

1. 목적

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

2. 추천 시스템의 개요

미래에 존재할 수 있는 다양한 경우를 고려해서 높은 점수를 얻을 수 있는 블록의 위치를 찾아주는 시스템을 추천 시스템이라 한다. 이는 사용자 없이 테트리스를 플레이하는 인공지능(Artificial Intelligence) 분야와 관련되어 있다. 여기서는 아주 기초적인 방법으로, 현재와 확실한 미래를 고려하는데, 여기서 확실한 미래의 의미는, 현재 블록이 필드에 쌓인 이후에 어떤 블록들이 나올지 모르는 상황인 불확실한 미래가 아닌, 어떤 블록이 필드에 연속적으로 나오게 되는지가 명확한 상황을 말한다. 1주차에서 구현한 테트리스에서는 화면상에서 확인할 수 있는 현재 블록은 현재를, 다음에 나올 2개의 블록들은 확실한 미래를 의미하고, 실제 테트리스 프로그램에서는 nextBlock의 배열의 크기를 변화시키고, 다음 블록을 몇 개를 고려하느냐에 따라 고려하는 미래의 길이가 달라질 수 있다. 이와 같이 추천 시스템은 명확한 미래를 갖는 경우, 현재와 미래에 대해 모든 가능한 경우를 고려하여 가장 높은 점수를 얻을 수 있는 위치를 알려주고, 사용자는 이를 이용해서 블록이 놓일 적절한 위치를 선택할 수 있다. 또한, 지능을 갖는 테트리스 플레이어를 직접 프로그래밍하여 자동으로 플레이 하도록 만들 수 있다.

테트리스 프로젝트 3주차에서 구현하는 추천 시스템은 앞서 설명했듯이, 현재와 확실한 미래의 정보를 동시에 고려하고, 이는 각각 현재 블록과 2개의 다음 블록들의 정보들에 해당한다. 이 추천 시스템은 각 블록에 대한 위치, 회전수에 대한 정보와 테트리스 필드의 정보들을 바탕으로 블록이 놓여질 위치를 추천한다. 이 때, 추천 시스템에서 추천된 블록의 위치를 결정하는 과정은 현재 블록이 놓여진 후, 새로운 블록이 화면에 나타나는 과정에서 수행된다. 이렇게 추천된 위치는 그림자 기능을 사용해서 테트리스 필드 상에 보여주고, 추천된 위치가 얼마나 좋은 위치인지 확인하기 위해서, 사용자없이 이 추천 시스템을 기반으로, 자동으로 플레이하는 테트리스 게임 모드를 구현하여, 미래에 고려되는 다음 블록 수에 따른 추천 시스템의 효율성을 확인한다.

3. 추천 시스템의 구조

추천 시스템에서는 현재 블록, 2개의 다음 블록들의 정보를 고려해서 사용자가 테트리스를 플레이 할 수 있는 모든 경우를 표현해야 한다. 즉, 각 블록의 ID, 회전수, 위치, 필드 상태 등의 정보를 하나의 집합체로 보고, 각 블록이 필드에 나오는 순서대로 각 블록에 대한 이 정보들의 집합들을 나열하면, 테트리스 게임 플레이 할 때, 각 블록들을 차례로 필드의 어느 위치에 블록을 놓을지에 대한 플레이 순서를 나타낼 수 있다. 따라서 이와 같은 정보들의 집합의 나열은 테트리스 게임에서 사용자가 플레이 가능한, 모든 연속적인 플레이를 표현할 수 있게 된다. 이 때, 이 연속적인 플레이는 플레이 시퀀스(play sequence)로

정의하고, 다음과 같은 형태로 표현된다.



(현재 블록의 ID, 현재 블록의 회전수, 현재 블록의 위치, 현재 블록의 필드 상태),
 (다음 블록1의 ID, 다음 블록1의 회전수, 다음 블록1의 위치, 다음 블록1의 필드 상태),
 (다음 블록2의 ID, 다음 블록2의 회전수, 다음 블록2의 위치, 다음 블록2의 필드 상태),
 ...

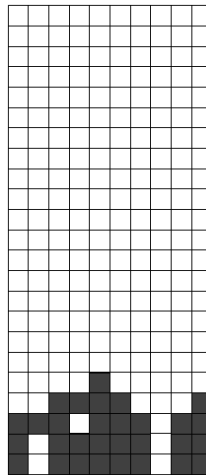
이 플레이 시퀀스는 그 수가 굉장히 많기 때문에, 어떤 규칙 없이 나열하게 되면, 이를 고려해서 어떤 동작을 수행하는데 있어 혼란을 야기하게 된다. 따라서 이와 같은 모든 플레이 시퀀스를 표현하기 위해, 추천 시스템에서는 tree 구조를 사용한다. 이 때, 현재 블록의 정보는 tree의 level 1에서 고려되고, 첫 번째 다음 블록의 정보는 tree의 level 2에서 고려되며, 두 번째 다음 블록의 정보는 tree의 level 3에서 고려된다. 이 tree의 구조는 다음 section의 추천 시스템의 간단한 예제를 통해서 좀 더 자세히 살펴본다.

4. 추천 시스템의 예제

추천 시스템을 위해 구축되는 tree의 각 노드는 블록의 ID, 회전수, 위치, 필드 상태와 플레이 한 후 얻게 되는 누적 점수로 구성된다고 가정하자.




- 블록의 ID(curBlockID)
- 블록이 놓여질 위치(recBlockX, recBlockY)
- 블록의 회전수(recBlockRotate)
- Field의 정보(recField)
- 누적된 score(accumulatedScore)


그리고 고려되는 블록의 수는 2개, 즉, 현재 블록과 다음 블록 1개를 고려한다고 가정하자. 현재 블록  이 필드에 떨어질 예정이고, 다음 블록은  이며, 현재 필드의 상태는 다음과 같다.

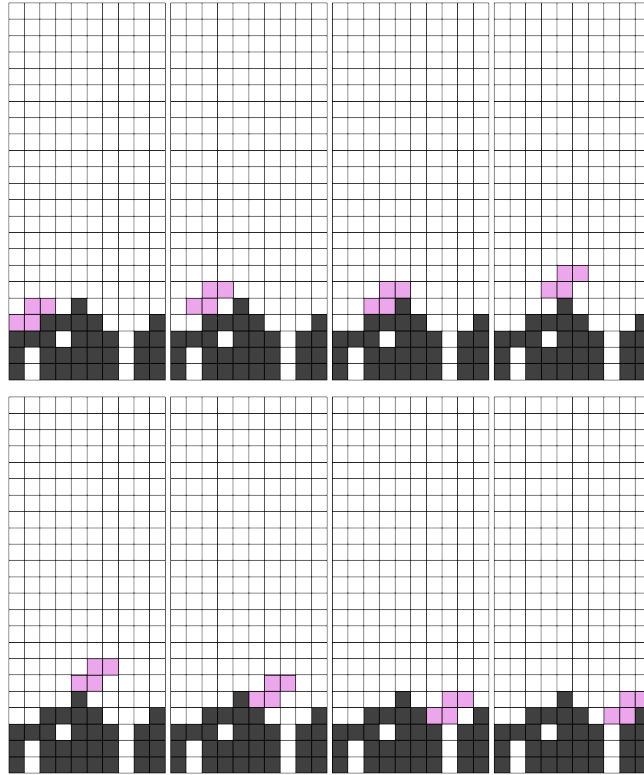



[그림 1] 테트리스 필드


이와 같은 상황이 주어졌을 때, 블록의 추천을 위한 tree가 구축되는 과정을 예제를 통해서 살펴본다.

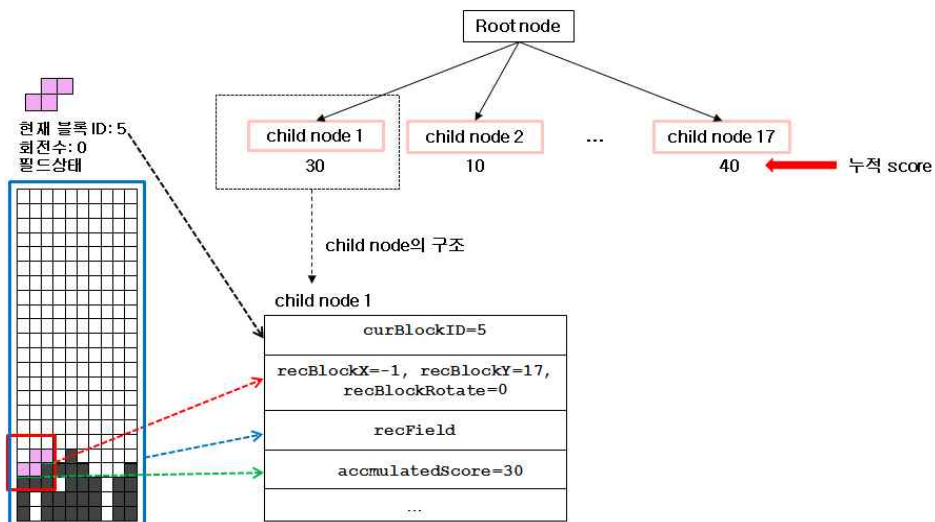
- 현재 블록의 모든 회전수를 고려했을 때, ,  로 두 가지 모양이 존재한다. 첫 번째 회전수를 갖는 블록  이 위 필드에 놓여질 수 있는 위치는 총 8가지이다. 현

재 고려하는 블록  을 놓을 수 있는 8가지 위치는 현재 블록을 가장 왼쪽에 놓았을 때부터, 차례로 오른쪽으로 한 칸씩 이동하면서, 가장 오른쪽에 놓았을 때의 경우까지 센 경우의 수이다. 이는 다음 그림을 통해서 확인할 수 있다.









[그림 2] 필드에 놓을 수 있는 블록  의 8가지 위치

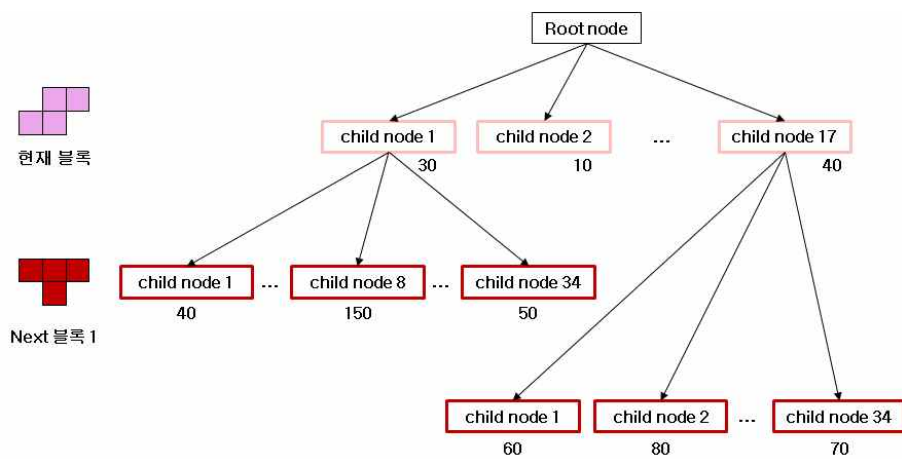
- 두 번째 회전수를 갖는 블록  에 대해서 그림 2의 방법을 사용해서 놓여질 수 있는 블록의 위치를 세면, 9가지로, 두 경우 모두 고려하면, 총 17가지 경우가 존재한다. 이 17가지 경우를 모두 tree구조로 구성하게 되는데, 17가지 경우는 17개의 노드로 표현되고, 구성 방법은 다음 그림과 같다.




[그림 3] tree의 구성 및 노드의 구조

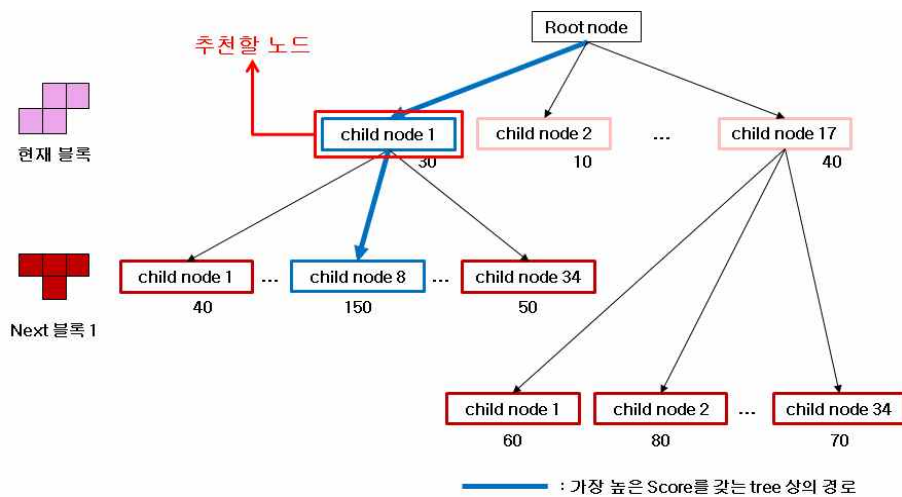
위 그림과 같이 17개의 각 각 노드는 현재 블록의 ID, 회전수, 블록의 좌표, 필드 상태, 누적된 점수를 갖고 있다. 이렇게 만들어진 17개의 노드들은 root 노드의 child 노드들이 된다.

- 이제 다음 블록  을 고려해보자. 이 경우에는 회전해서 나올 수 있는 블록의 경우는 , , ,  로 총 4가지이다. 이 때, 그림 2의 방법을 사용해 각각 회전수에 대해서 필드에 놓여질 수 있는 경우를 세면, 각 회전수에 대해서 8, 9, 8, 9 가지이므로 총 34가지 경우가 존재한다. 따라서 첫 번째 고려한 현재 블록  의 17 가지 각 경우에 대해서 34개의 child 노드들을 갖게 된다. 이를 tree로 표현하게 되면 다음과 같은 구조가 만들어 진다.



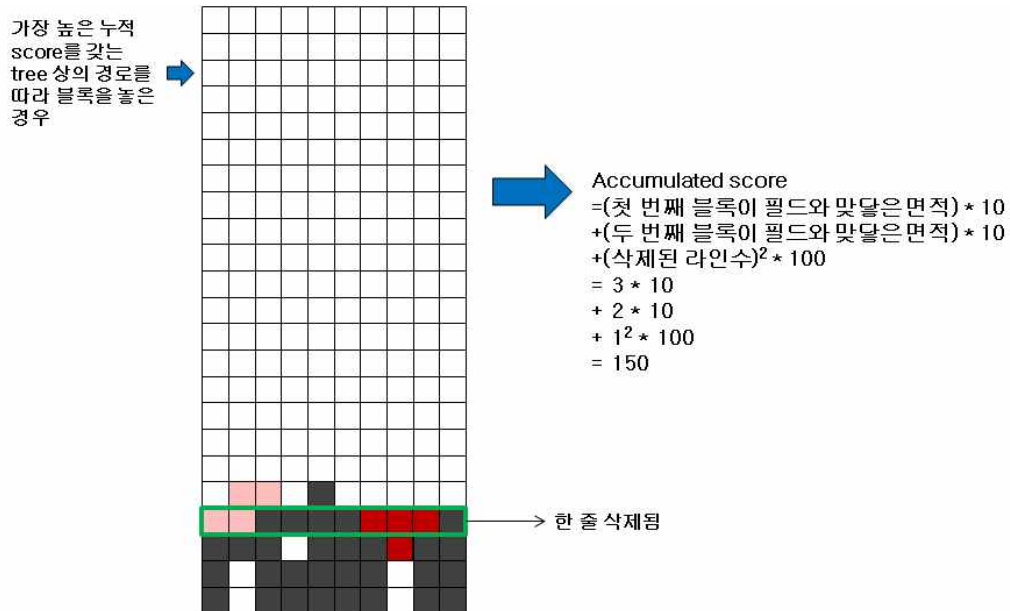
[그림 4] 블록  를 고려해서 구성된 tree

- 이제, 모든 블록과 그 회전수를 고려해서 tree 구성이 완료되었으므로, 마지막 노드 (leaf node)의 누적 점수를 고려해서 추천할 노드를 결정한다. 즉, 가장 많은 누적 점수를 얻게 되는 플레이 시퀀스(굵은 화살표)와 추천할 노드는 다음과 같다.



[그림 5] 가장 높은 점수를 달성하는 tree상의 경로와 추천할 노드

- 위 그림에서 가장 많은 점수를 얻는 플레이 시퀀스는 root 노드, tree의 level(or depth)이 1일 때, child 노드 1, tree의 level(or depth)이 2일 때, child 노드 8을 따라가는 경우(굵은 화살표)이다. 이와 같이 플레이 하는 경우는 실제로 한 줄을 지우게 되는 결과를 갖게 되어 100점을, 현재 블록과 다음 블록1을 필드에 놓았을 때, 필드와 각 블록이 맞닿은 면적의 수는 각 3, 2로 두 가지 모두 고려하면 5이므로, 50점을 획득한다. 따라서 총 150점을 얻게 된다. 이와 같은 결과는 2개의 블록이 놓여진 테트리스 필드를 나타내는 그림을 통해서 확인할 수 있다.



[그림 6] 2개의 블록이 놓여졌을 때, 얻은 점수

지금까지 설명된 tree 구조를 바탕으로 해서 추천된 블록의 위치가 결정된다. 비록 여기서는 2개의 블록을 고려했지만, 만약 더 많은 다음 블록을 알고 있고, 이들을 모두 고려하여 tree를 구성한다면, 적은 수의 블록을 고려했을 때 보다 더 나은 점수를 얻을 수 있는 위치를 추천할 수 있다. 실제로 위의 경우에서 살펴본 바와 같이 모든 플레이 시퀀스를 고려해서 테트리스를 게임을 수행하는 것은 많은 블록이 고려된다면, 블록이 놓일 수 있는 모든 위치를 고려하기 때문에 테트리스 게임에서 높은 점수를 얻을 수 있는 위치를 추천할 수 있는 방법임을 알 수 있다. 하지만, 여기에도 문제점이 발생하게 된다. 이 문제점에 대해서는 다음 section에서 살펴본다.

5. 추천 시스템의 비효율성

5-1. 시간의 비효율성

위와 같이 추천 시스템을 위한 tree구조를 구축할 때, 예제에서 보다 더 많은 블록을 고려한다고 가정하자. 그리고 최악의 경우 각 블록에 대해서 회전수와 필드를 고려해서 블록을 놓을 수 있는 좌표의 수, 즉 각 노드에 대해서 생성되는 child 노드들의 수는 총 34가지라고 가정하자. 만약 현재 블록을 포함해서 다음 블록 2개까지, 즉 총 3가지 블록을 고려하면, tree에서 생성하고 고려하게 되는 최대 노드의 수는 $34^3=39304$ 개이고, 4가지 블록을 고려한다면, $34^4=1336336$ 개이다. 실제로 블록 n개를 고려하게 되면, 노드의 수는 34^n 으로 노드의 개수는 고려되는 블록의 수가 증가함에 따라 기하급수적

으로 증가하게 된다. 따라서 더 높은 점수를 얻을 수 있는 블록의 위치를 추천하는 추천 시스템을 만들기 위해 더 많은 블록 수를 고려하는 것은 굉장히 많은 시간을 소모하고, 실제로 이를 위한 계산 시간으로 인해 테트리스 플레이를 지루하게, 혹은 불가능하게 만든다.

5-2 공간의 비효율성

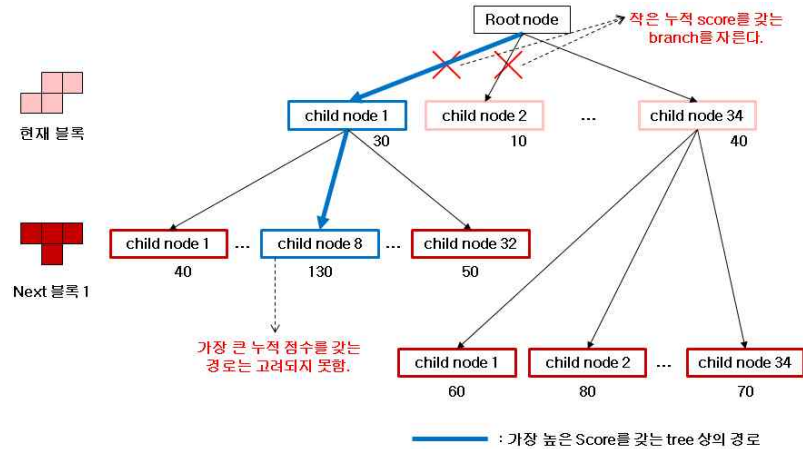
위의 시간의 비효율성에서 살펴보았듯이, 기하급수적으로 늘어나는 노드의 수만큼 노드가 차지하는 메모리 공간도 많아진다. 또한, 각 노드는 10×22 의 테트리스 필드를 저장해야 하므로, 하나의 노드가 차지하는 공간도 적지 않다. 만약 하나의 노드가 차지하는 공간을 c 라고 한다면, n 개의 블록을 고려할 때 소모되는 메모리 공간은 $c \times 34^n$ 이다. 여기서도 마찬가지로 추천을 위해 고려되는 블록의 수가 증가할수록 사용되는 메모리 공간의 수는 기하급수적으로 증가하게 되고, 이같은 많은 공간의 사용은 메모리 overflow 등의 문제와 함께 다른 프로그램의 수행에도 많은 영향을 미치게 된다.

5-3 비효율성을 해결하기 위한 방법의 예제

5-1, 5-2에서 살펴보았듯이 모든 플레이 시퀀스를 고려하게 되면 시간, 공간의 비효율성이 문제가 된다. 따라서 이를 해결하기 위한 방법이 필요하다. 이 방법에는 다양한 방법들이 존재하지만, 여기서는 그것들을 해결하기 위한 대표적인 2가지 방법을 소개하고, 아주 간단한 예제를 통해 각각의 특징을 살펴본다.

1. Pruning tree - 시간의 비효율성을 해결하기 위한 방법

시간의 효율성을 해결하기 위한 가장 대표적인 방법은 tree를 구성할 때, tree의 branch를 자르는 방법이다. Tree의 branch를 자름으로써, 모든 경우(모든 플레이 시퀀스)를 고려하지 않아도 되기 때문에, 구성된 tree에서 가장 좋은 성능을 보이는 경로를 찾기 위해 소모하는 시간이 줄어들게 된다. 하지만, branch를 자를 때 주의할 점은 자르게 될 branch에서 가장 좋은 성능(여기서는, 누적 점수)을 갖는 경로가 발생하지 않도록 branch를 잘라야 한다. 하지만, 자르려고 하는 branch를 갖는 경로가 어떤 성능(또는 결과)를 가질지 예측하여, 해당 branch가 좋지 않는 성능을 가질 것이라는 것이 명확한 경우에만 branch를 잘라야 하므로, 굉장히 어려운 과정임을 예측할 수 있다. 다음 간단한 예제를 통해 tree의 branch를 자르는 대표적인 한 가지 방법과 제시된 방법의 단점을 살펴본다. 앞에서 추천 시스템의 tree 구조를 설명하는 예제를 다시 생각해보자. 그리고 각 블록을 고려할 때, 작은 누적 점수를 갖는 branch를 자르는 방법을 생각하자. 이 때, tree의 level 1에서 잘려진 branch들 중 root 노드와 child 노드 1사이의 링크로부터 형성되는 경로가 가장 높은 누적 점수를 갖게 되는데, 여기서 사용된 방법은 이를 고려하지 못하는 문제가 발생한다. 이는 다음 그림을 통해서 할 수 있다.

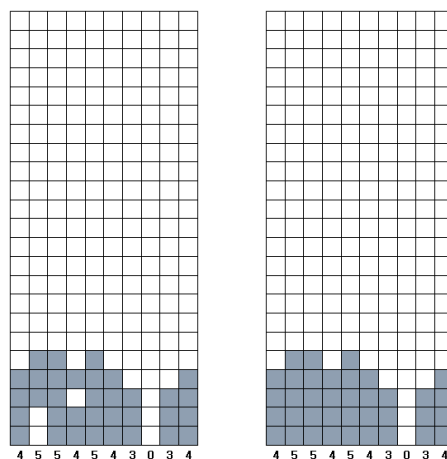


[그림 7] 작은 누적 점수를 갖는 링크를 자른 경우 발생하는 문제

위 그림에서 볼 수 있듯이, tree의 level 1에서 작은 누적 점수를 갖는 branch, 즉 root 노드와 child 노드 1사이의 branch와 root 노드와 child 노드 2사이의 branch를 자르게 되면, 고려하게 되는 branch의 수가 상당히 줄어들게 되어 시간적인 부담이 줄어들게 되지만, 위의 경우에서 살펴보듯 가장 좋은 점수를 얻을 수 있는 플레이 시퀀스(굵은 화살표로 표시)는 고려되지 못하는 결과가 발생한다. 이와 같이 tree의 branch를 pruning하는 방법을 통해서 고려되는 경우의 수를 줄이고자 할 때는 이와 같은 문제가 발생하지 않도록, 신중하게 pruning을 수행해야 한다.

2. Data Simplification - 공간의 비효율성을 해결하기 위한 방법

공간의 비효율성을 해결하기 위한 가장 좋은 방법은 저장되는 정보를 단순화하는 것이다. 추천 시스템의 tree 구성할 때 각 노드에서 가장 소모적으로 사용되는 부분은 필드의 정보를 저장하는 부분이다. 이 부분은 모든 필드의 정보를 배열로 저장하게 되기 때문에 많은 공간이 소모된다. 이 때, 필드의 높이만을 고려해서 저장한다면, 특히 많은 블록의 정보를 고려해서 tree를 구성할 때, 상당히 많은 공간을 줄일 수 있다. 다음 그림은 이를 보여주고 있는데, 왼쪽 그림은 실제 필드와 하단에 필드의 높이를 나타내는 그림이고, 오른쪽 그림은 저장된 필드의 높이만을 고려해서, 복원한 필드를 나타내는 그림이다.



[그림 8] 실제 필드(왼쪽)과 높이만을 저장한 후 복원한 필드(오른쪽)

이 방법은 필드의 정보를 저장하는 공간을 줄일 수 있고, 필드상에 블록이 놓일 위치를 결정하는데 아무런 문제가 없다. 하지만, 위의 그림과 같이 필드 정보를 저장하게 되면 실제 필드의 상세한 정보, 여기서는 실제 필드 상에 어느 부분이 비어있는지에 대한 정보를 손실하게 되는 위험성이 존재한다. 따라서 자료(data)를 단순화하는 방법도 이와 같은 단점을 최소화하여 프로그램의 수행에 큰 영향을 미치지 않으면서, 적은 양의 정보를 저장할 수 있는 방법이 필요하다.

6. 추천 시스템의 구현 이슈 및 구현 내용

6-1 추천 시스템 구현

앞에서 살펴본 추천 시스템을 위한 tree 구조에서처럼, 테트리스 게임에서 모든 가능한 플레이 시퀀스를 저장하고, 이에 대해 평가하여 추천될 블록의 좌표 및 회전수를 결정하는 tree를 구성하는 것이 목표이다. 이를 구현하여, 모든 가능한 플레이 시퀀스에 대한 정보를 저장하는 tree를 사용한 추천 시스템의 비효율성을 진단할 수 있다.

6-2 추천 시스템 구현 내용

1. 추천 시스템 구현을 위한 tree의 노드 구조(node structure)

```

Typedef struct _Node {
    // must-have elements
    int level;
    int accumualtedScore;
    char recField[HEIGHT][WIDTH];
    struct _Node **child;
    // optional elements
    int curBlockID;
    int recBlockX, recBlockY, recBlockRotate;
    struct _Node *parent;
    ...
} Node;

```

[그림 9] 추천 시스템 구현을 위한 tree의 노드 구조

구조체의 필수 원소

- int level : tree의 level(or depth)를 나타낸다.
- int accumulatedScore : 누적된 점수
- char recField[HEIGHT][WIDTH] : 추천된 블록의 위치와 회전수를 고려해서 블록을 테트리스 필드에 놓았을 때의 필드 상태
- struct _Node **child : tree에서 각 child를 가리키는 노드 포인터(node pointer)이고, child의 수만큼 동적 할당을 수행해야 한다.

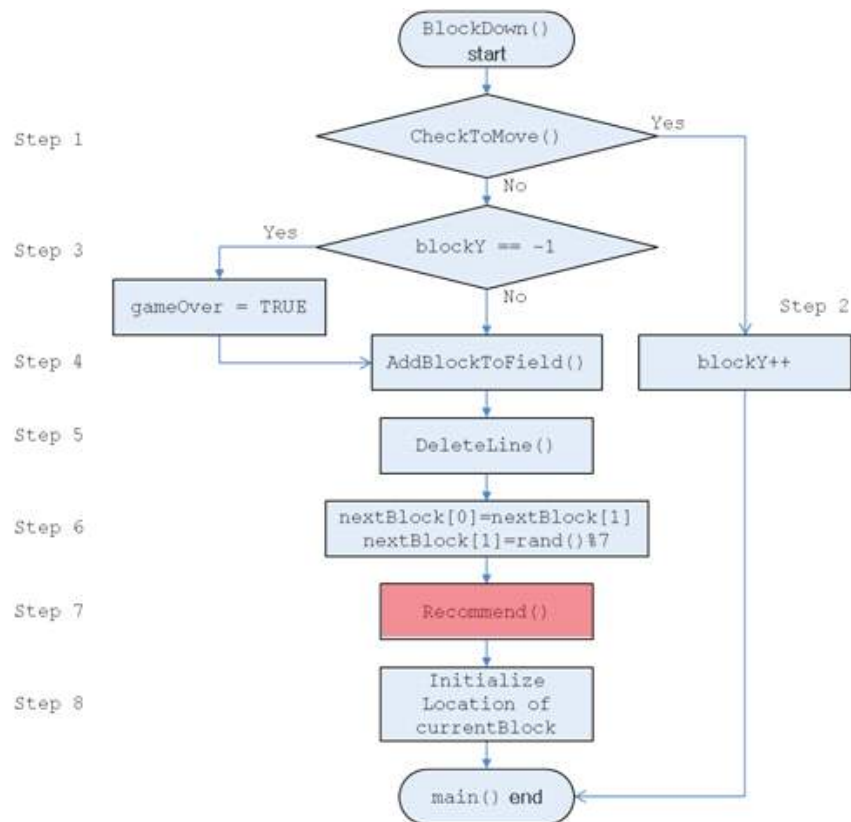
☞ 선택할 수 있는 구조체의 원소

- int curBlockID : tree에서 고려되는 블록의 ID
- int recBlockX, recBlockY, recBlockRotate : 블록의 위치와 회전수
- struct _Node *parent : tree에서 부모를 가리키는 노드 포인터(node pointer)

2. 구현 내용

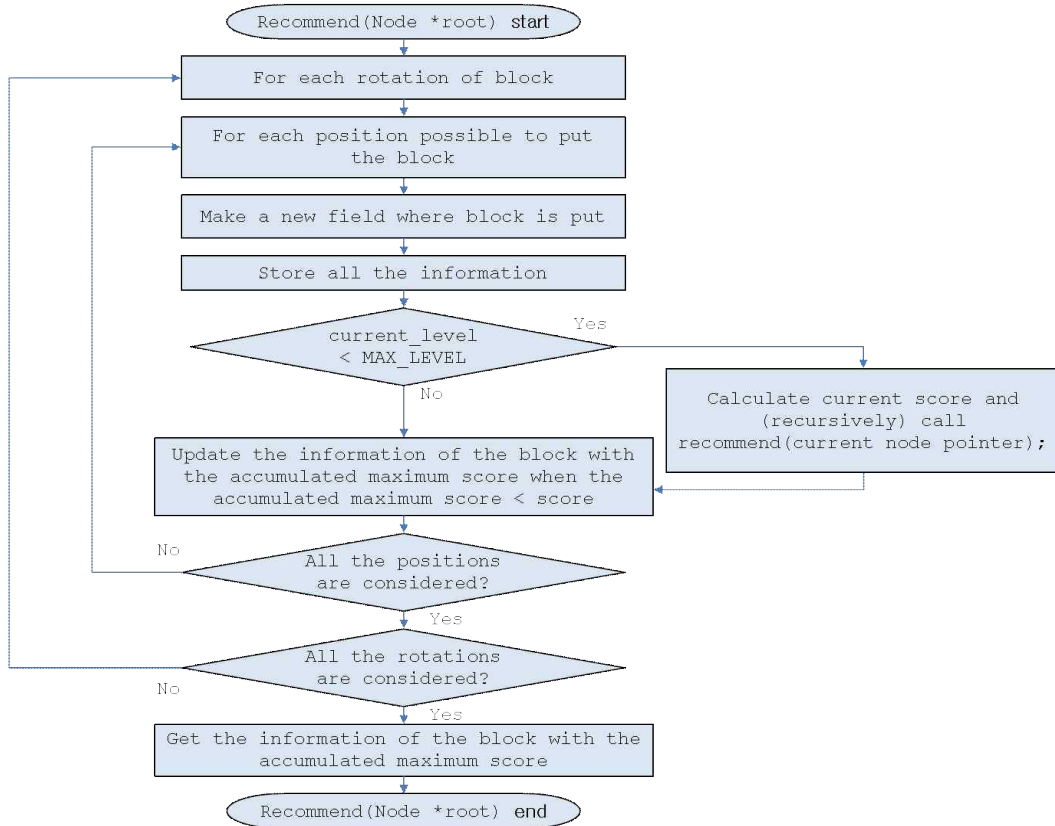
☞ int recommend()

- input : Node *root : root 노드(실제로 parent 노드)의 주소를 저장하고 있는 포인터
- output : tree에서 고려된 가장 큰 누적 score
- 현재 블록과 다음 2개의 블록을 고려해서 모든 플레이 시퀀스를 나타낼 수 있는 tree를 구성하고 tree의 정보를 바탕으로 사용자가 좋은 점수를 얻을 수 있는 현재 블록의 위치를 계산하는 기능을 갖는다. 이 함수는 전체적인 구성에서 다음과 같이 BlockDown() 함수에서 사용된다.
- 단, tree에서 고려하는 블록의 수를 조정하는 것에 의존하지 않고 함수가 수행되어야 한다. 만약 recommend() 함수가 블록의 수를 3개(현재 블록과 다음 2개의 블록, tetris.h에서 VISIBLE_BLOCKS)를 고려해서 작성되었다면, 다음 블록의 수가 늘어난 경우, 즉 VISIBLE_BLOCKS가 다른 수로 정의되어도, recommend() 함수에는 아무런 수정 없이 정상적으로 동작해야 한다.



[그림 10] recommend()가 포함된 BlockDown() 함수의 flow chart

실제 추천 시스템을 위한 tree를 구성하고, 추천 위치를 찾는 recommend() 함수의 flow chart는 다음과 같다.



[그림 11] recommend()의 flow chart

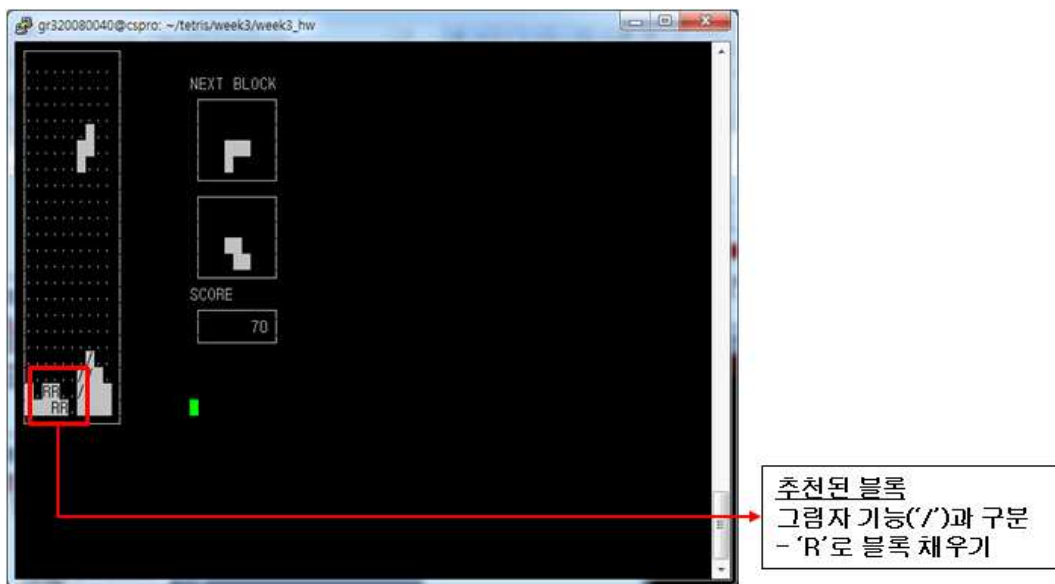
위의 flow chart에서 보여지듯이, 고려하고 있는 블록의 위치는 parameter인 root(실제로 parent)의 필드 정보를 기반으로 결정하고, 고려할 수 있는 모든 블록의 위치에 대해서 노드를 생성하여 정보를 저장하게 된다. 각 노드를 생성한 후에는 추천 시스템의 tree 구조에서 살펴본 것처럼, 각 노드의 부모(parent) 노드의 필드 상태를 바탕으로 블록의 좌표와 누적점수를 고려해야 한다. 따라서 이 부분을 구현하기 위해서는 recommend() 함수의 재귀적 호출이 필요한데, 이 부분이 위 flow chart에서 가장 중요한 부분이다. 위의 flow chart에서는 “(recursively) call recommend(current node pointer)”에 해당한다. 즉, 다시 말해서 이는 현재 고려하는 블록의 각 노드의 위치와 점수는 부모 노드의 필드 정보와 점수를 바탕으로 고려할 수 있도록 해주는 장치이다.

이 함수 recommend()의 flow chart에 대해 간략히 설명하면, 다음과 같다. 먼저, tree의 depth는 고려되는 블록의 수와 같다는 사실을 기억하자. 함수 recommend()에서는 tree의 children을 만드는 과정이 주로 진행되는데, 이는 2중 loop문을 사용해 구현한다. 첫 번째 loop는 블록의 회전수에 대한 것이고, 두 번째 loop는 필드 상에 놓일 수 있는 좌표의 수에 대한 것이다. 이 때, 주어진 블록의 회전수에 대해 필드 상에 놓일 수 있는 위치는 블록을 필드의 가장 왼쪽에서부터 오른쪽으로 이동하면서 블록이 고려될 수 있는 좌표들을 찾고, 찾은 좌표의 수를 센다. 이 때, 현재 고려하는 블록의 회전수(첫 번째 loop)와 좌표(두 번째 loop)는 결정되므로, 이를 바탕으

로 블록이 필드(이 필드가 바로 root(부모) 노드의 필드 정보이다)에 놓여졌을 때의 필드 정보를 업데이트하여 저장한다. 그리고 tree의 현재 level이 최대 level수보다 작으면, 누적 점수를 얻기 위해서 현재 고려하는 노드의 주소를 parameter로 하여 재귀적 함수 호출을 하게 된다(이 재귀적 호출은 tree의 하나의 노드가 생성될 때마다 호출되게 된다). 만약 tree의 현재 level이 최대 level 수와 같으면, 현재 점수만을 계산한다. 이 과정이 완료되면, 최대 누적 점수를 갱신하고, 그 때의 블록 정보(블록의 위치와 회전수)를 기억한다. 그리고 이 모든 과정이 완료되면, 위 과정을 통해 찾은 최대 누적 점수와 블록의 위치를 얻은 후, recommend() 함수를 종료한다.

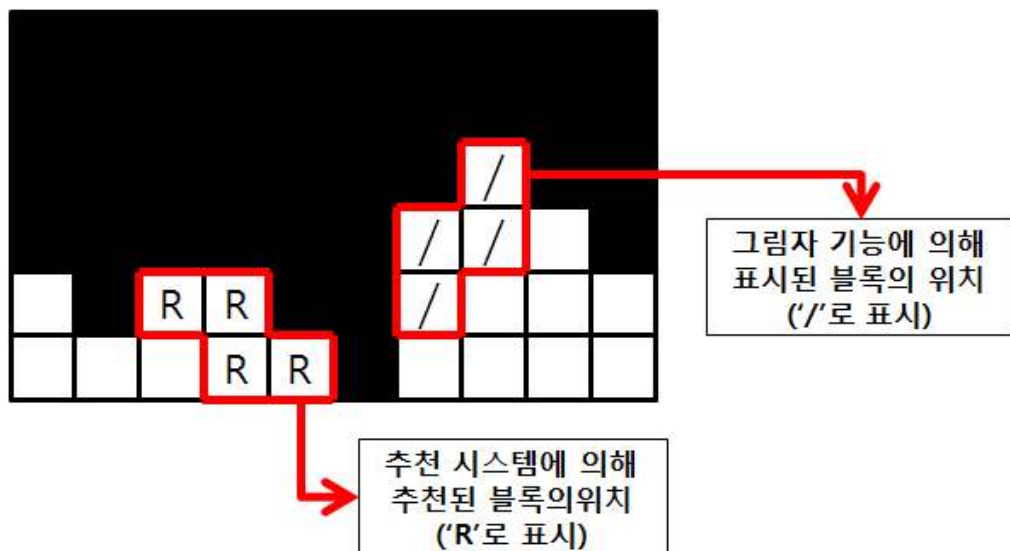
6-3 추천 시스템 구현 결과

위의 구성된 tree를 바탕으로 사용자에게 높은 점수를 얻을 수 있는 블록의 위치를 추천한다. 블록이 추천되는 위치는 character 'R'을 사용해서 나타낸다.



[그림 12] 추천 시스템 구현 화면

위의 실행 화면을 확대하면 다음과 같다.



[그림 13] 추천 시스템과 그림자 기능의 구분

7. 테트리스 프로젝트 3주차 실험 평가

7-1 recommend() 함수

1. 모든 플레이 시퀀스(play sequence)가 고려되어 tree가 구성되는가?
2. 구성된 tree에서 누적 score가 가장 높은 경로를 찾고, 현재 블록이 놓여질 위치에 대한 정보를 정확히 추출해 내는가?
3. 화면에 정확하게 그 위치를 표시하는가?
4. VISIBLE_BLOCKS의 값이 수정되어도 recommend() 함수는 아무런 수정 없이 잘 동작하는가?

7-2. 수행시간 측정 방법(수정 예정)

1. 수행시간 측정

C 프로그램에서 다음 방법으로 수행 시간을 측정하고 싶은 부분들에 대해 수행 시간을 측정할 수 있다.

```
...
#include <time.h>
...
time_t start; // 수행시간 측정 대상인 코드들이 시작되는 부분에서의 시각
time_t stop; // 수행시간 측정 대상인 코드들이 끝나는 부분에서의 시각
double duration; // 초 단위의 수행시간
...
start = time(NULL);
// 수행시간 측정 대상인 코드들
stop = time(NULL);
duration = (double)difftime(stop, start);
...
```

[그림 14] 수행시간 측정 방법

2. 메모리 사용량 측정

자료구조 구현 결과물의 실제 메모리 사용량을 측정하기 위해 다음의 프로시저를 활용할 수 있다. 모든 자료구조 구현 결과물을 하나 혹은 둘 이상의 리스트로 표현된다고 하자. 이 때, 각 리스트의 노드는 서로 공유될 수 있다. 다음 프로시저(procedure) evalsize(l)은 리스트 l 의 시작 노드에서부터 각 노드의 크기를 누적하여 메모리 사용량을 측정하는 프로시저이다.

```
/*  $l$ : a list */
/*  $h_l$ : the head node of the list  $l$  */
/*  $n_l$ : the number of the nodes of the list  $l$  */

1: evalSize( $l$ )
2: if  $h_l$  is visited, then return 0
```

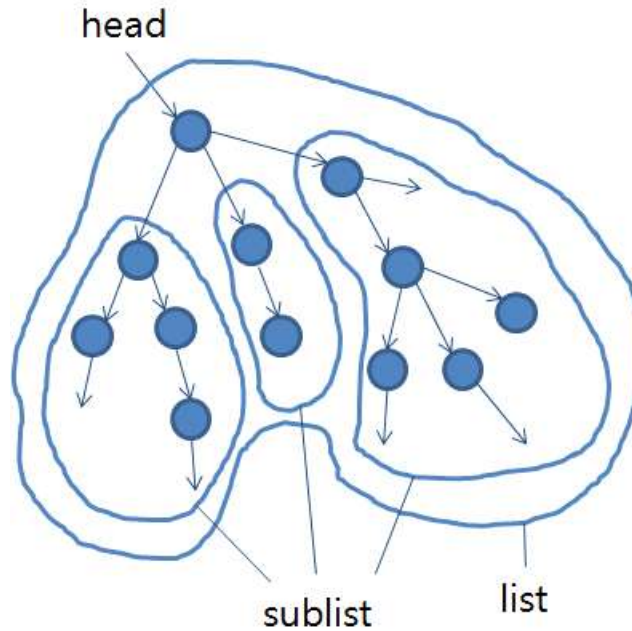
```

3:   $s := \text{the size of the node } h_l$ 
4:  if  $n_l > 0$ , then
5:    for each largest sublist  $l'$  such that  $h_l \notin l'$ ,
6:       $s := s + \text{evalSize}(l')$ 
7:  return  $s$ 

```

[그림 15] 메모리 사용량 측정 방법

위 라인 5에서 “each largest sublist”란, 주어진 리스트 l 의 시작 노드를 제거함으로써 얻을 수 있는 각각의 리스트를 말한다. 첫 $\text{evalSize}(l)$ 호출에서 아래의 head는 h_l 가 되고, 각 sublist는 위의 라인 5에서 고려되는 each largest sublist가 된다. 아래 그림은 [그림 15]의 메모리 사용량 측정 방법을 개념적으로 나타낸 그림이다.



[그림 16] 메모리 사용량 측정 방법을 개념적으로 도식화한 그림

만약, 자료구조 구현 결과물이 복수의 리스트로 표현된다면, 각 리스트마다 위의 프로시저를 수행하고 난 결과값들을 더하여 해당 자료구조를 구현하는데 필요한 전체 메모리 사용량을 측정할 수 있다. 단, 임의의 노드가 여러 노드 사이에서 공유된다면(여러 노드가 포인터를 사용하여 하나의 노드를 가리키고 있다면), 각 노드의 방문여부를 기록하여 이미 방문한 노드에 대해서는 메모리 사용량을 중복 합산하지 않도록 해야 한다. 다음 예제는 수행 시간 측정과 메모리 사용량을 측정하는 방법을 사용하여 주어진 정렬되지 않은 linked list를 insertion sort를 사용해서 정렬하는 예제이다.

```

/* goal: evaluate the memory size for a randomly generated
linked list and the elapsed time for sorting the list. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define REMAINDERS 10000

typedef struct _Node {
    int n;
    int *item;
    struct _Node *next;
} Node;

long evalSize(Node* head) {
    if (head->next == NULL) return 0;
    return sizeof(Node)+(head->n * sizeof(int)) +
        evalSize(head->next);
}

void sort(Node *head) {
    Node *node, *min;
    int t, *tptr;

    while (head->next != NULL) {
        min = node = head;
        while (node->next != NULL) {
            if (node->n < min->n) min = node;
            node = node->next;
        }
        /* swap */
        t = head->n;
        tptr = head->item;
        head->n = min->n;
        head->item = min->item;
        min->n = t;
    }
}

```

```

        min->item = tptr;

        head = head->next;
    }
}

int main(void) {
    Node *head, *node;
    int n, i;
    time_t start, stop;
    double duration;

    /* generate an arbitrary linked list */
    srand(time(NULL));
    n = rand()%REMAINDERS + 1;
    node = head = (Node *)malloc(sizeof(Node));
    node->next = NULL;
    for (i = 0; i < n; ++i) {
        node->n = rand()%REMAINDERS + 1;
        node->item = (int *)malloc(node->n * sizeof(int));
        node->next = (Node *)malloc(sizeof(Node));
        node = node->next;
        node->next = NULL;
    }
    printf("The total memory for the list : ");
    printf("%ld bytes\n", evalSize(head));

    /* sort the list */
    start = time(NULL);
    sort(head);
    stop = time(NULL);
    duration = (double)difftime(stop, start);
    printf("The total time for sorting : %lf seconds\n",
        duration);
    return 0;
}

```

[그림 17] 수행 시간과 메모리 사용량을 측정하는 예제(insertion sort)

위 예는 전체 크기가 임의로 정해지도록 linked list를 생성하여 메모리를 얼마나 사용하는지 측정하고, 그렇게 생성한 linked list를 삽입정렬 알고리즘으로 정렬하여 수행시간이 얼마인지를 알아보는 예이다. 이미 말했듯이, 이 예에서 사용하는 자료구조는 linked list로, 전체 노드 수는 임의로 결정된다. 뿐만 아니라, 각 노드에 동적으로 할당되는 배열이 존재하여, 노드의 크기도 임의적이다. 이렇게 생성한 리스트를 앞서 제시한 evalSize 프로시저를 통해 전체 리스트의 크기를 측정한다. 그 후에 리스트를 정렬하는데, 삽입정렬의 시작지점과 끝 지점에서의 시각을 기록하여 그 차이를 통해 정렬하는데 걸린 시간을 측정한다. 다음은 실행결과이다.

```
The total memory for the list : 175191676 bytes
The total time for sorting : 7.000000 seconds
```

8. 테트리스 프로젝트 3주차 숙제 및 보고서 작성

8-1. 예비보고서

1. 교재를 참조하여 테트리스 프로젝트 3주차에 구현하는 추천 기능은 어떤 원리로 작동되는지 설명하십시오. 그리고 추천 기능을 구현하는 tree 구조의 장점(효율성)과 단점(비효율성)을 기술하십시오.
2. Tree 구조의 비효율성을 해결할 방법에 대해서 2가지 이상 생각하고, 그 idea를 기술하십시오.

8-2. 숙제

☞ 숙제1: modified_recommend()

이 함수는 앞에서 설명한 tree구조의 시간과 공간의 비효율성을 해결하기 위해서 pruning 또는 data simplification의 방법 또는 자신만의 효율적인 방법을 고안해서 recommend() 함수를 수정하여 작성한다. 이 함수를 작성할 때는 자신이 고안한 방법이 다른 방법들과 비교해서 높은 시간 효율성과 공간 효율성을 달성할 수 있는지 고려해야 한다.

- modified_recommend() 함수의 평가

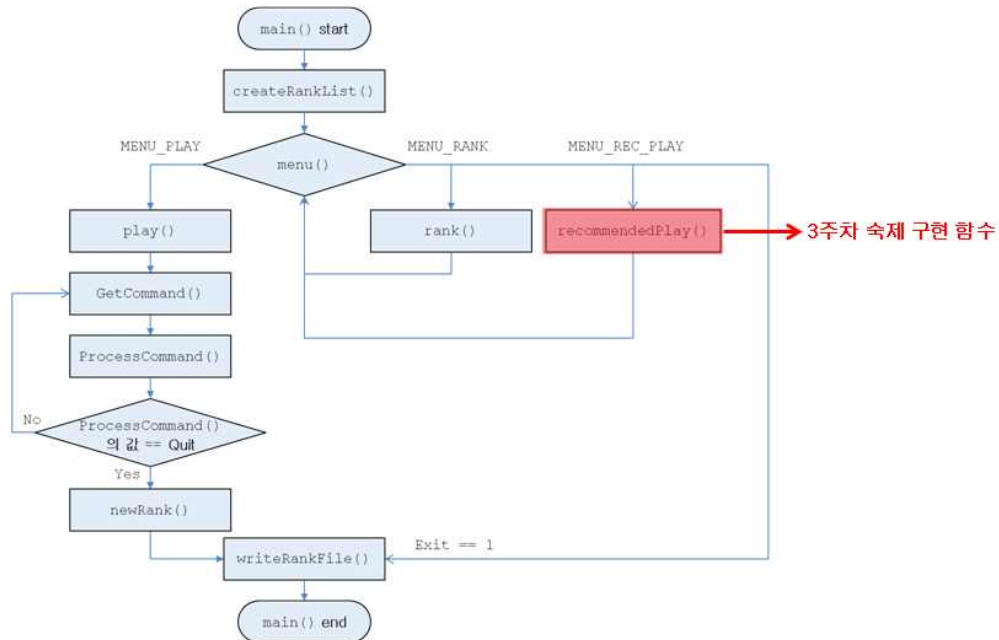
새로운 tree를 구성하여 테트리스를 플레이한 경우 일정 시간 t 동안 얻은 누적 score를 $score(t)$ 라 하고, t 시간 동안 tree의 정보를 구성하기 위해 소비된 누적 시간을 $time(t)$, t 시간 동안 tree의 정보를 구성하기 위해(자료구조에 정보를 저장하기 위해) 소비된 공간을 $space(t)$ 라고 하면, 시간 효율성과 공간 효율성은 다음과 같이 정의된다.

$$\begin{aligned}\text{시간 효율성}(t) &= score(t)/time(t), \\ \text{공간 효율성}(t) &= score(t)/space(t).\end{aligned}$$

이 때, 시간, 공간 효율성이 더 높은 tree를 구현한 학생에게 더 높은 점수를 부여한다. 이 효율성은 다음 recommended play를 통해서 측정된다.

☞ 숙제 2: recommended_play 모드

함수 `modified_recommend()`에서 추천된 블록의 정보만을 사용해서 테트리스를 게임을 하는 테트리스 게임 모드인 `recommended play`를 구현한다. 이 기능은 테트리스 프로그램의 메뉴 3번에 해당한다. 이 기능은 `play()` 함수를 유사하게 구현되는데, `play()` 함수와는 사용자가 플레이하는 것이 아니라, 추천 위치를 계산하고 그 위치에 블록을 놓는 동작을 일정시간마다 반복하도록 구현한다.



[그림 14] 전체적인 과정에서 `recommended_play()` 구현 위치

☞ 숙제 3: 시간 및 공간 효율성 계산

앞에서 설명된 `recommended_play()`를 사용해서, 플레이를 한 후, 화면에 플레이 시간 t , $score(t)$, $time(t)$, $space(t)$ 를 출력하고, 이로부터 계산된 시간, 공간 효율성도 화면을 통해 확인할 수 있도록 출력한다.

8-3. 결과 보고서

아래의 사항을 작성하여 다음 실험 시간에 제출하시오.

1. 실습 시간에 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술하시오. 완성한 알고리즘(추가 구현하게 되는 효율성을 고려한 tree도 포함)의 시간 및 공간 복잡도를 보이시오.
2. 모든 경우를 고려하는 tree 구조와 비교해서 어떤 점이 더 향상되고, 어떤 점이 그렇지 않은지 아울러 기술하시오.
3. 테트리스 프로젝트 3주 과정을 통해 습득한 내용이나 느낀 점을 기술하시오.

실험 PRJ-1 3주차 추천시스템 예비보고서

전공:

학년:

학번:

이름

실험 PRJ-1 3주차 추천시스템 결과보고서

전공:

학년:

학번:

이름

