8주차 결과보고서

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

1-1.

Tetris.h 에 RecNode tree 설정이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이번 실험은 recommend system을 구현하는 것이다. 예비보고서에서 공부했듯이 현재의 block 모양, field 상태, nextblock 들을 모두 고려하여 추천 rotate, recommendX, recommendY 좌표를 구해야 한다. 그래서 tree 구조를 사용하였다. 여기서 lv 은 현 노드가 가지는 depth, score는 현 노드에 있을 때 얼마나 점수가 축적되는지 저장하는 score, 이전 블록의 field 를 저장하는 field (\*f)[WIDTH], 그리고 이 node의 child node를 저장하는 \_RecNode struct 구조체가 있다. 현재 가장 높은 곳의 뿌리 node 는 treeroot 로 설정하였다.

Tetris.c 에서 main 함수의 일부이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 가장 처음 함수가 실행될 때 가장 뿌리 node 인 treeroot를 초기화 시켜주었다. Treeroot에 malloc을 사용하여 동적 메모리를 주고 level, score 역시 0으로 초기화 하였다.

InitTetris의 일부이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

테트리스 관련 내용을 처음 초기화 해줄 때 field를 만드는데 이때 treeroot의 f를 field로 설정을 해주고 recommend(treeroot)를 통해 첫 블록에 대한 recommend system을 작동하도록 하였다.

다음은 BlockDown 함수의 일부이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

일정 시간이 되면 반복적으로 이 함수는 실행되는데 우선 이 함수가 실행될 때마다 treeroot의 level, score를 초기화 시켜준다. 그리고 블록이 더 이상 움직이지 못해 쌓일 때 nextBlock[0]을 다음 nextBlock[1]로 바꿔주고 treeroot의 f 를 현재의 field로 저장시킨다. recommend(treeroot)를 실행하여 바뀐 nextBlock[0] 에 대해 추천 recommendX, recommendY, recommendR 를 계산한다. 이후 DrawWidhFeatures로 추천 기능 블록 역시 그려준다.

DrawBlockWithFeatures 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서는 추가로 추천 기능 블록 ‘R’ 모양으로 그려줘야 하는데 주의해야 할 점은 어느 블록을 먼저 그리는지에 따라 겹쳤을 때 보이는 게 다르다. 본 실험자는 처음에 DrawShadow를 중간에 두어 그림자 Block이 추천 Block에 겹쳐 안 보였는데 이는 play를 하는데 좋지 않아 DrawShadow를 가장 아래로 둬 겹칠 시 그림자 Block이 보이도록 하였다.

이번 실험의 가장 핵심 기능인 추천 block의 x,y,r을 계산하는 recommend 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Recommend 함수는 실험 시간에 구현한 모든 경우의 수를 고려한 tree를 이용한 것이다. 우선 최종적으로 Return 해주는 최고 점수인 recommendScore 를 0으로 초기화 한다. k는 후의 child node 의 배열 index를 의미하는 변수, rec\_x는 현재 node의 추천 x 좌표, rec\_r은 현재 node의 추천 y 좌표이다. child node를 사용하기 위해 malloc 을 사용하여 동적 메모리를 할당하고 child[i]->f 즉 child node의 field 역시 동적 메모리 할당을 해주었다. 이후 2중 For문을 돌면서 추천 좌표를 accumulatedScore를 통해 찾게 된다. 첫 for문은 rotate 에 대한 것으로 0부터 시작해 3까지 모든 경우의 rotate를 고려하였다. 다음 for문은 x 좌표에 대한 것으로 가장 field 기준 왼쪽부터 시작하여 한 칸 씩 오른쪽으로 이동하면서 모든 가능한 x 좌표를 고려하였다. 그리고 여기서 계속 새로운 node들이 만들어지기 때문에 accumulatedScore=0, rec\_y=0으로 초기화 시킨다. 이때 CheckToMove를 통해 rec\_y+1 가 y좌표일 때 움직일 수 있는지 확인해본다. 만약 한 칸 내려갈 수 없다면 처음부터 아래로 못 움직인다거나 아니면 rec\_x 가 field 범위를 넘어갔다는 의미이다. 그래서 continue를 사용하여 해당 for 문이 더이상 실행 되지 않게 하고 다음 x 좌표로 넘어간다. 이 과정이 없다면 child[i] 의 동적 메모리 할당 시 index를 최대 CHILDREN\_MAX 까지 했는데 2중 for 문의 rec\_r\*rec\_x 가 더 커 segmentation fault가 발생하고 또한 계산 과정에서 field를 넘어가는 추천 block이 생길 수도 있다.

만약 움직일 수 있다면 다시 while 반복문으로 CheckToMove를 통해 최대 y값이 어디까지 내려갈 수 있는지 확인하고 최종 y 값을 rec\_y로 둔다. 이후 child[k] 의 level을 input 값인 root의 level+1 로 두어 child node임을 알려준다. 다음은 child[k]의 field에 parent 인 root의 field를 채워준다. 이제 계산 과정이 들어가는데 기존의 점수 계산 함수인 AddBlockToField, DeleteLine을 field는 child[k]->f, BlockID 는 nextBlock[root->lv], rotate는 rec\_r, y좌표는 rec\_y, x좌표는 rec\_x로 두어 계산을 하고 accumulatedScore 에 저장한다. 다음은 child[k]의 level이 VISIBLE\_BLOCKS 즉 최대 level 보다 작다면 재귀함수 recommend(child[k]) 를 호출하여 child[k] 의 child node들을 만들고 비교해 계산된 점수를 return 값으로 받아 이를 accumulatedScore에 누적해 저장하고 이를 child[k]->score 에 넣어준다.

이제 recommendScore와 accumulatedScore와 비교하는데 root->lv이 0 일 때, 즉 가장 root node일 때 recommend 좌표, rotate 를 변경 시킨다. 만약 둘의 점수가 같다면 rec\_y와 recommendY값을 비교한다. rec\_y 가 더 크다는 것은 최종 추천 block의 y 값이 더 작다는 의미로 field 위쪽에 block이 쌓여야 한다. 이는 게임 상 block이 아래로 쌓여야 좋기 때문에 이를 고려했다. 즉 rec\_y>recommend 일 때 recommendX=rec\_x, recommendY=rec\_y, recommendR=rec\_r 로 변경시킨다. recommendScore<accumulatedScore 일때 역시 당연히 recommend 좌표를 바꿔야 한다. 그리고 recommendScore를 accumulatedScore로 업데이트 시킨다. 이후 k를 1 증가시켜 다음 child[k+1] node를 사용 할 수 있도록 해준다. 이 과정을 2중 for문을 돌면서 계속해줘 모든 경우의 수를 고려한 recommend 좌표 및 recommendScore를 계산한다.

해당 함수는 위에서 할당한 child[i] 의 메모리를 다시 free 시켜주는 과정 역시 포함이 되어있는데 메모리 절약을 위해서 추가하였다. For문을 돌면서 Child[i]->f 에 동적 메모리를 free 시키고 다음 child[i] 의 메모리를 free 시켜주었다. 마지막으로 재귀함수에서 사용할 recommendScore를 return 해준다.

해당 함수의 시간복잡도는 첫 for문인 Rotate 수, 두번째 for문인 WIDTH, 2번째 for문 속에서 field를 받아올 때 사용하는 2중 for문의 HEIGHT, WIDTH 를 고려 하면 O(NUM\_OF\_ROTATE\*WIDTH\*HEIGHT\*WIDTH) 이다. 이때 VISIBLE\_BLOCKS 에 따라서 tree 구조의 level 이 달라진다. 즉 재귀함수로 들어가는 과정 역시 시간 복잡도에 포함되어야 하는데 VISIBLE\_BLOCKS 만큼 O(NUM\_OF\_ROTATE\*WIDTH\*HEIGHT\*WIDTH) 가 있어야 한다. (실습에서는 3개의 VISIBLE\_BLOCKS 를 고려해서 ^3 해주면 된다.) 공간 복잡도는 root 에 따른 childnode를 CHILDREN\_MAX 만큼 만들었다. 역시 tree의 level 에 따라 재귀 함수로 만들어 지는 childnode 들 의 공간복잡도 역시 고려해야한다. 그래서 O(CHILDREN\_MAX^(VISIBLE\_BLOCKS)) 정도에 비례한다고 생각하면 될 것이다.

추가적으로 작성한 modified\_recommend 함수는 tree의 단점인 시간적, 공간적 효율을 극대화하기 위해서 우선 새롭게 ModiRecNode라는 tree를 tetris.h 에 구현하였다. 이는 각 parent node가 child node를 1개만 가지는 tree이다. 기존의 CHILDRENMAX(36)에서 1개로 줄었기 때문에 36배의 메모리 효율이 생길 것으로 예상된다. 그리고 기존은 rotation을 0~3 까지 즉 모든 block의 회전수를 4개로 보았지만 실제로는 그렇지 않다. 각 block에 맞는 rotation을 따로 저장하여 블록에 따라 첫 for문을 다르게 돌아가게 하였다. 이후 accumulatedScore<20 인 경우는 해당 node가 최대 점수가 안 될 가능성이 매우 높기 때문에 고려하지 않기 위해 재귀함수로 다시 들어가지 않고 continue를 통해 다음 for문으로 돌아가게 하여 tree의 가지치기를 하였다. 이후 큰 차이는 없지만 최대한 아래쪽에 block이 쌓이는게 좋기 때문에 rec\_y를 가중치로 줘 accumulatedScore에 더하여 아래쪽으로 block들이 더 쌓일 가능성이 높게 하였다.

이때 시간 복잡도는 첫 for문에서 가장 경우의 수가 많은 4개(NUM\_OF\_ROTATE)가 나올 수 있다. 또한 비록 pruning을 해주었지만 시간복잡도는 최악의 경우를 고려해야 하니 모든 accummulatedScoer가 20이 넘을 경우이기 때문에 기존의 recommend함수와 시간복잡도는 같다. 즉 VISIBLE\_BLOCKS 만큼 O(NUM\_OF\_ROTATE\*WIDTH\*HEIGHT\*WIDTH) 가 있어야 한다. 하지만 실제 프로그램을 실행하면 최악의 경우는 항상 나오지 않기 때문에 기존의 recommend 함수보다 훨씬 빠르게 구동된다. 기존의 공간복잡도는 O(CHILDREN\_MAX^(VISIBLE\_BLOCKS)) 였는데 modified\_recommend 에서는 CHILDREN\_MAX 정도의 메모리를 사용하지 않기 때문에 공간복잡도는 O(VISIBLE\_BLOCKS) 로 고려하면 된다.

2.

구현한 modified\_recommend 함수는 결과보고서 1번에서 설명한 것처럼 기존의 tree에 비해 시간적 효율, 메모리적 효율을 극대화 시켰다. 실제 시간 및 공간 효율성 계산 결과를 보면 약 recommend의 score/time은 최대 69 정도였고 modified\_recommend는 평균적으로 76 정도가 나왔다. 또한 recommend의 실험자가 계산한 score/space는 0.0002 정도였고 modified\_recommend 는 약 0.009 로 30배 이상이 차이가 났다. 그 이유는 pruning 과정과 tree의 단순화 과정을 거쳤기 때문이다.

우선 첫 for문의 rotation을 block에 따라 가지는 rotation 개수를 다르게 지정하여 시간 복잡도 측면에서는 큰 차이가 없지만 실제 프로그램이 작동하는 시간 효율적 측면에서는 훨씬 단축될 것이다. 이렇게 specific하게 지정해주었다. 하지만 tree를 단순화 하는 과정은 이 tetris의 recommend 시스템을 구현할 때처럼 특정한 상황에서 구현이 가능하다. 즉 accumulated\_score를 통한 recommend 시스템이 아니라 다른 것을 이용해서 구현한다면 모든 노드를 다 이용해야 할 것이다. 또한 pruning 과정 역시 과도하게 넣거나 이 과정이 조금이라도 있다면 시간적, 공간적 효율을 높아지지만 최종 목표인 Score를 더 높이는 것은 매우 어려웠다. 왜냐하면 accumulatedScore가 20보다 작은 극히 일부의 node들을 pruning 했음에도 불구하고 점수는 기존의 recommend 함수에 비해 거의 1/2~1/3배 정도 차이가 났음을 확인할 수 있었다.

그렇기 때문에 예비보고서에 작성했던 것처럼 tradeoff (일부가 좋아지면 일부가 나빠진다) 이다.

3.

우선 이번 실험에는 한 번도 사용해 보지 않은 tree 자료구조를 이용하여 recommend system을 구현하였다. 그래서 처음 이해하는 데 오래걸렸고 코드 상 불필요한 코드들 역시 존재하고 또 이를 삭제하는 과정이 힘들었다.( 2주차의 자료구조인 linkedlist는 그래도 지난 실습 때 구현을 해보아서 자료구조는 알고 있는 상태였다.) Tree 구조는 특히 모든 경우의 수를 다 고려해야 하기 때문에 저번 결과보고서에 코드를 구현할 때 가장 중요 요소로 언급한 시간 복잡도, 메모리 할당, 공간 복잡도 역시 매우 복잡해지고 커지는 것을 알게 되었다. 그래서 코드가 돌아가는데는 전혀 문제는 없지만 최적화를 위해 조금이라도 적은 메모리를 사용하려고 childnode의 free 와 같은 과정도 포함을 시켰다.

또한 tree 구조가 생소한 만큼 많은 시간이 걸리고 오류도 많았다. 그래서 2주차에서 배웠던 gdb를 이용하여 해당 오류가 어느 줄에서 생겼는지 디버깅도 할 수 있었다. 이때 Makefile 에서 a.out을 만들 때 -g 를 붙이지 않으면 gdb를 이용해도 오류 정보를 확인 할 수 없었음을 다시 인지했다.

테트리스 프로젝트 실험이 3주에 걸쳐 끝이 났다. 이렇게 큰 프로젝트를 C언어로 혼자서 진행하는 것은 처음이라 걱정이 되었지만 그래도 순차적으로 주에 걸쳐 테트리스 구현 내용이 심화되었기 때문에 앞서지는 못했지만 그래도 따라간 것 같다. 특히 이렇게 큰 코드에서 주석 처리는 나중에 어떤 코드였는지 잘 기억이 나지 않을 때 큰 효과를 발휘했다. 그래서 3주차 과제에서 autoBlockDown를 구현할 때 1주차에서 진행한 BlockDown의 주석을 읽고 손 쉽게 일부 코드만 고쳐 바로 진행 할 수 있었다. 그리고 Linkedlist, tree 와 같은 3년 전 자료구조에서 배워 매우 중요하지만 머리 속에서 잊고 있었던 자료구조들을 다시 공부하고 실제 어떻게 쓰이는지 알게 되었다. 앞으로 코드를 구현할 때 지금의 경험을 토대로 어떤 자료구조가 좋을지 고민 해 볼 것 같다. 또한 이번 3주간의 실험처럼 앞으로 큰 프로젝트가 있다면 우선 관련된 내용들을 먼저 묶고 독립적인 내용들은 순차적으로 하나 씩 진행할 것이다.