**8주차 Tetris-3 결과보고서**

전공: 수학과 학년: 4 학번: 20161255 이름: 장원태

1. **실험시간에 작성한 프로그램의 알고리즘과 자료구조를 요약하여 기술하시오. 완성한 알고리즘의 시간 및 공간 복잡도를 보이시오.**

우선, 자료구조는 다음과 같다.

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

이번 실험시간에 작성한 프로그램에서 이용한 자료구조는 트리(tree) 구조이다. 이는 현재 블록에 대해서 모든 경우의 수를 고려하고, 그 각 경우의 수에 따라서 그 다음 블록에 대한 모든 경우의 수를 고려하고, 그 뒤에 또 다음 블록에 대한 모든 경우의 수를 고려하는 식으로 구성할 것이므로, 하나의 parent 노드에서 여러 child를 파생시킬 수 있는 트리 구조를 채택했다. 각 노드는 몇 번째 블록에 대해 고려하는지에 연관이 되어 있는 레벨, 각 경우에 축적되는 점수, 이전 경우까지 블록이 쌓일 때의 필드 상태를 나타내는 필드, 그리고 child 노드들에 대한 포인터를 성분으로 가진다.

다음으로, 각 함수에 대해 살펴보기에 앞서, 추천 시스템을 실제로 구현한 것은 다음과 같은 식을 InitTetris 함수와 BlockDown 함수에 대입함을 통해 이행했다.

|  |
| --- |
| // InitTetris 함수 중  recRoot = (treepointer)malloc(sizeof(struct \_RecNode));  recRoot->lv = 0;  recRoot->score = 0;  recRoot->f = field;  recommend(recRoot); |
| // BlockDown 함수 중  recRoot->f = field; recRoot->lv = 0; recRoot->score = 0;  recommend(recRoot); |

이와 같이 트리의 루트(root)를 초기화해준 뒤 recommend 함수를 호출해줌을 통해 각 경우에 대해 어느 곳을 추천해줄 지 표시해준다.

이어서, 추천한 블록을 그려주는 함수들이다.

|  |
| --- |
| void DrawRecommend(int y, int x, int blockID,int blockRotate){  // user code  DrawBlock(y, x, blockID, blockRotate, 'R');  }  void DrawBlockWithFeatures(int y, int x, int blockID, int blockRotate){  DrawRecommend(recommendY, recommendX, blockID, recommendR);  DrawShadow(y, x, blockID, blockRotate);  DrawBlock(y, x, blockID, blockRotate, ' ');  } |

이는 단순히 블록을 그려주는 일만 행하며, DrawRecommend함수의 경우는 단순히 해당 y, x, blockID, blockRotate의 블록 위치에 ‘R’을 반전시킨 무늬의 블록을 그려주는 함수이다. DrawBlockWithFeatures의 경우, 이전에 shadow까지 그렸던 함수에, shadow를 그리기 전에 DrawRecommend함수를 먼저 호출함을 통해, 무늬가 겹칠 경우 R이 가장 순위가 밀리고, 그 다음이 그림자, 그 다음이 실제 블록이 가장 순위가 높도록 설정한다.

추천 시스템을 구현한 본 함수이다.

|  |
| --- |
| int recommend(RecNode \*root){  int max=0; // 미리 보이는 블럭의 추천 배치까지 고려했을 때 얻을 수 있는 최대 점수  int rotate, Xpos, x, y; int Ypos=0;  int i=0; int touched; int accumulatedScore;  // user code  for (rotate=0; rotate<NUM\_OF\_ROTATE; rotate++){  for (Xpos=-1; Xpos<WIDTH; Xpos++){  Ypos=0;  while (Ypos<HEIGHT && CheckToMove(root->f, nextBlock[root->lv], rotate, Ypos, Xpos)) Ypos++;  Ypos--;  if (Ypos == -1) continue;  root->c[i] = (treepointer)malloc(sizeof(struct \_RecNode));  root->c[i]->lv = root->lv+1;  root->c[i]->f = (char(\*)[WIDTH])malloc(sizeof(char)\*HEIGHT\*WIDTH);  for (y=0; y<HEIGHT; y++){  for (x=0; x<WIDTH; x++){  root->c[i]->f[y][x] = root->f[y][x];  }  }  touched = 0;  for (y=0; y<4; y++){  for (x=0; x<4; x++){  if (block[nextBlock[root->lv]][rotate][y][x] == 1){  if (Ypos+y+1 == HEIGHT || root->c[i]->f[Ypos+y+1][Xpos+x] == 1) touched++;  root->c[i]->f[Ypos+y][Xpos+x] = 1;  }  }  }  accumulatedScore = root->score + touched\*10;  accumulatedScore += DeleteLine(root->c[i]->f);  root->c[i]->score = accumulatedScore;  if (root->c[i]->lv < VISIBLE\_BLOCKS){  accumulatedScore = recommend(root->c[i]);  }  if (max <= accumulatedScore){  max = accumulatedScore;  if (root->lv == 0){  recommendR = rotate; recommendX = Xpos; recommendY = Ypos;}  }  i++;  }  }  return max;  } |

큰 틀은 루트의 child 레벨이 2가 될 때까지 recursion을 이용하는 것이다. 우선, 각 회전 수와 각 X좌표에 대하여 for loop을 돌린다. 그 상태에서, Y의 경우는 해당 위치에서, 필드에 있는 블록이나 틀에 닿을 때까지 계속 increment를 해준다. 만약 Y가 변화가 없었다면(해당 코드에서는 while문을 거친 후에 -1을 해주므로, Y=-1인 경우), 그 위치에는 추천을 할 수 없으므로 통과(continue)한다. 그렇지 않다면, 해당 경우에 대한 노드를 생성하여, 레벨은 부모 노드보다 1 높게, 필드는 부모 노드와 같게 설정하며, 점수는 아직 초기화해주지 않아도 된다. 그리고, AddBlockToField 함수를 이용하여 직접적으로 블록을 놓게 되면 실제 필드까지 바뀌어버리는 현상이 발생하여, 그 함수와 같은 매커니즘을 가지고 점수를 계산할 수 있도록 touched 변수를 선언한다. 그리고, 블록의 경우도 실제 필드가 아니라 해당 경우의 가상 필드에 쌓을 수 있도록 별도로 for문을 구축하여 블록을 그려준다. 동시에, AddBlockToField 함수에서 점수를 계산하는 바와 같이, 해당 블록을 놓는 위치의 아래에 블록이나 맨 아래 경계가 닿는 수만큼 touched를 증가시켜준다. 그리고, 이에 따라 점수는 부모 노드의 점수에 touched의 10배, 그리고 해당 경우의 필드에서 완성된 줄을 지운 경우의 점수를 모두 더하여 누적 점수를 구하여 그 점수를 해당 노드의 점수로 설정한다. 만약 레벨이 보이는 블록 개수인 VISIBLE\_BLOCKS에 미치지 못하면, 이를 recursion으로 다시 호출한다. 그 과정을 마치고 나면, max 점수를 각 경우에 대해 갱신해준다. 해당 레벨에서의 가장 높은 점수를 받도록 한다. 이 때, recursion에 의해서, 더 레벨이 낮은 노드일 수록 높은 레벨의 노드에서 점수를 계속 누적하여 받아오게 된다. 그러므로, 레벨이 0인 root 노드의 child에서 최대 점수를 가리면 되며, 그 경우의 rotate, X좌표, Y좌표를 전역 변수를 통해 기억해낸다.

이 함수의 시간 복잡도의 경우, WIDTH에 따른 for문을 구성하고, 그 안에서 또 필드를 받아오는 과정에서 HEIGHT\*WIDTH만큼의 횟수의 for문을 들어가게 되므로, 시간 복잡도는 O(WIDTH\*WIDTH\*HEIGHT)가 된다. 같은 이유로 공간 복잡도 또한 O(WIDTH\*WIDTH\*HEIGHT)이 된다.

1. **모든 경우를 고려하는 tree 구조와 비교해서 어떤 점이 더 향상되고, 어떤 점이 그렇지 않은지 아울러 기술하시오.**

조금 더 효율적으로 작성해본 코드는 다음과 같다.

|  |
| --- |
| int modified\_recommend(RecNode \*root){  int max=0; // 미리 보이는 블럭의 추천 배치까지 고려했을 때 얻을 수 있는 최대 점수  int rotate, Xpos, x, y; int Ypos=0;  int i=0; int touched; int accumulatedScore;  if (!root) return 0;  // user code  for (rotate=0; rotate<NUM\_OF\_ROTATE; rotate++){  if (rotate >= 1 && nextBlock[root->lv]==4) break; // 2x2 정사각형 블록은 회전이 1회만 필요  else if (rotate >= 2 && (nextBlock[root->lv] == 0 || nextBlock[root->lv] >= 5)) break; // 회전이 2회만 필요한 블록  for (Xpos=-1; Xpos<WIDTH; Xpos++){  Ypos=0;  while (Ypos<HEIGHT && CheckToMove(root->f, nextBlock[root->lv], rotate, Ypos, Xpos)) Ypos++;  Ypos--;  if (Ypos == -1) continue;  root->c[i] = (treepointer)malloc(sizeof(struct \_RecNode));  root->c[i]->lv = root->lv+1;  root->c[i]->f = (char(\*)[WIDTH])malloc(sizeof(char)\*HEIGHT\*WIDTH);  for (y=0; y<HEIGHT; y++){  for (x=0; x<WIDTH; x++){  root->c[i]->f[y][x] = root->f[y][x];  }  }  touched = 0;  for (y=0; y<4; y++){  for (x=0; x<4; x++){  if (block[nextBlock[root->lv]][rotate][y][x] == 1){  if (Ypos+y+1 == HEIGHT || root->c[i]->f[Ypos+y+1][Xpos+x] == 1) touched++;  root->c[i]->f[Ypos+y][Xpos+x] = 1;  }  }  }  accumulatedScore = root->score + touched\*10;  accumulatedScore += DeleteLine(root->c[i]->f);  root->c[i]->score = accumulatedScore;  if (root->lv == 0 && root->c[i]->score <= 10){ // 첫 블록이 닿을 때 10점이 나오면 해당 가지를 삭제(pruning tree)  free(root->c[i]->f);  free(root->c[i]);  continue;  }  if (root->c[i]->lv < VISIBLE\_BLOCKS-1){  accumulatedScore = modified\_recommend(root->c[i]);  }  if (max <= accumulatedScore){  max = accumulatedScore;  if (root->lv == 0){  recommendR = rotate; recommendX = Xpos; recommendY = Ypos;}  }  i++;  }  }  return max;  } |

위에서 언급한 recommend 함수와 큰 틀은 같다. 그러나, recommend함수의 경우는 가능한 모든 경우를 고려하는 반면, 이 modified\_recommend 함수의 경우는 다음 세 가지를 이용하여 효율성을 증대시켰다.

1. 블록의 개수가 현재 블록과 다음 블록 2개의 총 3개가 보이지만, 그 중 현재 블록과 그 다음 블록까지만 고려하도록 설정한다.
2. Root 노드(레벨 0)에서 점수를 10점밖에 얻지 못하는 가지는 잘라버린다.
3. 불필요한 rotate의 redundancy를 줄이기 위해, blockID가 0, 4, 5, 6인 블록에 대해서는 rotate을 조정한다.

위와 같은 조정을 거치면, 우선 고려하는 경우의 수가 크게 감소한다. 1번째 사항에 의해서, 고려하는 블록의 개수를 줄임으로써 메모리 사용을 최대 34배까지 단축하게 된다. 2번째 사항의 경우, 테트리스에서 일반적으로 높은 점수를 받을 수 있는 경우는 공통적으로 첫 블록에서 최소 20점은 받아야 한다고 판단을 했다. 10점 이하로 받는 경우, 즉 10점만 받는 경우는 해당 블록 중 한 면만이 필드에 있는 블록이나 아래쪽에 닿는 것인데, 이는 플레이 측면에서 비효율적이고 점수를 높게 받는 것을 도와주지 않기 때문이다. 이에 따라, 현재 블록이 10점 이하로 점수를 받게 되는 경우만을 가지를 잘라버렸다. 즉, 해당 노드는 free시켰다. 3번째 사항의 경우, 기존 recommend 함수에서는 불필요한 회전 중복이 많이 일어났다. 예를 들어, 2x2 정사각형 블록의 경우는 회전이 필요 없지만, 해당 경우에도 rotate을 3까지 고려하였기 때문에 불필요한 중복이 크게 발생했다. 이는 과제로 구성한 recommendedPlay를 구현할 때도, 블록이 회전할 필요가 없을 때도 회전하는 현상이 발생하게 했다. 그래서, 메모리 사용도 줄이고 recommendedPlay 모드에서도 불필요한 회전이 일어나지 않도록, 해당 문제가 있는 블록들에 대해서는 모든 rotate에 대해 고려가 끝나면 바로 프로시저를 종료하도록 했다. 이와 같은 일련의 과정을 통해, 사용하는 메모리 양이 크게 감소하는 효과를 일으킨다.

그러나, 이전에 비해 더 정확하지 않게 추천 시스템이 작동하는 문제가 있다. 위에서 언급한 세 가지 사항 중 3번째 사항의 경우는 문제가 되지 않지만, 1번째 사항과 2번째 사항에서 문제가 발생한다. 1번째 사항의 경우, 보이는 블록이 총 3개임에도 불구하고 2개까지만 고려하였기 때문에, 3번째 블록까지 놓을 미래를 고려하지 않아 높은 점수를 얻을 기회를 놓칠 가능성도 없지 않다. 2번째 사항의 경우, 가능한 한 경우의 수를 놓치지 않기 위해 “10점”으로만 점수를 제한하기는 했지만, 예기치 않게도 블록의 한 면만이 필드에 닿도록 하는 플레이가 높은 점수로 이어질 가능성도 생각해볼 수는 있다. 예를 들어, 1x4 기다란 블록의 경우, 세로로 길게 블록을 놓아도 효과를 발휘하는 경우가 많다. 물론 이는 줄을 삭제하는 경우 효과를 발휘하는 것이 대부분이겠지만, 그렇지 않더라도 효과를 낼 수 있는 경우가 있을 수도 있다.

1. **본 테트리스 프로젝트를 통해 습득한 내용을 한 내용이나 느낀 점을 기술하시오.**

이번 테트리스 프로젝트에서는 많은 것이 새로웠다. 시초적으로는, ncurses 라이브러리의 함수들을 사용해보는 것이었다. 단순히 printf와 같이 스크린 내에서 위에서부터 출력하거나, 단순한 계산 등을 구현하는 것이 아닌, printw와 같이 스크린 내의 정해진 위치에 출력하거나, 화면을 지우거나, 키보드에 입력한 키에 따라 명령을 바로바로 수행하거나, 1초에 한 번씩 블록을 떨어트리게 하는 등 새로운 함수들을 많이 접하게 되었다. 또한, 단순히 수치적으로만 생각했던 이전 문제들 및 프로젝트들과는 달리, 이번 프로젝트는 본인이 작성한 코드가 “필드”와 “블록”이라는 시각적인 물체를 통해 보여지는 것이 특별했다.

또한, 랭킹 시스템과 추천 시스템을 통해 링크드 리스트와 트리를 구현하는 데에 더 익숙해졌다. 기존에 배열에만 익숙했었으나, 이제는 링크드 리스트와 트리라는, 조금은 어색한 자료구조도 구현을 해보고, 그 자료구조에 삽입 및 삭제도 해보면서 해당 자료구조들을 다루는 데에 더 익숙해짐을 느꼈다.

마지막으로, 이와 같이 700~800줄 정도가 되는 방대한 양의 코드를 다뤄보는 것은 처음이었는데, 이에 따라 각 코드를 작성할 때 고려해야 할 부분이 많았고, 오류가 발생할 경우 디버깅하는 과정도 쉽지 않았다. 아직 완벽하게 이들을 익히지는 못했을 지라도, 향후 또 다른 방대한 코드를 다룰 때에 더 익숙하게 다룰 수 있을 것이라는 생각이 들었다.