AI for Atari GO in Netlogo

Luca Manzi
Ingeniería Informática
ETSII
Sevilla

January 5, 2021

1 Introducción : que es Atari Go

Atari Go es una variante de Go, un antiguo juego oriental. En Go, el objetivo es de conquistar la mayor cantidad de tablero posible.

Atari Go es para 2 jugadores, la partida se realizará sobre un tablero 19×19 (13×13, 9×9) con dos colores de piedras: negra y blanca, se puede poner una piedra en cualquier casilla del tablero. Cada piedra (o grupo de piedras del mismo color) tiene que tener al menos una **liberty**, es decir, un espacio vacío adyacente en la dirección cardinal.

Los grupos sin liberties están muertos y son capturados por el otro jugador. La diferencia entre Atari Go y Go está en que el primero que capture al menos una piedra, gana el juego. De hecho Atari indica con precisión la situación en la que un grupo está en "jaque", es decir, sólo tiene una liberty.

2 Implementación

2.1 Tablero

Cada estado del juego está representado con una lista de esta forma:

[matrix player]

matrix es una extensión de Netlogo que me permite representar matrices, es mas cómodo acceder a elementos de la lista a través de dos índices. Player es el jugador, 1 o 2.

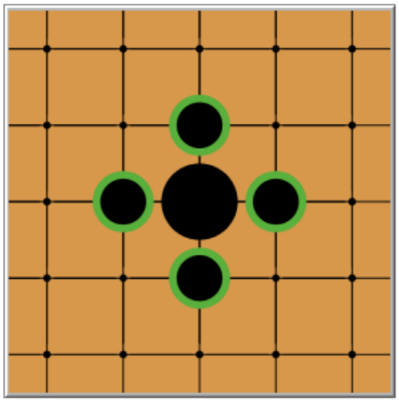

Figure 1: liberties de una pieza (obtenido con count-liberties board2state 1 2 2 y debug = 1)

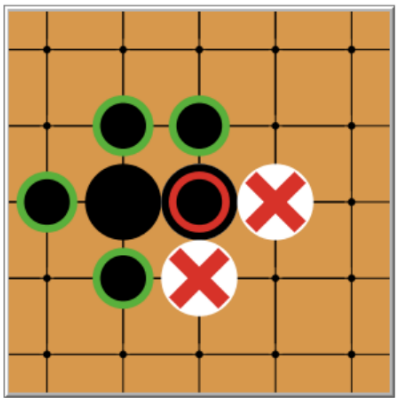

Figure 2: mismo código de la fig 1, despues de 3 jugadas

El tablero, en su lugar, se representa en el modelo de Netlogo como un cuadrado de patches y se puede cambiar de la pantalla principal. El software tomará por si mismo el lado de ese cuadrado y lo guardará en la variable global *boardsize*, entonces se puede jugar en un tablero de cualquier tamaño. Cada patch tomará una propiedad *value* entre 0,1,2 respectivamente si está vacio o con una pieza del jugador 1 o 2.

Arriba se colocarán piezas de *breed [pieces piece]*

Solo hay un pequeño cambio que hacer para pasar desde coordenadas de Netlogo y coordenadas de matriz, porque el tablero por Netlogo está representado con origen en la parte inferior izquierda, el origen podría fijarse en otro lugar. En cambio, en las matrices como estamos acostumbrados a ver su origen estaría en la parte superior izquierda, entonces he tenido que cambiar las x y para las interacciones entre los dos y mantener el WYSIWYG (what you see is what you get).

$$\begin{bmatrix} 0,0 & 0,1 & \cdots & 0,n \\ 1,0 & 1,1 & \cdots & 1,n \\ \vdots & \vdots & \ddots & \vdots \\ n,0 & n,1 & \cdots & n,n \end{bmatrix} \rightarrow \begin{bmatrix} 0,n & 1,n & \cdots & n,n \\ 1,n-1 & 1,n-1 & \cdots & n,n-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0,0 & 1,0 & \cdots & n,0 \end{bmatrix} \iff (x,y) \rightarrow ((y),(n-x))$$

2.2 Algunas funciones útiles

Hay algunas funciones que si descriptas rapidamente puedon ayudar para hecer un debug o inspeccionar el proyecto. De matrix:

- matrix:get m x y $\longrightarrow$ devuelve elemento en la posición x y de la matriz m
- matrix:set m x y val $\longrightarrow$ asigna val al elemento en la posición x y de la matriz m
- matrix:pretty-print-text m $\longrightarrow$ devuelve stringa de la matriz m en diseño de matriz

De representacion/debug:

- board2state $\longrightarrow$ construye y devuelve una matriz m desde el tablero actual
- pone [i x y] $\longrightarrow$ pone una pieza del jugador i en la posición x y del tablero (0 para eliminar)
- terminal? [s] $\longrightarrow$ devuelve true si existe una pieza/grupo con 0 liberties
- count-liberties [m x y p] $\longrightarrow$ la mas importante, devuelve el número de liberties dada: matrix m, posición x y, jugador p

El código es:

```
to-report count-liberties [m x y p]
  set tl []
  set tl lput list x y tl   ;;setting a global list tl to have just this node,
  ;then my recursive part will fill it with traversed nodes
  if debug [print "count-liberties launched for"
    ifelse (p = 1)
    [print "BLACK"]
    [print "WHITE"]
  ]
  if (matrix:get m x y = get-other-player p)[report 0]
  ;;if i am looking in an adversary occupied space, just return 0
  report reduce + (list
    count-liberties-in m (x + 1) y p
    count-liberties-in m x (y + 1) p
    count-liberties-in m (x - 1) y p
    count-liberties-in m x (y - 1) p
  );;recursion now, tl global to speed up and not double check nodes
   ;;(i wouldn try to parallelize this tho)
end
```

Y al final la parte recursiva, que trabaja con la lista tl (traversed nodes) es simple:

- si voy a un espacio vacío devuelvo 1
- si el espacio lo ocupa el otro jugador devuelvo 0
- si lo ocupa el mismo jugador lanzo recursivamente una búsqueda NSEW (NorthSouthEastWest) en el otro espacio.

```
to-report count-liberties-in [m x y p]
  let lib 0
  if debug [print "launched on "] ;; now check for good index
  if (x < boardsize and y < boardsize and x >= 0 and y >= 0)
  [if (debug)[print "position -> "
    print list x y
    print "found -> "]
  let found matrix:get m x y
  if (debug and found = 1)
    [print "BLACK"]
  if (debug and found = 2)
    [print "WHITE"]
  if (debug and found = 0)
    [print "EMPTY"]
  if debug [  print "traversed nodes -> "
      print tl]
  ]
  if (x >= boardsize or y >= boardsize or x < 0 or y < 0
          or member? (list x y) tl)[ ;; not valid
    if (debug) [
      print "not valid"
      ask patch  y ((boardsize - 1) - x) [
        sprout-temps 1[
          set color red
          set shape "o"
        ]
      ]
    ]
    report 0 ;; I can't go in that direction
  ]

  set tl lput list x y tl
  if-else (matrix:get m x y = 0)
  [if (debug)[
    print "spotted one liberty in"
    ask patch  y ((boardsize - 1) - x) [
      sprout-temps 1 [
        set color green
        set shape "o"
      ]
    ]
   ]
    if debug [
      print list x y
      print "\n"
    ]
```

```
    report 1 ;;case empty, found one lib
  ]
  [if (matrix:get m x y = get-other-player p)
    [
      if (debug)[
        print "other player stone, break"
        ask patch  y ((boardsize - 1) - x) [
          sprout-temps 1 [
            set color red
            set shape "x"
          ]
        ]
      ]
      report 0]
  ]
  ;;ofc dont do nothing, dont add liberties
  if (matrix:get m x y = p)[;;case I find the same p, recursive NSEW search
    ;print "launched recursion"
    if (debug) [
      ask patch  y ((boardsize - 1) - x) [
        ask temps-here [
          set color grey
          set shape "recurs"
        ]
      ]
    ]
    report reduce + (list
    count-liberties-in m (x + 1) y p
    count-liberties-in m x (y + 1) p
    count-liberties-in m (x - 1) y p
    count-liberties-in m x (y - 1) p
  )
  ]
end
```

2.3 Posibile heurística

También se puede escribir una función *safe*, con los mismos parámetros que devuelva True si hay más de 1 liberty. Obviamente me quedo con la misma complejidad, pero se corta antes, y puedo ahorrar un poco de tiempo.

Continuaré con esta función porque se puede intentar poner una heurística con la información más completa que devuelve *getlib*. Por ejemplo poner que la solución es mejor si se maximiza el número *mínimo* de liberties

3 Monte Carlo Tree Search

3.1 Porque MCTS

Este tipo de juego, con muchas configuraciones posibles, no solo porque el tablero es muy grande, sino también porque cada movimiento tiene muchas posibilidades. Es así que puedo colocar una pieza casi en cualquier lugar, entonces el tamaño del árbol de jugadas/estados puede ser tan grande que resulta impracticable explorarlo completamente.

Además la estratégia no es así clara a priori.

Monte Carlo Tree Search soluciona ambos problemas, no requiere un conocimiento específico del juego y (después de hacer jugadas al azar) va a explorar solo las ramas más interesantes del árbol completo, pero sí explorará de manera estadística el árbol, puede ser que no encuentre la solución óptima.

4 Funciones

Hay algunas funciones básicas para interactuar con los estados, son:

- MCTS:get-content [s] ⟶ devuelve la configuración del tablero del estado
- MCTS:get-playerJustMoved [s] ⟶ devuelve el jugador que generó el estado
- MCTS:create-state [c p] ⟶ genera un estado desde el tablero y un jugador
- MCTS:apply [r s] ⟶ aplica una regla a un estado, osea, coloca una pieza en las coordenadas dadas por la regla y cambia el jugador que generó el último, devuelve el nuevo estado

Y las dos más interesantes, que voy a analizar y estudiar, son:

- MCTS:get-rules [s] ⟶ devuelve una lista de todas las reglas aplicables al estado s
- MCTS:get-result [s p] ⟶ dado un estado final s y un jugador p, devuelve 1 si p ganó, 0 si perdió, 0.5 si empató. Para ver si el estado es final o no, Monte Carlo mira si se puede aplicar más reglas o no a partir de este estado, entonces necesito que el *get-rules* devuelva una lista vacia para un estado ganador o, por el contrario, perdido.

Estas dos funciones usan las *get-liberties*

4.1 Posibile get-rules

```
to-report get-rules-full [s] ;; matrix player ;; [[STATE]] p deducibile da state
  let op item 1 (s) ;; old player
  let p get-other-player(op)
  let m (item 0 s)
  let vm-list []  ;;valid moves
  let x 0
  let y 0 ;;setting variables to cycle all matrix
  while [x < boardsize][
    while [y < boardsize][
      if (matrix:get m x y = 0)[ ;; if empty/playable
        if (count-liberties m x y p > 0)
                    ;; if i am not suiciding myself

        or (((y + 1 < boardsize) and (matrix:get m x (y + 1) = op)
              and (count-liberties m x (y + 1) op = 1))
        or ((y - 1 >= 0) and (matrix:get m x (y - 1) = op)
              and(count-liberties m x (y - 1) op = 1))
        or ((x + 1 < boardsize) and (matrix:get m (x + 1) y = op)
              and (count-liberties m (x + 1) y op = 1))
        or ((x - 1 >= 0) and (matrix:get m (x - 1) y = op)
              and (count-liberties m (x - 1) y op = 1)))
                    ;; or if i am suiciding but capturing (so winning)
              [
          set vm-list lput (list x y) vm-list ;;list all possibilities
        ]
      ]
```

```
      set y y + 1
    ]
    set y 0
    set x (x + 1)
  ]
  report vm-list
end
```

También podría permitir que el AI juegue en cada espacio, y de esta forma solo en el siguiente estado el algoritmo se dará cuenta de que ha perdido y, por lo tanto, deja de generar hijos. Es solo una cuestión de qué representación del juego queremos elegir y avanzar de una manera lógica.

4.2 MCTS:get-rules

Por ejemplo, dado que tengo que devolver movimientos válidos y mi función prácticamente pasa una prueba de victoria, no tiene sentido que no devuelva los movimientos ganadores directamente, entonces puedo modificar la *get-rules*.

Como ya he comentado, tengo también que ver si ya estoy en un estado terminal, y llamo la función *terminal?* descripta anteriormente.

Sin embargo, mantengo una versión completa que devuelva todos los movimientos, así podría analizar el movimiento del usuario antes de cambiar el estado.

```
to-report MCTS:get-rules [s] ;; matrix player ;; [[STATE]] tomo p desde state
  let op item 1 (s) ;; old player
  let p get-other-player(op)
  let m (item 0 s)
  let vm-list []  ;;valid moves
  let wm-list []  ;;winning moves
  let x 0
  let y 0
  if (terminal? s)[report []]
  while [x < boardsize][
    while [y < boardsize][
      if (matrix:get m x y = 0)[ ;; if empty/playable
          if(((y + 1 < boardsize) and (matrix:get m x (y + 1) = op)
          and (count-liberties m x (y + 1) op = 1))
          ;;  if it captures valid and winning so reporting ASAP, check NSEW
          or ((y - 1 >= 0) and (matrix:get m x (y - 1) = op)
            and(count-liberties m x (y - 1) op = 1))
          or ((x + 1 < boardsize) and (matrix:get m (x + 1) y = op)
            and (count-liberties m (x + 1) y op = 1))
          or ((x - 1 >= 0) and (matrix:get m (x - 1) y = op)
            and (count-liberties m (x - 1) y op = 1)))[
          set wm-list lput (list x y) wm-list ;; making a list of wm, but
          ;report (list list x y) ;;I could have reported only a winning move
        ]
        if (count-liberties m x y p > 0)[ ;; check if its not a suicide
          set vm-list lput (list x y) vm-list ;;list all possibilities
        ]
      ]
      set y y + 1
    ]
    set y 0
    set x (x + 1)
  ]
```

```
  ifelse (not empty? wm-list)[report wm-list][report vm-list]
end
```

4.3 MCTS:get-result

Por último, la función *get-result [s p]* simplemente comprueba, dado un estado terminal, quién es el jugador ganador. Toma como entrada un estado y un jugador de referencia p: se p gana *get-result [s p]* = 1 , sino 0. Lo complicado es ver que el jugador que hizo el último movimiento también puede haber colocado la pieza en una zona con cero liberties y aún así haber ganado. Las victorias se guardarán (de punto de vista de *MCTC:get-playerJustMoved [s]*) en dos variables lphit y lploss (last player), y después se hará un control con el jugador p.

```
to-report MCTS:get-result [s p]
;; care! last moving player point of view has precendence!
;;if he wins it will be ok even if he has a 0-liberties area
  let m (item 0 s)
  let traversedlist []
  let tempgroup []
  let x 0
  let y 0
  let lphit false ;; last player hit
  let lploss false
  let lp MCTS:get-playerJustMoved s ; last player
  let lpa get-other-player MCTS:get-playerJustMoved s
  ; last player's adversary, just to check his liberties
  while [x < boardsize][
    while [y < boardsize][
      let check matrix:get m x y
      if (check != 0 and count-liberties m x y (check) = 0)[
        ifelse (check = lpa)[set lphit true][set lploss true]
      ]
      set y y + 1
    ]
    set y 0
    set x (x + 1)
  ]
  if (lphit)[
    ifelse (lp = p)[report 1][report 0]
  ]
  if (lploss)[
    ifelse (lp = p)[report 0][report 1]
  ]
  if (empty? get-rules-full s) [
    report 0.5
  ]
  report [false]
end
```

4.4 MCTS:UTC

El último paso es el de ejecutar un bucle, el *go*, donde el jugador puede empezar la partida (o dejar a MonteCarlo comenzar) y a continuación guardará el estado generado en el tablero, y se llama *MCTS:UTC*.

UTC empieza creando el nodo raíz que va generando hijos (a traves de otra función) con *state=s, wins=0, visits=0, unTriedRules=all applicable rules.*

Desde aquí se va tomando una regla a la vez, entre las *unTriedRules*. Se ejecutan los *apply [r s]* hasta encontrar un estado final, y se devuelve hacia atrás los resultados obtenidos.

Devuelve la regla con más posibilidad de ganar que ha encontrado en una de la MaxIterations, que es otra variable que tenemos que pasar a UTC.

5 Posibles desarrollos interesantes: GO

Para desarrollar este proyecto me ha resultado muy interesante pensar cómo podría implementar el mismo procedimiento en el juego oficial GO. Por ejemplo como almacenar los estados y como escribir las funciones en la representación.

Cada estado puede ser en la forma de

[matrix player (stonesleft, optional) captured passed laststate]

o sino para ser mas similar a lo de antes:

[matrix (stonesleft, optional) captured passed laststate] player]

Este debe ser así ya que necesito más información para calcular los puntos en los estados finales. Stonesleft *[blackstonesleft whitestonesleft]* se puede usar para almacenar cuantas piezas tenga, y una vez acabadas, el juego se tiene que parar en un estado final.

Además necesito guardar sí un jugador capturó piezas del otro (Captured es*[capturedbyblack capturedbywhite]*), para añadirlos al final al cálculo de los puntos. Y finalmente, la gestión de los KO: un estado que se va generando no podrá ser igual al estado de antes, entonces tengo que sacar de *get-rules* la regla que me genera un estado ya visitado (que no es el actual del nodo/estado, sino lo inmediatamente anterior; por ello necesito una variable matriz).

Se pueden añadir funciones útiles como por ejemplo:

- get-group [m x y] $\longrightarrow$ construye y devuelve una grupo de piezas iguales desde una matriz m
- capture[m g] $\longrightarrow$ captura y pone a 0 elementos de un grupo g en la matriz m
- perimeter-color[m group] $\longrightarrow$ devuelve (sí todo un área (vacia) tiene el perímetro del mismo color) el color del perímetro (1,2), sirve para calcular la puntuación, pero tiene complicaciones!

5.1 Perimeter-color: GO

En GO, el final de una partida se decide cuando ambos jugadores dejan pasar la jugada terminando con la pieza blanca. Entonces puedo poner que si un estado llega con el jugador que pasó, (quizas porque tenia vacia *get-moves*), sí la AI tiene más puntos que el otro, pasa y gana. Pero, la *get-result* usa la *perimeter-color* para calcular la puntuación.

Problema: sí hay un area vacia que intento a medir, y hay al menos una pieza de otro juador *perimeter-color* devuelve que non tiene todo el mismo color, osea no es un territorio de nadie. Pero existen estas *dead stones* que no se pueden defender, porque en todos los casos (si el jugador adversario no comete errores) acaban capturadas.

No es nada sencillo, pero dado el origen matemático del problema, se puede encontrar una solución en algún algoritmo: por ejemplo MINIMAX, que explora todo completamente el árbol, relacionandolo a un problema lógico para verificar una tautología. De este modo se puede ver si un grupo está muerto o no algoritmicamente.