Programming Languages (Project 1)

Gabriel Howard Jadderson (gajad16)

November 17, 2017

Abstract

The goal of this project is simulate the game of kalaha using haskell.

Contents

1 The Kalaha game with parameters (n, m)

```
module Kalaha where

type PitCount   = Int
type StoneCount = Int
data Kalaha     = Kalaha PitCount StoneCount deriving (Show, Read, Eq)

type KPos       = Int
type KState     = [Int]
type Player     = Bool
```

1.1 The function **startStateImpl**

Very straightforward using replicate and concatenation

```
startStateImpl :: Kalaha -> KState

startStateImpl (Kalaha n m) = (replicate n m)++[0]++(replicate n m)++[0]
```

1.2 The function **movesImpl**

This function first calculates the lists of both players and depends on p calls positions who filters the elements that are equals to zero

```
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl g p s = let n = extract g
                      l1 = take n s
                      l2 = init $ drop (n+1) s
                  in case p of True -> positions l2 (n+1)
                               False -> positions l1 0
                  where extract (Kalaha n _) = n
                        positions [] _ = []
                        positions (s:ss) pos = if(s>0) then pos:(positions ss (pos+1))
    else (positions ss (pos+1))
```

1.3 The function **valueImpl**

This function first calculates the difference between two pits

```
valueImpl :: Kalaha -> KState -> Double
valueImpl g s = fromIntegral $ (s !! (extract g)) - (s !! ((*2) (extract g) + 1))
                where extract (Kalaha n _) = n
```

1.4 The function **moveImpl**

Instead of iterating over the list as we would naively do we can compute how many times we would iterate over the entire list. Easy to note that this is equal to the number of stones divided by $2n + 1$. The remainder of this division would be the number of stones left after all the "entire" rounds. Now this is easily done in O(n) by adding the number of entire rounds to all the elements (except the other's player kahala) and then iterating only once to add the last stones.

Instead of having two cases, one for each player, we used the fact that the board is kind of symmetric. We only implemented the function for player False and then reduced the case of player True to player False.

```
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)

moveImpl g True s pos =
  let (npos, nst) = moveImpl g False (change_player s) (pos-(extract g+1))
  in (not npos, change_player nst)
  where change_player s = drop (extract g +1) s ++ take (extract g+1) s
        extract (Kalaha n _) = n



moveImpl g False s pos =
    let
      var = s!!pos
      divi = div var (2*n+1)
      modi = mod var (2*n+1)
      s1 = change pos 0 s
      s2 = map (+divi) (init s1) ++ [last s1]

      final = (pos+1) `mod` (2*n+1)
      initial = (pos+modi) `mod` (2*n+1)

      nKstate = if modi == 0 then s2
                  else map (func final initial) (zip s2 [0..])
      nKstate' = if nKstate!!initial == 1 && initial<n then takeInicial initial nKstate
                     else nKstate
    in (initial /= n, checkfinal nKstate')
    where
      n = extract g
      extract (Kalaha n _) = n
      change _ _ [] = []
      change pos val (a:as) = let na = if (pos == 0) then val
                                       else a
                              in na:(change (pos-1) val as)
      func final initial (x, pos) = let r = final>initial
                              in if (not r && final <= pos && pos <= initial)
                                    || (r && pos /=(2*n+1)
                                        && (pos <= initial || final <= pos)) then x+1
                                 else x
      takeInicial pos s = let var2  = s!!(2*n-pos) + s!!(n) + 1
                         in  change n (var2) . change pos 0
                                . change (2*n-pos) 0 $ s
      checkfinal s = if movesImpl g True s == [] || movesImpl g False s == [] then
                            let p1 = s!!n + (sum $ take n s)
                                p2 = s!!(2*n+1) + (sum $ (take n . drop (n+1)) s)
                            in replicate n 0 ++ [p1] ++ replicate n 0 ++ [p2]
                     else s
```

1.5 The function `showGameImpl`

As stated, we compute $maxLen = length(show(n * m * 2))$, then for a number x we know we have to put $maxLen - length(showx)$ spaces when printing. This can be done using replicate.

```
showGameImpl :: Kalaha -> KState -> String
showGameImpl g s =
  prints ++ concatMap putsp b
  ++ "\n" ++ (putsp (s!!(2*n+1)) ++ concatMap (replicate n) prints ++ putsp (s!!n))
  ++ "\n" ++ prints ++ concatMap putsp a
  where   maxLen = length (show (n*m*2))
          putsp x = replicate (maxLen - length (show x)) ' ' ++ show x
          prints = replicate maxLen ' '
          a = take n $ s
          b = reverse . take n . drop (n+1) $ s
          n = extractN g
          extractN (Kalaha n _) = n
          m = extractM g
          extractM (Kalaha _ m) = m
```

2 Trees

```
data Tree m v  = Node v [(m,Tree m v)] deriving (Eq, Show)
```

2.1 The function **takeTree**

We recursevily go down the tree and when we get to the height we want we cut all the children.

```
takeTree :: Int -> Tree m v -> Tree m v
takeTree 0 (Node v ls) = Node v []
takeTree h (Node v ls) = Node v (map (\(x,y) -> (x, takeTree (h-1) y)) ls)
```

3 The Minimax algorithm

```
data Game s m = Game {
    startState    :: s,
    showGame      :: s -> String,
    move          :: Player -> s -> m -> (Player,s),
    moves         :: Player -> s -> [m],
    value         :: Player -> s -> Double}

kalahaGame :: Kalaha -> Game KState KPos
kalahaGame k = Game {
    startState = startStateImpl k,
    showGame   = showGameImpl k,
    move       = moveImpl k,
    moves      = movesImpl k,
    value      = const (valueImpl k)}

startTree :: Game s m -> Player -> Tree m (Player,Double)
startTree g p = tree g (p, startState g)
```

3.1 The function **tree**

Implemented as said on the statement. We create a node with the given player and the value of the state. And then recursevily called the tree constructor function on its children.

```
tree      :: Game s m -> (Player, s) -> Tree m (Player, Double)
tree g (p,s) = Node (p, (value g) p s)
            (zip (moves g p s) (map (tree g . move g p s) (moves g p s)))
```

3.2 The function **minimax**

We note that when the statement means the value it refers to the minimax value, not to the value function on the game data.

```
minimax   :: (Ord m) => Tree m (Player, Double) -> (Maybe m, Double)
minimax (Node (p, val) []) = (Nothing, val)
minimax (Node (True, val) ls) = swap'.maximum $ zip (map (snd.minimax.snd) ls) (map (Just . fst) ls)
    where   swap' (a,b) = (b,a)
minimax (Node (False, val) ls) = swap'.minimum $ zip (map (snd.minimax.snd) ls) (map (Just . fst) ls)
    where   swap' (a,b) = (b,a)
```

3.3 The function **minimaxAlphaBeta**

We simulate a for loop by using a foldl function. Everything else is just a translation of the provided code (and wikipedia's article code).

```
inf = 10000000000.0
type AlphaBeta = (Double,Double)

minimaxAlphaBeta :: (Ord m) => AlphaBeta -> Tree m (Player, Double) -> (Maybe m, Double)
minimaxAlphaBeta ab (Node (b, val) []) = (Nothing, val)
minimaxAlphaBeta (al, be) (Node (True, val) ls) =
    let (v, mov, al', be') = foldl fmax (-inf, fst.head $ ls, al, be) $ (map swap' ls)
    in (Just mov, v)
        where fmax (v, mov, al', be') (t, m2) =
                if be' <= al' then (v, mov, al', be')
                else let    nv = v `max` (snd $ minimaxAlphaBeta (al', be') t)
                            nmov = if nv > v then m2
                                   else mov
                            nal' = al' `max` nv
                     in (nv, nmov, nal', be')
              swap' (a,b) = (b,a)

minimaxAlphaBeta (al, be) (Node (False, val) ls) =
    let (v, mov, al', be') = foldl fmin (inf, fst.head $ ls, al, be) $ (map swap' ls)
    in (Just mov, v)
        where fmin (v, mov, al', be') (t, m2) =
                if be' <= al' then (v, mov, al', be')
                else let    nv = v `min` (snd $ minimaxAlphaBeta (al', be') t)
                            nmov = if nv < v then m2
                                   else mov
                            nbe' = be' `min` nv
                     in (nv, nmov, al', nbe')
              swap' (a,b) = (b,a)
```

4 Testing and sample executions

```
runhaskell MainTestRun.hs “Kalaha 6 6” “(Minimax, 5)” “(AlphaBeta, 4)”
moveImpl g False [6,6,6,6,6,6,0,6,6,6,6,6,6,0] 0 == (False,[0,7,7,7,7,7,1,6,6,6,6,6,6,0])
moveImpl g False [0,7,7,7,7,7,1,6,6,6,6,6,6,0] 2 == (True,[0,7,0,8,8,8,2,7,7,7,6,6,6,0])
moveImpl g True [0,7,0,8,8,8,2,7,7,7,6,6,6,0] 7 == (False,[1,7,0,8,8,8,2,0,8,8,7,7,7,1])
moveImpl g False [1,7,0,8,8,8,2,0,8,8,7,7,7,1] 0 == (True,[0,8,0,8,8,8,2,0,8,8,7,7,7,1])
moveImpl g True [0,8,0,8,8,8,2,0,8,8,7,7,7,1] 8 == (False,[1,9,1,8,8,8,2,0,0,9,8,8,8,2])
moveImpl g False [1,9,1,8,8,8,2,0,0,9,8,8,8,2] 0 == (True,[0,10,1,8,8,8,2,0,0,9,8,8,8,2])
moveImpl g True [0,10,1,8,8,8,2,0,0,9,8,8,8,2] 9 == (False,[1,11,2,9,9,8,2,0,0,0,9,9,9,3])
moveImpl g False [1,11,2,9,9,8,2,0,0,0,9,9,9,3] 0 == (True,[0,12,2,9,9,8,2,0,0,0,9,9,9,3])
moveImpl g True [0,12,2,9,9,8,2,0,0,0,9,9,9,3] 11 == (False,[1,13,3,10,10,0,2,0,0,0,9,0,10,14])
moveImpl g False [1,13,3,10,10,0,2,0,0,0,9,0,10,14] 0 == (True,[0,14,3,10,10,0,2,0,0,0,9,0,10,14])
moveImpl g True [0,14,3,10,10,0,2,0,0,0,9,0,10,14] 10 == (False,[1,15,4,11,11,1,2,0,0,0,0,1,11,15])
moveImpl g False [1,15,4,11,11,1,2,0,0,0,0,1,11,15] 0 == (True,[0,16,4,11,11,1,2,0,0,0,0,1,11,15])
moveImpl g True [0,16,4,11,11,1,2,0,0,0,0,1,11,15] 12 == (False,[1,17,0,12,12,2,2,1,1,1,0,1,0,22])
moveImpl g False [1,17,0,12,12,2,2,1,1,1,0,1,0,22] 0 == (True,[0,18,0,12,12,2,2,1,1,1,0,1,0,22])
moveImpl g True [0,18,0,12,12,2,2,1,1,1,0,1,0,22] 7 == (False,[0,18,0,12,12,2,2,0,2,1,0,1,0,22])
moveImpl g False [0,18,0,12,12,2,2,0,2,1,0,1,0,22] 4 == (True,[1,19,1,13,0,3,3,1,3,2,1,2,1,22])
moveImpl g True [1,19,1,13,0,3,3,1,3,2,1,2,1,22] 7 == (False,[1,19,1,13,0,3,3,0,4,2,1,2,1,22])
moveImpl g False [1,19,1,13,0,3,3,0,4,2,1,2,1,22] 0 == (True,[0,20,1,13,0,3,3,0,4,2,1,2,1,22])
moveImpl g True [0,20,1,13,0,3,3,0,4,2,1,2,1,22] 8 == (False,[0,20,1,13,0,3,3,0,0,3,2,3,2,22])
moveImpl g False [0,20,1,13,0,3,3,0,0,3,2,3,2,22] 2 == (True,[0,20,0,14,0,3,3,0,0,3,2,3,2,22])
moveImpl g True [0,20,0,14,0,3,3,0,0,3,2,3,2,22] 9 == (False,[0,20,0,14,0,3,3,0,0,0,3,4,3,22])
moveImpl g False [0,20,0,14,0,3,3,0,0,0,3,4,3,22] 3 == (True,[1,21,1,1,2,4,4,1,1,1,4,5,4,22])
moveImpl g True [1,21,1,1,2,4,4,1,1,1,4,5,4,22] 7 == (False,[1,21,1,1,2,4,4,0,2,1,4,5,4,22])
moveImpl g False [1,21,1,1,2,4,4,0,2,1,4,5,4,22] 0 == (True,[0,22,1,1,2,4,4,0,2,1,4,5,4,22])
moveImpl g True [0,22,1,1,2,4,4,0,2,1,4,5,4,22] 8 == (False,[0,22,1,1,2,4,4,0,0,2,5,5,4,22])
moveImpl g False [0,22,1,1,2,4,4,0,0,2,5,5,4,22] 2 == (True,[0,22,0,2,2,4,4,0,0,2,5,5,4,22])
moveImpl g True [0,22,0,2,2,4,4,0,0,2,5,5,4,22] 9 == (False,[0,22,0,2,2,4,4,0,0,0,6,6,4,22])
moveImpl g False [0,22,0,2,2,4,4,0,0,0,6,6,4,22] 3 == (True,[0,22,0,0,3,5,4,0,0,0,6,6,4,22])
moveImpl g True [0,22,0,0,3,5,4,0,0,0,6,6,4,22] 10 == (False,[1,23,1,0,3,5,4,0,0,0,0,7,5,23])
moveImpl g False [1,23,1,0,3,5,4,0,0,0,0,7,5,23] 4 == (True,[1,23,1,0,0,6,5,1,0,0,0,7,5,23])
moveImpl g True [1,23,1,0,0,6,5,1,0,0,0,7,5,23] 7 == (False,[1,23,1,0,0,6,5,0,0,0,0,7,5,24])
moveImpl g False [1,23,1,0,0,6,5,0,0,0,0,7,5,24] 0 == (True,[0,24,1,0,0,6,5,0,0,0,0,7,5,24])
moveImpl g True [0,24,1,0,0,6,5,0,0,0,0,7,5,24] 11 == (False,[1,25,2,1,1,6,5,0,0,0,0,0,6,25])
moveImpl g False [1,25,2,1,1,6,5,0,0,0,0,0,6,25] 5 == (True,[1,25,2,1,1,0,6,1,1,1,1,1,6,25])
moveImpl g True [1,25,2,1,1,0,6,1,1,1,1,1,6,25] 7 == (False,[1,25,2,1,1,0,6,0,2,1,1,1,6,25])
moveImpl g False [1,25,2,1,1,0,6,0,2,1,1,1,6,25] 0 == (True,[0,26,2,1,1,0,6,0,2,1,1,1,6,25])
moveImpl g True [0,26,2,1,1,0,6,0,2,1,1,1,6,25] 8 == (False,[0,26,2,1,1,0,6,0,0,2,2,1,6,25])
moveImpl g False [0,26,2,1,1,0,6,0,0,2,2,1,6,25] 3 == (True,[0,26,2,0,2,0,6,0,0,2,2,1,6,25])
moveImpl g True [0,26,2,0,2,0,6,0,0,2,2,1,6,25] 10 == (False,[0,26,2,0,2,0,6,0,0,2,0,2,7,25])
moveImpl g False [0,26,2,0,2,0,6,0,0,2,0,2,7,25] 2 == (True,[0,26,0,1,3,0,6,0,0,2,0,2,7,25])
moveImpl g True [0,26,0,1,3,0,6,0,0,2,0,2,7,25] 9 == (False,[0,26,0,1,3,0,6,0,0,0,1,3,7,25])
moveImpl g False [0,26,0,1,3,0,6,0,0,0,1,3,7,25] 1 == (True,[2,2,2,3,5,2,8,2,2,2,3,5,9,25])
moveImpl g True [2,2,2,3,5,2,8,2,2,2,3,5,9,25] 7 == (False,[2,2,2,3,5,2,8,0,3,3,3,5,9,25])
moveImpl g False [2,2,2,3,5,2,8,0,3,3,3,5,9,25] 0 == (True,[0,3,3,3,5,2,8,0,3,3,3,5,9,25])
moveImpl g True [0,3,3,3,5,2,8,0,3,3,3,5,9,25] 9 == (False,[0,3,3,3,5,2,8,0,3,0,4,6,10,25])
moveImpl g False [0,3,3,3,5,2,8,0,3,0,4,6,10,25] 2 == (True,[0,3,0,4,6,3,8,0,3,0,4,6,10,25])
moveImpl g True [0,3,0,4,6,3,8,0,3,0,4,6,10,25] 8 == (False,[0,3,0,4,6,3,8,0,0,1,5,7,10,25])
moveImpl g False [0,3,0,4,6,3,8,0,0,1,5,7,10,25] 1 == (True,[0,0,1,5,7,3,8,0,0,1,5,7,10,25])
moveImpl g True [0,0,1,5,7,3,8,0,0,1,5,7,10,25] 9 == (False,[0,0,1,5,7,3,8,0,0,0,6,7,10,25])
moveImpl g False [0,0,1,5,7,3,8,0,0,0,6,7,10,25] 2 == (True,[0,0,0,6,7,3,8,0,0,0,6,7,10,25])
moveImpl g True [0,0,0,6,7,3,8,0,0,0,6,7,10,25] 10 == (False,[1,1,1,6,7,3,8,0,0,0,0,8,11,26])
moveImpl g False [1,1,1,6,7,3,8,0,0,0,0,8,11,26] 0 == (True,[0,2,1,6,7,3,8,0,0,0,0,8,11,26])
moveImpl g True [0,2,1,6,7,3,8,0,0,0,0,8,11,26] 12 == (False,[1,3,0,7,8,4,8,1,1,1,0,8,0,30])
moveImpl g False [1,3,0,7,8,4,8,1,1,1,0,8,0,30] 0 == (True,[0,4,0,7,8,4,8,1,1,1,0,8,0,30])
moveImpl g True [0,4,0,7,8,4,8,1,1,1,0,8,0,30] 8 == (False,[0,4,0,7,8,4,8,1,0,2,0,8,0,30])
moveImpl g False [0,4,0,7,8,4,8,1,0,2,0,8,0,30] 1 == (True,[0,0,1,8,9,5,8,1,0,2,0,8,0,30])
moveImpl g True [0,0,1,8,9,5,8,1,0,2,0,8,0,30] 9 == (False,[0,0,1,8,9,5,8,1,0,0,1,9,0,30])
moveImpl g False [0,0,1,8,9,5,8,1,0,0,1,9,0,30] 2 == (True,[0,0,0,9,9,5,8,1,0,0,1,9,0,30])
moveImpl g True [0,0,0,9,9,5,8,1,0,0,1,9,0,30] 10 == (False,[0,0,0,9,9,5,8,1,0,0,0,10,0,30])
moveImpl g False [0,0,0,9,9,5,8,1,0,0,0,10,0,30] 5 == (True,[0,0,0,9,9,0,9,2,1,1,1,10,0,30])
moveImpl g True [0,0,0,9,9,0,9,2,1,1,1,10,0,30] 7 == (False,[0,0,0,9,9,0,9,0,2,2,1,10,0,30])
moveImpl g False [0,0,0,9,9,0,9,0,2,2,1,10,0,30] 4 == (True,[0,0,0,9,0,1,12,1,3,3,2,11,0,30])
moveImpl g True [0,0,0,9,0,1,12,1,3,3,2,11,0,30] 7 == (False,[0,0,0,9,0,1,12,0,4,3,2,11,0,30])
moveImpl g False [0,0,0,9,0,1,12,0,4,3,2,11,0,30] 3 == (True,[0,0,0,0,1,2,13,1,5,4,3,12,1,30])
moveImpl g True [0,0,0,0,1,2,13,1,5,4,3,12,1,30] 7 == (False,[0,0,0,0,1,2,13,0,6,4,3,12,1,30])
moveImpl g False [0,0,0,0,1,2,13,0,6,4,3,12,1,30] 4 == (True,[0,0,0,0,0,3,13,0,6,4,3,12,1,30])
moveImpl g True [0,0,0,0,0,3,13,0,6,4,3,12,1,30] 8 == (False,[1,0,0,0,0,3,13,0,0,5,4,13,2,31])
moveImpl g False [1,0,0,0,0,3,13,0,0,5,4,13,2,31] 5 == (True,[1,0,0,0,0,0,14,1,1,5,4,13,2,31])
moveImpl g True [1,0,0,0,0,0,14,1,1,5,4,13,2,31] 9 == (False,[2,0,0,0,0,0,14,1,1,0,5,14,3,32])
moveImpl g False [2,0,0,0,0,0,14,1,1,0,5,14,3,32] 0 == (True,[0,1,0,0,0,0,20,1,1,0,0,14,3,32])
moveImpl g True [0,1,0,0,0,0,20,1,1,0,0,14,3,32] 12 == (False,[1,2,0,0,0,0,20,1,1,0,0,14,0,33])
moveImpl g False [1,2,0,0,0,0,20,1,1,0,0,14,0,33] 0 == (True,[0,3,0,0,0,0,20,1,1,0,0,14,0,33])
moveImpl g True [0,3,0,0,0,0,20,1,1,0,0,14,0,33] 7 == (False,[0,3,0,0,0,0,20,0,2,0,0,14,0,33])
moveImpl g False [0,3,0,0,0,0,20,0,2,0,0,14,0,33] 1 == (True,[0,0,1,1,0,0,23,0,0,0,0,14,0,33])
moveImpl g True [0,0,1,1,0,0,23,0,0,0,0,14,0,33] 11 == (False,[1,1,2,2,1,1,23,1,1,1,1,1,2,34])
moveImpl g False [1,1,2,2,1,1,23,1,1,1,1,1,2,34] 0 == (True,[0,2,2,2,1,1,23,1,1,1,1,1,2,34])
moveImpl g True [0,2,2,2,1,1,23,1,1,1,1,1,2,34] 7 == (False,[0,2,2,2,1,1,23,0,2,1,1,1,2,34])
moveImpl g False [0,2,2,2,1,1,23,0,2,1,1,1,2,34] 1 == (True,[0,0,3,3,1,1,23,0,2,1,1,1,2,34])
moveImpl g True [0,0,3,3,1,1,23,0,2,1,1,1,2,34] 8 == (False,[0,0,3,3,1,1,23,0,0,2,2,1,2,34])
moveImpl g False [0,0,3,3,1,1,23,0,0,2,2,1,2,34] 4 == (True,[0,0,3,3,0,2,23,0,0,2,2,1,2,34])
moveImpl g True [0,0,3,3,0,2,23,0,0,2,2,1,2,34] 10 == (False,[0,0,3,3,0,2,23,0,0,2,0,2,3,34])
moveImpl g False [0,0,3,3,0,2,23,0,0,2,0,2,3,34] 2 == (True,[0,0,0,4,1,3,23,0,0,2,0,2,3,34])
moveImpl g True [0,0,0,4,1,3,23,0,0,2,0,2,3,34] 9 == (False,[0,0,0,4,1,3,23,0,0,0,1,3,3,34])
moveImpl g False [0,0,0,4,1,3,23,0,0,0,1,3,3,34] 4 == (True,[0,0,0,4,0,4,23,0,0,0,1,3,3,34])
moveImpl g True [0,0,0,4,0,4,23,0,0,0,1,3,3,34] 10 == (False,[0,0,0,4,0,4,23,0,0,0,0,4,3,34])
moveImpl g False [0,0,0,4,0,4,23,0,0,0,0,4,3,34] 3 == (True,[0,0,0,0,1,5,24,1,0,0,0,4,3,34])
moveImpl g True [0,0,0,0,1,5,24,1,0,0,0,4,3,34] 11 == (False,[1,1,0,0,1,5,24,1,0,0,0,0,4,35])
moveImpl g False [1,1,0,0,1,5,24,1,0,0,0,0,4,35] 4 == (True,[1,1,0,0,0,6,24,1,0,0,0,0,4,35])
moveImpl g True [1,1,0,0,0,6,24,1,0,0,0,0,4,35] 7 == (False,[1,1,0,0,0,6,24,0,0,0,0,0,4,36])
moveImpl g False [1,1,0,0,0,6,24,0,0,0,0,0,4,36] 5 == (True,[1,1,0,0,0,0,25,1,1,1,1,1,4,36])
moveImpl g True [1,1,0,0,0,0,25,1,1,1,1,1,4,36] 7 == (False,[1,1,0,0,0,0,25,0,2,1,1,1,4,36])
moveImpl g False [1,1,0,0,0,0,25,0,2,1,1,1,4,36] 1 == (True,[1,0,0,0,0,0,27,0,2,1,0,1,4,36])
moveImpl g True [1,0,0,0,0,0,27,0,2,1,0,1,4,36] 12 == (False,[2,1,1,0,0,0,27,0,2,1,0,1,0,37])
moveImpl g False [2,1,1,0,0,0,27,0,2,1,0,1,0,37] 0 == (True,[0,2,2,0,0,0,27,0,2,1,0,1,0,37])
moveImpl g True [0,2,2,0,0,0,27,0,2,1,0,1,0,37] 8 == (False,[0,2,0,0,0,0,27,0,0,2,0,1,0,40])
moveImpl g False [0,2,0,0,0,0,27,0,0,2,0,1,0,40] 1 == (True,[0,0,1,0,0,0,30,0,0,0,0,1,0,40])
moveImpl g True [0,0,1,0,0,0,30,0,0,0,0,1,0,40] 11 == (False,[0,0,0,0,0,0,31,0,0,0,0,0,0,41])
No available moves!
```