IA Juego de la cantera

Juan Casado Ballesteros, David Menoyo Ros, Álvaro Vaya Arboleda

5/17/2019

Contents

Trabajo realizado

Se ha implementado el juego solicitado con la funcionalidad solicitada. Sobre el juego se ha creado un agente inteligente que podrá tomar decisiones a cerca de la mejor jugada a realizar en cada uno de los estados del juego. Dicho agente utilizará minmax apoyado por distintos métodos de optimización: poda alphabeta, creación de nodos en lazzy, inferir valor de los nodos hoja de forma anticipada y uso de una caché para almacenar el valor de los nodos ya calculados.

Las técnicas de optimización agilizan la toma de las decisiones de modo que esto pueda hacerse en el menor tiempo posible teniendo como resultado poder ejecutar la búsqueda sobre bloques de mayor tamaño. De toda ejecución se recogen metadatos que nos informan sobre cómo ha sido realizada.

Por último y para hacer la experiencia de juego más amena se ha creado un script de python con el que nos comunicaremos para que nos pinte los bloques en función de su tamaño.

Optimización de la búsqueda

Implementar una búsqueda mediante minmax fue sencillo. No obstante, rápidamente nos dimos cuenta de que sin realizar alguna mejora para optimizar la búsqueda nos estábamos restringiendo a solo poder realizarla sobre espacios de búsqueda muy reducidos. Iterativamente y de forma modular fuimos implementando distintos métodos para optimizar dicha búsqueda para lograr reducir el coste en tiempo de realizarla. Se podrá elegir que optimizaciones usar y cuales no para cualquier ejecución.

Poda alpahabeta

Para poderla realizar añadimos valores alpha y beta a la estructura de datos de los nodos. Mediante dichos valores pudimos predecir cuando no sería necesario evaluar más hijos de cierto nodo pues no influirían en la decisión final. Fue necesario comprobar qué orden de ejecución de los nodos favorecía más a la poda pues la capacidad de corte de esta se ve influida por la forma del árbol. Con ello mejoramos en gran medida el rendimiento aumentando el espacio de búsqueda en menos de un minuto a bloques hasta $(9, 9, 9)$ cuando la poda nos favorecía y hasta $(7, 7, 7)$ en caso de que no lo hiciera.

Creación de nodos en lazzy

Pensando en maximizar los beneficios de la poda alphabeta decidimos crear los nodos del árbol de búsqueda en lazzy. Un nodo no se crearía hasta que no fuera a ser evaluado ya fuera en la búsqueda en anchura como en la búsqueda en profundidad, inicialmente y según la implementación de minmax tradicional esto solo se hace para la búsqueda en profundidad. Esto redujo nuestro uso de memoria y aumentó la velocidad de ejecución. No obstante, no logramos aumentar el tamaño de bloque evaluable en menos de un minuto aplicando solo esta optimización.

Inferir valor de los nodos hoja de forma anticipada

Decidimos crear una función que predijera el valor de los nodos inferiores a bloques de $(3, 3, 3)$ mediante una heurística que analizara el bloque y nos dijera en función del tipo de nodo, MIN o MAX el valor para ese camino. Con esto logramos reducir la profundidad del árbol de búsqueda en 8 niveles por lo que el rendimiento aumentó considerablemente pudiendo evaluar en menos de un minuto bloques de $(6, 6, 6)$ por minmax y bloques de $(12, 12, 12)$ por alphabeta.

Caché para almacenar el valor de los nodos ya calculados

Finalmente optamos por crear una caché que nos permitiera almacenar los resultados de las ramas ya evaluadas. Dicha caché la creamos con una tabla hash que indexara los nodos para obtener el resultado de su evaluación. Gracias a ella logramos aumentar el tamaño de bloque explorable hasta $(200, 200, 200)$ en menos de un minuto para alpahabeta.

Recogida de metadatos para cada ejecución

Como método para ir comprobando que los resultados obtenidos al aplicar cada optimización mejoraban los resultados sin ella creamos un sistema para almacenar metadatos de una ejecución con el cual recogemos la siguiente información.

- **CREATED**: Cantidad de nodos del árbol que se han creado.
- **LAZZY SAVINGS**: Nodos que fueron creados pero que no se evaluaron. En caso de estar activa la optimización de crear nodos en lazzy este valor no solo indica los nodos no evaluados por haber sido podados si no los no creados, pues si no se evalúan no se llegan a crear. Valdrá 0 si no estamos en alphabeta pues minmax acaba evaluando todos los nodos del árbol.
- **EVALUATED**: Nodos evaluados.
- **CUTS**: Cortes producidos por poda alphabeta.
- **INFERED**: Nodos cuyo valor ha sido inferido. Valdrá 0 si esta optimización no se aplica.
- **CACHE HIT**: Aciertos en la caché, caminos que ya fueron evaluados previamente. Valdrá 0 si no se usa dicha optimización.
- **LEAF**: Nodos hoja del árbol de búsqueda.
- **DEPTH**: Profundidad máxima del árbol de búsqueda.
- **DIFERENT NODES**: Cantidad de valores distintos almacenados en la caché, valdrá 0 si no se usa dicha optimización.

La recogida de metadatos se implementa mediante una tabla hash.

Resultados obtenidos de la optimización

Se muestran a continuación los ejemplos de algunas ejecuciones con su salida completa ante el mismo bloque variando la configuración del agente.

```
(minmax (rootChild 4 4 4) #f #f #f)

CREATED:        24135
LAZZY SAVINGS:  0
EVALUATED:      24135
CUTS:           0
INFERED:        0
CACHE HIT:      0
LEAF:           9918
DEPTH:          15
DIFERENT NODES: 0
cpu time: 169 real time: 169 gc time: 37
'((4 4 1) -inf.0 +inf.0 0 #t ())
```

Este es el peor caso de ejecución posible, en él no se aplica ningún método de optimización de los que se dispone, no obstante mediante él se obtendrá el mejor resultado, al igual que con el resto siempre que el espacio de búsqueda no sea demasiado grande en cuyo caso será muy lento.

```
(minmax (rootChild 4 4 4) #t #f #f)

CREATED:        24135
LAZZY SAVINGS:  0
EVALUATED:      24135
CUTS:           0
INFERED:        0
CACHE HIT:      0
LEAF:           9918
DEPTH:          15
DIFERENT NODES: 0
cpu time: 155 real time: 156 gc time: 21
'((4 4 1) -inf.0 +inf.0 0 #t ())
```

Activando la creación de nodos en lazzy se siguen evaluando todos los nodos pues el método empleado es minmax, no obstante, se ve que el tiempo de ejecución se reduce debido al ahorro en uso de memoria que se produce ya que los nodos se generan antes de ser usados en la búsqueda en anchura y no cuando son descubiertos.

```
(minmax (rootChild 4 4 4) #f #t #f)

CREATED:        2781
LAZZY SAVINGS:  0
```

```
EVALUATED:      2781
CUTS:           0
INFERED:        1752
CACHE HIT:      0
LEAF:           318
DEPTH:          7
DIFERENT NODES: 0
cpu time: 30 real time: 29 gc time: 0
'((4 4 1) -inf.0 +inf.0 0 #t ())
```

Aplicando la heurística de inferir los nodos raíz a partir del tamaño $(3, 3, 3)$ vemos como la profundidad de árbol se reduce, se evalúan menos nodos y la ejecución es mucho más rápida.

```
(minmax (rootChild 4 4 4) #f #f #t)

CREATED:        516
LAZZY SAVINGS:  0
EVALUATED:      516
CUTS:           0
INFERED:        0
CACHE HIT:      381
LEAF:           18
DEPTH:          7
DIFERENT NODES: 118
cpu time: 8 real time: 8 gc time: 0
'((4 4 1) -inf.0 +inf.0 0 #t ())
```

Los mejores resultados en rendimiento se obtienen con el uso de una caché que almacena el resultado de evaluar cada nodo ya que nos ahorramos evaluar varias veces los mismos nodos.

```
(minmax (rootChild 4 4 4) #t #t #t)

CREATED:        360
LAZZY SAVINGS:  0
EVALUATED:      360
CUTS:           0
INFERED:        147
CACHE HIT:      141
LEAF:           6
DEPTH:          7
DIFERENT NODES: 67
cpu time: 7 real time: 6 gc time: 0
'((4 4 1) -inf.0 +inf.0 0 #t ())
```

Mediante la ejecución con poda a pesar de no utilizar ninguna optimación adicional obtendremos de base mejores resultados que con minmax. Se debe

mencionar que el resultado obtenido aunque equivalente al obtenido con minmax cambia de $(4, 4, 1)$ a $(1, 4, 4)$, esto se debe a que la ejecución de la poda produce resultados distintos al evaluar el árbol por un lado o por otro. Ya que en nuestro caso se obtenía mejores resultados experimentalmente al podar por la izquierda y no por la derecha alteramos el orden de evaluación de los nodos para fomentar la poda. Se ven en el apartado LAZZY SAVINGS los nodos creados que no se llegaron a evaluar.

```
(alphabeta (rootChild 4 4 4) #f #f #f)

CREATED:        424
LAZZY SAVINGS:  29
EVALUATED:      395
CUTS:           19
INFERED:        0
CACHE HIT:      0
LEAF:           166
DEPTH:          15
DIFERENT NODES: 0
cpu time: 6 real time: 10 gc time: 0
'((1 4 4) -inf.0 +inf.0 0 #t ())
```

Activando la creación de nodos en lazzy se logra con alphabeta que los nodos del apartado de LAZZY SAVINGS no solo no hayan sido evaluado si no tampoco creados. No obtente al ser el espacio de búsqueda tan pequeño no se aprecian ventajas de rendimiento.

```
(alphabeta (rootChild 4 4 4) #t #f #f)

CREATED:        424
LAZZY SAVINGS:  29
EVALUATED:      395
CUTS:           19
INFERED:        0
CACHE HIT:      0
LEAF:           166
DEPTH:          15
DIFERENT NODES: 0
cpu time: 8 real time: 8 gc time: 0
'((1 4 4) -inf.0 +inf.0 0 #t ())
```

Acortando la profundidad del árbol gracias a predecir el valor de los nodos hoja anticipadamente logramos mejores resultados todavía pues evaluamos menos nodos.

```
(alphabeta (rootChild 4 4 4) #f #t #f)

CREATED:        121
```

```
LAZZY SAVINGS:  14
EVALUATED:      107
CUTS:           4
INFERED:        63
CACHE HIT:      0
LEAF:           15
DEPTH:          7
DIFERENT NODES: 0
cpu time: 3 real time: 3 gc time: 0
'((1 4 4) -inf.0 +inf.0 0 #t ())
```

Con el uso de la caché se reducen los nodos evaluados a pesar de que los nodos creados son más que al utilizar la heurística. Dicha reducción se magnifica exponencialmente para espacio de búsqueda mayores.

```
(alphabeta (rootChild 4 4 4) #f #f #t)

CREATED:        125
LAZZY SAVINGS:  20
EVALUATED:      105
CUTS:           10
INFERED:        0
CACHE HIT:      54
LEAF:           14
DEPTH:          7
DIFERENT NODES: 38
cpu time: 1 real time: 2 gc time: 0
'((1 4 4) -inf.0 +inf.0 0 #t ())
```

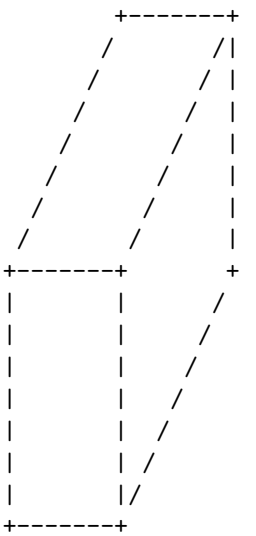
Finalmente, la mejor ejecución se logra activando todas las optimizaciones realizadas donde se reduce la cantidad de nodos evaluados de 24135 sin ninguna optimización a tan solo 80 haciendo por tanto que los espacios de búsqueda que pueda manejar el algoritmo sean mucho mayores en tiempos razonables.

```
(alphabeta (rootChild 4 4 4) #t #t #t)

CREATED:        80
LAZZY SAVINGS:  14
EVALUATED:      66
CUTS:           4
INFERED:        35
CACHE HIT:      11
LEAF:           4
DEPTH:          7
DIFERENT NODES: 17
cpu time: 3 real time: 3 gc time: 0
'((1 4 4) -inf.0 +inf.0 0 #t ())
```

Comunicar Racket con python

Para dibujar los bloques se ha creado un pequeño script de python cuya llamada debe realizarse con tres números que indicarán las dimensiones del bloque que se desea pintar. El script creará una representación ASCII-art del bloque representado por sus dimensiones y lo verterá sobre la salida estándar. Para usar dichos scripts desde racket enlazaremos mediante una tubería dinámica la salida estandar del programa racket en ejecución con la salida del script de python de modo que cuando este sea ejecutado mostrará su resultado en la misma consola que racket en ese momento lo estuviera haciendo.



```
        +0123456+
       6       66
      5       5 5
     4       4  4
    3       3   3
   2       2    2
  1       1     1
 0       0      0
+0123456+       +
6       6      6
5       5     5
4       4    4
3       3   3
2       2  2
1       1 1
0       00
+0123456+
```

Comenzar una partida

El jugador al comenzar una partida podrá elegir y podrá configurar al agente de modo que este realice la búsqueda utilizando uno de los métodos creados para que esta se haga a mayor o menor velocidad. Se elija la configuración que se elija el agente realizará una acción válida sobre el bloque de juego con la intención de ganar a su oponente. El juego se iniciará con la sentencia:

```
(play)
```

Tras la cual se le harán las siguientes cuestiones para configurar al agente:

- **"MODO DE JUEGO (ALPHABETA|MINMAX)"** ante la que se espera una respuesta de *minimax* o *alphabeta*.
- **"USO DE MEMORIA (LAZZY|COMPLETE)"** ante la que se espera una respuesta de *lazzy* o *complete*.
- **"USO DE MEMORIA (CACHE|SIMPLE)"** ante la que se espera una respuesta de *cache* o *simple*.
- **"INFERENCIA DE NODOS TERMINALES (INFERE|LEAF)"** ante la que se espera una respuesta de *infere* o *leaf*.

En caso de introducir valores erróneos en alguna de las preguntas se aplicarán los valores por defecto de *alphabeta lazzy cache* e *infere*. Posteriormente se preguntará por el estado inicial del bloque con el que se jugará:

- **"Introduce la X:"** ante la que se espera un *número natural*.
- **"Introduce la Y:"** ante la que se espera un *número natural*.
- **"Introduce la Z:"** ante la que se espera un *número natural*.

En todos los casos si se introdujera un valor erróneo este se convertiría a uno válido, es decir uno en el rango $[1, \infty]$.

Adicionalmente se podrán utilizar las sentencias:

```
(minmax (rootChild a b c) c1 c2 c3)
(alphabeta (rootChild a b c) c1 c2 c3)
```

Para ver el resultado que se obtendría si el estado del bloque fuera de dimensiones a, b, c con $a, b, c \in [1, \infty]$. $c1, c2, c3$ hacen referencia a variables booleanas $c1, c2, c3 \in [T, F]$ que configuran al ajente en su uso de memoria y en si este infiere el valor o no de los nodos. En caso de ser verdaderas T aplicarán el valor por defecto mencionado anteriormente y en caso de ser falsas aplican el otro valor correspondiente en cada caso.

Ejemplo de una partida

La partida comenzará concediendo aleatoriamente el primero turno a la máquina o al jugador.

```
(play)
MODO DE JUEGO (ALPHABETA|MINMAX)
alphabeta
USO DE MEMORIA (LAZZY|COMPLETE)
lazzy
USO DE MEMORIA (CACHE|SIMPLE)
cache
INFERENCIA DE NODOS TERMINALES (INFERE|LEAF)
infere
Introduce la X:
3
Introduce la Y:
3
Introduce la Z:
3

TURNO DE LA MAQUINA
    +---+
   /   /|
  /   / |
 /   /  |
+---+   +
|   |  /
|   | /
|   |/
+---+
x = 3
y = 3
z = 3

CREATED:         32
LAZZY SAVINGS:   7
EVALUATED:       25
CUTS:            2
INFERED:         15
CACHE HIT:       0
LEAF:            2
DEPTH:           3
DIFERENT NODES: 9
cpu time: 1 real time: 1 gc time: 0
```

```
TURNO DEL JUGADOR
    +-+
   / /|
  / / |
 / /  |
+-+   +
| |  /
| | /
| |/
+-+
x = 1
y = 3
z = 3
Introduce de dónde eliminar:
y
Introduce cuanto quitar:
1

TURNO DE LA MAQUINA
    +-+
   / /|
  / / |
 / /  +
+-+  /
| | /
| |/
+-+
x = 1
y = 2
z = 3

CREATED:         10
LAZZY SAVINGS:   0
EVALUATED:       10
CUTS:            0
INFERED:         0
CACHE HIT:       1
LEAF:            4
DEPTH:           3
DIFERENT NODES: 6
cpu time: 1 real time: 1 gc time: 0

TURNO DEL JUGADOR
   +-+
  / /|
 / / |
```

```
+-+  +
| | /
| |/
+-+
x = 1
y = 2
z = 2
Introduce de dónde eliminar:
z
Introduce cuanto quitar:
1

TURNO DE LA MAQUINA
  +-+
 / /|
+-+ |
| | +
| |/
+-+
x = 1
y = 2
z = 1

CREATED:        1
LAZZY SAVINGS:  0
EVALUATED:      1
CUTS:           0
INFERED:        0
CACHE HIT:      0
LEAF:           1
DEPTH:          1
DIFERENT NODES: 1
cpu time: 0 real time: 1 gc time: 0

TURNO DEL JUGADOR
  +-+
 / /|
+-+ +
| |/
+-+
x = 1
y = 1
z = 1

HA GANADO LA MAQUINA
```

Conclusión

Trabajar en el campo de la IA involucra tres aspectos clave.

- Crear nuevas formas de poder manejar e inferir nueva información a partir del conocimiento existente.
- Crear algoritmos que permitan materializar y realizar esas manipulaciones de datos.
- Optimizar los algoritmos de modo que se puedan aplicar en tiempo razonable o para que se puedan adaptar de forma rápida a cambios online sobre los datos de entrada.

A lo largo de esta práctica se ha hecho un estudio detallado del último apartado partiendo de ideas preexistentes para los dos primeros. Se han implementado formas de mejorar y perfeccionar el rendimiento en tiempo del algoritmo minmax para buscar la mejor jugada válida a realizar a partir de un bloque en el marco del juego definido.

Dichas optimizaciones nos han permitido poder buscar mejores jugadas para nodos de tamaño $(200, 200, 200)$ cuando inicialmente tras crear la primera versión del algoritmo era complicado llegar a evaluar nodos de tamaño $(5, 5, 5)$.

Código

Racket

Algoritmo de búsqueda minmax, bucle de juego y estructuras de datos.

```
;-----------------CUBE-----------------

#lang racket
(provide (all-defined-out))

(define python-path
  (let ((os (system-type 'os)))
    (cond
      [(equal? os 'windows) "U:\\Python\\Python IDE\\python.exe"]
      [else "/usr/bin/python"]
    )
  )
)

(define (cube x y z name)
  (define (print-cube x y z)
    (display "x = ")(display x)
    (display "\ny = ")(display y)
    (display "\nz = ")(display z)(display "\n")
```

```
  )
        (if (system* python-path name
                  (number->string x) (number->string y) (number->string z))
                  (print-cube x y z) (print "no python found"))
)

(define (ncube x y z) (cube x y z "ncube.py"))
(define (lcube x y z) (cube x y z "lcube.py"))

;-----------------OBJECT-----------------

;POSICIONES DE LOS ELEMENTOS
(define iId       1)
(define iAlpha    2)
(define iBeta     3)
(define iWeight   4)
(define iType     5)
(define iBest     6)
(define lenNodo   6)
;Obtener un elemento de una lista en una posicion dada
(define (get list index)
  (cond
    [(empty? list) "Las has liado chaval en get"]
    [(= 1 index) (car list)]
    [else  (get (cdr list) (- index 1))]
  )
)
;Poner un elemento en una lista en una posicion dada
;QUITANDO lo que hubiera en esa posicion
(define (set list elem index)
  (cond
    [(empty? list) elem]
    [(= 1 index) (cons elem (cdr list))]
    [else (cons (car list) (set (cdr list) elem (- index 1)))]
  )
)
;Quitar un elemento en una lista en una posicion dada
(define (remove list index)
  (cond
    [(empty? list) null]
    [(= 1 index) (cdr list)]
    [else (cons (car list) (remove (cdr list) (- index 1)))]
  )
)
;Poner un elemento en una lista en una posicion dada
;SIN QUITAR lo que hubiera en esa posicion
```

```
(define (put list elem index)
  (cond
    [(empty? list) elem]
    [(= 1 index) (cons elem list)]
    [else (cons (car list) (put (cdr list) elem (- index 1)))]
  )
)
;PUTO O SET SEGUN CORRESPONDA
(define (establish list elem index)
  (if (< (length list) lenNodo)
      (put list elem index)
      (set list elem index)
  )
)

;Devuelve la id de un nodo
(define (getId nodo) (get nodo iId))
(define (setID nodo ID)
  (establish nodo ID iId)
)

(define (getX nodo)(car(getId nodo)))
(define (getY nodo)(cadr(getId nodo)))
(define (getZ nodo)(caddr(getId nodo)))

(define (getNX nodo)(- (getX nodo) 1))
(define (getNY nodo)(- (getY nodo) 1))
(define (getNZ nodo)(- (getZ nodo) 1))

;Devuelve el alpha de un nodo
(define (getAlpha nodo) (get nodo iAlpha))

;Establece el alpha de un nodo
(define (setAlpha nodo alpha)
  (establish nodo alpha iAlpha)
)

;Devuelve el beta de un nodo
(define (getBeta nodo) (get nodo iBeta))

;Establece el beta de un nodo
(define (setBeta nodo beta)
  (establish nodo beta iBeta)
)

;Devuelve el peso de un nodo
```

```
(define (getW nodo) (get nodo iWeight))

;Establece el peso de un nodo
(define (setW nodo w)
  (establish nodo w iWeight)
)

;Devuelve el mejor hijo de un nodo
(define (getBest nodo) (get nodo iBest))

;Establece el mejor hijo de un nodo
(define (setBest nodo best)
  (establish nodo best iBest)
)

;Devuelve el tipo (MAX o MIN) de un nodo
(define (getType nodo) (get nodo iType))

;Hace que el id de un nodo sea un valor mayor o igual a 1
(define (validate-number number) (if (< number 1) 1 number))

;Generador de hijos
(define (newChild x y z alpha beta tipo)
  (list (list (validate-number x)  (validate-number y)  (validate-number z))
   alpha beta 0 tipo '())
)
(define (rootChild x y z)
  (list (list (validate-number x) (validate-number y) (validate-number z))
   -inf.0 +inf.0 0 #t '())
)
(define (bestChild id)
  (list id -inf.0 +inf.0 0 #t '())
)
(define (winnerChild id)
  (list id -inf.0 +inf.0 1 #t id)
)
(define (looserChild id)
  (list id -inf.0 +inf.0 -1 #t id)
)

;Pinta un nodo
(define (draw nodo)
  (lcube (getX nodo) (getY nodo) (getZ nodo))
)

;-----------------METADATA-----------------
```

```
(define created "CREATED")
(define evaluated "EVALUATED")
(define infered "INFERED")
(define leaf "LEAF")
(define cache "CACHE")
(define cut "CUT")
(define expanded "EXPANDED")
(define depth "DEPTH")

(define metadata (make-hash))
(define (newMetadata)
  (hash-set! metadata created 0)
  (hash-set! metadata evaluated 0)
  (hash-set! metadata infered 0)
  (hash-set! metadata leaf 0)
  (hash-set! metadata cache 0)
  (hash-set! metadata cut 0)
  (hash-set! metadata expanded 0)
  (hash-set! metadata depth 0)
)

(define (adder name value)
  (hash-set! metadata name (+(hash-ref metadata name) value))
)

(define (addCreated value)
  (adder created value)
)

(define (addEvaluated value)
  (adder evaluated value)
)

(define (addInfered value)
  (adder infered value)
)

(define (addLeaf value)
  (adder leaf value)
)

(define (addHit value)
  (adder cache value)
)
```

```
(define (addCut value)
  (adder cut value)
)

(define (addExpanded value)
  (adder expanded value)
)

(define (setDepth value)
  (let* [
        (current (hash-ref metadata depth))
        ]
    (if (> value current)
        (adder depth value)
        ""
  ))
)

(define (printMetadata)
  (display "\nCREATED:        ")
  (display (hash-ref metadata created))
  (display "\nLAZZY SAVINGS:  ")
  (display (- (hash-ref metadata created) (hash-ref metadata expanded)))
  (display "\nEVALUATED:      ")
  (display (hash-ref metadata evaluated))
  (display "\nCUTS:           ")
  (display (hash-ref metadata cut))
  (display "\nINFERED:        ")
  (display (hash-ref metadata infered))
  (display "\nCACHE HIT:      ")
  (display (hash-ref metadata cache))
  (display "\nLEAF:           ")
  (display (hash-ref metadata leaf))
  (display "\nDEPTH:          ")
  (display (hash-ref metadata depth))
)

;-----------------MINMAX-----------------

;Devuelve cuantos hijos se pueden generar desde un nodo
(define (getChildCount nodo)
  (+ (getNX nodo) (getNY nodo) (getNZ nodo))
)

;Genera el siguiente hijo dependiendo de la cantidad de hijos ya generados segun MINIMAX
(define (nextChild nodo index)
```

```
  (let* [
          (limity (+ (getX nodo) (getY nodo)))
          (x (if (< index (getX nodo)) index (getX nodo)))
          (y (if (and (>= index (getX nodo)) (< (+ index 1) limity)) (- (+ index 1)
           (getX nodo)) (getY nodo)))
          (z (if (>= (+ index 1) limity) (- (+ index 2) limity) (getZ nodo)))
          (alpha (getAlpha nodo))
          (beta (getBeta nodo))
        ]
    (newChild x y z alpha beta (not (getType nodo)))
  )
)
;Genera todos los hijos de un nodo
(define (createChildren nodo)
  (define (_createChilds nodo index)
    (if (< index 1) '()
        (cons (nextChild nodo index) (_createChilds nodo (- index 1)))
    )
  )
  (reverse(_createChilds nodo (getChildCount nodo)))
)

;Devuelve si un hijo es o no un nodo hoja
(define (isLeaf nodo)
  (> 1 (getChildCount nodo))
)

;Actualiza un nodo hijo, pone en su valor quien gana
(define (update-leaf nodo)
  (define (value nodo) (if (getType nodo) -1 1))
  (addLeaf 1)
  (setBest (setW nodo (value nodo)) (getId nodo))
)

;Actualiza un nodo, su alpha o beta según corresponda, su peso y cambia su id
;por la de su mejor hijo
(define (update root-node child)
  (define (update-max root-node child);TOMAR MAYOR PESO
    (let* [
          (action (or(> (getAlpha root-node) (getW child))
                     (and(= (getAlpha root-node) (getW child)) (< (getChildCount child)
                     (getChildCount (bestChild (getBest root-node)))))))
          (val  (if action (getAlpha root-node) (getW child)))
          (best (if action (getBest root-node) (getId child)))
          (other (getBeta child))
          ]
```

```
      (setBest (setBeta (setAlpha (setW root-node val) val) other) best)
  ))
  (define (update-min root-node child);TOMAR MENOR PESO
    (let* [
          (action (or(< (getBeta root-node) (getW child))
                      (and(= (getBeta root-node) (getW child)) (< (getChildCount child)
                      (getChildCount (bestChild (getBest root-node)))))))
          (val  (if action (getBeta root-node) (getW child)))
          (best (if action (getBest root-node) (getId child)))
          (other (getAlpha child))
          ]
      (setBest (setAlpha (setBeta (setW root-node val) val) other) best)
  ))
  (addEvaluated 1)
  (if (getType root-node)
        (update-max root-node child)
        (update-min root-node child)
  )
)

;INFIERE EL VALOR DE UN NODO INFERIOR A (3 3 3)
(define (infere-leaf node)
  ;COUNT 1
  (define (count1 node)
    (+ (if (= 1 (getX node)) 1 0) (if (= 1 (getY node)) 1 0) (if (= 1 (getZ node)) 1 0))
  )
  ;SUMA PAR
  (define (sumEven nodo)
    (even? (getChildCount nodo))
  )
  ;SUMA IMPAR
  (define (sumOdd nodo)
    (odd? (getChildCount nodo))
  )
  (let* [
        (count (count1 node))
        (result (cond
                  [(and (= 2 count) (getType node)) #t]
                  [(and (= 1 count) (getType node) (sumOdd node)) #t]
                  [(and (= 1 count) (not(getType node)) (sumEven node)) #t]
                  [else #f]
                  ))
        (finalNode (if result (winnerChild (getId node)) (looserChild (getId node))))
        ]
    (addInfered 1)
    finalNode
```

```
))

(define ht (make-hash))
;MINIMAX
(define (minmax nodo lazzy infer cache)
  (define (cuter root-node) #f)
  (newMetadata)
  (hash-clear! ht)
  (if lazzy
      (optimize nodo cuter infer cache)
      (optimize-not-lazzy nodo cuter infer cache)
))
;ALPHABETA
(define (alphabeta nodo lazzy infer cache)
  (define (cuter root-node) (> (getAlpha root-node) (getBeta root-node)))
  (newMetadata)
  (hash-clear! ht)
  (if lazzy
      (optimize nodo cuter infer cache)
      (optimize-not-lazzy nodo cuter infer cache)
))
(define (inferable node)
  (and (<= (getX node) 3) (<= (getY node) 3) (<= (getZ node) 3))
)

;OPTIMIZADOR CON LAZZY
(define (optimize nodo check infer cache)
  (define (profundidad node depth)
    (setDepth depth)
    (if (isLeaf node)
        (update-leaf node)
        (if (and infer (inferable node) (> depth 2))
            (infere-leaf node)
            (if cache
                (if (hash-has-key? ht node)
                    (let [] (addHit 1) (hash-ref ht node))
                    (let [(result(anchura node node 1 depth))]
                      (hash-set! ht node result)
                      (addCreated (getChildCount node))
                      result
                ))
            (let [] (addCreated (getChildCount node)) (anchura node node 1 depth))
      ))
  ))
  (define  (anchura original root-node expansion depth)
    (if (> expansion (getChildCount original)) root-node
```

```
        (if (check root-node)
            (let [] (addCut 1) root-node)
            (let [] (addExpanded 1) (anchura original
            (update root-node (profundidad (nextChild original expansion)
            (+ 1 depth))) (+ expansion 1) depth))
    )))
  (time
    (let [(best (bestChild (getBest (profundidad nodo 0))))] (printMetadata)
    (display "\nDIFERENT NODES: ") (display (hash-count ht)) (display "\n") best)
))

;OPTIMIZADOR SIN LAZZY
(define (optimize-not-lazzy nodo check infer cache)
  (define (profundidad node depth)
    (setDepth depth)
    (if (isLeaf node)
        (update-leaf node)
        (if (and infer (inferable node) (> depth 2))
            (infere-leaf node)
            (if cache
                (if (hash-has-key? ht node)
                    (let [] (addHit 1) (hash-ref ht node))
                    (let [(result(anchura node (createChildren node) depth))]
                      (addCreated (getChildCount node)) (hash-set! ht node result)
                      result
                ))
            (let [] (addCreated (getChildCount node))
            (anchura node (createChildren node) depth))
      ))
  ))
  (define  (anchura root-node child-list depth)
    (if (null? child-list) root-node
        (if (check root-node)
            (let [] (addCut 1) root-node)
            (let [] (addExpanded 1)  (anchura (update root-node (profundidad
            (car child-list) (+ 1 depth))) (cdr child-list) depth))
    )))
  (time
    (let [(best (bestChild (getBest (profundidad nodo 0))))] (printMetadata)
    (display "\nDIFERENT NODES: ") (display (hash-count ht)) (display "\n") best)
))

;-----------------MAIN-----------------

(define (ask question)
      (display question)
```

```
      (display "\n")
      (string->number(read-line (current-input-port)))
)
(define (ask-str question)
      (display question)
      (display "\n")
      (string-downcase(read-line (current-input-port)))
)

(define (remove node where ammount)
  (let [(formated-ammount (validate-number ammount))]
  (cond
    [(string=? where "x") (if (<= (getX node) 1) (remove node "y" ammount)
    (rootChild (-(getX node) formated-ammount) (getY node) (getZ node)))]
    [(string=? where "y") (if (<= (getY node) 1) (remove node "z" ammount)
    (rootChild (getX node) (-(getY node) formated-ammount) (getZ node)))]
    [(string=? where "z") (if (<= (getZ node) 1) (remove node "x" ammount)
    (rootChild (getX node) (getY node) (-(getZ node) formated-ammount)))]
    [else  (rootChild (getX node) (getY node) (getZ node))]
  ))
)

(define (play)
  (define (ganador turno)
    (if turno (display "\nHA GANADO EL JUGADOR\n") (display "\nHA GANADO LA MAQUINA\n"))
  )
  (define (bucle-juego min-max lazzy cache infer nodo turno)
    (if turno (display "\nTURNO DE LA MAQUINA\n") (display "\nTURNO DEL JUGADOR\n"))
    (draw nodo)
    (if (isLeaf nodo)
        (ganador turno)
        (if turno
            (if min-max
                (bucle-juego min-max lazzy cache infer (minmax nodo lazzy infer cache)
                (not turno))
                (bucle-juego min-max lazzy cache infer (alphabeta nodo lazzy infer cache)
                (not turno))
            )
            (let [(user (ask-str "Introduce de dónde eliminar: "))]
              (if (string=? user "auto")
                  (if min-max
                      (bucle-juego min-max lazzy cache infer
                      (minmax nodo lazzy infer cache) (not turno))
                      (bucle-juego min-max lazzy cache infer
                      (alphabeta nodo lazzy infer cache) (not turno))
                  )
```

```
                  (bucle-juego min-max lazzy cache infer (remove nodo user
                  (ask "Introduce cuanto quitar: ")) (not turno))
            ))
  )))
  (bucle-juego (string=? (ask-str "MODO DE JUEGO (ALPHABETA|MINMAX)") "minimax")
                (string=? (ask-str "USO DE MEMORIA (LAZZY|COMPLETE)") "lazzy")
                (not(string=? (ask-str "USO DE MEMORIA (CACHE|SIMPLE)") "simple"))
                (string=? (ask-str "INFERENCIA DE NODOS TERMINALES (INFERE|LEAF)") "infere")
                (rootChild
                (ask "Introduce la X: ")
                (ask "Introduce la Y: ")
                (ask "Introduce la Z: "))
                (>(random) 0.5))
)
```

Python

Pintar los bloques.

```
#-----------------LCUBE.py-----------------

def cuboid(x,y,z):
    t = {(n,m):' ' for n in range(3+x+z) for m in range(3+y+z)}
    xrow = ['+'] + ['-'  for i in range(x)] + ['+']
    for i,ch in enumerate(xrow):
        t[(i,0)] = t[(i,1+y)] = t[(1+z+i,2+y+z)] = ch
    ycol = ['+'] + ['|' for j in range(y)] + ['+']
    for j,ch in enumerate(ycol):
        t[(0,j)] = t[(x+1,j)] = t[(2+x+z,1+z+j)] = ch
    zdepth = ['+'] + ['/' for k in range(z)] + ['+']
    for k,ch in enumerate(zdepth):
        t[(k,1+y+k)] = t[(1+x+k,1+y+k)] = t[(1+x+k,k)] = ch
    return '\n'.join(''.join(t[(n,m)] for n in range(3+x+z)).rstrip()
                         for m in reversed(range(3+y+z)))


if __name__ == '__main__':
    import sys
    print(cuboid(int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3])))

#-----------------NCUBE.py-----------------

def cuboid(x,y,z):
    t = {(n,m):' ' for n in range(3+x+z) for m in range(3+y+z)}
```

```
    xrow = ['+'] + ['%i' % (i % 10) for i in range(x)] + ['+']
    for i,ch in enumerate(xrow):
        t[(i,0)] = t[(i,1+y)] = t[(1+z+i,2+y+z)] = ch
    ycol = ['+'] + ['%i' % (j % 10) for j in range(y)] + ['+']
    for j,ch in enumerate(ycol):
        t[(0,j)] = t[(x+1,j)] = t[(2+x+z,1+z+j)] = ch
    zdepth = ['+'] + ['%i' % (k % 10) for k in range(z)] + ['+']
    for k,ch in enumerate(zdepth):
        t[(k,1+y+k)] = t[(1+x+k,1+y+k)] = t[(1+x+k,k)] = ch
    return '\n'.join(''.join(t[(n,m)] for n in range(3+x+z)).rstrip()
                     for m in reversed(range(3+y+z)))

if __name__ == '__main__':
    import sys
    print(cuboid(int(sys.argv[1]), int(sys.argv[2]), int(sys.argv[3])))
```