Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Goodbye Haskell

Ultima perla: Unbeatable Tic-Tac-Toe

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **16**, 29 aprile 2021

Lezione 16a

Il gioco del Filetto

Il gioco del filetto

Dovrebbe essere noto a tutti il gioco del filetto (o tris, o tic-tac-toe, o noughts and crosses, etc.).

Si gioca (usualmente) in una griglia (matrice) 3×3: ogni giocatore a turno sceglie una casella dove porre il proprio simbolo.

Vince il giocatore che riesce a mettere **3 simboli** lungo una **riga**, **colonna** o **diagonale** (**Fig. 1**).

Se la matrice si riempie senza filetti, la partita è patta (**Fig. 2**).

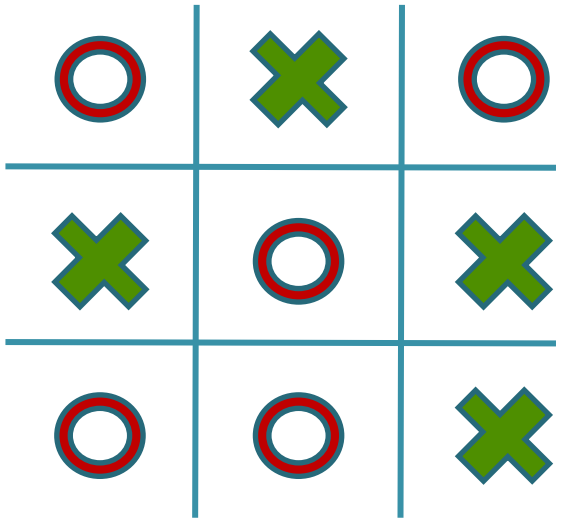


Fig. 1: O vince

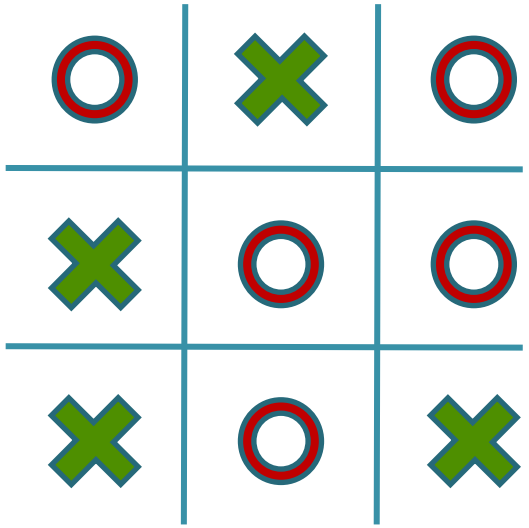


Fig. 2: patta

Rappresentazione del Gioco

Parametrizziamo la dimensione della griglia.

Per i giocatori definiamo un **tipo induttivo** apposito con **3 costruttori** (senza parametri), quindi **un tipo finito con 3 differenti valori**: il terzo (B per "blank) ci serve per modellare le caselle vuote.

Alla fine, lo stato del gioco è rappresentato da una lista di liste, che è il modo standard di rappresentare una matrice in Haskell.

```
-- parametrizziamo il gioco risp. alla dimensione:
size = 3
-- tipo induttivo per i giocatori (B=blank, vuoto):
data Player = O | B | X
     deriving (Eq, Ord, Show)
-- la matrice del gioco è una lista di liste
type Grid = [[Player]]
```

Definizioni base: griglia, turno etc.

Per usare funzioni di utilità su caratteri (ad es. `isDigit`) liste (ad esempio `transpose`) e input/output, importiamo delle librerie.

Oltre alla griglia, lo stato del gioco è determinato dal turno. Scriviamo una funzione che **cambia il turno**.

Alcune utilità sulla griglia: costante `empty` e test `isFull`.

```
import Data.Char
import Data.List
import System.IO

-- funzione per cambiare turno:
next :: Player -> Player
next O = X
next B = B
next X = O

-- griglia vuota:
empty :: Grid
empty = replicate size (replicate size B)

-- test di griglia piena:
isFull :: Grid -> Bool
isFull = all (/=B) . concat
```

Turno e validità di una mossa

In realtà il **turno** è un'informazione **ridondante**. Potrebbe facilmente essere dedotta dalla griglia (sapendo che O muove sempre per primo).

Indichiamo una mossa semplicemente con un numero intero tra 1 e `size^2`: la griglia viene linearizzata per verificare se il numero è corretto e riferito a una **casella ancora vuota** (ricordare che gli elementi della lista **partono da 0**)

```
turn :: Grid -> Player
turn g = if os <= xs then O else X
   where
      os = length (filter (== O) ps)
      xs = length (filter (== X) ps)
      ps = concat g

valid :: Grid -> Int -> Bool
valid g i = 0 < i && i <= size^2 &&
                        concat g !! (i-1) == B
```

Eseguire mosse/verifica della vittoria

Per eseguire la mossa, occorre trovare la riga e infilare il nuovo simbolo al posto giusto.

Controllare la vittoria è abbastanza facile: basta che ci sia almeno (`any`) una linea con lo stesso simbolo (`all (==p)`).

Le colonne si ottengono trasponendo la matrice.

Le diagonali un po' più difficile. Osservare come ottenere la diagonale "ascendente" (`diag (map reverse g)`).

```
move :: Grid -> Int -> Player -> Grid
move (g:gs) i p =
  if i > size then g : move gs (i-size) p
     else (take (i-1) g ++ [p] ++ drop i g):gs
wins :: Player -> Grid -> Bool
wins p g = any line (rows ++ cols ++ dias)
   where line = all (== p)
         rows = g
         cols = transpose g
         dias = [diag g, diag (map reverse g)]
         diag g = [g !! n !! n | n <- [0..size-1]]
```

Stampa della Griglia

Trasformiamo una cella in una stringa…

quindi una riga…

e infine la matrice.

A questo punto è sufficiente stampare la stringa ottenuta.


```
showCell :: Player -> String
showCell X = " x "
showCell B = " . "
showCell O = " o "

rowToString :: [Player] -> String
rowToString  rs = foldr1 (++) (map showCell rs) ++"\n"

gridToString :: [[Player]] -> String
gridToString gs = foldr1 (++) (map rowToString gs)

putGrid :: Grid -> IO ()
putGrid g = putStr (gridToString g)
```

Inserimento di una mossa

Input di una mossa.

```
getNat :: String -> IO Int
getNat prompt =
   do putStr prompt
      xs <- getLine
      if xs /= [] && all isDigit xs
        then return (read xs)
        else do putStrLn "Error: Invalid Number\n"
                getNat prompt
```

Ciclo interattivo uomo vs uomo

Ecco finalmente…

```
tictactoe :: IO ()
tictactoe = run empty O

run :: Grid -> Player -> IO ()
run g p = do putGrid g
             run' g p

run' :: Grid -> Player -> IO ()
run' g p| wins O g  = putStrLn "Player O wins!\n"
        | wins X g  = putStrLn "Player X wins!\n"
        | isFull g  = putStrLn "Draw!\n"
        | otherwise =
           do i <- getNat ("Inserisci Mossa: ")
              if valid g i
                then run (move g i p) (next p)
                else do putStrLn ("Mossa Errata!\n")
                        run' g p
```

Lezione 16b

Game Trees e Algoritmo minimax

Progettiamo un giocatore artificiale

Cominciamo con il mostrare un approccio che "idealmente" si applica a qualsiasi gioco a due giocatori a "somma zero", come il ticTacToe, ma anche Scacchi, Dama etc.

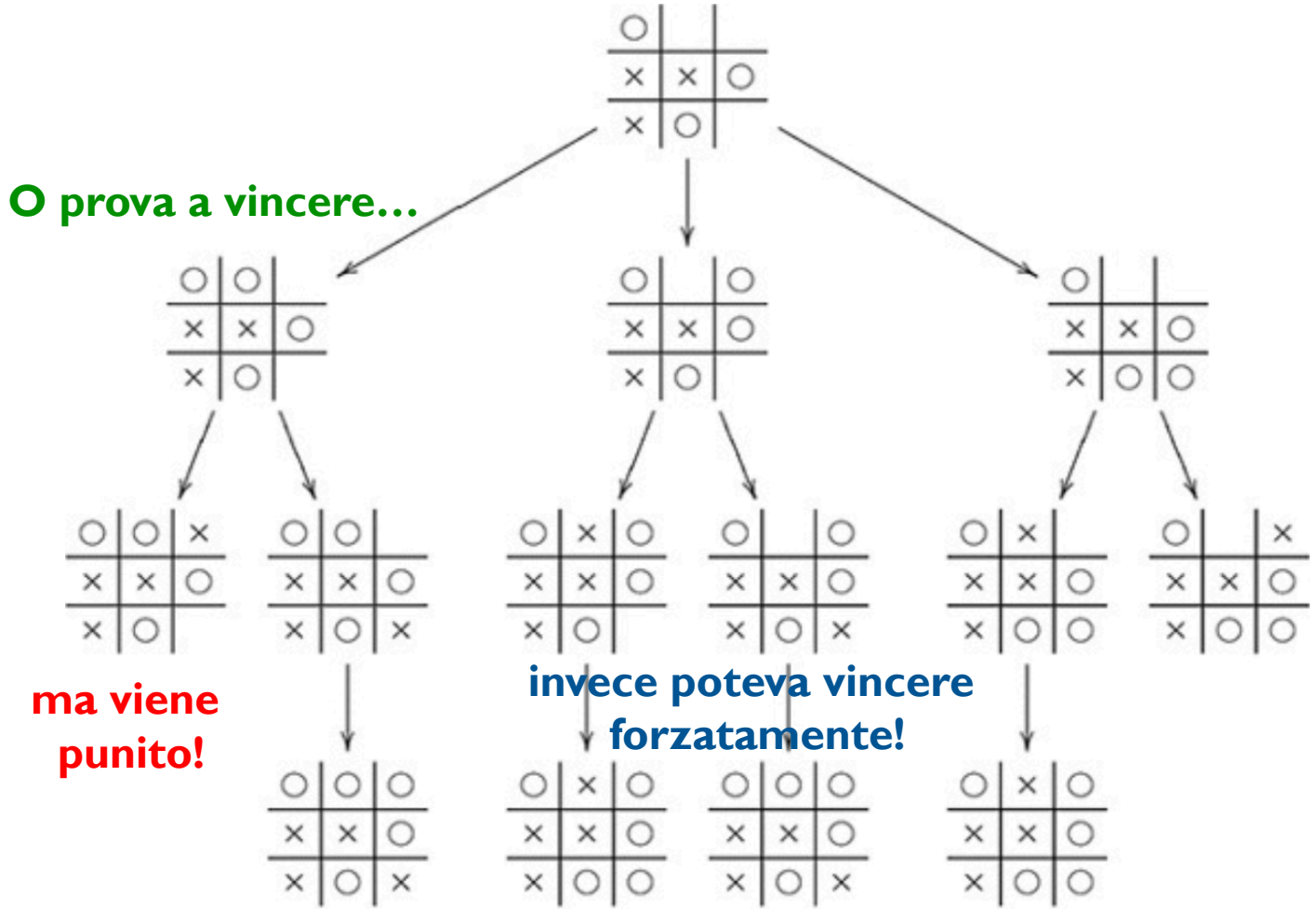
Idea:

- generare "tutte" le posizioni future in un albero;
- valutare le posizioni che si trovano a una certa profondità
- applicare l'algoritmo *minimax*, cioè cercare la mossa che massimizza il vantaggio del giocatore, assumendo le migliori mosse dell'avversario.

Faremo uso degli alberi *n*-ari.

L'albero dei futuri

Esempio di analisi dei futuri nel TicTacToe.



Definizioni Base

Immaginiamo innanzitutto di avere una funzione che genera tutte le mosse possibili in una certa posizione.

Diventa relativamente banale generare **tutto l'albero del gioco**...

… eventualmente **potato** a un certo livello

Grazie alla **laziness**, di sicuro **non viene mai generato l'albero** a profondità superiori a quella prevista da `prune` (osservate che è una forma di `take` generalizzata agli alberi).

```
-- supponiamo di avere un tipo position, nel nostro
-- caso Grid
moves :: position -> [position]
data GameTree = Node position [GameTree]

gameTree :: position -> GameTree
gameTree p = Node p (map gameTree (moves p))

prune :: Int -> GameTree -> GameTree
prune 0 (Node x ts) = Node x []
prune n (Node x ts) = Node x (map (prune (n-1)) ts)
```

Algoritmo minimax 1

Supponiamo di avere una funzione:

```
static :: Ord a => position -> a
```

che assegna una **static evaluation** a una posizione. Un giocatore deve ottimizzare questa funzione assumendo le migliori mosse dell'avversario.

Assumiamo che se la valutazione di una posizione è *w* per un giocatore, allora è *-w* per l'altro.

Questi giochi sono anche detti a **somma zero**, infatti la vittoria di un giocatore implica la sconfitta dell'altro.

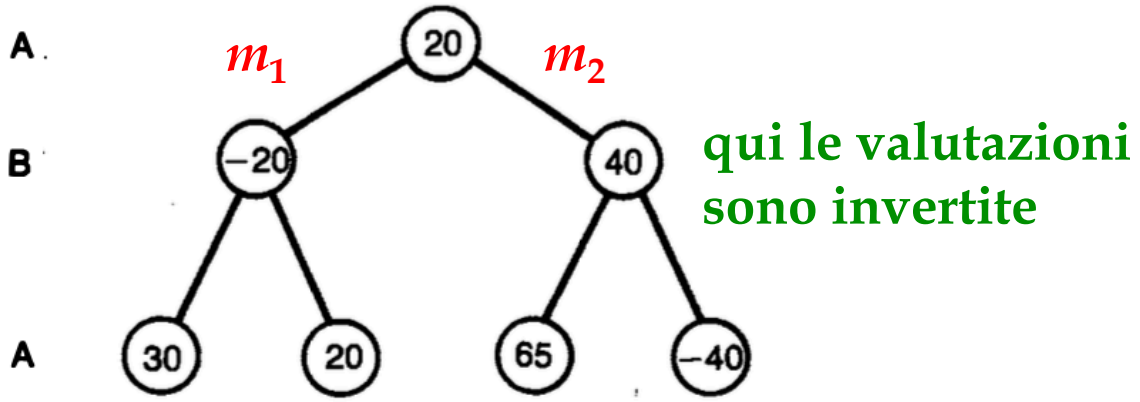
Assumendo le posizioni più "lontane nel tempo" più importanti, è possibile scegliere la mossa e valutare quindi la posizione corrente.

Algoritmo minimax 2

Nell'esempio in Figura, un **principiante** al posto del giocatore A potrebbe scegliere m_2 attratto dalla posizione valutata 65… tuttavia l'avversario potrebbe giocare meglio, portandolo in tal caso a una posizione che valuta -40.

Viceversa un **maestro**, sceglierà sempre m_1, perché garantisce, alla peggio una utilità di 20.

Osservate che i valori dei nodi dei primi 2 livelli sono implicati dalla valutazione delle foglie (non serve quindi calcolare la loro valutazione statica e con lazy evaluation non si farà!.



Riassumendo in Haskell

Quindi, minimax calcola la valutazione statica sulle foglie.

Ignora invece il valore nei **nodi interni**, il cui valore **dipende** dal calcolo **minimax**.

Osservate che a ogni chiamata viene invertito il segno, perché occorre **ottimizzare** la scelta del **giocatore corrente**.

Osservate che la **laziness** assicura che static sarà calcolata **solo sulle foglie**.

La funzione scritta sotto valuta la posizione, ma **non dice qual è la sequenza ottimale** calcolata (cioè le mosse che realizzano il minimax): modificare il programma programma per ottenere questa informazione (**Esercizio**).

```
minimax (Node x []) = x
minimax (Node x gs) = - min (map minimax gs)

dynamic  n =
     minimax . fmap static . prune n . gameTree
```

Applichiamo al TicTacToe

Nel caso del ticTacToe tuttavia è possibile generare tutto l'albero del gioco e possiamo fissare la profondità di `prune` a 9.

Rispetto a quanto visto, dobbiamo essenzialmente definire le funzioni `moves` e `static`.

Per le mosse, usiamo la funzione `valid`.

È sufficiente una una valutazione "**qualitativa**", cioè se in un sottoalbero vince O, X o nessuno dei due.

```
-- definiamo tutte le mosse possibili in una pos.
moves :: Grid -> Player -> [Grid]
moves g p =[move g i p | valid g i, i <- 0..ncells]

static :: Grid -> Player
static g
  | wins g O  = O
  | wins g X  = X
  | otherwise = B
```

Definiamo best move

Siccome in questo caso è possibile generare tutto l'albero del gioco, fisseremo la profondità di prune a 9!

`minimum` e `maximum` dipendono dall'ordinamento dato sui costruttori di player `O < B < X`.

```
bestMove :: Grid -> Player -> Grid

bestMove g p =
   head [g' | Node (g',p') _ |<- ts, p'= best]
      where tree = prune depth (gameTree g)
            Node (_, best) ts = minimax tree

minimax :: Tree Grid -> Tree (Grid, Player)

minimax (Node g []) = Node (g, static g) []
minimax (Node g ts)  =
  | turn g == O = Node (g, minimum ps) ts'
  | turn g == X = Node (g, maximum ps) ts'
where
   ts'= map minimax ts
   ps = [p | Node (_, p) _ <- ts']
```

Lezione 16

Goodbye Haskell!

Grazie per l'attenzione…

…Domande?