8주차 예비보고서

전공: 수학/컴퓨터공학 학년: 3학년 학번: 20181288 이름: 윤성호

1. 3주차 실습에서 구현하는 추천 기능은 점수를 극대화할 수 있도록 현재 내려오고 있는 블럭의 배치 방법을 추천해준다. 이를 통해 플레이어는 블럭을 추천된 위치에 놓으면 높은 점수를 얻을 수 있다는 생각을 가지고 게임을 진행할 수 있게 된다. 이를 구현하기 위해서는 트리 자료구조를 사용하며, 트리의 각 노드에 블럭을 배치할 수 있는 모든 경우에 대한 정보를 집어넣어준다. 그러나 현재 내려오고 있는 블럭만을 고려하여 점수를 극대화하는 프로그램을 짜기 위해서는 트리 자료구조는 불필요하다. 그러나 테트리스 게임의 특성상 현재 내려오고 있는 블럭을 통해 점수를 극대화하는 그리디 알고리즘을 이용하게 되면 문제가 발생하게 된다. 현재 블럭만을 고려하였을 때 가장 큰 점수를 얻지는 못하여도, 그 다음 블럭까지 이용하게 되면 결과적으로 훨씬 높은 점수를 획득 할 수도 있기 때문이다. 즉, 현재 내려오는 블럭의 배치상황을 고려한 후 그 다음 블럭의 배치상황을 모두 고려해주기 위해서 트리 자료구조를 이용하는 것이고, 이를 통해 모든 상황을 고려하려고 하는 것이다. 여기서 한가지 더 생각해볼 수 있는 것은, 미래에 등장할 블럭들을 많이 알면 알수록 더 높은 점수를 획득할 수 있는 배치를 추천받을 수 있게 된다. 여러 수 앞을 먼저 생각하여 테트리스 게임을 진행하게 되는 것이기 때문이다. 이것만 생각하게 되면 앞으로 나올 블럭들을 최대한 많이 고려하면 된다고 생각이 들지만, 언제나 그렇게 생각할 수 만은 없다. 미래의 정보를 많이 고려하면 할 수록 고려해야할 경우의수가 기하급수적으로 늘어나서, 시간 및 메모리가 아주 많이 사용될 수 있기 때문에 프로그래머가 적절히 타협하여 프로그램을 작성해야 한다.

Int recommend(RecNode \*root) 함수의 작동과정을 살펴보며 추천 기능의 원리에 대해 설명해보고자 한다. 먼저, 인자로는 모든 정보가 초기화되어있는 루트 노드를 하나 받는다. 루트 노드 아래 최대 CHILDREN\_MAX개의 자식 노드를 받을 수 있도록 포인터 배열이 초기화 되어있다. 그 이후, 현재 내려오는 블럭에 대해서 2중 loop를 이용하여 배치될 수 있는 모든 상황을 생각한다. 이 때, 재귀적으로 함수를 작성하여, 만약 현재 탐색중인 노드의 레벨이 VISIBLE\_BLOCK 이라면, 더 이상 볼 블럭이 없기 때문에 거기까지 해서 얻을 수 있는 점수를 저장하면 된다. 그러나, 만약 노드의 레벨이 VISIBLE\_BLOCK 보다 작다면, 그 이후 등장하는 블럭들을 더 고려해주어야 하므로, 다시 recommend 함수에 자식노드를 인자로 넣어 호출해준다. 그렇게 트리형태로 모든 상황을 고려해주고, 점수의 최댓값이 나오는 배치를 찾았다면, recommendX, recommendY, recommendR 변수에 각각 추천해줄 위치의 X좌표, Y좌표, 회전횟수를 저장해주고, 현재 상황에서 얻을 수 있는 점수의 최댓값을 리턴해준다. 점수의 최댓값을 리턴해주는 이유는 함수를 재귀적으로 호출할 때 subtree를 고려했을 때의 최댓값도 계속 함께 고려를 해주어야 하기 때문이다. 만약 반환값이 없는 함수를 만들고 싶다면, 함수의 인자로 최댓값을 추가로 전달해주면 같은 기능을 하는 함수를 구현할 수 있다.

추천 기능을 구현하는 트리 구조는 모든 상황을 여러번 고려하는데 매우 효율적이다. 만약 트리를 사용하지 않고 backtracking 기법을 사용하여 모든 상황을 고려한다면, 다음 블록이 새롭게 등장했을 때, 매번 새롭게 다시 모든 상황을 고려해주어야 하는데, 이는 고려해야하는 블록의 개수 (VISIBLE\_BLOCKS)가 커질수록 시간이 기하급수적으로 오래걸리게 된다. 그러나, 트리 자료구조는 가장 처음 상황에서 VISIBLE\_BLOCKS 만큼의 레벨을 가지는 트리를 만들고 나서는, 그 다음 블럭을 고려하는 상황에는 트리의 레벨을 하나만 더 추가해주면 되기 때문에 시간복잡도 측면에서 굉장히 효율적이다. 그러나, VISIBLE\_BLOCKS의 수가 많아지면 많아질수록 만들어야 하는 트리의 크기가 굉장히 커진다. 이 커지는 정도는 backtracking 기법을 이용하여 모든 상황을 고려할 때 시간복잡도가 증가하는 정도랑 같다. 즉, VISIBLE\_BLOCKS 의 수가 1만 증가한다고 하더라도 트리를 만들기 위해 필요한 메모리의 크기는 기하급수적으로 커지게 된다. 즉, 공간복잡도 측면에서는 비효율적이다.

1. 먼저, 위에서 설명했듯이, 트리 자료구조는 시간복잡도 측면에서는 굉장히 효율적이지만, 공간복잡도 측면에서는 비교적 비효율적이다. 즉, 이를 개선하기 위해서는 공간복잡도를 줄여나가야 된다.

이를 위한 첫번째 방법은 노드의 개수를 줄이는 방법이다. 즉, pruning tree를 만드는 건데, 이때 pruning의 기준은 현재 내려오는 블럭의 모든 경우 중, 최댓값 점수를 가지는 노드는 제외하고 나머지 노드들 중 무작위로 절반을 삭제하는 것이다. 일단, 이 방법은 고려해야 하는 노드중 많은 부분을 삭제하기 때문에 시간복잡도와 공간복잡도를 동시에 줄일 수 있다. 최댓값 노드는 무조건 남기는 이유는 많은 경우에 추천 기능은 각각의 블럭이 매순간 최대 점수를 받는 상황에서 나오기 때문이다. 그런 와중에 최댓값 노드가 아닌 모든 노드를 삭제하지 않고 무작위로 절반만 삭제한 것은, 항상 매순간 최댓값 점수를 얻는 블럭배치가 최종적으로 최댓값을 얻을 수 있는 배치가 아니기 때문이다. 또한, 특정 기준이 아니고 무작위로 절반을 삭제하는 이유는 위의 상황이 적용되는 노드가 남아있을 확률이 특정 기준을 적용해 지우는 것보다 훨씬 크기 때문이다.

다음 방법은 노드에 담긴 정보의 크기를 줄이는 것이다. 노드에 점수나 레벨에 대한 정보는 정수형이라 크게 줄일 방법이 없지만, 각각의 상황에서의 필드 정보를 담는 이차원 배열 정보는 비트마스킹 기법을 이용하면 크기가 WIDTH 인 정수형 배열로 변환이 가능하다. 이는 ppt에서 소개한 필드 높이를 기억하는 아이디어보다 나은데, 그 이유는 데이터의 손실이 없기 때문이다. 현재 필드의 높이는 22이다. 각 열에 배치된 블럭정보를 이진수로 보고 정수형으로 변환하는 것이다. 즉, 빈칸은 0, 빈칸이 아니면 1로 보고 n번째 높이에서 빈칸이면 0, 빈칸이 아니면 2^(n-1)을 배열의 값에 추가하는 것이다. 이렇게 하면 배열에 담기는 정수의 최댓값은 2^22로 약 4백만이다. 이정도의 크기라면 int형 배열로 충분히 만들 수 있고, 이 정보를 노드에 담아주는 것이 공간복잡도 측면에서 훨씬 효율적이다. 이진수로 저장된 정보를 사용하는것도 c언어에서 기본적으로 제공하는 비트 연산자를 활용하면 필드 자체를 받아오는 연산과 동일한 과정으로 트리를 생성할 수 있다.