Ali Arda Eker 03.01.2016

150 130 217

ISE 308, Artificial Intelligence & Expert Systems, Fall 2015-2016

Project Report

I followed the idea of greedy algorithm to implement my bot. I did not perform a search with greedy algorithm but instead I used it to find reasonable locations for incoming blocks. Greedy solutions is one of the informed search strategies like A*. The difference is greedy solution uses just h(n) (heuristic function) to get f(n). It is similer to Uniform-Cost Search but since UCS is uninformed, it uses g(n) (path cost) instead of h(n) to get f(n).

Since my f(n) = h(n), I had to choose a rational heuristic function for my bot and it had to be compatible with the game. As the game consists of blocks I thought choosing manhattan distance for my h(n) would be nice. The main logic which I will tell in detail later is measuring the manhattan distance between the coming block and appropriate locations for it. After I calculated distances I picked the best one which got the highest score. This is how my bot starts:

```
for (int i=0; i<20; i++)
{
    for (int j=0; j<10; j++)
    {
        if (f.GetCell(j,19-i).IsEmpty()) myField[i][j] = false;
        else myField[i][j] = true;
    }
}
```

First of all I created a double array to represent my game field so that I can check existing blocks and available locations. A false cell indicates it is empty and true is for blocked ones. Later I calculated a manhattan distance for coming blocks. To do that I needed to choose what type of block is coming. So I implemented this which I go deeper in 2 steps later.

```
 46      if (s == Shape::ShapeType::O)
 47      {
 71
 72      if (s == Shape::ShapeType::I)
 73      {
130
131      if (s == Shape::ShapeType::T)
132      {
268
269      if (s == Shape::ShapeType::J)
270      {
405
406      if (s == Shape::ShapeType::L)
407      {
542
543      if (s == Shape::ShapeType::S)
544      {
602
603      if (s == Shape::ShapeType::Z)
604      {
662
```

Every conditional statement is for a different type of block as can be expected. So the related actions will be performed considering the type of the block. After this step I filled my move vector with the calculated moves in those if statements like this:

```
if (manhattanDistance < 0) for (int k=0; k>manhattanDistance; k--) moves.push_back(Move::MoveType::LEFT);
else for (int k=0; k<manhattanDistance; k++) moves.push_back(Move::MoveType::RIGHT);

moves.push_back(Move::MoveType::DROP);
return moves;
```

Here the calculated manhattan distance is between -4 and 5. It is the range of the game field in x coordinate if we take the coming blocks' entry point to the field as 0. The rotations are performed for each type of block if necessary in their if statements. Now we see the square block. Since it is not effected by a left or right rotation it is the simplest one.

```
if (s == Shape::ShapeType::O)
{
    for (int i=0; i<19; i++)
    {
        for (int j=0; j<9; j++)
        {
            if (myField[i][j] == false && myField[i][j+1] == false && myField[i+1][j] == false && myField[i+1][j+1] == false)
            {
                pathBlocked = false;
                for (int m=i; m<19; m++)
                {
                    for (int n=j; n<j+2; n++) if (myField[m][n]) pathBlocked = true;
                }

                if (!pathBlocked)
                {
                    manhattanDistance = j - 4;
                    sentinel = true;
                    break;
                }
            }
        }
        if (sentinel) break;
    }
}
```

Here we have 3 nested for loop. The outer 2 are for traversing the game field to find a space for the square. But of course it is not enough. If it finds a suitable place then the 3'rd for loop traverses the field to find if the square can travel there or not. If the path is blocked the outer loops keep searching. Once a location with open a path is found, the manhattan distance is calculated. Of course this is not like the real manhattan distance because we consider just x axis. We will consider y asis later when we should consider rotations. And as a break statement breaks only the inner loop we keep a variable named "sentinel" to manage that.

Now we will see the "I" statement representing stick block. Since when it is rotated twice it will be the same we will consider just one rotation and it will be same for both directions. The "Z" and "S" blocks are also behave like that. So I will not explain them in detail, they are very similar to "I" except some boundry checkings. This is "I":

```
for (int i=0; i<20; i++)
{
    for (int j=0; j<7; j++)
    {
        if (myField[i][j] == false && myField[i][j+1] == false && myField[i][j+2] == false && myField[i][j+3] == false)
        {
            pathBlocked = false;
            for (int m=i; m<19; m++)
            {
                for (int n=j; n<j+4; n++) if (myField[m][n]) pathBlocked = true;
            }

            if (!pathBlocked)
            {
                depth1 = i;
                manhattanDistance1 = j - 3;
                sentinel = true;
                break;
            }
        }
    }
    if (sentinel) break;
}
sentinel = false;

for (int i=0; i<16; i++)
{
    for (int j=0; j<10; j++)
    {
        if (myField[i][j] == false && myField[i+1][j] == false && myField[i+2][j] == false && myField[i+3][j] == false)
        {
            pathBlocked = false;
            for (int m=i; m<19; m++)
            {
                if (myField[m][j]) pathBlocked = true;
            }

            if (!pathBlocked)
            {
                depth2 = i;
                manhattanDistance2 = j - 4;
                sentinel = true;
                break;
            }
        }
    }
    if (sentinel) break;
}
```

Here we see the inside of the “I” statement. The 2 nested structures are for checking if the rotations are necessary. The first one keeps the depth which is actually the y axis of the manhattan distance for a non-rotated stick. And second one keeps depth if a rotation is done. Then we will check which one finds a deeper place. Since placing a block as deep as possible is a good act we will do a rotation based on “depth1” and “depth2” like this:

```
if (depth1 <= depth2) manhattanDistance = manhattanDistance1;
else
{
    manhattanDistance = manhattanDistance2;
    moves.push_back(Move::MoveType::TURNLEFT);
}
```

Here if we see the depth without a rotation is deeper we choose its manhattan distance. Otherwise we choose other’s one and make a left rotation. The main idea for “J”, “L” and “T” is very similar to this but since they are effected by number of rotations they will need 4 nested loops and we will choose the deepest again. But this time we will consider a scoring value. “J”, “L” and “T” is very similar to each other except some boundry conditions so I will mention the “T” only.

```
for (int i=0; i<19; i++)
{
    for (int j=0; j<8; j++)
    {
        if (myField[i][j+1] == false && myField[i+1][j] == false && myField[i+1][j+1] == false && myField[i+1][j+2] == false)
        {
            pathBlocked = false;
            for (int m=i+1; m<19; m++)
            {
                for (int n=j; n<j+3; n++) if (myField[m][n]) pathBlocked = true;
            }

            if (!pathBlocked)
            {
                if (myField[i][j] == true && myField[i][j+2] == true) depth3 = i-2;
                else depth3 = i;
                manhattanDistance3 = j - 3;
                sentinel = true;
                break;
            }
        }
    }
    if (sentinel) break;
}
sentinel = false;

for (int i=0; i<18; i++)
{
    for (int j=0; j<9; j++)
    {
        if (myField[i][j] == false && myField[i+1][j] == false && myField[i+2][j] == false && myField[i+1][j+1] == false)
        {
            pathBlocked = false;
            for (int m=i+1; m<19; m++)
            {
                for (int n=j; n<j+3; n++) if (myField[m][n]) pathBlocked = true;
            }

            if (!pathBlocked)
            {
                if (myField[i][j+1] == true) depth4 = i-1;
                else depth4 = i;
                manhattanDistance4 = j - 3;
                sentinel = true;
                break;
            }
        }
    }
    if (sentinel) break;
}
```

Here we see a part from the inside of “T” statement. Actually it has 4 nested loops like above but to explain I put just 2 on them to the paper. A “T” block has 4 shape actually. One without a rotation and two is one left rotation and it is identical to 3 right rotation, third is same with 2 left or right rotations and fourth is 3 left rotation or 1 right rotation. This is why we need 4 nested for loops for “T”.

Here the main idea is same with the other blocks but we have a additional part. It is the most inner if conditions. We have a extra if-else statement to give score for better replacements of “T” even they are shallower. Foe example to put a “T” block perfectly by filling a hole a good thing. So we would want it although they are shallower than the other found location. So the first nested for loop structure has a “-2” value to be added on its depth. This will be done if a hole is filled completely. This is done to give a higher score by seting a depth lower than its actual value if we really want it. We may give “-3” instead of “-2”. It is due to the programmers’ game strategy.

Now we will see the depth evalating parts for “T”, “J” and “L”. They are identical:

```
int min = depth1;
int numbers[4] = {depth1,depth2,depth3,depth4};
for (int i=1; i<4; i++) if (numbers[i] < min) min = numbers[i];
for (int i=0; i<4; i++)
{
    if (min == numbers[i])
    {
        depthNo = i;
        break;
    }
}
if (depthNo == 0) manhattanDistance = manhattanDistance1;
else if (depthNo == 1)
{
    manhattanDistance = manhattanDistance2;
    moves.push_back(Move::MoveType::TURNLEFT);
}
else if (depthNo == 2)
{
    manhattanDistance = manhattanDistance3;
    moves.push_back(Move::MoveType::TURNLEFT);
    moves.push_back(Move::MoveType::TURNLEFT);
}
else if (depthNo == 3)
{
    manhattanDistance = manhattanDistance4;
    moves.push_back(Move::MoveType::TURNLEFT);
    moves.push_back(Move::MoveType::TURNLEFT);
    moves.push_back(Move::MoveType::TURNLEFT);
}
else cerr << "Something went wrong in block T" << endl;
```

This is for choosing the best score for rotations. Although it says “depthNo” it represents more than a depth value because we credited desirable ones. First of all we choose the minumim value between depths. Then regarding which is the minimum we take its manhattan distance and do left rotations to satisfy it.

As we see my bot follows a very simple algorithm implemented with the idea of greedy search. Even though it doesn’t perform a search operation between paths or action, it finds a suitable location for blocks by evaluating a heuristic which is manhattan distance. It gives scores for desirable locations and does rotations regarding the score.