1. Identification
NanDai 11162942 nbd596
Solo

2. Implementation

Wights have worth of 20, dragons have worth of 30, queen has worth of 300+10*x.

Each move is a different state, given the depth d, it search d levels deep and use minimax to compute the best move from current state.

From example, at the current state, a Wight could move up,down or left,right, there is four different states in first level,
then it’s queen’s turn, and for each state extended from first level,the queen/dragon could has several different moves, those states are second level.
So when depth is 1, the number of maximum possible states is 8, when depth is 2, each possible state could have a maximum of 8 nextMoves, so it would be 8*=64 possible states.
Always assume the opponent can make the best decision, so every second level need to find the minimum value.

Here is what my program does:
Search for a leaf node,
node=LeafNode, then find it’s parent
node=LeafNode.parent, then search the best score among its children, and assign current node.estimate to this best score,
after done this for all leaf nodes, recursively find the best estimate for its parent level nodes.
When reach the first level, we have all nodes with a minimax value, and pick the highest value from first level, make that move.

For example:

```
                . D Q
                W . .
                . W .
                <-  ->
        WDQ             .WQ
        . ..            . ..
        .W.             .W.
     <-    ->           <-  ->
D.Q         W.Q      .Q.        .W.
. . .       .D.      . ..       .Q.
.W.         .W.      .W.        .W.
```

W=20 D=-30 Q=-100-10X

```
        ||
        ||

           -90
    -90           -60
 -110  -90     -80   -60
```
find minimum of bottom level then assign its parent value to this value

```
        ||
        ||
         -90
  -110          -80
```
find the maximum of bottom level then assign its parent value to this value

```
        ||
        ||
        -80
```

Now we reach the top level, so -80 is the best move.

///

I didn't use alpha and beta value. In my implementation, the value of each level is calculated differently.

For example, if it's wights' turn, and search depth is 3, it would be

```
                     Move        (current state)
                    /    \      -> max
                   move  move   (next move for wights)
                   / \   / \    -> min
                  m  m  m  m       (next move for queen)
                 /\ /\  /\ /\    -> max
                m m m m m m m m         (next move for wights)
```

```
def getScore(state):
    wights=state.wights
    dragons=state.dragons
    queen=state.queen
    score=0
    for wi in wights:
        score+=wi.worth
    for di in dragons:
        score-=di.worth
    for q in queen:
        score-=q.worth
    return score

def getScoreQ(state):
```

```
    wights=state.wights
    dragons=state.dragons
    queen=state.queen
    score=0
    for wi in wights:
        score-=wi.worth
    for di in dragons:
        score+=di.worth
    for q in queen:
        score+=q.worth
    return score
```

When wights’ turn, the value of wights is positive and value of dragon and queen is negative; when queen’s turn, the value of wights is negative and value of dragon and queen is positive.

getScoreQ compute the sum of all pieces’ worth in the state, for Queen. (Queen/dragon has positive value, wights have negative value)
getScore compute the sum of all pieces’ worth in the state, for Wights.(Queen/dragon has negative value, wights have positive value)
So every level, it just need to pick the best move for itself.
It does the same thing as alpha-beta.

The value of possible states for wights’ turn is calculated by getScore()

And the value of possible states for queen’s turn is calculated by getScoreQ(). And just find the maximum value for every level. Do not need to find max value for max level and min value for min level.

///

Class and method I created in my implementation:
‘’’’

Class: Wights. Has field x and y, represent the position on the board.
Id: tell the search function it’s a wights
Worth: How much per wight worth, use to calculate the value for minimax
up,down,left,right: the coordinate of these direction

Class: Dragons. Has field x and y, represent the position on the board.
Id: tell the search function it’s a dragon
Worth: How much per wight worth, use to calculate the value for minimax

Class: Queen: Same as Dragons, only different is id, which tell the function it’s a queen.

Class: State. Has field wights, dragons, queen, and estimate.
I consider the current pieces on the board as a state, list of wights, dragons and queen construct a state. Each state has a estimate value
to keep record of minimax value.

Class: Node. Make each state a node, each node has a field state, has a list of children, one parent and a estimate value.
Node class methods:
addChild() add a child node and set the child node’s parent to self
isLeaf() check if current node is a leaf node
wSetESTFromChild() find the maximum value from its children and set estimate value
qSetESTFromChild() find the maximum value from its children and set estimate value
getChild(i) return the child node that is ith in the list
numChild() return the length of children
getParent() return its parent node
getChildrens() return the children list
setParent(parent) set the parent of current node
setState(state) set the state of current node

Class: Board.
CreateBoard() create board with initial pieces
printBoard(state) display the board of given state
getPos(board) return the positions of wights,dragons,queen on the board
findpiece(x,y,wights,dragons,queen) return the piece on location x,y
moveW(x,y,d,state) give the coordinate of piece and direction, move wights on the coordinate
d can be1,2,3,4,5,6,7,8, represent 8 different directions
moveD(x,y,d,state) move dragon or queen

wTurn() when player controlling wights, it ask for input and check if
it’s legal to move, if true, move, if false, ask again

qTurn() when player controlling queen, it ask for input and check if it’s legal to move

possibleMoves(state,player) return a list of all possible moves of player side in current state, return three values, wights,dragons,queen

printPossibleMoves(): print board of possible moves

wCanUp,wCanDown…. check if a wight is able to move up/down…

wCanTopLeft,wCanTopRight…… check if there is a dragon or queen on its top left… if there is, return true, else false

wForward,wBackward… move the wight

wForwardLeft…. capture the dragon/queen on its top left, then move to there

dCanUp,dCanDown…same a wCanUp…

dUpWard,dDownward…. move the dragon/queen

searchBestMoveW(states) from a list of state, return the state with highest value for wights

searchBestMoveQ(states) from a list of state, return the state with highest value for queen

transStates(possibleMoves) combine every three wights,dragons,queen to a state

getScore() compute the total score of pieces current on board for wights side

getScoreQ() compute the total score of pieces current on board for queen side

makeTree() construct the search tree

search() search the best move

recurSearch() recursively search

‘,,

3. Results

AI VS AI: When depth = 1, the wights win mostly of it. When depth = 2, about fifty percent it’s a draw, another fifty percent is wights win. Then depth =3, the wights win using less turns. When depth is higher, the probability of draw is higher.

ME VS AI: When depth = 1, its easy to win. When depth = 2, the AI easily walk in to my bait. When depth becomes higher, I can make it a tie if I just move one piece between two coordinates, but hard to win.

        Search node created:
        When depth = 1, each piece has 8 direction to move, so number of piece * 8
        When depth = 2, there are #Piece*8 nodes, need to create node for each nodes, so (#Piece*8)^2
        So the search node create is (#Piece*8)^depth
        Search node visited:
        Visited every node in the tree. (#Piece*8)^depth
        Depth of search: The search visited every node in the tree
        Time to find moves: number of piece*8^depth
        Memory used:(#Piece*8)^depth

The program first create the search tree level by level, like creating tree using BFS, when it finished construct the tree, it compare the first group of node on the last level, find the best move from the group, then pass the estimate value to the group's parent. Then move to next group of node on the last level, do the something, when finished, it moves to the second last level, and compare the estimate value group by group, pass the value to their parent, repeatedly until reach the second level. Then it find the move with best estimate value, make that move.

ME VS AI at depth 1:

```
Please enter the X,Y coordinates of piece and the direction you would
like to move in order.
Direction code:
1: Forward                2: Backward                3:Left
4:Right
5:Forward-Left Diagonal   6:Forward-Right Diagonal    7: Backward-Left
Diagonal    8: Backward-Right Diagonal
Please select your character: 1.Wights   2.Queen     3.None  4.Both1
Please enter the depths of AI search level: 1
----------------------Begin---------------------------
. . Q . .
. D D D .
. . . . .
. . . . .
W W W W W
Wights' turn:
X coord: 5
Y coord: 5
Direction:1
. . Q . .
. D D D .
. . . . .
. . . . W
```

```
W W W W .
Queen's turn:
. . Q . D
. D D . .
. . . . .
. . . . W
W W W W .
Wights' turn:
X coord: 4
Y coord: 5
Direction:1
. . Q . D
. D D . .
. . . . W
. . . . .
W W W W .
Queen's turn:
. . . Q D
. D D . .
. . . . W
. . . . .
W W W W .
Wights' turn:
X coord: 5
Y coord: 4
Direction:1
. . . Q D
. D D . .
. . . . W
. . . W .
W W W . .
Queen's turn:
. . . . D
. D D . Q
. . . . W
. . . W .
W W W . .
Wights' turn:
X coord: 5
Y coord: 1
Direction:1
. . . . D
. D D . Q
. . . . W
W . . W .
. W W . .
Queen's turn:
. . . . D
. D D . .
. . . . Q
```

```
W . . W .
. W W . .
Wights' turn:
X coord: 4
Y coord: 4
Direction:6
. . . . D
. D D . .
. . . . W
W . . . .
. W W . .
Wights Won!
9
```

Compare: For a mini level, alpha beta pruning stop search its sibling if the value of current search node is smaller than the parent’s sibling node’s value, because it grandparent only want the maximum value of its parent and its parent’s siblings, so it would be useless to still search remaining node. And same for the max level.

In my implementation, I didn’t use alpha beta value, but the value of each level is calculated differently.

For example, if it’s wights’ turn, and search depth is 3, it would be

```
                     Move        (current state)
                    /    \      -> max
                  move  move   (next move for wights)
                  / \   / \    -> min
                 m  m  m  m       (next move for queen)
                /\ /\  /\ /\    -> max
               m m m m m m m m         (next move for wights)
```

The value of possible states for wights’ turn is calculated by getScore()

And the value of possible states for queen’s turn is calculated by getScoreQ(). And just find the maximum value for every level. Do not need to find max value for max level and min value for min level.

But I did not implement the alpha beta pruning search with my structure, the data structure of my search tree is mess, I can’t find a way to fix it.

4. Discussion

My implementation was stucked, I want to make the implementation better and I really tied, but there are about 2300 lines of code in my implementation, it is hard to debug my code, I should have organized data structure in a more professional way.

I think my idea was right, but I’m having problem to find all the leaf nodes and some other thing, so I used a similar approach to compute the best move, which I don’t think it is calling it self recursively in the right way. The problem is the construction of the tree, I can not figure out how to find the leafNodes group by group.