lezione20

November 30, 2023

[ ]:

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 20 - 30 novembre 2023

2 NOI CI SIAMO

2.1 25 novembre:

2.2 Giornata Internazionale per l'eliminazione della violenza sulle donne

2.3 Did This Really Happen?

3 RECAP:

- Alberi di gioco
  - Somma di coppie pari o dispari in una sequenza di interi
  - Anagramma e numero di scambi
  - Dare i resti se si hanno certe monete

3.1 Una classe per visualizzare l'albero con Graphviz

[1]:
```
from pygraphviz import AGraph
class GameNode:
    _num_nodi = 0
    def __init__(self):
        GameNode._num_nodi += 1              # contatore condiviso per dare ID␣
 ↪diversi alle istanze
        self.__id = GameNode._num_nodi
        self._sons = []
        self._shape = 'box'                  # default shape
        self._color = 'black'                # default color
        self._style = 'solid'                # default style

    def dot(self, G:AGraph) -> None:
```

```
        "Costruisco la rappresentazione dell'albero da visualizzare con␣
↪Graphviz"
        G.add_node(self.__id, label=self, color=self._color, style=self._style)
        for son in self._sons:
            G.add_edge(self.__id, son.__id)  # un arco per figlio
            son.dot(G)                       # e ricorsivamente aggiungo i suoi␣
↪figli

    def show(self):
        "layout e visualizzazione dell'albero"
        G = AGraph()
        G.node_attr['shape'] = self._shape
        self.dot(G)
        G.layout('dot')
        return G
```

4 GIOCO: Filetto/Tris

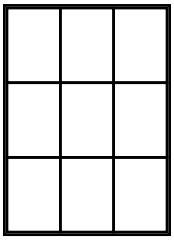
- 2 giocatori
- **configurazione**: scacchiera 3x3 con cimboli **o** oppure **x** (oppure spazio per la casella vuota)
- **mosse possibili**: inserire il simbolo del giocatore di turno in una casella vuota
- **vincita**: 3 simboli uguali in fila (riga, colonna o diagonale)
- **convergenza**: max 9 caselle quindi max 9 mosse

[2]:
```
import jdc

# nuovo tipo di errore per comunicare errori del gioco
class FilettoError(Exception):
    pass

class Filetto(GameNode):
    def __init__(self, configurazione=None):
        "creazione di un nuovo nodo a partire da una data configurazione,␣
↪rappresentata come matrice 3x3"
        super().__init__()
        self._shape = 'plain' # 'Mrecord' oppure 'record'
        if configurazione is None:                          # se non viene␣
↪passata
            configurazione = [ [' ']*3 for i in range(3)]   # ne creiamo una␣
↪vuota
        self._configurazione = configurazione
        self._sons = []                                     # all'inizio non ci␣
↪sono figli

    def __repr__(self):
        "rappresentazione sotto forma di stringa della tabella"
```

```
        # graphviz mostra una scacchiera se il nodo ha shape='record'
        # return '{' + '|'.join(['{'+ '|'.join(riga)+'}' for riga in self.
↪_configurazione ]) + '}'
        # altrimenti se shape='plain' e label è racchiusa tra "<>" è un pezzo␣
↪di HTML (in questo caso una tabella)
        return ('<<table border="1" cellpadding="5" cellspacing="0"><tr>' +
                    ('</tr><tr>'.join( [ ('<td width="20">' +
                                          '</td><td width="20">'.join(riga) +
                                          '</td>' )
                     for riga in self._configurazione ] ) ) +
                '</tr></table>>')
```

[3]:
```
F = Filetto()
F.show()
```

[3]:



4.1 Le mosse valide sono tutte le caselle libere (MA SOLO se non è finita la partita)

[4]:
```
%%add_to Filetto
# --- elenco delle mosse valide
def mosse_valide(self):
    """Trovo le mosse valide (ma non ce ne sono se patta o qualcuno ha vinto)"""
    if self.vittoria('X'): return []
    if self.vittoria('O'): return []
    if self.patta():       return []
    return [ (x,y)   # posso muovere in x,y se c'è uno spazio
                     for y,riga    in enumerate(self._configurazione)
                     for x,casella in enumerate(riga)
                     if casella == ' ']

# definirò fra poco i metodi 'patta' e 'vincente'
```

4.2 il prossimo giocatore si ottiene contando le mosse

- si inizia sempre con 'o'
- però torniamo None se non ci sono più caselle libere

[5]:
```
%%add_to Filetto
# --- Prossimo giocatore
def prossimo_giocatore(self):
    "si inizia sempre col simbolo 'o' quindi basta contare gli ' ' per sapere a␣
 ↪chi tocca"
    conteggio = sum( 1
                    for riga in self._configurazione
                    for cell in riga
                    if  cell == ' ')
    if conteggio == 0:        # se non ci sono spazi
        return None           # non è il turno di nessuno
    elif conteggio % 2 == 1:  # inizia sempre 'o' (con 9 caselle libere)
        return 'O'
    else:
        return 'X'
```

[6]:
```
Filetto().prossimo_giocatore()
```

[6]:
```
'O'
```

4.3 La partita è finita alla pari se non ci sono più mosse disponibili

(ovvero nessuno ha già vinto)

[7]:
```
%%add_to Filetto
# --- Configurazione che dà patta
def patta(self):
    "torno True se siamo in una patta (non ci sono caselle libere)"
    return self.prossimo_giocatore() is None
```

[8]:
```
# Una configurazione piena e senza filetti è
Filetto([[1,2,3],[4,5,6],[7,8,9]]).patta()
```

[8]:
```
True
```

4.4 una configurazione è vincente per un certo giocatorese ci sono 3 simboli in fila uguali al suo

[9]:
```
%%add_to Filetto
# --- Configurazione vincente per un giocatore G o K (non intendo strategia)
def vittoria(self, giocatore):
    """la configurazione corrente è vincente per il giocatore se ci sono 3
    dei suoi simboli in riga, colonna o diagonale"""
    [[A,B,C],
     [D,E,F],   # spacchetto le celle
     [G,H,I]] = self._configurazione
    return (A == B == C == giocatore  # prima riga
         or D == E == F == giocatore  # seconda riga
```

```
         or G == H == I == giocatore  # terza riga
         or A == D == G == giocatore  # prima colonna
         or B == E == H == giocatore  # seconda colonna
         or C == F == I == giocatore  # terza colonna
         or A == E == I == giocatore  # diagonale
         or C == E == G == giocatore  # antidiagonale
        )
```

[10]:
```
# tre 1 allineati in diagonale
Filetto( [[1,1,1],
          [2,1,2],
          [3,3,1]]).vittoria(1)
```

[10]: True

[11]:
```
# le mosse valide all'inizio sono 9
Filetto().mosse_valide()
```

[11]: [(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]

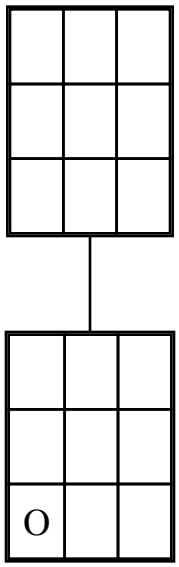
4.5 Per applicare una mossa basta mettereil simbolo GIUSTO nella casella

[12]:
```
%%add_to Filetto
# --- Mosse: inserire il simbolo di turno in una delle caselle vuote
def applica_mossa(self, x, y):
    G = self.genera_figlio(x,y)         # costruisco il figlio con la mossa x,y
    self._sons.append(G)                # e lo aggiungo ai figli
    return self                         # torno self per concatenare metodi
```

[13]:
```
%%add_to Filetto
def genera_figlio(self, x, y):
    """data una coordinata x,y inserisco il giocatore di turno,
    e costruisco un nuovo figlio con la nuova configurazione"""
    if self._configurazione[y][x] != ' ':       # se la casella NON è libera␣
 ↪lancio errore
        raise FilettoError(f"La casella {x} {y} è già occupata")
    # altrimenti tutto OK
    copia = [ riga.copy() for riga in self._configurazione ]    # copio la␣
 ↪configurazione
    copia[y][x] = self.prossimo_giocatore()                     # e ci metto il␣
 ↪simbolo alle coordinate x,y
    return self.__class__(copia)    # e creo un altro oggetto dello stesso tipo␣
 ↪(NOTA: potrebbe essere una sottoclasse))
```

[14]:
```
Filetto().applica_mossa(0,2).show()
```

[14]:

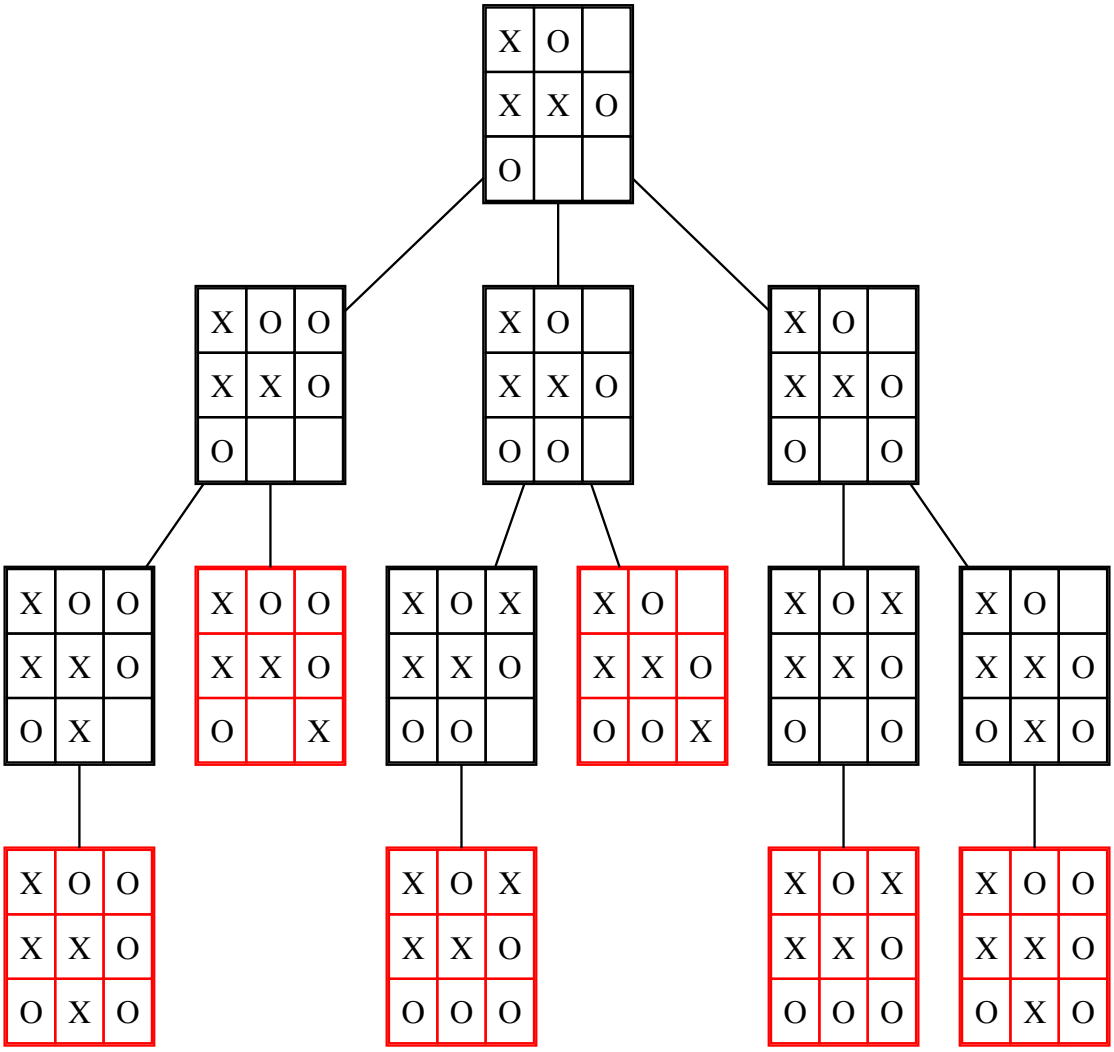


4.6 Finalmente generiamo tutto l'albero di gioco

[15]:
```
%%add_to Filetto
# --- generare l'albero di gioco
def genera(self):
    "Se ci sono mosse valide genero i figli, quindi espando anche i figli"
    mosse = self.mosse_valide()
    if not mosse:
        self._color = 'red'            # coloro le foglie di rosso
    for x,y in mosse:
        self.applica_mossa(x, y)
    for figlio in self._sons:
        figlio.genera()
    return self         # torno self per concatenare metodi
```

[16]:
```
board = [['X','O',' '],
         ['X','X','O'],
         ['O',' ',' ']]
Filetto(board).genera().show()
```

[16]:



4.7 Strategia vincente per il giocatore G

- casi base
  - se patta NO
  - se vincente per G SI
  - se vincente per K NO
- altrimenti: esiste una mossa per G tale che per tutte le mosse di K si arriva sempre ad una posizione vincente x G?
  - se è il turno di G e basta UNA mossa che porti alla vittoria di G
  - se è il turno di K e devono TUTTE portare alla vittoria di G

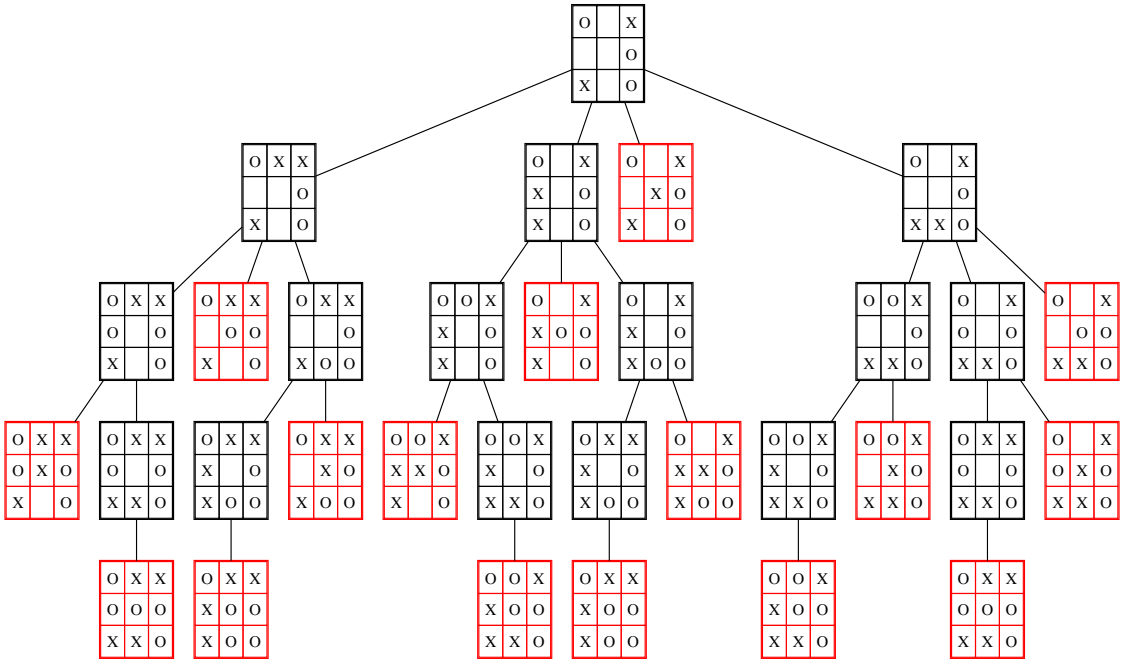
[17]:
```
%%add_to Filetto
# --- Strategia vincente per il giocatore G
def esiste_strategia_vincente(self, giocatore):
    "vedo se questa posizione ha una strategia vincente per il giocatore"
```

```
    if self.vittoria(giocatore):    # sì se ho già vinto
        return True
    altro = 'O' if giocatore == 'X' else 'X'
    if self.vittoria(altro):        # no se ha vinto l'altro giocatore
        return False
    if self.patta():                # no se siamo alla patta
        return False
    # altro modo: se non ho figli e ho vinto True else False
    pg = self.prossimo_giocatore()
    if giocatore == pg:             # se tocca a me
        for figlio in self._sons:  # e c'è almeno un figlio che è vincente␣
↪posso muovermi lì e quindi questa posizione è vincente per me
            if figlio.esiste_strategia_vincente(giocatore):
                return True
        else:
            return False            # altrimenti nessuno dei figli è vincente,␣
↪questa posizione non è vincente
    else:                           # se invece non è il giocatore a dover␣
↪giocare
        for figlio in self._sons:  # per vincere, nessuna delle scelte␣
↪dell'avversario deve essere vincente
            if figlio.esiste_strategia_vincente(altro):    # se una lo è,␣
↪questa posizione NON è vincente per me
                return False
        else:                       # se nessuno dei figli è vincente per␣
↪l'avversario
            return True             # allora io posso vincere da qui

# TODO: estrarre la sequenza di mosse vincenti
```

[18]:
```
f1 = Filetto([
    ['O', ' ', 'X'],
    [' ', ' ', 'O'],
    ['X', ' ', 'O'],
    ])
print('è posizione vincente per X?      ', f1.vittoria('X'))
print('prossimo giocatore               ', f1.prossimo_giocatore())
print('è patta?                         ', f1.patta())
f1.genera()
print("c'è una strategia vincente per X?", f1.esiste_strategia_vincente('X'))
f1.show()
```

```
è posizione vincente per X?       False
prossimo giocatore                X
è patta?                          False
c'è una strategia vincente per X? True
```

[18]:



4.8 Ottimizzazione: EVITIAMO di generare configurazioni EQUIVALENTI

- questo riduce molto l'albero di gioco

[19]:
```
class FilettoEfficiente(Filetto):
    "specializzazione di Filetto che ricorda se due nodi sono equivalenti"
    def __init__(self, board=None):
        super().__init__(board)                                    # come Filetto
        self._equivalent = None                                    # per default non␣
 ↪sono equivalente a nessuno
```

4.8.1 quando due configurazioni sono EQUIVALENTI?

- due configurazioni sono equivalenti se:
  - sono uguali per rotazioni (4 direzioni)
  - sono uguali per riflessione (4 direzioni)

[20]:
```
%%add_to FilettoEfficiente
def is_equivalent(self, other:AGraph) -> bool :
    "controllo le 4 rotazioni e 4 riflessioni"
    C1 = self._configurazione
    [[A,B,C],[D,E,F],[G,H,I]] = other._configurazione
    return ( C1 == [[A,B,C],[D,E,F],[G,H,I]]              # identica (non serve?)
             or
             C1 == [[C,F,I],[B,E,H],[A,D,G]]              # rot 90° antioraria
             or
```

```
            C1 == [[I,H,G],[F,E,D],[C,B,A]]             # rot 180°
            or
            C1 == [[G,D,A],[H,E,B],[I,F,C]]             # rot 90° oraria
            or
            C1 == [[A,D,G],[B,E,H],[C,F,I]]             # diag. princ.
            or
            C1 == [[A,D,G],[B,E,H],[C,F,I]]             # diag. second.
            or
            C1 == [[G,H,I],[D,E,F],[A,B,C]]             # sopra sotto
            or
            C1 == [[C,B,A],[F,E,D],[I,H,G]]             # sinistra destra
          )
```

4.9 Un nodo equivalente non ha figli (niente mosse valide)

[21]:
```
%%add_to FilettoEfficiente
def mosse_valide(self):
    if self._equivalent:                # mosse vuote se equivalente
        return []
    return super().mosse_valide()       # altrimenti come al solito
```

4.10 in applica_mossa se sono equivalente mi ricordo di chi

- basta che guardo i "fratelli"?
- oppure tutte le configurazioni costruite finora? (a questo livello) TODO

[22]:
```
%%add_to FilettoEfficiente
def applica_mossa(self, x, y):
    "nell'applicare una mossa controllo se un altro figlio è equivalente e lo␣
 ↪ricordo"
    G = self.genera_figlio(x,y)         # la generazione del nodo è come prima
    for son in self._sons:
        # se però è una configurazione equivalente ad una già generata
        if son.is_equivalent(G):
            G._equivalent = son         # ricordo il nodo di cui sono␣
 ↪equivalente
            break                       # mi fermo al primo equivalente che␣
 ↪trovo
    self._sons.append(G)                # e lo aggiungo ai figli per farlo␣
 ↪vedere
    return self
```

4.11 nel cercare se una configurazione è vincente, l'equivalente guarda all'altro nodo

[23]:
```
%%add_to FilettoEfficiente
# --- Strategia vincente per il giocatore G
def esiste_strategia_vincente(self, giocatore):
    if self._equivalente:
        return self._equivalente.esiste_strategia_vincente(giocatore)
    else:
        return super().esiste_strategia_vincente(giocatore)
```

4.12 Mettiamo assieme il codice (super() non lavora bene son jdc)

[24]:
```
class FilettoEfficiente(Filetto):
    "specializzazione di Filetto che ricorda se due nodi sono equivalenti"
    def __init__(self, board=None):
        super().__init__(board)                                  # come Filetto
        self._equivalent = None                                  # per default non␣
 ↪sono equivalente a nessuno

    def is_equivalent(self, other:AGraph) -> bool :
        "controllo le 4 rotazioni e 4 riflessioni"
        C1 = self._configurazione
        [[A,B,C],[D,E,F],[G,H,I]] = other._configurazione
        return ( C1 == [[A,B,C],[D,E,F],[G,H,I]]                 # identica (non␣
 ↪serve?)
                 or
                 C1 == [[C,F,I],[B,E,H],[A,D,G]]                 # rot 90°␣
 ↪antioraria
                 or
                 C1 == [[I,H,G],[F,E,D],[C,B,A]]                 # rot 180°
                 or
                 C1 == [[G,D,A],[H,E,B],[I,F,C]]                 # rot 90° oraria
                 or
                 C1 == [[A,D,G],[B,E,H],[C,F,I]]                 # diag. princ.
                 or
                 C1 == [[A,D,G],[B,E,H],[C,F,I]]                 # diag. second.
                 or
                 C1 == [[G,H,I],[D,E,F],[A,B,C]]                 # sopra sotto
                 or
                 C1 == [[C,B,A],[F,E,D],[I,H,G]]                 # sinistra destra
               )

    def mosse_valide(self):
        if self._equivalent:                    # mosse vuote se equivalente
            return []
        return super().mosse_valide()           # altrimenti come al solito
```

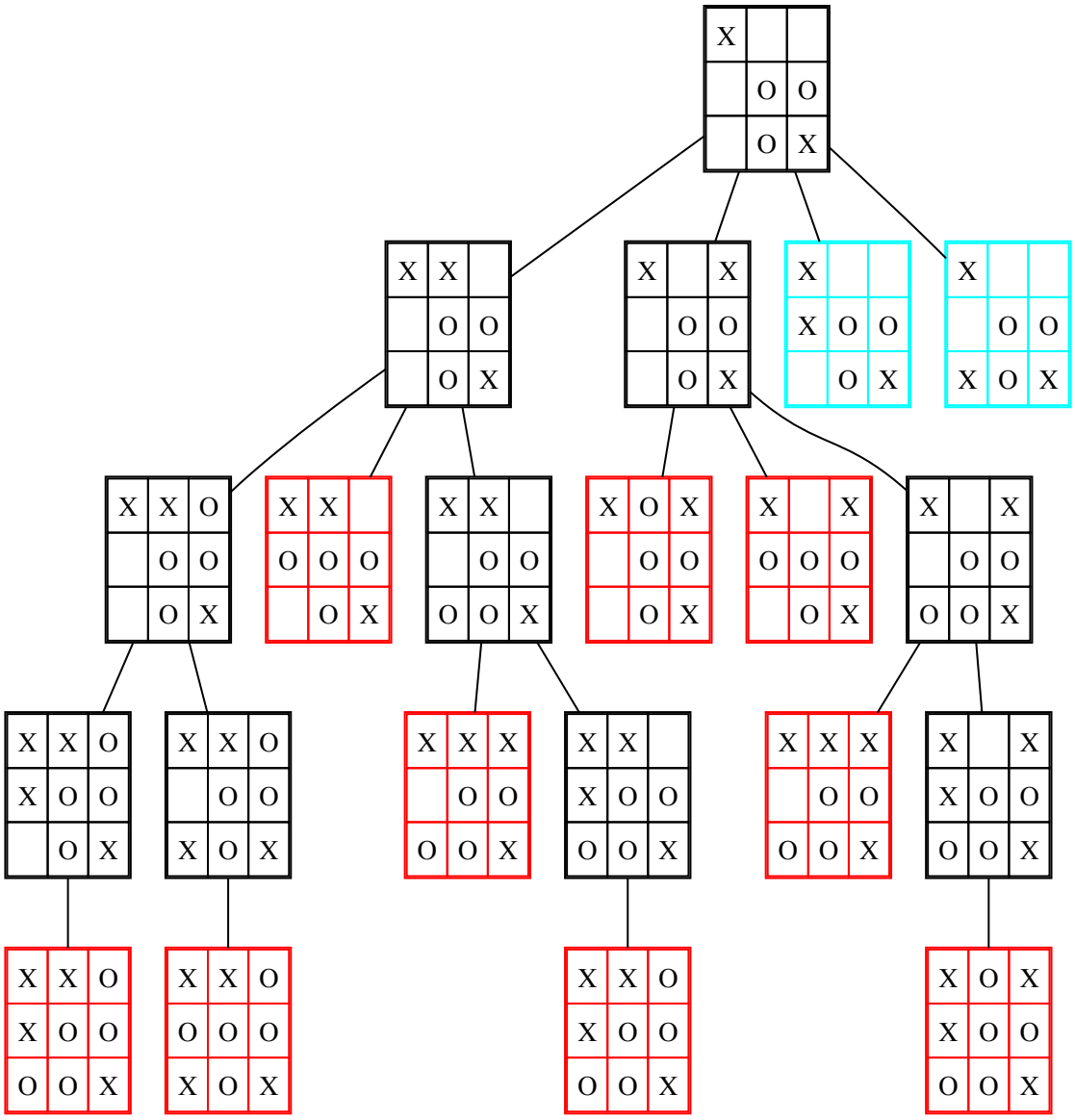
```
    def applica_mossa(self, x, y):
        "nell'applicare una mossa controllo se un altro figlio è equivalente e␣
↪lo ricordo"
        G = self.genera_figlio(x,y)          # la generazione del nodo è come␣
↪prima
        for son in self._sons:
            # se però è una configurazione equivalente ad una già generata
            if son.is_equivalent(G):
                G._equivalent = son          # ricordo il nodo di cui sono␣
↪equivalente
                G._color = 'cyan'            # e mi coloro di azzurro
                break                        # mi fermo al primo equivalente che␣
↪trovo
        self._sons.append(G)                 # e lo aggiungo ai figli per farlo␣
↪vedere
        return self

    def dot(self, G:AGraph):
        if self._equivalent: self._color = 'cyan'   # visualizzo gli␣
↪equivalenti azzurri
        return super().dot(G)

    # --- Strategia vincente per il giocatore G
    def esiste_strategia_vincente(self, giocatore):
        if self._equivalent:
            # se equivalente chiedo all'equivalente
            return self._equivalent.esiste_strategia_vincente(giocatore)
        else:
            # altrimenti esamino la situazione come al solito
            return super().esiste_strategia_vincente(giocatore)
```

4.13 Esempio di albero di gioco con equivalenze

[25]:
```
f2 = FilettoEfficiente([
    ['X', ' ', ' '],
    [' ', 'O', 'O'],
    [' ', 'O', 'X'],
    ])
print(f2.mosse_valide())
f2.genera().show()
```

```
[(1, 0), (2, 0), (0, 1), (0, 2)]
```

[25]:



4.13.1 Vediamo quali strategie sono disponibili

[26]:
```
print('prossimo giocatore                ', f2.prossimo_giocatore())
print('è patta?                          ', f2.patta())
player = f1.prossimo_giocatore()
print(f'è posizione vincente per {player}?      ', f2.vittoria(player))
player = 'O'
print(f'è posizione vincente per {player}?      ', f2.vittoria(player))
print(f"c'è una strategia vincente per {player}?", f2.
 ↪esiste_strategia_vincente(player))
player = 'X'
```

```
print(f"c'è una strategia vincente per {player}?", f2.
  ↪esiste_strategia_vincente(player))
```

```
prossimo giocatore                X
è patta?                          False
è posizione vincente per X?       False
è posizione vincente per O?       False
c'è una strategia vincente per O? True
c'è una strategia vincente per X? False
```