남는다. 해당 recommend 함수(해당 노드의 lv이 0인지 확인하여 판단한다.)에서 가장 점수가 높았던 경우를 기준으로 추천 회전 상태와 위치를 결정한다.

위와 같은 방법은 이진 트리의 순회 방법 중 preorder traversal을 응용한 것이다. 루트 노드를 탐색 후 왼쪽 자식 노드(서브 트리)를 재귀적으로 탐색하고 오른쪽 자식 노드를 재귀적으로 탐색하는 preorder traversal을 응용하여 루트 노드를 탐색 후, 각 자식 노드를 할당하면서 탐색하는 것이다. 이진 트리를 넘어서 일반적인 트리 및 그래프에서의 깊이 우선 탐색(depth first search, DFS) 방법과 같다.

34진 트리의 경우 depth=VISIBLE\_BLOCKS=N일 때 노드의 개수는 O(34N)이다. 따라서 시간 복잡도와 공간 복잡도는 최악의 경우 O(34N)이다.

input : 추천 트리의 루트 노드의 주소 RecNode\* root

output: 블록을 놓았을 때 얻을 수 있는 최대 점수

위의 함수들을 정의하고 기존에 정의된 함수에서 호출해야 한다.

기존의 정의된 InitTetris 함수와 BlockDown 함수에서 recommendStart 함수를 호출한다. 테트리스 게임이 시작되어 처음 추천을 하기 위해 InitTetris 함수에서 한 번 호출하고, 매번 블록이 다 내려올 때마다 다음 블록으로 갱신되면서 새로운 추천을 해야 하므로 BlockDown 함수에서 호출된다.

play 함수와 BlockDown 함수에서 deleteRecNode 함수를 호출한다. play 함수의 경우, 만약 Q 버튼을 입력해 게임이 종료된 경우 추천 트리를 해제하기 위해 호출한다. BlockDown 함수의 경우 블록이 다 내려오면 새로운 블록에 대하여 추천 트리를 할당해야 하므로 그 전에 deleteRecNode 함수를 호출하여 이전 트리를 해제한다.

DrawBlockWithFeatures 함수에서 DrawRecommend 함수를 호출한다. DrawShadow에 의해 그려질 그림자와 추천 위치가 겹쳐질 경우 추천 위치가 사라지지 않게 하기 위해 해당 함수가 호출될 때마다 DrawRecommend 함수를 호출하여 블록이 완전히 내려오기 전까지 추천 위치가 화면에 유지되도록 한다. DrawBlockWithFeatures에서 DrawRecommend를 호출할 떄에는 DrawShadow, DrawBlock과 달리 recommend 함수에 의해 결정된 recommendR, recommendY, recommendX를 전달한다.

3. 실습 환경

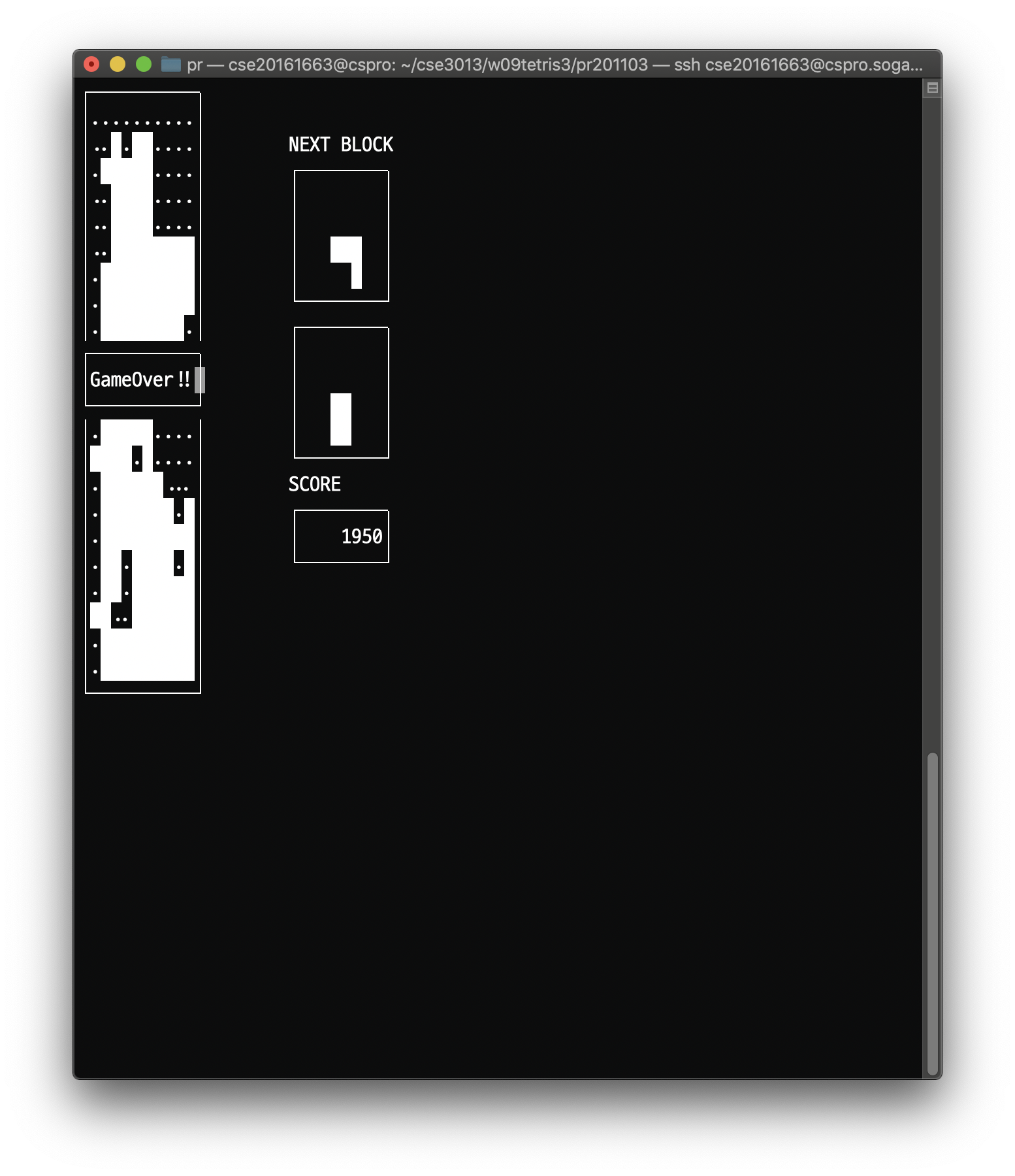
CSPRO 서버 : CSPRO.sogang.ac.kr

macOS의 터미널에서 원격 접속을 통해 CSPRO 서버에 접속하여 실습을 진행하였다. 필요에 따라Windows에서 Putty애 접속하여 실습을 진행했다.

4. 실습 결과 및 분석

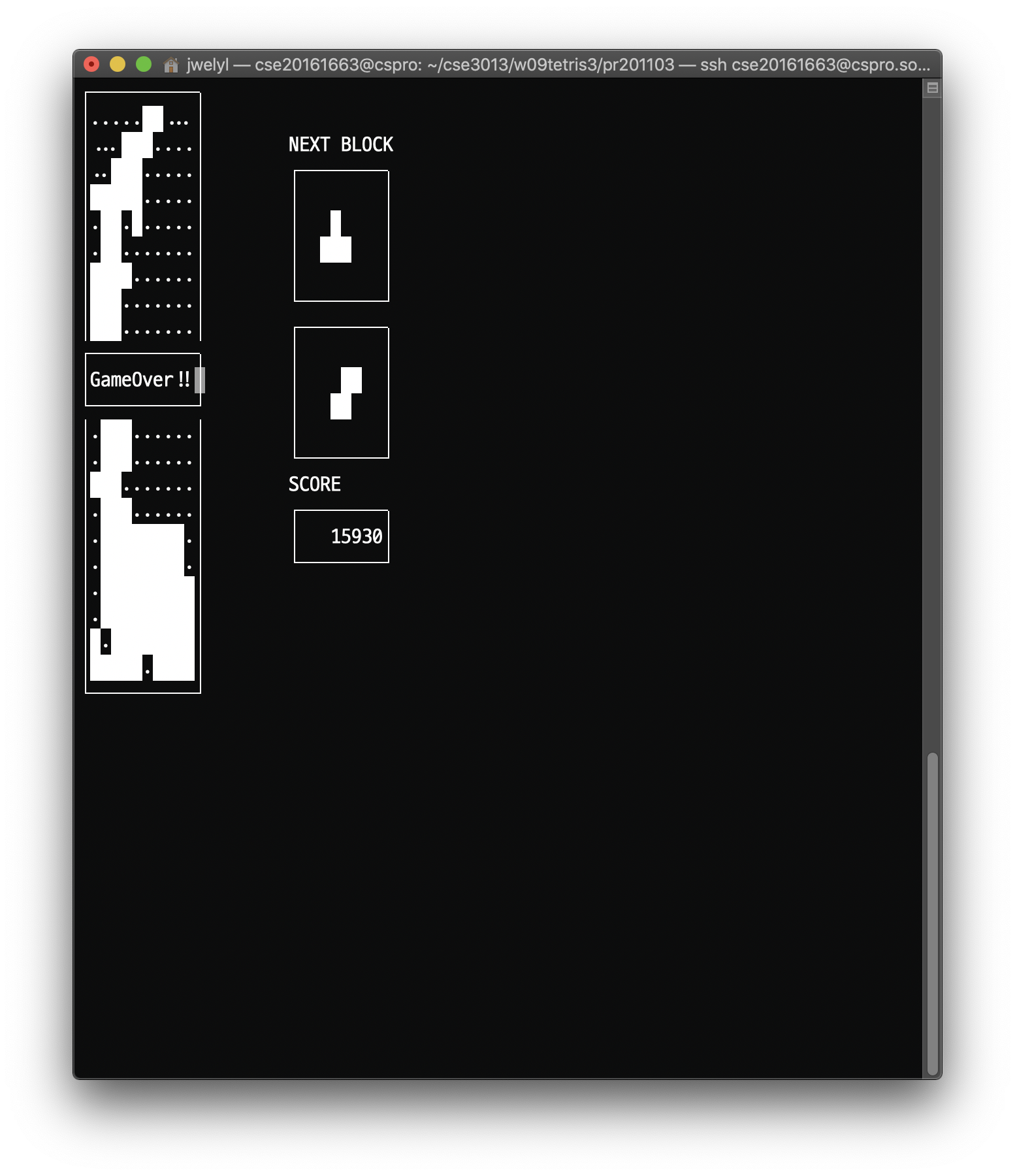
recommend 트리의 depth는 고려할 블록의 개수로 작성한 프로그램에서는 매크로로 정의한VISIBLE\_BLOCKS와 같다. depth가 N이면, 즉 고려할 블록의 개수가 N개라면, 현재 블록 1개와, 앞으로 나올 다음 블록의 개수가 N-1개, 총 N개의 블록을 고려한다. 플레이어에게 미리 보여주는 블록의 개수는 2개(즉, 플레이어가 알 수 있는 블록의 개수는 현재 블록 1개, 앞으로 나올 블록 2개로 총 3개이다.)로 고정되어 있다 가정하자. depth가 2, 3, 4일 경우 결과를 확인해본다.

depth가 2일 때의 게임 종료(GameOver!!) 후 결과 점수는 다음과 같다. 현재 블록과 다음 블록 1개만을 고려한다.



게임 종료 시 점수를 보면 점수가 낮은 것을 알 수 있다. 게임 종료 시 까지 블록이 놓여 있는 것을 보면 블록의 배치도 비효율적인 것을 알 수 있다. 플레이어가 미리 볼 수 있는 NEXT BLOCK이 2개로 총 3개의 블록을 고려할 수 있는데, recommend 트리의 depth가 2일 경우 현재 블록 1개와, 다음 블록 1개만 고려하므로 플레이어가 직접 판단하는 것보다 더 나은 판단을 할 수가 없다. 트리의 깊이가 2밖에 되지 않으므로 생성 및 소멸해야 할 자식 노드의 개수는 적어서 트리의 생성 및 소멸 시간은 매우 빠르지만 효과적인 블록 놓기가 불가능하다.

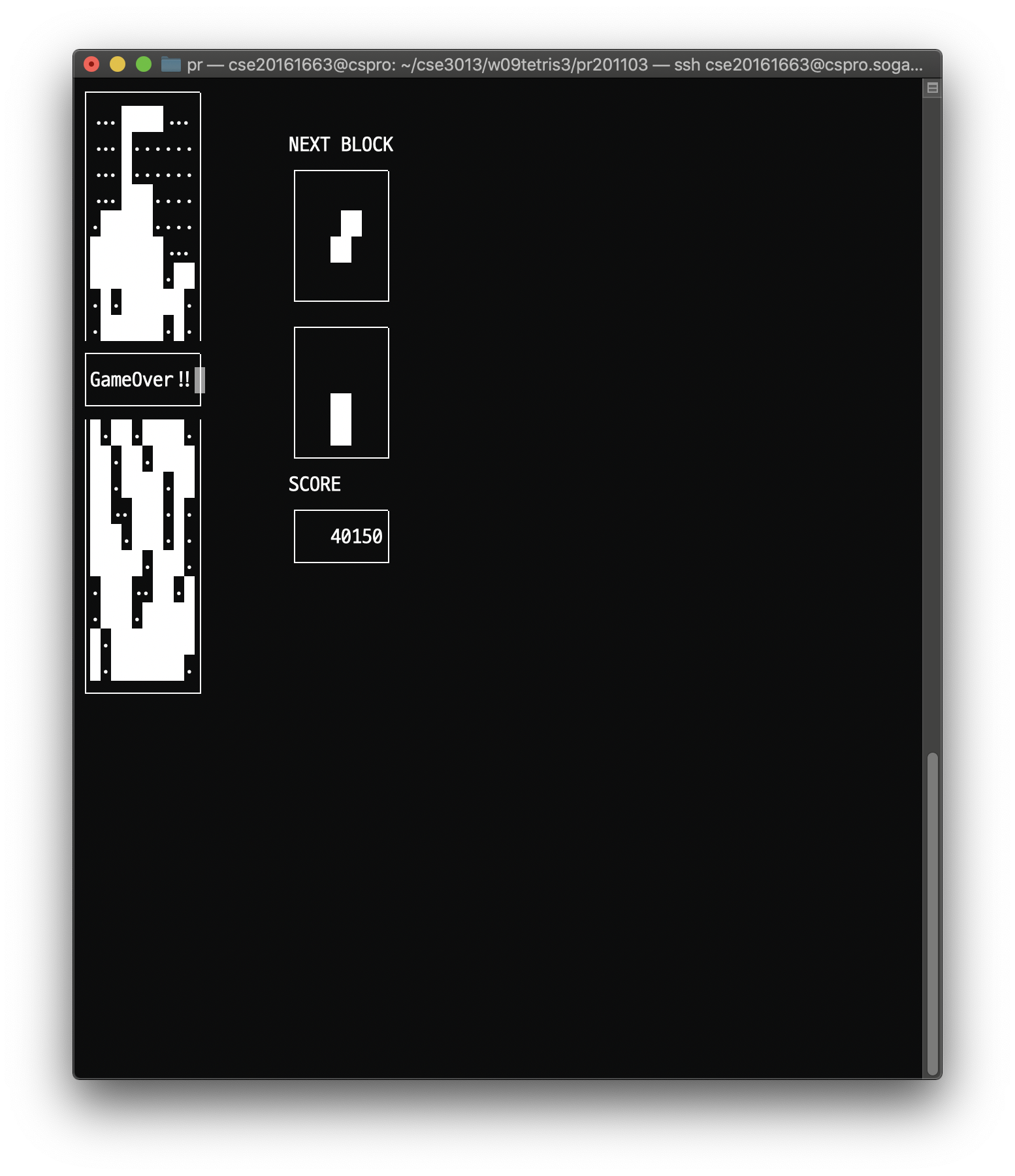
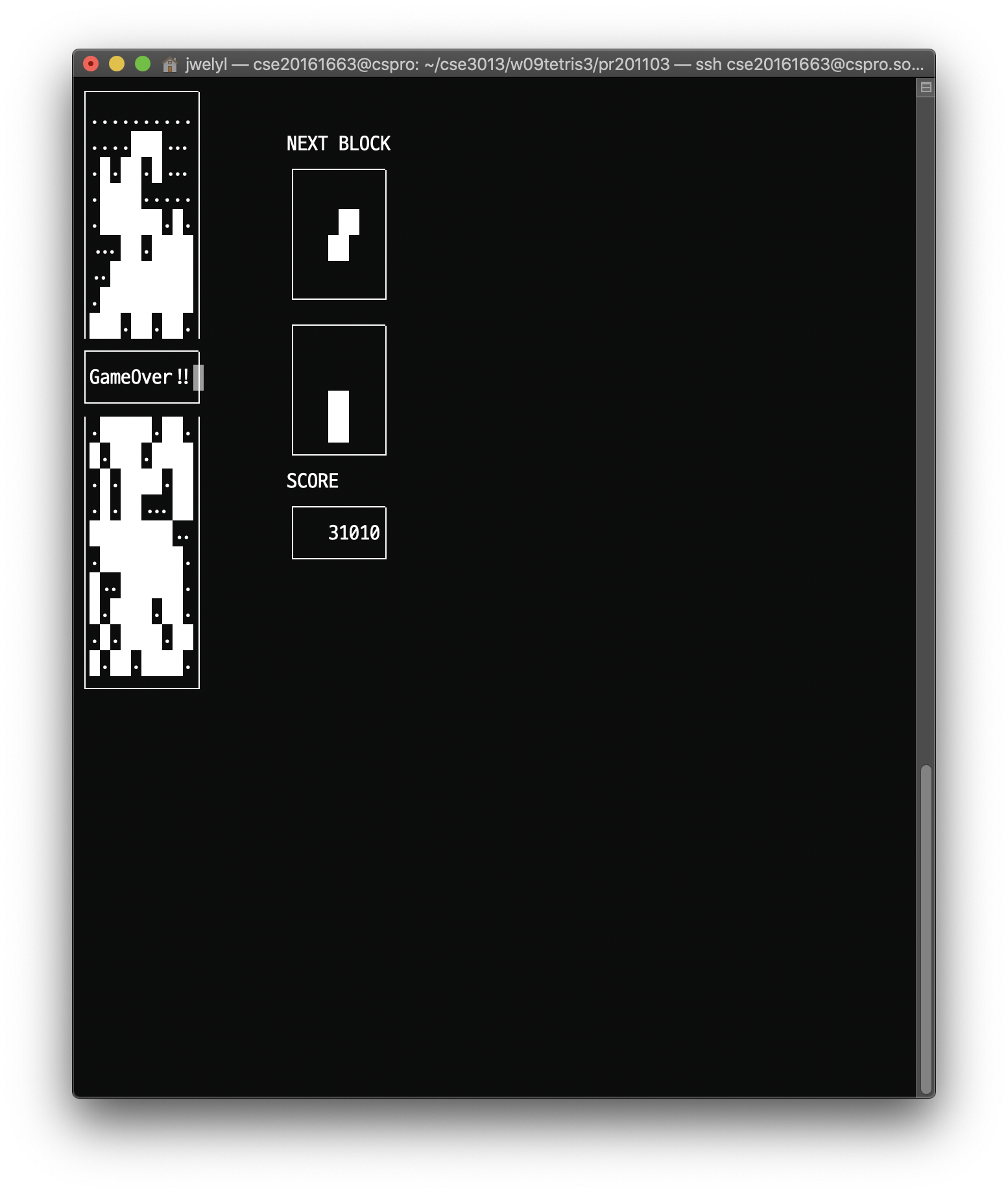
depth가 3일 때의 게임 종료(GameOver!!) 후 결과 점수는 다음과 같다. 현재 블록과 다음 블록 2개를 고려한다.



게임 종료 시 점수를 보면 depth가 2일 때와 비교했을 때 점수가 크게 높아진 것을 알 수 있다. 플레이어가 미리 볼 수 있는 NEXT BLOCK이 2개로 총 3개의 블록을 고려할 수 있는데, recommend 트리의 depth가 3일 경우 현재 블록 1개와, 다음 블록 2개를 고려하므로 이론적으로는 플레이어가 직접 판단하는 것과 같은 판단을 할 수가 있다. depth가 3이 되어 dept생성 및 소멸해야 할 자식 노드의 개수는 최대 343 = 39,304개의 노드를 더 생성하고 더 소멸시켜야 하므로 트리의 생성 및 소멸 시간은 depth=2일 때보다 이론적으로 최대 약 34배의 시간이 더 소모된다. depth가 2일 때보다 차지하는 공간도 최대 34배 증가한다. 시간의 증가는 처음 게임을 시작하여 처음 블록 3개에 대한 트리를 만들 때, 블록을 필드에 놓고 해당 트리를 소멸 후 새로운 트리를 만들 때 확인할 수 있다.

depth=3일 경우에도 미리 볼 수 있는 블록을 모두 고려하지만 블록을 놓는 데 비효율적인 부분이 있음을 알 수 있다. 특히 현재 볼 수 있는 블록들만 있을 때, 블록이 마땅히 놓을 곳이 없을 경우(어딜 놓으나 비슷한 점수를 얻을 경우) 다음 수를 고려하지 못하고 왼쪽으로 치우치게 블록이 놓여 장기적으로 봤을 때, 더 큰 점수를 얻기 힘들고 더 빨리 게임이 종료될 가능성이 높아진다.(depth=2일 경우도 마찬가지이다.)

depth가 4일 때의 게임 종료(GameOver!!) 후 결과 점수는 다음과 같다. 현재 블록과 다음 블록 3개를 고려한다. 플레이어가 미리 볼 수 있는 블록 2개 외에도 1개의 블록을 미리 알고 고려한다.



게임 종료 시 점수를 보면 depth가 3일 때와 비교했을 때 점수가 최소 1만 점 이상 높은 것을 알 수 있다. 플레이어가 미리 볼 수 있는 NEXT BLOCK이 2개로 플레이어는 총 3개의 블록을 고려할 수 있는데, recommend 트리의 depth가 4일 경우 현재 블록 1개와, 다음 블록 3개를 고려하므로 플레이어보다 1개의 블록에 대한 정보를 더 알 수 있으므로 더 많은 경우의 수를 따질 수 있다. depth가 4일 때부터 이론적으로는 플레이어보다 더 나은 플레이를 할 수 있다. 실제로는 플레이어는 당장 얻는 점수가 적더라도(recommend 트리로 계산한 경우의 수에서 최적의 점수가 아니더라도) 테트리스 필드의 이미 쌓인 필드 사이에 빈 칸이 최소화되도록 쌓으므로 블록이 적절하게 나와주면 게임을 거의 무한정으로 이어 나가서 더 높은 점수를 얻을 수 있다. recommend 기능은 앞으로 놓을 수 있는 블록을 고려하여 얻을 수 있는 최대 점수를 기준으로 동작하므로 필연적으로 점수를 위해 비효율적으로 블록을 배치하는 경우가 생기게 된다. 이러한 비효율적인 배치가 누적되어 게임 오버가 된다.

고려하는 블록의 개수가 적을수록 단순히 블록을 놓았을 때 배치 면적이 크게 놓는 경우를 찾게 된다. 여러 줄을 지우는 경우는 여러 개의 블록에 의해 이루어지기 때문에 고려하기 힘들어지고(특히 4줄을 한 번에 지우는 경우는 거의 발생하지 못한다.) 이로 인해 점수가 올라가는 시간이 오래 걸리며, 블록도 비효율적으로 쌓여 게임 오버에 더 빨리 가까워진다.

또한 단순히 블록과 필드에 쌓여 있는 블록의 맞닿은 면적만을 고려하여 블록을 놓는 경우, 맞닿은 면적이 같다면 필드의 왼쪽에 놓이게 된다. 블록이 왼쪽에 치우치게 쌓이게 되어 다음 블록을 놓거나 줄을 지우기 어려워져 블록의 배치가 비효율적으로 된다.

depth가 4일 경우, depth가 2나 3일 경우보다 이러한 비효율성이 크게 감소한다. 스크린샷을 비교해보면 depth 4일 경우 depth가 2이거나 3일 경우보단 블록의 배치가 한쪽으로 편향되지 않은 것을 알 수 있다.

depth가 4일 경우, depth가 3일 경우보다 최대 344=1,336,336개의 노드를 더 생성하고 제거해야 한다. 트리의 생성 및 소멸에 걸리는 시간, 트리를 생성하는 데 필요한 공간은 depth가 3일 때보다 최대 34배 가량 더 증가하여 게임 플레이 속도가 depth가 3일 때보다 확연히 느려진다. depth가 2에서 3으로 증가했을 때 체감 시간 차이는 크게 느껴지지 않지만, depth가 3에서 4로 증가할 경우의 체감 시간 차이는 매우 커서 게임으로서 부적합하다. 과제에서 이러한 시간, 공간 비효율성을 최소화한다.

5. 과제

recommendedPlay 기능을 구현한 결과를 정리한다. 추천 위치를 정하는 방법을 실습과 같은 방법으로 하고, 추천 위치에 자동으로 블록이 놓이는 recommendedPlay 기능을 구현한다. 단순히 자동 플레이가 되는 recommendedPlay만을 구현하기 위해서는 다음과 같은 매크로와 함수가 추가 및 수정되어야 한다.

헤더 파일 tetris.h에 다음 매크로를 추가한다.

|  |
| --- |
| #define MENU\_REC\_PLAY ‘3’ |

자동 플레이 기능을 구현하기 위해 다음과 같은 함수들을 추가한다.

(1) void recommendDown(int sig)

6주차 실습에서 정의한 BlockDown 함수를 변형한 함수로 자동 플레이 시 블록이 추천된 위치에 바로 내려오도록 하는 함수이다. 기존의 BlockDown 함수와 다르게 블록이 내려오는 궤적을 보일 필요가 없으므로 DrawChange 함수를 호출하지 않는다. BlockDown 함수와 마찬가지로 CheckToMove 함수를 이용해 블록이 내려올 수 없는 경우를 판별하여 gameOver를 1로 바꾼다. 블록이 내려올 수 있을 경우, 추천된 회전 상태와 필드의 위치에 블록을 놓아서 점수를 추가한다. 점수를 추가할 때는 6주차 실습과 과제에서 구현한 AddBlockToField 함수와 DeleteLine 함수를 이용한다. 점수를 추가 후, 새로운 현재 블록의 위치를 초기화하고 점수와 필드를 화면에 출력한다. 9주차 실습의 BlockDown 함수에서 기존의 추천 트리를 해제하기 위해 deleteRecNode 함수를 호출한 것처럼 recommendDown 함수에서도 deleteRecNode 함수를 호출한다. 기존의 트리를 해제 후, 새로운 블록에 대하여 새로운 위치를 추천받기 위해 실습에서 구현한 recommendStart 함수를 호출하여 추천 트리를 할당하여 추천 회전 상태, 위치를 알아내고, DrawBlockWithFeatures 함수를 호출하여 현재 블록의 위치와, 아래로 떨어졌을 떄의 그림자, 추천 위치를 그린다. 위의 과정이 끝났으면 recommendDown이 끝났음을 알리기 위해 timed\_out을 1로 설정한다. recommendDown의 입력 int sig는 BlockDown의 입력과 같다. 후술할 recommendedPlay 함수의 act.sa\_handler = recommendDown; 문장은 신호가 오면 recommendDown 함수를 실행한다는 의미로 recommendDown 함수의 입력 sig는 해당 신호이다.

input : 신호에 해당하는 정수형 변수 sig

output : 없음

(2) int recommendCommand(int command)

기존의 play 함수에서 호출했던 ProcessCommand 함수처럼, recommededPlay 함수에서 ProcessCommand 함수를 대신해 호출할 명령 처리 함수이다. recommendedPlay는 블록이 추천 위치에 놓이므로 사용자가 입력할 수 있는 명령은 플레이를 강제로 종료시키는 QUIT 명령어뿐이다. 따라서 GetCommand로 입력 받은 command가 QUIT일 경우를 제외하고 다른 명령(방향키 이동 명령)을 모두 무시한다. 방향키 입력을 무시하므로 방향키 입력대로 움직일 수 있는 지 확인하는 CheckToMove 함수와 움직일 수 있을 때, 블록의 움직임을 그리는 DrawChange 함수를 호출하지 않는다. 입력 명령 command가 QUIT일 경우만 QUIT을 반환하여 게임이 종료되도록 한다.

input : recommandedPlay 함수에서 GetCommand로 입력받은 명령 command

output : 입력 command가 QUIT일 경우 QUIT을 반환, 그렇지 않을 경우 1을 반환

(3) void recommendedPlay()

플레이어가 직접 방향키로 블록의 위치를 조정하는 테트리스 게임을 구현한 play 함수를 대체하는 함수이다. Play 함수와 차이점은 act.sa\_handler = BlockDown; 문장 대신 위에서 정의한 recommendDown 함수를 이용하여 act.sa\_handler = recommendDown; 문장을 삽입한다. 또한 ProcessCommnd 함수를 호출하는 대신, 위에서 정의한 recommendCommand 함수를 호출하여 recommendedPlay 함수가 실행될 동안 게임을 강제 종료하는 QUIT 명령( ‘q’ 입력)을 제외한 방향키 입력을 모두 무시하여 플레이어가 블록을 움직일 수 없도록 한다. 그 외에는 실습에서 수정한 play 함수와 같다.

input : 없음

output : 없음

recommendedPlay 함수를 추가 후, main 함수를 수정한다. main 함수에서 menu로 3. recommended play를 입력받을 수 있도록 switch 문의 case에 MENU\_REC\_PLAY를 추가한다. 해당 메뉴를 선택했을 경우 recommended play가 가능하도록 위에서 정의한 recommendedPlay 함수를 호출한다.

Recommended play를 구현한 결과는 다음과 같다.



Putty로 두 개의 창을 띄워서 2개의 게임을 동시에 플레이하는 모습이다. 자동 플레이가 아닐 경우 불가능하다. 6주차 과제에서 구현한 그림자와 별개로 9주차 실습에서 구현한 추천 위치(R로 표시)도 잘 나타난다.



VISIBLE\_BLOCKS가 4일 때, recommendPlay로 게임 오버될 때까지 플레이한 결과이다.

기타

recommendPlay를 구현하면 트리의 depth에 따른 점수의 비교를 더 쉽게 할 수 있다. 플레이어가 직접 추천 위치에 놓는 경우, 플레이어의 실수에 의해 블록의 위치를 잘못 놓는 경우가 발생할 수 있다. 이 경우 추천 시스템이 추천하는 경우와 어긋나게 되어 정확한 추천 성능을 알기 어렵다. 또한 depth가 깊어질 경우, 한 블록을 놓고 다음 블록이 새로 나타날 때까지 기다려야 하는 시간이 길어져서 테스트 시간이 지나치게 길어진다. 플레이어가 직접 플레이할 경우, 해당 시간을 낭비하게 된다. recommendPlay는 자동으로 플레이하므로 테트리스 게임이 플레이될 동안 플레이어는 다른 일을 할 수 있으며, 후에 게임이 종료되면 결과만 확인할 수 있다. 마지막으로 플레이어가 직접 방향키로 블록을 조종할 경우, 필드에 블록이 많이 쌓일수록(즉 게임 오버에 가까워 질수록) 좌우 이동이 자유롭지 않아서 추천하는 위치에 놓지 못할 가능성이 높아진다. 하지만 recommendPlay의 경우, 추천해 주는 위치에 바로 놓기 때문에 게임 오버가 되기 전까지는 추천 위치에 블록을 놓지 못하는 일은 발생하지 않는다. recommendPlay는 플레이어에 의해 생길 수 있는 모든 변수(플레이어의 실수, 블록을 빈칸 없이 정돈하기 위해 점수를 희생하는 배치 등)를 배제하고 추천 시스템에 의하여 이론적으로(트리를 통한 점수 비교로) 얻을 수 있는 최대 점수를 달성하는 기능이다.