8주차 예비보고서

전공: 수학과,컴퓨터공학과 학년: 3학년 학번: 20181256 이름: 김도현

1-1.

이번에 진행할 실습 내용은 테트리스 게임에서 어느 곳에 블록을 두어야 가장 점수가 높게 나올 수 있는지 추천하는 기능을 구현하는 것이다. 이는 tree 자료구조를 통해 구현이 가능하다. 현재 block, next block1, next block2에 대한 위치 ,ID, 회전수, field 정보 등을 묶어서 가능한 모든 경우의 수를 계산하고 가장 높은 accumulated score 를 가지는 위치에 추천을 해준다. Tree의 처음 level 1 에서는 현재 block에 따른 모든 경우의 수를 고려하여 각각의 node들을 만들어 해당 정보들을 저장한다. 그리고 level 2에서는 level 1에서 만든 각 node들에 nextblock 에 따른 child node를 가지도록 모든 경우의 수를 node를 만들어 연결하고 이 과정을 원하는 depth=level 까지 진행하면 될 것 이다. 이러면 결국 root node 로부터 마지막 level의 childnode 까지 고려하여 가장 높은 accumulated score를 가진 노드가 있기에 거기에 따라 다시 parent node를 타고 올라가 현재 block의 추천 위치를 return 하면 된다.

1-2.

Tree는 모든 경우의 sequence를 쉽게 다 확인 할 수 있다는 장점이 있다. 2주차 예비보고서에서 고민했던 array 같은 경우 continuous하기 때문에 현재 block들의 내용을 저장한 array[k]에서 nextblock의 내용을 저장하려면 다차원 array를 사용해야 하는 매우 복잡한 과정을 거쳐야 한다. 또한 이전에 사용한 linkedlist는 node가 있을 때 link 한 개로만 앞, 뒤 node와 직접적으로 연결되어 있었기 때문에 하나의 node에서 여러 상황을 고려하지 못한다. 하지만 tree는 root node부터 마지막 level에 있는 여러 child node까지 거의 바로 접근 가능하고 새로운 node 연결, 또 연결만 제거하면 해당 node 삭제도 용이하여(pruning) 이번 실습과 같이 추천기능, 즉 모든 경우를 고려해야하는 상황에 tree는 효율성이 높다.

하지만 tree의 가장 큰 문제점은 시간적, 공간적 낭비가 클 수 있다. 왜냐하면 모든 경우의 수를 고려하는 tree는 우리의 실습 내용에서 하나의 block으로 생성가능한 node의 수가 34개라고 가정하자면 4개의 block만 고려해도 depth가 4일 때는 34\*34\*34\*34=1336336 개의 node들을 고려해야 하고 전체 node 수를 생각하면 depth1, depth2, depth3, depth4(1+ 34+ 34\*34 +34\*34\*34+ 34\*34\*34\*34) 노드 정도가 만들어진다. 만약 n개의 block을 고려한다면 이 수는 더욱 커지기 때문에 계산 과정이 매우 많다. 즉 엄청난 시간적 비효율이 생길 것이다.

이와 비슷하게 이런 수 많은 node들을 저장하는 메모리 역시 많이 들것이다. 메모리 overflow가 발생할 가능성이 높고 해당 프로그램이 메모리에 차지하는 비율이 많을 수록 다른 작업을 수행할 때도 메모리가 금방 꽉 차 프로그램이 죽어버리는 경우도 발생할 것이다.

2-1.

사실 이런 문제점을 해결하기 위한 방법은 크게 2개로 나뉘는데 하드웨어적 측면, 소프트웨어적 측면(시간, 공간적 비효율성을 줄이기 위한 코드 수정) 이다.

비록 비효율성을 줄이지는 못하더라도 tree의 가장 큰 장점인 모든 경우의 수를 고려하는 거에 집중을 하자면 결국 하드웨어를 바꾸면 해결이 된다. 과거에 공간 복잡도가 중요했던 이유는 메모리와 관련한 기술이 부족해서 메모리 사용에 민감했기 때문이다. 이처럼 더욱 큰 메모리를 장착하고 더 계산 능력이 빠른 cpu, gpu를 사용하면 프로그램은 잘 돌아갈 것이다.

하지만 이는 금전적인 문제도 있고 이 문제가 요구하는 답은 소프트웨어적 측면이라고 생각한다.

그래서 tree 의 문제는 모든 경우의 수를 고려하는 것이기 때문에 이를 줄여야 한다.

그래서 고려하는 block의 수를 줄이는 방법이 있다. 만약 고려 block을 1개 줄이면 현재 고려하는 모든 node의 합에서 34를 나누게 된다. 이는 계산할 과정이 1/34로 줄어들게 된다는 의미이다. 그리고 공간적 측면에서도 node 저장할 개수가 줄어들기 때문에 효율이 더 좋다. 하지만 이는 tree의 가장 좋은 장점인 모든 경우의 수를 전부 확인은 못해 아마 score 는 더 낮게 나올 수는 있으나 이는 결국 trade off 이다.

또 다른 방법은 pruning이다. 가지치기라는 말로 그 중 greedy 한 방법을 생각해보면 해당 parent node 에서 가장 높은 score를 가지는 child node를 제외한 모든 노드를 다 제거하고 그 노드에서 다시 child node를 계산하는 방법이다. 이는 시간적, 공간적 측면을 매우 효율적으로 만들어 주지만 제거된 곳에서 실제 가장 높은 score를 가지는 node가 존재할 수도 있기 때문에 추천문제 측면에서는 효율적이지 못할 수 있어 이 역시 trade off 이다. 또 다른 방법은 확률적으로 해당 노드의 score가 높으면 다음 childnode에서 score도 높을 가능성이 있기 때문에 parent node에서 특정 점수를 넘기지 못하면 제거하는 방법도 있다. 가장 높은 score를 제외하고 child node들 중 일부 node 들만 무작위로 제거 하는 방법도 있을 수 있다. 하지만 이 역시 모든 경우의 수를 고려한 tree와 비교했을 때 같은 결과를 도출하지 못할 수 있다. 그래도 이 두 방법 모두 가장 높은 score 를 가지는 child node를 제외한 나머지 모든 node를 제거하는 것보다는 고려하는 node 수가 줄어 시간적, 메모리 즉 공간적 측면에서 효과적일 것이다.

다음은 data simplication으로 node들은 field의 정보를 저장하고 있는데 모든 정보를 저장하지 않고 단순히 쌓인 block의 가장 윗 부분 정보만 저장하여 메모리 효율을 높이는 과정이다. 결국nextblock의 위치를 결정짓는데 큰 문제는 없지만 이 역시 field에 대한 정보 손실이 존재한다.

마지막 방법은 모든 경우의 수를 전부 child node로 두어 여러 개를 사용해 메모리를 많이 사용하는 것이 아니라 child node 1개만 사용하여 여기에 2중 for문이 돌아갈수록 계속 update 하는 방법도 존재 한다. 이는 원래 기존의 방법은 한 level 당 34개의 node들을 만들어야 하지만 1개의 node만 만들면 된다는 것이다. 각 재귀함수에서도 노드를 1개만 만들기 때문에 한 level 당 거의 33개의 node를 줄이는 것이다. 이는 엄청난 메모리 효율이다. 이게 가능한 이유는 결국 재귀 함수 안의 변수인 accumulatedScore만 순차적으로 for문이 돌아갈 때마다 update 해주면 될 것이기 때문이다. 그리고 이를 recommendScore와 비교하여 결국 최종 recommendScore를 return 해주면 된다. 이는 기존의 방법과 동일한 성능을 보여주지만 메모리 사용은 매우 적을 것으로 예상한다.