Un juego de tablero

El pasado mes de noviembre de 2019, con el proyecto de “Asteroids” y el libro “Programación Retro del Commodore 64 Volumen II” ya terminados, adelantaba cuáles podrían ser mis siguientes pasos: un juego de tablero.

La verdad es que me ha costado encontrar la inspiración. Nos ha tocado vivir un tiempo raro (meses de marzo y abril de 2020) y he estado descentrado. Finalmente, superada esa primera fase de desconcierto, creo que he encontrado la fuerza para sacar adelante el proyecto.

Os iré contando durante los próximos días y semanas. De momento, lo más importante: salud para todos.

Colossus Chess

Durante los 80, mis aficiones principales eran tres:

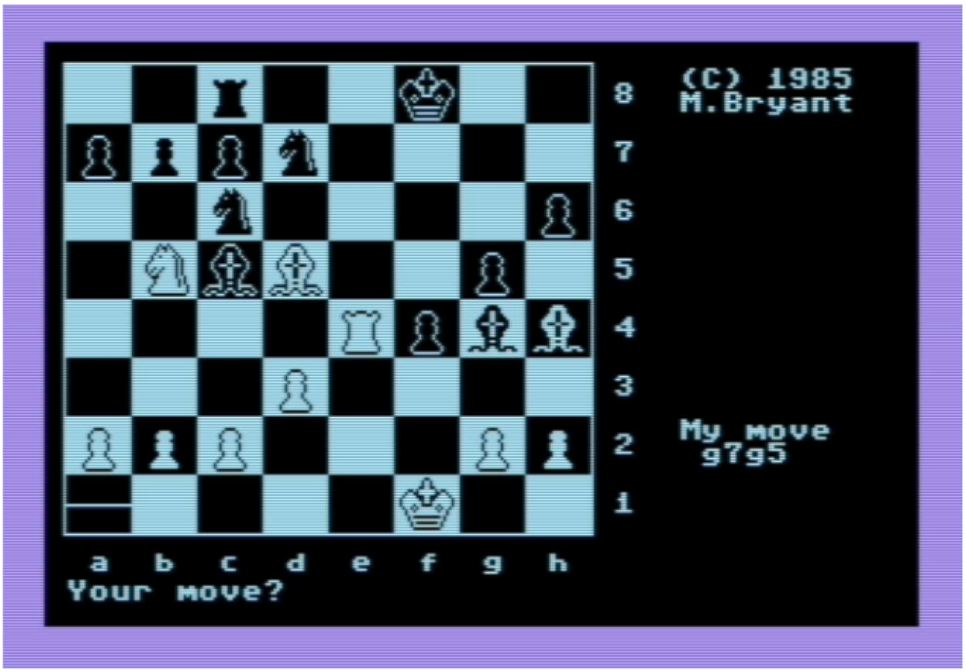
- Mi Commodore 64.
- El baloncesto.
- El ajedrez.

Por eso me gustaban juegos como “One on One” o “International Basketball”. Y relativos al ajedrez, yo creo que estaremos todos de acuerdo en que el mejor juego, de largo, era “Colossus Chess”.

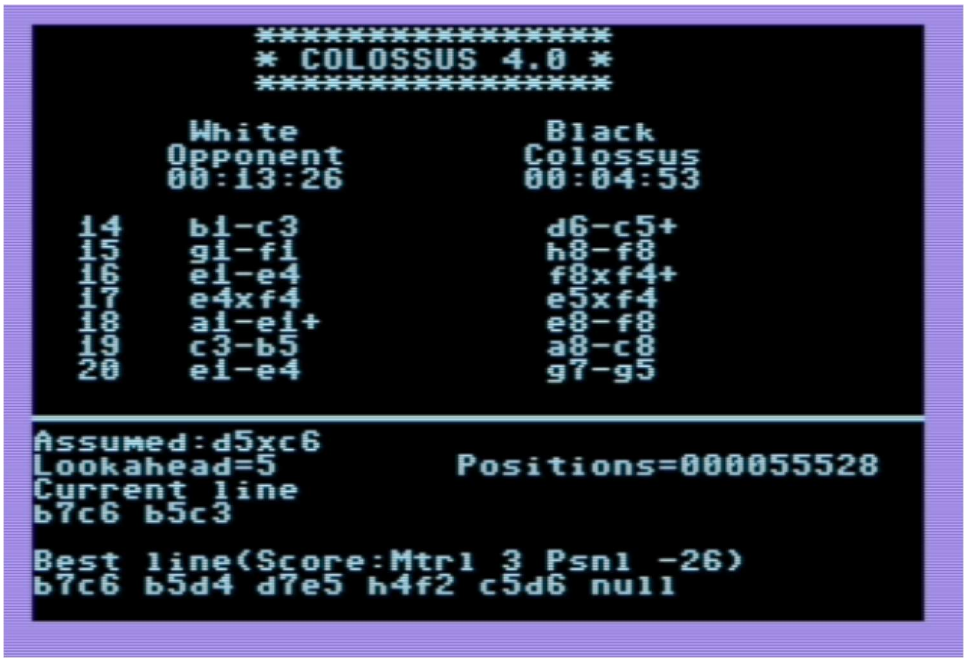
Yo entonces no lo sabía, pero por lo que leo ahora, “Colossus Chess” fue una serie de juegos de ajedrez ganadores de muchos premios durante los 80. Lo programó Martin Bryant y se portó a casi todos los equipos de 8 bits de la época.

Muchas veces jugué contra “Colossus Chess”. Ya no recuerdo si conseguí ganarle en alguna ocasión, pero sí recuerdo haberlo puesto contra las cuerdas más de una vez. Al final, casi siempre cometía algún fallo garrafal, se me revolvía, y me daba un buen repaso.

Aquí os dejo un par de pantallas de una partida reciente. Este es el tablero de juego (tenía una vista 2D y otra 3D):



Y este es el registro de la partida y lo que está pensando la máquina:



El juego tenía muchísimas funcionalidades, entre las que destacan:

- Tablero de juego con vistas 2D y 3D. Se pueden cambiar los colores. Se puede cambiar la orientación del tablero.
- Registro de jugadas de la partida en notación algebraica, incluyendo los tiempos de juego de cada jugador.
- Información sobre el análisis de jugadas que está haciendo Colossus (*lookahead*: profundidad de análisis, *positions*: número de tableros analizados, *best line*: evaluación que hace de esos tableros, etc.).
- Evaluación de tableros atendiendo a criterios materiales (número y tipos de piezas), de posición (posición de las piezas en el tablero), y otros.
- El juego no sólo “piensa” durante su turno, sino que también analiza jugadas mientras piensa el contrincante humano. Para ello, hace una hipótesis sobre la jugada que jugará el humano (*assumed*), y analiza jugadas a partir de esa hipótesis.

- Se permite alterar la posición de las piezas, para así llevar el tablero a una situación X desde la que se quiera continuar una partida.
- Se permite deshacer movimientos, por ejemplo, para rectificar errores; también se permite repetir movimientos.
- Guardar partidas a cinta o disco, y cargarlas de nuevo.
- Cambiar de bando. Pedir a Colossus que haga una jugada por ti.
- Que Colossus juegue contra sí mismo.
- Aplicar una base de datos de aperturas y finales de partida.
- Repetir una partida, en el sentido de reproducir todos los movimientos del registro de jugadas, desde el primero hasta el último.
- Modificar los parámetros de configuración del juego (si Colossus usa o no la base de datos de aperturas / finales, si Colossus "piensa" o no durante el turno del contrincante, profundidad de análisis del árbol de jugadas, etc.).
- Etc.

En definitiva, un juego completísimo e imbatido cuando se midió contra programas similares en otros equipos de 8 bits (Apple II, Spectrum, Atari, etc.).

Por supuesto, nuestro objetivo no va a ser destripar cómo estaba hecho Colossus Chess. Y tampoco intentar hacer un programa similar. Pero sí conocer los principios básicos de diseño de esto tipo de juegos, y esbozar en ensamblador un juego de tablero, aunque sea mucho más modesto.

¿Interesante? ¡¡A mí me parece que sí!!

Principios básicos de diseño

Juegos de tablero hay muchos:

- Ajedrez.
- Go.
- Damas.
- Othello, también llamado Reversi.
- Backgamon.
- Etc.

Todos estos juegos se juegan sobre tableros de 8 x 8 casillas o tableros similares. De todos ellos, tradicionalmente se ha considerado que el ajedrez es el rey, aunque el go se ha puesto muy de moda últimamente desde que Google sacara AlphaGo en 2016. IBM ya había sacado Deep Blue veinte años antes, en 1996.

Pues bien, todos estos juegos se pueden programar de forma similar. Los principios de diseño de estos programas los leí hace años en un libro titulado "Aventuras Informáticas", escrito por Alexander K. Dewdney en 1990.

En ese libro hay un capítulo titulado "¿Un programa que juega a las damas sin perder jamás?" donde, precisamente, se describen los criterios de diseño. Me encantaría enlazar aquí una copia del capítulo, así que he escrito a su autor, que afortunadamente todavía vive (nació en 1941), y le he solicitado permiso para añadirlo aquí.

Mientras tanto, tendréis que conformaros con mi resumen:

Todos los programas que desarrollan juegos de tablero, ya sea ajedrez, damas u otros juegos similares, tienen cuatro elementos principales y uno opcional:

- Un árbol de juego.
- Un generador de jugadas.
- Una función de evaluación del tablero.
- Un procedimiento minimax.
- Un procedimiento de poda (opcional).

Por supuesto, a estos elementos básicos se pueden añadir otros, como bases de datos de aperturas, bases de datos de finales, bases de datos de partidas, etc.

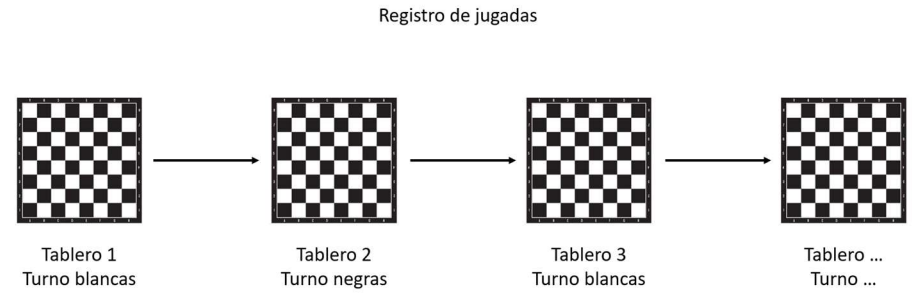
En las entradas que siguen iremos describiendo estos elementos.

El árbol de juego

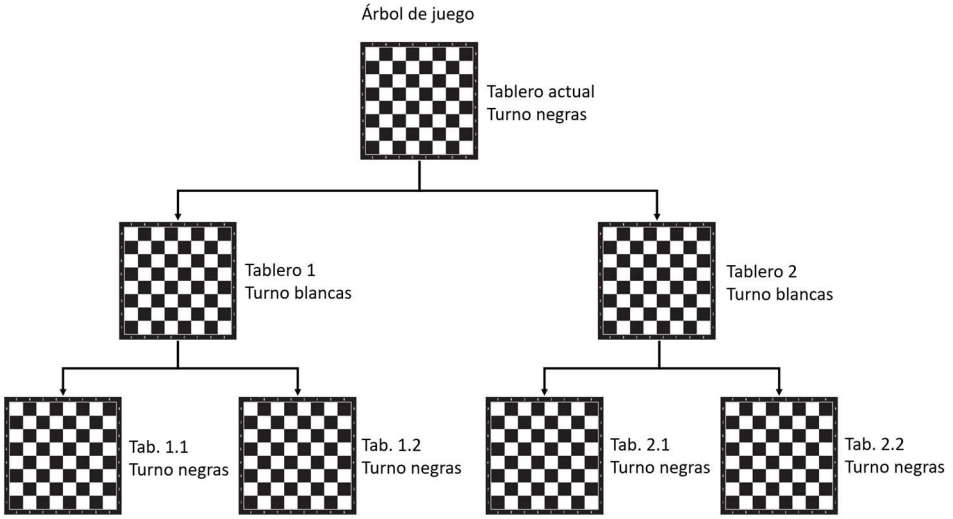
El árbol de juego es una estructura de datos. Es un árbol cuyos nodos son tableros.

Es importante tener claro que el árbol de juego no es la secuencia de tableros por los que discurre la partida (el registro de jugadas de Colossus Chess, para entendernos). Esa secuencia de tableros sólo se necesita si se quieren ofrecer

funcionalidades como rectificar una jugada, grabar y cargar partidas, etc. Si no se quieren ofrecer esas funcionalidades con llevar cuenta del tablero actual es suficiente. Además, para esa secuencia de tableros no haría falta un árbol; llegaría con una lista enlazada.



El árbol de juego es un árbol de posibles futuras jugadas, tanto de un bando como del otro, y hasta una profundidad N, que se utiliza para decidir qué jugada hará el ordenador.



Cada vez que le toque mover al ordenador, se generará un árbol de este tipo partiendo del tablero actual, y se decidirá qué jugada hacer. Luego le tocará

mover al humano, para lo cual bastará con pedirle al usuario qué jugada quiere efectuar. Y luego se repetirá el proceso. Por tanto, aunque hablemos de "el árbol de juego" no habrá uno único. Habrá muchos, uno por cada jugada del ordenador.

El árbol de juego es el nexo de unión entre el generador de jugadas, la función de evaluación del tablero, y el procedimiento minimax. El árbol de juego es una estructura de datos; los otros tres son algoritmos o programas que generan y procesan el árbol de juego.

El generador de jugadas

El generador de jugadas es el programa que desarrolla el árbol. Partiendo del tablero actual (raíz del árbol), va generando jugadas de las diferentes piezas, y va dando lugar a nuevos tableros hijo. Y así sucesivamente, alternando bandos, hasta una profundidad N (profundidad de análisis), que normalmente es configurable.

El árbol tiene que tener una profundidad limitada porque, obviamente, ni la memoria del ordenador es ilimitada, ni el juego puede tardar un tiempo excesivo en decidir la siguiente jugada.

Si el ordenador tuviera tanta memoria como para almacenar el árbol de juego completo, y tanta capacidad de cálculo como para generarlo y tomar una decisión en un tiempo razonable, entonces estos programas serían invencibles o llevarían siempre a tablas (si el juego tuviera algún tipo de limitación o característica intrínseca que lo abocara a acabar en tablas si ambos jugadores jugaran de manera perfecta). Desde luego no es el caso del C64, que tiene 64 KB de memoria y 1 MHz de frecuencia de reloj.

Los tableros

Para que el generador de jugadas pueda generar tableros, lo primero que necesitamos es una forma de representar los tableros y las piezas. Y la forma más sencilla es mediante una matriz de M x M posiciones donde cada posición almacena (cualquier otra codificación sería válida):

- 0 si la casilla está vacía.

- 1, 2, 3, …, para las diferentes piezas de un bando.
- -1, -2, -3, …, para las diferentes piezas del otro bando.

Por ejemplo, para el juego de las damas de 8 x 8 (1 = peón blanco; -1 = peón negro):

```
-1, 0,-1, 0,-1, 0,-1, 0
 0,-1, 0,-1, 0,-1, 0,-1
-1, 0,-1, 0,-1, 0,-1, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 1, 0, 1, 0, 1, 0, 1
 1, 0, 1, 0, 1, 0, 1, 0
 0, 1, 0, 1, 0, 1, 0, 1
```

De este modo, conociendo el tablero de partida, el bando que juega, y las reglas del juego (los movimientos permitidos), es posible generar todos los tableros hijo a los que daría lugar y vincularlos con su padre. Y así sucesivamente alternando los bandos.

La matriz es la solución obvia y la más natural. Pero no es la única ni la más "compacta" (la que menos memoria consume). De hecho, si observáis la matriz anterior comprobaréis que el 62% de las casillas están vacías (0).

¿Seguro que no se puede "comprimir" esa información? Por supuesto que se puede, pero si lo hacemos se nos complicará la generación de jugadas o, más probablemente, tendremos que andar convirtiendo entre tableros comprimidos (el formato a utilizar para almacenar en memoria) y tableros sin comprimir (el formato a utilizar para generar jugadas y evaluar). Es decir, que ahorraremos memoria, pero gastaremos más computación. Así es la vida…

La función de evaluación

La función de evaluación es una función (una rutina en el caso del C64) que, dado un tablero, determina mediante un número cómo de favorable es la situación para un bando y para el otro.

Para ese número se puede usar el rango y la codificación que se quiera, pero, típicamente, si las blancas se codifican en la matriz con números positivos y las negras con números negativos, entonces una evaluación positiva será favorable a las blancas y una evaluación negativa será favorable a las negras. Y cuanto más positiva o más negativa más favorable será para blancas o negras.

En realidad, los valores absolutos que se asignen a los tableros son lo de menos. Lo importante es que esos valores permitan comparar tableros y tomar decisiones correctas. Es decir, si un tablero recibe el valor 7 y otro el valor 13, el segundo tablero debería ser para las blancas notablemente mejor que el primero. Pero podríamos llegar a la misma conclusión con los valores 14 y 26.

Una buena función de evaluación debe contemplar muchos criterios. Como poco debería tener en cuenta:

- Aspectos materiales: qué piezas hay sobre el tablero y de qué tipos son.
- Aspectos posicionales: qué posiciones ocupan esas piezas.

Los aspectos materiales sólo son importantes en los juegos en que se puede “comer” o capturar, porque en los otros siempre hay las mismas piezas.

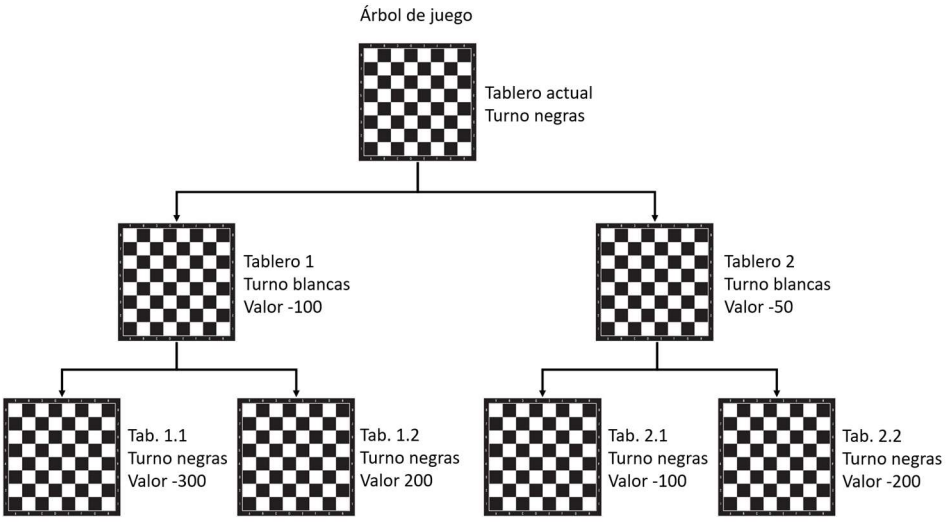
En el fondo, la función de evaluación es el complemento necesario de un árbol de juego limitado. Si el árbol de juego pudiera ser completo, entonces llegaría con que la función de evaluación se limitara a detectar que, para el tablero X, ganan las blancas o las negras. Pero como el árbol de juego no puede ser completo, es necesario tener una función que mida cómo de bueno o malo es un tablero, cómo de prometedor es para cada bando llegar hasta ahí.

En mi opinión, una buena función de evaluación es la pieza clave que distingue un buen programa de otro que no lo es, porque todo lo demás es más o menos rutinario. Todos los programas de ajedrez o damas tendrán que ser capaces de generar jugadas conforme a las reglas del juego, o no serán correctos. Y todos los juegos aplicarán el procedimiento minimax, porque no es más que pura lógica. Es asumir que cada bando tomará la jugada que más le beneficia.

El procedimiento minimax

Ya tenemos una función de evaluación que permite asignar un valor a cada tablero. Ahora podemos aplicar esa función de forma aislada a cada tablero o, mejor todavía, relacionando unos tableros con otros.

Me explico. Supongamos que le toca mover al ordenador (negras) y, partiendo del tablero actual, generamos un árbol de juego de profundidad dos:



Supongamos, también, que el tablero 1 tiene muy buena valoración para las negras (-100), mejor que el tablero 2 (-50). ¿Sería correcto que el ordenador optara por el tablero 1?

Pues depende, en general no. Dependerá de los tableros que siguen. Si los tableros 1.1 y 1.2, o al menos uno de ellos, son mejores para las blancas (peores para las negras) que los tableros 2.1 y 2.2, entonces el ordenador no estaría decidiendo correctamente. Debería optar por el tablero 2.

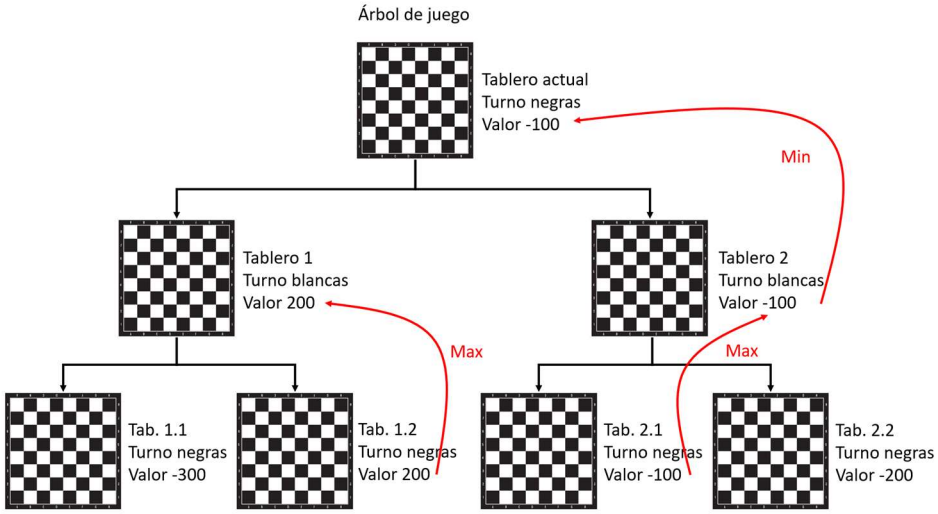
Y este razonamiento hay que aplicarlo en cascada hasta las hojas del árbol de juego. Más allá de ese punto no se puede llevar, porque el árbol de juego tiene una profundidad limitada.

Y esto es, precisamente, el procedimiento minimax. Es pura lógica. Es asumir que, igual que tú quieres la mejor jugada para ti, tu contrincante querrá la mejor jugada para sí (salvo que esté jugando al despiste).

Por tanto, dado que a las blancas les benefician las valoraciones positivas y a las negras les benefician las valoraciones negativas, las primeras intentarán maximizar el valor en su turno, y las segundas intentarán minimizarlo en el suyo. De ahí el nombre del procedimiento: minimax, de mínimo y máximo.

En el fondo esto se traduce en que la función de evaluación sólo hay que aplicarla sobre las hojas del árbol, que son los destinos a los que podemos llegar, y, a partir de ahí, hay que hacer subir los valores hasta la raíz en función del turno. Si el turno es de las blancas se tomará el máximo valor de los hijos; si el turno es de las negras se tomará el mínimo valor de los hijos.

En nuestro ejemplo:



Este proceso no sólo nos da una valoración de los tableros poniéndolos en relación entre sí, incluida la raíz, sino que además nos dice por qué jugada o tablero debe optar el ordenador, en nuestro ejemplo por el tablero 2.

El procedimiento de poda

Aunque el árbol de juego tenga una profundidad limitada, con unos pocos niveles (cuatro o cinco), podemos estar hablando de millones de tableros. Téngase en cuenta que el crecimiento en el número de tableros es exponencial con la profundidad. Cada tablero dará lugar a tantos hijos como el número de jugadas posibles, y así sucesivamente en cada nivel. Por tanto:

Número de tableros = (número de jugadas posibles) ^ profundidad

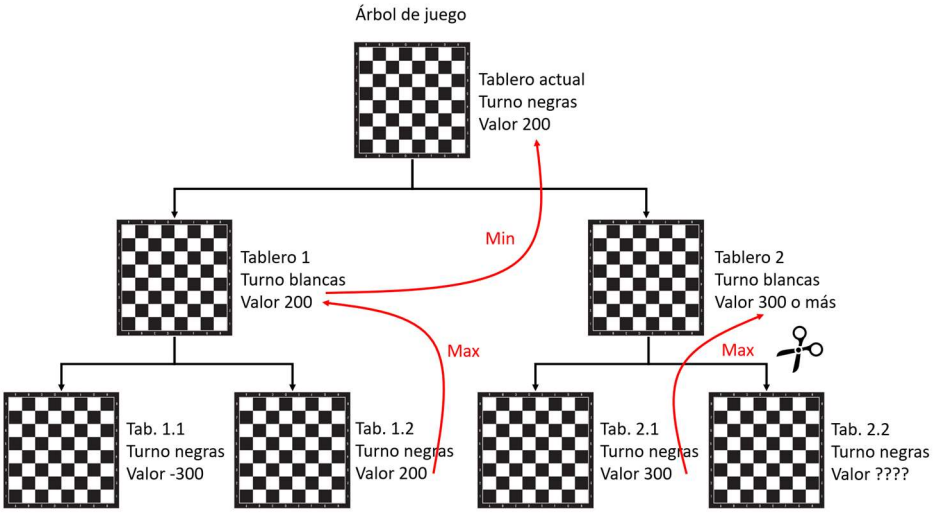
El número de jugadas posibles depende del juego y sus reglas. Por ejemplo, en el caso del ajedrez, si nos centramos en el tablero de inicio, hay veinte jugadas posibles (8 peones x 2 movimientos + 2 caballos x 2 movimientos = 20). No todos los tableros tendrán el mismo número de jugadas posibles, pero de forma estimativa, si asumimos ese valor y cinco niveles, tenemos:

Número de tableros = 20 x 20 x 20 x 20 x 20 = 20 ^ 5 = 3,2 millones

En este contexto, interesa podar todas las ramas del árbol que se pueda, aquellas que no tenga sentido analizar. No es obligatorio podar el árbol, pero sí una opción muy interesante.

Para poder podar el árbol hay que evaluarlo según se va construyendo. Si el programa se diseña de tal modo que primero se construye el árbol completo y luego se evalúa, no tiene tanto sentido podarlo, ya que ya se habrá incurrido en el coste de construirlo.

Supongamos que tenemos una situación como ésta:



Es decir, la rama 1 del árbol ya ha sido desarrollada y evaluada, y ha arrojado un valor de 200. Si al empezar a desarrollar la rama 2 surge un tablero con un valor de 300, dado que es el turno de las blancas, elegirán ese tablero u otro con valor superior. Por tanto, esa rama 2 ya nunca va a interesar a las negras, y puede dejar de construirse y evaluarse.

A mi entender, son situaciones un poco difíciles de detectar, y sólo compensa intentarlo cuando la profundidad de análisis es alta, porque es cuando más ahorro puede suponer.

Elección de un juego de tablero

Ya tenemos las bases del diseño:

- Matrices para codificar los tableros.
- El tablero actual para llevar control de dónde está la partida.
- Un generador de jugadas (o tableros).
- El árbol de juego para decidir las jugadas del C64.
- Una función de evaluación.
- Un procedimiento minimax.

No implementaremos un registro de jugadas, aunque no sería complejo, porque no vamos a dar funcionalidades como rectificar jugadas ni grabar / cargar partidas. Y tampoco implementaremos un procedimiento de poda.

De todo lo anterior, las partes más complejas son:

- El generador de jugadas.
- La función de evaluación.

El generador de jugadas es tanto más complejo como complejo sea el juego. En el caso del ajedrez, que hay muchos tipos de piezas, muchos movimientos posibles, y muchas reglas, el generador de jugadas será necesariamente complejo. En el caso de las damas, que hay menos tipos de piezas, menos movimientos posibles, y menos reglas, el generador de jugadas será más sencillo. Y en el caso de Othello / Reversi, cuyo movimiento principal consiste en colocar una ficha sobre una casilla vacía y flanqueando al contrincante, el generador de jugadas será sencillo.

La función de evaluación también es compleja, pero no tanto porque sea difícil de programar, sino porque es difícil definir criterios que midan bien si los tableros son buenos o malos para los bandos. Los criterios materiales son más claros (tantos puntos por pieza de tal tipo); pero los criterios posicionales son difíciles de identificar y definir. Y son tanto más difíciles como complejo sea el juego.

En resumen, un juego como el ajedrez por supuesto es abordable, incluso con un C64, y con mucho éxito. A los hechos me remito (véase Colossus Chess). Pero su complejidad hace que esté fuera del alcance este blog. Un juego como las damas o el Othello / Reversi sí son abordables al nivel amateur.

En todo caso, como el objetivo es ejemplificar todos estos conceptos, y además en ensamblador del C64 (ya de por sí complejo), vamos a optar por un juego de tablero sencillo: el ratón y los gatos.

En la siguiente entrada describiremos el juego.

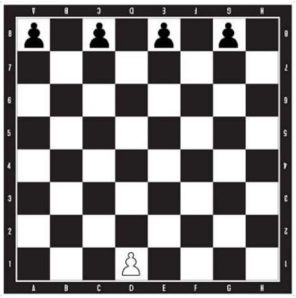
El ratón y los gatos (RYG)

El ratón y los gatos es un juego de tablero sencillo. Se juega con un tablero clásico de 8 x 8 casillas. Hay dos bandos:

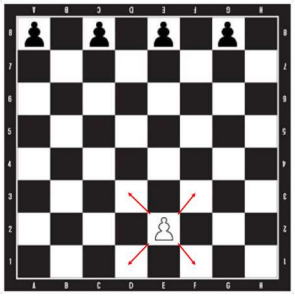
- Los gatos, que consta de cuatro peones negros.
- El ratón, que consta de un único peón blanco.

Normalmente los gatos se ponen en la parte superior del tablero y se mueven hacia abajo, y el ratón se coloca en la parte inferior y se mueve hacia arriba, aunque lógicamente esto depende de cómo queremos orientar el tablero. Ambos tipos de piezas, ratón y gatos, se colocan sobre casillas del mismo color, supongamos que sobre casillas blancas.

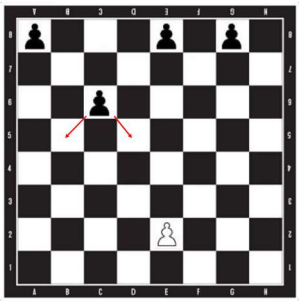
La situación inicial de la partida es como sigue:



Empieza moviendo el ratón, que puede moverse en diagonal hacia adelante o hacia atrás, siempre una casilla:



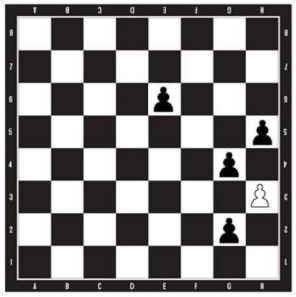
Los gatos pueden moverse uno por turno, y nuevamente en diagonal y una casilla, pero sólo hacia delante (es decir, hacia abajo). No pueden retroceder:



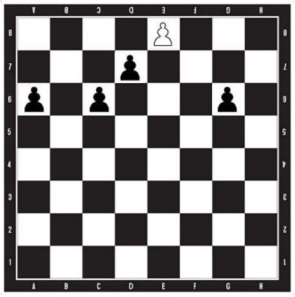
No hay capturas, es decir, los gatos no pueden comerse al ratón, ni mucho menos el ratón comerse a los gatos ☺ .

El objetivo del juego es que los gatos rodeen al ratón, y lo dejen sin posibilidad de moverse, o que el ratón llegue hasta la fila superior del tablero. En el primer caso ganan los gatos; en el segundo gana el ratón:

Ganan gatos



Gana ratón



Así de simple. Pero el reto es programarlo en ensamblador del C64…

Como los gatos tienen pinta de malos, dejaremos que el C64 juegue por los gatos. El humano será el ratón e intentará esquivarlos.

RYG: estructura de datos del tablero

Lo primero que hace falta para empezar a trabajar es una estructura de datos que permita almacenar un tablero, ya que los tableros van a ser nuestra materia prima.

Como hemos comentado anteriormente, lo más básico es tener una matriz de 8 x 8, pero hacen falta más cosas:

- El nivel: la profundidad del tablero en el árbol de juego.
- El turno: si le toca mover al ratón o a los gatos.
- El valor: el resultado de la función de evaluación o del procedimiento minimax, según sea una hoja o un tablero intermedio.
- El padre: el tablero del que procede este tablero.
- Los hijos: los tableros que proceden de este tablero.
- La matriz: el tablero propiamente dicho.

De este modo, podemos diseñar una estructura de datos así (ver fichero "Tableros.asm"):

```
; Un tablero básicamente es una estructura de datos en una posición de memoria

; Constantes

Vacio   = $00
Raton   = $01
Gato    = $ff

; Estructura de datos (85 bytes)

;nivel   byte $00
;turno   byte $00
;valor   byte $00

;padreLo byte $00
;padreHi byte $00

;hijosLo byte $00,$00,$00,$00,$00,$00,$00,$00
;hijosHi byte $00,$00,$00,$00,$00,$00,$00,$00

;tablero byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00
```

En total son 85 bytes que se reparten así:

- Nivel: Es el nivel de profundidad que ocupa el tablero en el árbol de juego. Con un byte será suficiente.

- Turno: Es el turno, es decir, si le toca mover a ratón ($01) o gatos (-1 = $ff). Nuevamente, con un byte será suficiente.
- Valor: Es el valor que la función de evaluación o el procedimiento minimax asignan al tablero. Será suficiente con un byte.
- Padre: El tablero en curso procederá de un tablero padre. Ese tablero padre será otra estructura de 85 bytes. Con tener la dirección de memoria de su primer byte (su comienzo) será suficiente. Para esa dirección necesitamos dos bytes, el byte "lo" y el byte "hi".
- Hijos: El tablero en curso podrá dar lugar, como mucho, a ocho tableros hijo (un ratón puede moverse a un máximo de cuatro casillas, y cada gato a un máximo de dos). Por tanto, necesitaremos ocho direcciones de memoria para llevar control de toda la progenie. Como en el caso del padre, cada dirección tendrá su byte "lo" y su byte "hi".
- Tablero: Es una matriz de 8 x 8 para especificar la posición de las piezas sobre el tablero. Equivale a una secuencia de 8 x 8 = 64 bytes. Codificaremos con $00 las casillas vacías, con $01 el ratón, y con -1 = $ff los gatos.

En definitiva, un tablero será una estructura de datos de 85 bytes que vendrá especificada por la dirección de memoria de su primer byte (el nivel). Conociendo la dirección de ese primer byte, y utilizando el modo de direccionamiento indirecto – indexado, es posible acceder a cualquiera de los campos del tablero:

Tableros en la memoria del C64

En particular, usando el puntero al padre y los punteros a los hijos es posible navegar por el árbol de juego, tanto hacia arriba (hacia la raíz) como hacia abajo (hacia las hojas).

Cuando una dirección no esté definida (ej. el padre o algún hijo), usaremos la dirección $0000 para indicarlo.

Más adelante meteremos una pequeña variación en esta estructura de datos porque no es muy práctico que ocupe 85 bytes. Sería mucho más práctico que ocupara 88 bytes porque, al ser un múltiplo de 8 (88 = 11 x 8), se facilita mucho la visualización de los tableros en el depurador de CBM prg Studio durante la fase de depuración del programa. Para esto, llega con meter un relleno o prefijo de tres bytes que, además, tiene la ventaja de marcar la frontera en memoria entre tableros si se elige un patrón fácilmente reconocible de forma visual.

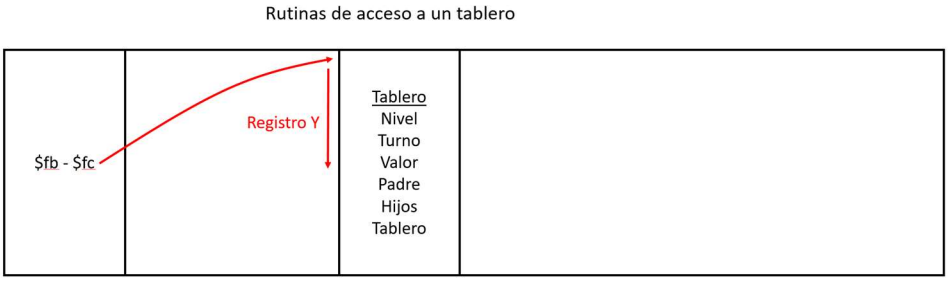
RYG: rutinas para manejar tableros

Ahora que tenemos definida la pieza clave del proyecto, que es la estructura de datos para manejar tableros, resultado muy conveniente definir una serie de rutinas para manejar tableros (ver fichero “Tableros.asm”). De este modo, se pueden definir rutinas para:

- Inicializar un tablero.
- Fijar y recuperar sus datos básicos (nivel, turno y valor).
- Fijar y recuperar su padre.
- Fijar y recuperar sus hijos.
- Fijar y recuperar la pieza de una posición (fila, columna).
- Etc.

Esta librería de rutinas la iremos completando y perfeccionando según avance el proyecto y según vayan surgiendo nuevas necesidades.

Todas estas rutinas, en general, se apoyarán en el modo de direccionamiento indirecto – indexado. Configuraremos el puntero de página cero $fb – $fc apuntando al primer byte del tablero (nivel) y, a partir de ahí, usando el registro Y como índice, iremos al campo de nuestro interés para leer (instrucción “lda ($fb),y”) o escribir (instrucción “sta ($fb),y”):



Como sabemos, cuando un programa llama a una rutina, no hay garantía de que la rutina mantenga inalterados los registros del microprocesador (acumulador, registro X y registro Y). Pero como en este proyecto vamos a usar mucho los registros X e Y para tareas como recorrer los hijos de un tablero o los campos de una estructura de datos, nos interesa que las rutinas sí conserven estos registros. Por ello, aquellas rutinas que vayan a modificar X y/o Y tendrán un campo TempX y/o TempY donde resguardar los valores de los registros al empezar y recuperarlos justo antes de terminar.

Y como todas las rutinas enumeradas anteriormente son parecidas, nos limitaremos a explicar una rutina de cada tipo:

Rutina “fijaContenido”:

El código de la rutina es así:

```
418 ; Rutina para fijar el contenido de una posición (fila,columna)
419
420 fcTableroLo     byte $00
421 fcTableroHi     byte $00
422 fcFila          byte $00
423 fcColumna       byte $00
424 fcFicha         byte $00
425
426 fcTempY         byte $00
427
428 fijaContenido
429
430         sty fcTempY
431
432         lda fcFila
433         sta doFila
434
435         lda fcColumna
436         sta doColumna
437
438         jsr dameOffset
439
440         lda fcTableroLo
441         sta $fb
442
443         lda fcTableroHi
444         sta $fc
445
446         ldy doOffset
447
448         lda fcFicha
449         sta ($fb),y
450
451         ldy fcTempY
452
453         rts
454
```

La rutina recibe como parámetros de entrada:

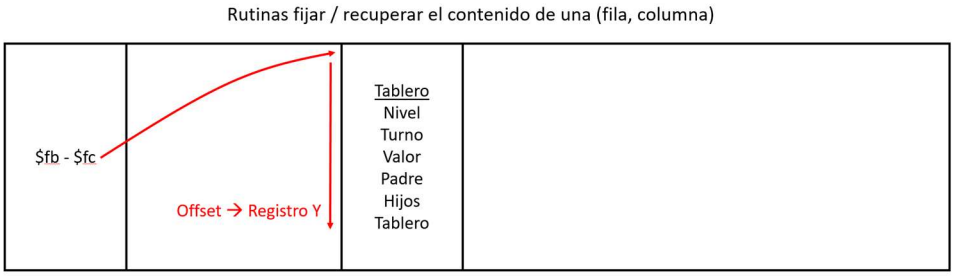
- Un puntero al tablero, con sus bytes “lo” y “hi”.
- La fila y columna cuyo contenido se quiere modificar.
- El contenido o ficha a poner en (fila, columna), es decir, $00 para vacío, $01 para ratón, y -1 = $ff para gato.

Obsérvese que también hay un byte “fcTempY”. Esto no es un parámetro de entrada ni de salida. Simplemente es un campo en el que vamos a guardar temporalmente el valor del registro Y (“sty fcTempY”) para recuperarlo luego justo antes de terminar (“ldy fcTempY”). De este modo, aunque la rutina modifique el valor del registro Y, y lo modifica para usar el modo de direccionamiento indirecto – indexado (“sta ($fb),y”), la rutina o programa llamante no se verá afectado. El programa llamante puede confiar en que no le cambiarán el valor de Y.

Esquemáticamente, lo que hace la rutina es:

- Resguarda el valor del registro Y, porque lo va a modificar.
- Convierte las coordenadas (fila, columna) en un offset contado desde el comienzo del tablero. Este offset es el número de bytes que hay que contar desde comienzo del tablero (byte de nivel) para llegar al byte de la (fila, columna) que hay que modificar. Se obtiene con la rutina complementaria “dameOffset”.
- Prepara el puntero $fb - $fc para que apunte al tablero.
- Carga el offset en el registro Y.
- Modifica el contenido de la posición (fila, columna) con “sta ($fb),y”.
- Recupera el valor del registro Y.

Fácil, ¿no? Básicamente es usar el modo de direccionamiento indirecto – indexado. Se prepara un puntero al comienzo del tablero (indirección), se calcula el offset de la posición (fila, columna) respecto al comienzo del tablero, y se modifica la posición en ese offset (indexación).



Cuando lo que fija la rutina es el nivel, el turno, el valor, etc., no hace falta calcular ningún offset. La rutina correspondiente ya “conoce” por diseño de la estructura de datos cuál es el offset de esos campos. Por ejemplo, el nivel tiene offset 0, el turno tiene el offset 1, y el valor tiene el offset 2.

El cálculo del offset sólo es necesario cuando se quiere fijar o recuperar el contenido de una posición (fila, columna), es decir, cuando ya entramos en la matriz de datos.

Rutina dameOffset:

La matriz o tablero propiamente dicho empieza en el offset 21 de la estructura de datos. Ese byte correspondiente a la posición (0, 0) del tablero.

A partir del offset 21, cada fila añade otros 8 bytes de offset. Es decir: 21, 29, 37, 45, 53, 61, 69 y 77. Creamos una tabla con estos datos y accedemos a ella usando como índice el registro X (previo cargar en él el número de fila, claro).

Eso nos da el offset del comienzo de cada fila. Luego basta sumar la columna. Para sumar no hay que olvidarse de borrar antes el acarreo con "clc".

Fácil, ya tenemos la rutina "dameOffset":

```
492 ; Rutina para convertir (fila,columna) a offset
493
494 doFila          byte $00
495 doColumna       byte $00
496 doOffset        byte $00
497
498 doTempX         byte $00
499
500 tblOffsetFilas  byte 21, 29, 37, 45, 53, 61, 69, 77
501
502 dameOffset
503
504         stx doTempX
505
506         lda doColumna
507
508         ldx doFila
509
510         clc
511         adc tblOffsetFilas,x
512
513         sta doOffset
514
515         ldx doTempX
516
517         rts
```

Ya hemos comentado otras veces que, en ensamblador, es mucho más fácil resolver las fórmulas matemáticas mediante tablas de datos que mediante operaciones aritméticas (offset = 21 + 8 * fila + columna).

Rutina "dameContenido":

Esta rutina es completamente análoga a la rutina "fijaContenido", sólo que en vez de escribir datos con "sta ($fb),y" los lee con "lda ($fb),y". No necesita mucha más explicación.

```
455 ; Rutina para recuperar el contenido de una posición (fila,columna)
456
457 dcTableroLo     byte $00
458 dcTableroHi     byte $00
459 dcFila          byte $00
460 dcColumna       byte $00
461 dcFicha         byte $00
462
463 dcTempY         byte $00
464
465 dameContenido
466
467         sty dcTempY
468
469         lda dcFila
470         sta doFila
471
472         lda dcColumna
473         sta doColumna
474
475         jsr dameOffset
476
477         lda dcTableroLo
478         sta $fb
479
480         lda dcTableroHi
481         sta $fc
482
483         ldy doOffset
484
485         lda ($fb),y
486         sta dcFicha
487
488         ldy dcTempY
489
490         rts
```

Y el resto de rutinas del fichero “Tableros.asm”, al menos las rutinas de esta primera versión del proyecto, son todas muy parecidas. Todas utilizan el modo de direccionamiento indirecto – indexado.

La librería de rutinas se irá mejorando con otras nuevas con el avance del proyecto. Y las iremos comentando.

RYG: pintado de tableros

Tanto para dotar al programa de una interfaz de usuario como para poder hacer tareas de depuración, se hace necesario pintar tableros por pantalla.

De momento optaremos por un pintado sencillo, en modo texto. Más adelante se podrá mejorar el aspecto con caracteres personalizados o, incluso, con sprites.

El pintado de tableros no es difícil. Básicamente consiste en utilizar algunas rutinas de la librería “LibText” que desarrollamos durante los proyectos del volumen I y el volumen II. Cuando se trata de pintar números o direcciones usamos “pintaHex” y cuando se trata de pintar cadenas de texto usamos “pintaCadena”.

Aunque no es difícil, sí es farragoso pintar un tablero entero. Por ello, la nueva rutina “pintaTablero”, que es larga y tiene subrutinas, la implementamos en un nuevo fichero “PintaTableros.asm”.

La rutina “pintaTablero” no tiene mucho misterio, así que sólo daré un par de pinceladas sobre su implementación. Básicamente se trata de dividir el trabajo en varias subrutinas:

```
; Rutina para pintar un tablero

ptTableroLo     byte $00
ptTableroHi     byte $00

pintaTablero

        ; Separador
        jsr pintaSeparador

        ; Datos básicos
        jsr pintaDatosBasicos

        ; Padre
        jsr pintaPadre

        ; Hijos
        jsr pintaHijos

        ; Línea en blanco
        lda #13
        jsr chrout

        ; Línea 0...7
        jsr pintaLinea07

        ; Tablero
        jsr pintaTabla

        rts
```

Cada una de estas subrutinas (“pintaSeparador”, “pintaDatosBasicos”, etc.) se encarga de una función más pequeña y más específica.

Quizás el mayor misterio esté en la subrutina “pintaTabla”, que es la que pinta la matriz de datos del tablero. Básicamente es un bucle que pinta la matriz fila por fila:

```
312 ; Rutina para pintar el tablero propiamente dicho
313
314 pintaTabla
315
316         ldy #$0
317
318 ptBucle
319
320         sty plFila
321
322         jsr pintaFila
323
324         iny
325
326         cpy #$08
327         bne ptBucle
328
329         rts
330
```

Y, a su vez, pintar una fila es pintar las fichas (ratón, gato o vacío) de todas las columnas de esa fila. Y para obtener esa información nos apoyamos en la ya conocida “dameContenido”:

```
331 ; Rutina para pintar una fila
332
333 plFila byte $00
334
335 pintaFila
336
337         ; Número de fila
338         lda plFila
339         clc
340         adc #48
341         jsr chrout
342
343         ; Posiciones de la fila
344         lda ptTableroLo
345         sta dcTableroLo
346
347         lda ptTableroHi
348         sta dcTableroHi
349
350         lda plFila
351         sta dcFila
352
353         ldx #$00
354
355 pfBucle
356
357         stx dcColumna
358
359         jsr dameContenido
360
361         lda dcFicha
362         sta pfFicha
363         jsr pintaFicha
364
365         inx
366
367         cpx #$08
368         bne pfBucle
369
370         lda #13
371         jsr chrout
372
373         rts
374
```

Por último, la rutina “pintaFicha” pinta una “R” para el ratón, una “G” para los gatos, y un punto para las casillas vacías:

```
375 ; Rutina para pintar una ficha
376
377 pfFicha byte $00
378
379 pintaFicha
380
381         lda pfFicha
382
383 pfRaton
384
385         cmp #Raton
386
387         bne pfGato
388
389         lda #"r"
390         jsr chrout
391
392         rts
393
394 pfGato
395
396         cmp #Gato
397
398         bne pfVacio
399
400         lda #"g"
401         jsr chrout
402
403         rts
404
405 pfVacio
406
407         lda #"."
408         jsr chrout
409
410         rts
411
```

En resultado es algo así (de momento el tablero no tiene información de padre ni hijos, pero sí está inicializado con las piezas en su sitio):

Como digo, esto se puede convertir en algo más vistoso más adelante mediante el uso de caracteres personalizados, pero no vamos a empezar la casa por el tejado.

RYG: programa principal

Hasta ahora tenemos:

- Una estructura de datos para los tableros.
- Una librería de rutinas para gestionar los tableros.
- Y una rutina para pintar tableros.

Por tanto, lo siguiente es ya ponernos manos a la obra con el programa principal. Este programa lo definimos en el fichero “RYG.asm” y de momento hace muy poca cosa:

```
; Juego del ratón y el gato
; Fichero principal

; Crea el tablero inicial y lo pinta

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

        ; Inicializa el tablero
        lda #<tableroActual
        sta itTableroLo

        lda #>tableroActual
        sta itTableroHi

        jsr inicializaTablero

        ; Pinta el tablero
        lda #<tableroActual
        sta ptTableroLo

        lda #>tableroActual
        sta ptTableroHi

        jsr pintaTablero

        rts

```

Es decir, el programa principal tiene un cargador BASIC en $801 (10 SYS 2064) y se carga a partir de $810. Hace esto:

- Inicializa un tablero llamado “tableroActual”.
- Pinta “tableroActual”.
- Y termina con “rts”.

La etiqueta “tableroActual” apuntará al tablero actual, es decir, al tablero que contiene la situación actual de la partida. Como no vamos a llevar un registro de jugadas no nos hace falta una lista enlazada de tableros; nos llega con el tablero actual.

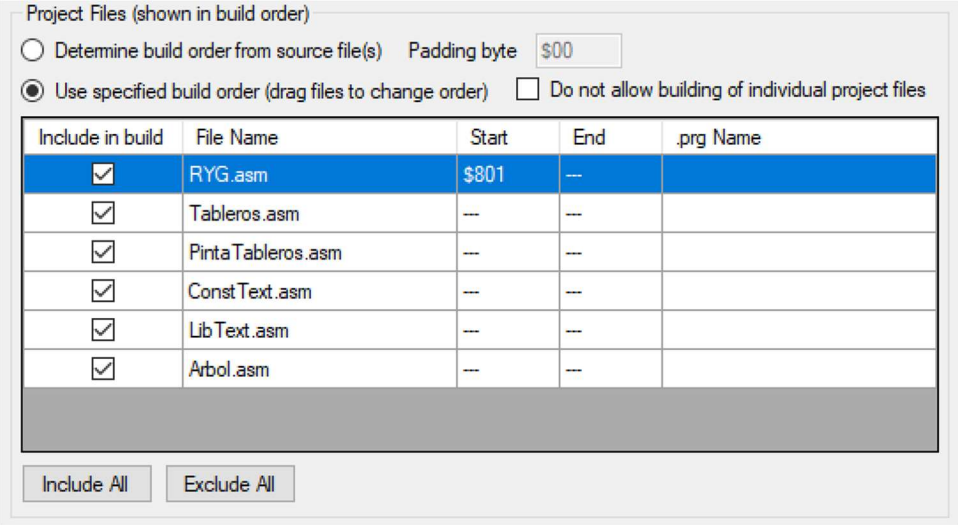
Esta etiqueta no está definida en “RYG.asm”, sino que está definida en otro fichero: el fichero “Arbol.asm”. En este fichero iremos metiendo más etiquetas

(es decir, más variables), no sólo para el tablero actual, sino también para el árbol de juego.

Arbol.asm

```
1  ; Fichero para el árbol de jugadas
2  ; Debe ir al final de los programas
3
4  tableroActual
5
```

Y dado que el árbol de juego va a crecer, se va a tomar una decisión de juego sobre él, se va a tirar, se va a generar otro árbol de juego, y así sucesivamente, necesita memoria libre que no esté ocupada por código. Por eso es importante que el fichero “Arbol.asm” vaya al final de la lista de construcción del proyecto (en CBM prg Studio menú Project > Properties):



De hecho, deberíamos hacer una estimación, aunque sea aproximada, de la memoria que vamos a necesitar en función de la profundidad del árbol. Los gatos tienen un máximo de ocho movimientos y el ratón de dos, aunque no todos los movimientos son posibles siempre. Así:

- 1 nivel → 85 bytes x 8 movimientos = 680 bytes.
- 2 niveles → 85 bytes x 8 x 4 = 2720 bytes.
- 3 niveles → 85 bytes x 8 x 4 x 8 = 21760 bytes.

- 4 niveles → 85 bytes x 8 x 4 x 8 x 4 = 87040 bytes.

Es decir, todo parece indicar que, con esta estructura de datos, lo máximo a lo que vamos a poder llegar es a 3 niveles de profundidad. Para llegar a 4 niveles, es decir, a 8 x 4 x 8 x 4 = 1024 tableros, harían falta casi 87 KB que lógicamente no están disponibles. Pero sí sería posible diseñar una estructura de datos más compacta.

Bueno, el resultado del programa principal, de momento, es el ya conocido: un tablero inicializado y pintado por pantalla. Poco a poco vamos construyendo.

RYG: jugadas del ratón – rutina básica

El siguiente paso es generar las jugadas del ratón, es decir, empezar ya con el generador de jugadas.

Con el ratón va a jugar el humano, es decir, el usuario. Entonces, ¿por qué necesitamos generar sus jugadas? Pues porque en el árbol de juego habrá tableros que proceden de movimientos de los gatos y tableros que proceden de movimientos del ratón. Por tanto, el C64 no sólo tiene que saber mover los gatos; también tiene que saber cómo se mueve el ratón.

Para no mezclar el generador de jugadas con nada de lo anterior ("Tableros.asm", "PintaTableros.asm", "RYG.asm" y "Arbol.asm") estrenamos un nuevo fichero: "GenJugadas.asm".

La primera parte de este fichero contiene estas dos tablas para el ratón:

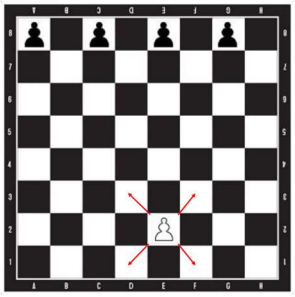
```
1  ; Fichero para generar jugadas, es decir, desarrollar el árbol
2
3  ; Jugadas que puede hacer el ratón
4
5  tblRatonF       byte $ff, $ff, $01, $01
6  tblRatonC       byte $ff, $01, $01, $ff
7
```

La tabla "tblRatonF" nos da los cambios permitidos en la fila del ratón; la tabla "tblRatonC" nos da los cambios permitidos en la columna del ratón. Recorriendo ambas tablas con un índice común, combinando ambos cambios, y recordando que $ff es lo mismo que -1, los cambios permitidos son:

- Índice 0 → (-1, -1).

- Índice 1 → (-1, +1).
- Índice 2 → (+1, +1).
- Índice 3 → (+1, -1).

Es decir, si los pintamos gráficamente sobre un tablero:



No todos estos cuatro movimientos serán siempre válidos. En ocasiones, la casilla de destino estará ocupada por un gato o caerá fuera del tablero. Pero esto lo dejamos para un momento posterior. De momento sólo vamos a generar todas las posibilidades.

La segunda parte del fichero "GenJugadas.asm" tiene otro par de tablas ("tablaGatosF" y "tablaGatosC"), pero ahora dedicadas al movimiento de los gatos. Por tanto, no vamos a verlas de momento.

La última parte del fichero contiene la rutina "generaJugadaRaton", que genera una jugada para el ratón. Esta rutina recibe como parámetros de entrada:

- Fila del ratón.
- Columna del ratón.
- Número de jugada, numeradas desde el 0 hasta el 3.

Y devuelve como parámetros de salida:

- La nueva fila del ratón.
- La nueva columna del ratón.

El campo “gjrTempX”, como ya hemos comentado, no es más que un almacén temporal para el registro X, puesto que vamos a modificar su valor para acceder con él a la tabla de movimientos del ratón.

El código de la rutina “generaJugadaRaton” es así:

```
; Rutina para generar una jugada del ratón

gjrFila         byte $00
gjrColumna      byte $00
gjrNumJugada    byte $00
gjrNuFila       byte $00
gjrNuColumna    byte $00

gjrTempX        byte $00

generaJugadaRaton

        stx gjrTempX

        ldx gjrNumJugada

        lda gjrFila
        clc
        adc tblRatonF,x
        sta gjrNuFila

        lda gjrColumna
        clc
        adc tblRatonC,x
        sta gjrNuColumna

        ldx gjrTempX

        rts

```

Es decir:

- Guarda el valor del registro X.
- Carga el número de jugada (0 … 3) en el registro X.
- Usando el registro X (número de jugada) como índice, obtiene el cambio en la fila y lo suma a la fila actual del ratón. El resultado lo pone en la salida.
- Usando el registro X (número de jugada) como índice, obtiene el cambio en la columna y lo suma a la columna actual del ratón. El resultado lo pone en la salida.
- Recupera el valor original del registro X.

De este modo, variando el número de jugada desde el valor 0 hasta el valor 3 podemos generar todas las jugadas del ratón. Cuestión diferente es que sean válidas o no.

RYG: jugadas del ratón – nuevas rutinas de apoyo

El siguiente paso sería ampliar el programa principal “RYG.asm” para generar y pintar las jugadas del ratón. Pero, antes de eso, o más bien al intentar hacerlo, surge la necesidad de desarrollar nuevas rutinas en “Tableros.asm”.

Rutina “dameRaton”:

La rutina “generaJugadaRaton” no mueve el ratón allí donde esté. Necesitan que le digan dónde está el ratón y, a partir de esta información, genera la nueva posición de salida.

Por tanto, necesitamos una rutina que localice el ratón dentro de un tablero. Esa rutina es la rutina “dameRaton” del fichero “Tableros.asm”. Su código es así:

```
669 ; Rutina para localizar el ratón
670
671 drTableroLo     byte $00
672 drTableroHi     byte $00
673 drOffset        byte $00
674
675 drTempY         byte $00
676
677 dameRaton
678
679         sty drTempY
680
681         ldy #21
682
683         lda drTableroLo
684         sta $fb
685
686         lda drTableroHi
687         sta $fc
688
689 drBucle
690
691         lda ($fb),y
692         cmp #Raton
693
694         bne drCont
695
696         sty drOffset
697         ldy drTempY
698
699         rts
700
701 drCont
702
703         iny
704
705         cpy #85
706
707         bne drBucle
708
709         ldy #$ff        ; No localizado
710         sty drOffset
711
712         ldy drTempY
713
714         rts
715
```

La rutina funciona así:

- Recibe un tablero como entrada.
- Guardar el registro Y para recuperarlo al final.
- Prepara el registro Y para que apunte al offset 21, que es el comienzo de la matriz, es decir, la posición (0, 0).
- Prepara el puntero $fb – $fc para que apunte al comienzo del tablero.
- Con el modo de direccionamiento indirecto – indexado, lee el contenido de la posición apuntada por Y. Mira si es un ratón ($01).

- Si el contenido es un ratón, mueve Y a la salida (offset del ratón) y recupera el valor original de Y.
- Si el contenido no es un ratón, incrementa Y y vuelve a intentarlo.
- Todo esto hasta el offset 85, que es el final de la matriz. Si llegamos hasta aquí es que no hemos encontrado el ratón y tiene que haber un error. Se señalizaría con el offset $ff, que no es válido.

En definitiva, esta rutina localiza el ratón en el tablero, pero nos devuelve su posición en formato offset, es decir, como desplazamiento o número de bytes desde el comienzo del tablero. Esto tiene la ventaja de que es fácil de obtener, pero es poco práctico de manejar.

Es más práctico de manejar el formato (fila, columna), que de hecho es el formato que acepta "generaJugadaRaton". Por tanto, necesitamos una rutina de conversión. Esa rutina es "dameFilaCol" y se describe más adelante.

Rutina "dameGato":

Esta rutina hace lo mismo que "dameRaton" pero para un gato. La principal diferencia es que gatos hay cuatro, mientras que sólo hay un ratón. Por tanto, hay que indicarle qué gato estamos buscando (0 … 3).

La implementación es también muy parecida. La principal diferencia es que, cuando encontramos un gato ($ff), tenemos que mirar si es el que buscamos o no. Si es el gato que buscamos devolvemos su offset; si no lo es, seguimos buscando hasta dar con él.

Rutina "dameFilaCol":

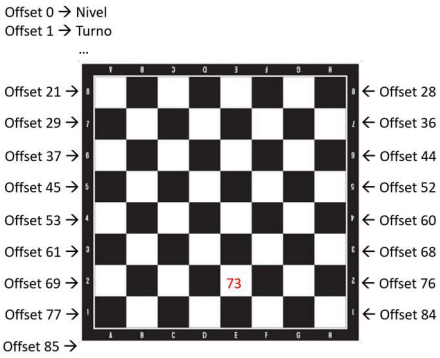
La rutina "dameFilaCol" convierte un offset en (fila, columna). Es complementaria a la rutina "dameOffset", que hace justo lo contrario.

La rutina "dameOffset" se basaba en una tabla de datos. En función de la fila, obteníamos un offset de base (21, 29, 37, …) y luego le sumábamos la columna.

Para hacer una implementación similar de "dameFilaCol" necesitaríamos una tabla con tantas entradas como offsets posibles (21 – 84). En realidad, dos tablas, una para las filas y otra para las columnas. Se puede hacer, pero no es una

solución muy elegante y consume bastante memoria (dos tablas de 64 posiciones cada una).

Por ello, se adopta otra solución. Supongamos que tenemos el offset 73:



Lo vamos comparando por orden contra el comienzo de cada fila, empezando por la última fila (fila 7) y terminando por la primera (fila 0). En realidad, empezamos comparando contra una fila inexistente, la fila 8, para detectar si el offset es mayor que el permitido:

- Fila 8 (inexistente): ¿es el offset mayor o igual que 85? En el caso de 73, no. Pero en caso de serlo, la rutina devolvería fila = $ff y columna = $ff, es decir, una señal de error.
- Fila 7 (última): ¿es el offset mayor o igual que 77? En el caso de 73, no.
- Fila 6: ¿es el offset mayor o igual que 69? En el caso de 73, sí. Por tanto, devuelve fila 6 y columna 73 – 69 = 4. Es decir, devuelve (6, 4).
- …

El proceso anterior hubiera continuado, caso de ser necesario, hasta la fila 0. De hecho, si el offset fuera menor a 21 (comienzo de la fila 0), también se devolvería una señal de error ($ff, $ff).

El código de la rutina es largo, pero muy repetitivo. Se muestra a continuación un extracto suficiente para entender su funcionamiento:

```
519 ; Rutina para convertir de offset a (fila,columna)
520
521 dfcOffset       byte $00
522 dfcFila         byte $00
523 dfcColumna      byte $00
524
525 dameFilaCol
526
527 dfcFila8
528
529         lda dfcOffset
530         sec
531         sbc #85
532
533         bcc dfcFila7
534
535         lda #$ff        ; Offset no válido
536         sta dfcColumna
537         sta dfcFila
538
539         rts
540
541 dfcFila7
542
543         lda dfcOffset
544         sec
545         sbc #77
546
547         bcc dfcFila6
548
549         sta dfcColumna
550
551         lda #7
552         sta dfcFila
553
554         rts
555
556 dfcFila6
```

Con estas rutinas de apoyo (“dameRaton”, “dameGato” y “dameFilaCol”) ya es posible mejorar el programa principal y generar las jugadas del ratón. Esto lo haremos en la siguiente entrada.

RYG: jugadas del ratón – nuevo programa principal

Ahora ya sí, con las nuevas rutinas de apoyo, podemos ampliar el programa principal “RYG.asm”. Manteniendo lo anterior (inicializar y pintar el tablero actual), añadimos los siguiente:

```
36          ; Localiza el ratón
37          lda #<tableroActual
38          sta drTableroLo
39
40          lda #>tableroActual
41          sta drTableroHi
42
43          jsr dameRaton
44
45          ; Convierte a fila y columna
46          lda drOffset
47          sta dfcOffset
48
49          jsr dameFilaCol
50
51          ; Genera la jugada X del ratón
52          lda dfcFila
53          sta gjrFila
54
55          lda dfcColumna
56          sta gjrColumna
57
58          ldx #0
59
60  bucle
61
62          stx gjrNumJugada
63
64          jsr generaJugadaRaton
65
66          lda gjrNuFila
67          sta numeroHex
68          jsr pintaHex
69
70          lda gjrNuColumna
71          sta numeroHex
72          jsr pintaHex
73
74          lda #13
75          jsr chrout
76
77          inx
78
79          cpx #4
80          bne bucle
81
82          rts
83
```

Es decir:

- Localizamos el ratón con la nueva rutina “dameRaton”.
- Convertimos la posición del ratón de formato offset a formato (fila, columna) mediante el uso de la nueva rutina “dameFilaCol”.
- Hacemos un bucle de 0 a 3 variando el número de jugada del ratón.
- Solicitamos las jugadas 0 a 3 con la rutina “generaJugadaRaton”.
- Pintamos las nuevas posiciones del ratón con “pintaHex”.

El resultado es éste:

Como podéis observar, en cierto modo el programa ya sabe mover el ratón. Sabe que el ratón se podría mover a las posiciones:

- (6, 2).
- (6, 4).
- (8, 4).
- (8, 2).

Lo que no sabe todavía es que las posiciones (8, 4) y (8, 2) no son válidas, ya que están fuera del tablero. Este será el siguiente paso: validar las jugadas.

RYG: jugadas del ratón – validación de jugadas

Las condiciones para que una jugada del ratón sea válida son tres:

- Que respete las reglas de movimiento, es decir, que sea en diagonal, hacia delante o hacia atrás, y moviendo una única casilla.

- Que la casilla de destino esté dentro de los márgenes del tablero.
- Que la casilla de destino esté libre, puesto que no hay capturas.

La primera condición se cumple siempre, puesto que la forma de generar los movimientos mediante dos tablas lo garantiza.

Las condiciones dos y tres de momento no están garantizadas. Hay que comprobarlas. Lo bueno es que son las mismas condiciones que necesitan los gatos, de modo que con una única rutina de validación podemos validar los movimientos de ratón y gatos.

La rutina de validación se llama “validaJugada”, y es una nueva rutina del fichero “GenJugadas.asm”:

```
43  ; Rutina para validar una jugada
44
45  ; Para que una jugada sea válida el destino tiene que caer dentro
46  ; del tablero y estar libre; vale tanto para ratón como para gatos
47
48  vjNuFila        byte $00
49  vjNuColumna     byte $00
50  vjValida        byte $00
51
52  validaJugada
53
54          lda #$ff
55          sta vjValida
56
```

Recibe como parámetros de entrada la nueva fila y columna, es decir, la posición de destino. Y devuelve si el movimiento es válido o no con $00 (válido) o $ff (no válido). Nada más empezar inicializamos la salida asumiendo que no será válido.

La validación de los límites del tablero se hace así:

```
57  vjFilaNeg
58
59          lda vjNuFila
60
61          cmp #$ff
62          bne vjFilaPos
63
64          rts
65
66  vjFilaPos
67
68          cmp #$08
69          bne vjColNeg
70
71          rts
72
73  vjColNeg
74
75          lda vjNuColumna
76
77          cmp #$ff
78          bne vjColPos
79
80          rts
81
82  vjColPos
83
84          cmp #$08
85          bne vjOcu
86
87          rts
88
```

Es decir, verificamos si la nueva fila es -1 = $ff u 8. Lo mismo con la columna. Y, como las coordenadas válidas van desde 0 hasta 7, ambas incluidas, cualquier fila o columna que valga -1 u 8 están fuera del tablero. No es necesario verificar valores menores que -1 ni mayores que 8, puesto que las fichas sólo se mueven una casilla en cada movimiento.

Por otro lado, la verificación de si el destino está ocupado o no es así:

```
88          ---
89  vjOcu
90
91          lda vjNuFila
92          sta dcFila
93
94          lda vjNuColumna
95          sta dcColumna
96
97          jsr dameContenido
98
99          lda dcFicha
100         cmp #Vacio
101
102         beq vjOK
103
104         rts
105
106 vjOK
107
108         lda #$00
109         sta vjValida
110
111         rts
112
```

Es decir, se recupera el contenido de la posición de destino con la rutina "dameContenido" y se comprueba si es vacío.

Si cualquiera de las validaciones anteriores (límites o contenido) falla, la rutina termina con un "rts" y devuelve el valor $ff (jugada no válida). Si todas las validaciones se superan, la rutina devuelve $00 (jugada válida).

La rutina anterior tiene una limitación que de momento es aceptable: sólo trabaja sobre el tablero actual (que está implícito). Pero cuando construyamos el árbol de juego para que el C64 juegue por los gatos hará falta validar jugadas sobre un tablero arbitrario del árbol. Llegados ese momento habrá que mejorarla.

RYG: jugadas del ratón – obtención de todas las jugadas válidas

Una vez que ya somos capaces de validar una jugada, el siguiente paso es obtener todas las jugadas válidas del ratón (y sólo las validas). Esto lo conseguimos con la nueva rutina "generaJugadasValidasRaton" del fichero "GenJugadas.asm" que, resumidamente, es una combinación de las rutinas "generaJugadaRaton" y "validaJugada".

La rutina "generaJugadasValidasRaton" está declarada así:

```
113 ; Rutina para obtener todas las jugadas validas del ratón
114
115 gjvrFila        byte $00
116 gjvrColumna     byte $00
117 gjvrNuFilas     byte $00, $00, $00, $00
118 gjvrNuColumnas  byte $00, $00, $00, $00
119
120 gjvrTempX       byte $00
121
122 generaJugadasValidasRaton
123
124         stx gjvrTempX
125
```

Es decir, recibe como entrada la posición actual del ratón (fila y columna) y devuelve como salida hasta cuatro posiciones válidas.

El cuerpo de la rutina es así:

```
125 |
126         ldx #$00
127
128 gjvrBucle
129
130         lda gjvrFila
131         sta gjrFila
132
133         lda gjvrColumna
134         sta gjrColumna
135
136         stx gjrNumJugada
137
138         jsr generaJugadaRaton
139
140         lda gjrNuFila
141         sta vjNuFila
142
143         lda gjrNuColumna
144         sta vjNuColumna
145
146         jsr validaJugada
147
148         lda vjValida
149
150         cmp #$ff
151         beq gjvrNoValida
152
153 gjvrValida
154
155         lda gjrNuFila
156         sta gjvrNuFilas,x
157
158         lda gjrNuColumna
159         sta gjvrNuColumnas,x
160
161         jmp gjvrCont
162
163 gjvrNoValida
164
165         lda #$ff
166         sta gjvrNuFilas,x
167         sta gjvrNuColumnas,x
168
169 gjvrCont
170
171         inx
172
173         cpx #$04
174         bne gjvrBucle
175
176         ldx gjvrTempX
177
178         rts
```

Es decir, esencialmente es un bucle desde X=0 hasta X=3, donde para cada valor del registro X:

- Genera la jugada X-ésima del ratón.

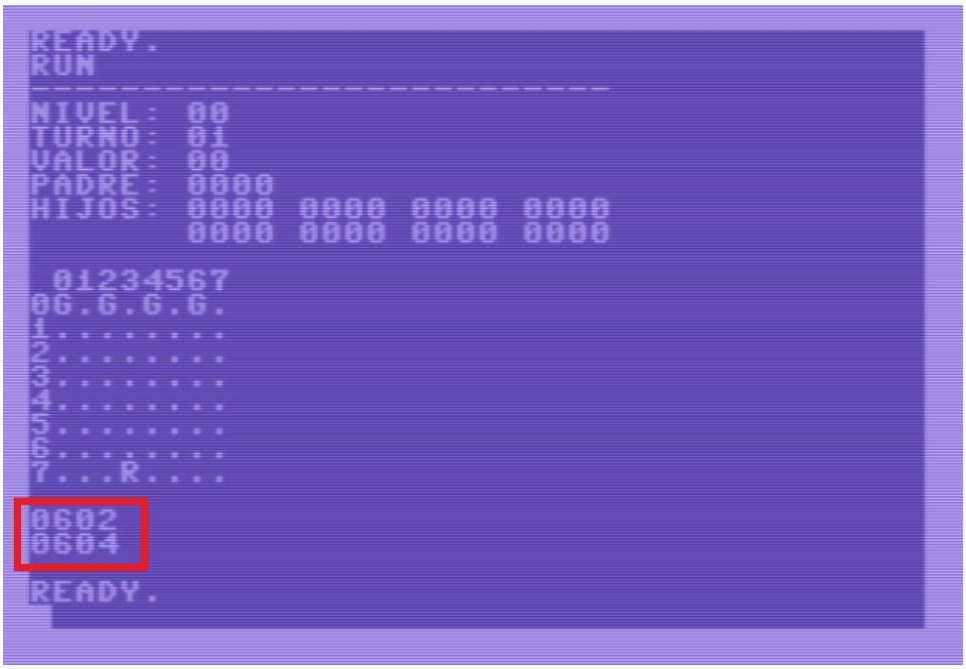
- Valida la jugada, es decir, verifica que el destino está vacío y dentro del tablero.
- Si es válida (etiqueta “gjvrValida”), almacena la jugada en la tabla de salida.
- Si no es válida (etiqueta “gjvrNoValida”), almacena ($ff, $ff) en la tabla de salida.

De este modo, al terminar la rutina, en la tabla de salida tenemos las jugadas válidas (nueva fila y columna) o ($ff, $ff) si una jugada es inválida.

De este modo, podemos volver a modificar el programa principal “RYG.asm”. En vez de generar todas las jugadas del ratón con “generaJugadaRaton”, generamos sólo las válidas con “generaJugadasValidasRaton”, y éstas son las que pintamos:

```
        ; Localiza el ratón
        lda #<tableroActual
        sta drTableroLo

        lda #>tableroActual
        sta drTableroHi

        jsr dameRaton

        ; Convierte a fila y columna
        lda drOffset
        sta dfcOffset

        jsr dameFilaCol

        ; Genera las jugadas válidas del ratón
        lda dfcFila
        sta gjvrFila

        lda dfcColumna
        sta gjvrColumna

        jsr generaJugadasValidasRaton

        ldx #$00

bucle

        lda gjvrNuFilas,x

        cmp #$ff
        beq cont

        sta numeroHex
        jsr pintaHex

        lda gjvrNuColumnas,x
        sta numeroHex
        jsr pintaHex

        lda #13
        jsr chrout

cont

        inx

        cpx #$04
        bne bucle

        rts

```

El resultado es así:



Ahora ya podemos afirmar que, definitivamente, el C64 sabe mover el ratón. Se puede probar a mover el ratón a otra posición inicial y comprobar que los movimientos generados son correctos.

El siguiente paso será pedirle al usuario qué jugada quiere efectuar.

RYG: jugadas del ratón – solicitud al usuario

El objetivo de generar los movimientos válidos del ratón no es ayudar al usuario. Se supone que el usuario sabe jugar.

El objetivo de generar los movimientos válidos del ratón es generar el árbol de juego. Para esto también necesitaremos generar los movimientos válidos de los gatos, pero esto ya lo haremos más adelante.

No obstante, “aprovechando que el Pisuerga pasa por Valladolid”, como se suele decir, podemos ofrecerle al usuario las jugadas válidas del ratón, y que éste elija cuál le interesa más. La alternativa sería hacerle una pregunta abierta del tipo “¿a

qué casilla quiere mover el ratón?" y, a posteriori, validar que el destino es válido. Ya que hemos llegado hasta aquí, casi mejor hacer lo primero.

Para ello, lo primero que vamos a hacer es mejorar el fichero "PintaTableros.asm" con una nueva rutina "pintaJugada", que simplemente pinta una jugada en el formato:

Número de jugada> (fila origen, columna origen) – (fila destino, columna destino)

Y lo siguiente que hacemos es mejorar el programa principal, fichero "RYG.asm", para hacer todo lo que veníamos haciendo hasta ahora (inicializar el tablero actual, pintar el tablero actual, pintar las jugada válidas del ratón – ya con la nueva rutina "pintaJugada" –) y, además, solicitar al usuario qué jugada le interesa más.

Además, aprovechamos el viaje para estructurar un poco el programa principal porque, según vamos avanzando, se va complicando más y más:

```
; Juego del ratón y el gato
; Fichero principal

; Pregunta al usuario qué jugada quiere y la valida

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        rts

```

La principal novedad (reestructuración aparte) es la solicitud de la jugada al usuario, para lo que usamos la rutina “leeTeclado” de la librería “LibText.asm”:

```
121 solicitaJugadaRaton
122
123         lda #<sjrPeticion
124         sta cadenaLo
125
126         lda #>sjrPeticion
127         sta cadenaHi
128
129         jsr pintaCadena
130
131         jsr leeTeclado
132
133         lda #13
134         jsr chrout
135
136         ldx byteLeido
137
138         cpx #$04
139         bcs solicitaJugadaRaton
140
141         lda gjvrNuFilas,x
142
143         cmp #$ff
144         beq solicitaJugadaRaton
145
146         rts
147
148 sjrPeticion     text "que jugada desea? "
149                 byte $00
150
```

El número introducido por el usuario (“byteLeido”) se contrasta contra el valor 4 y, si es mayor o igual que este valor, se vuelve a preguntar. Las jugadas válidas del ratón se numeran desde la 0 hasta la 3.

Igualmente, usando “byteLeido” como índice (registro X), se comprueba si la jugada elegida es válida o si vale ($ff, $ff). En este último caso, se vuelve a preguntar.

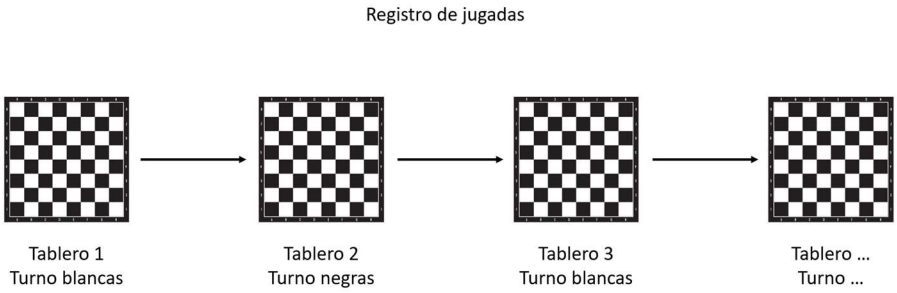
En conclusión, ya le presentamos al usuario sus jugadas válidas y le preguntamos cuál quiere:

Lo siguiente es aplicar esa jugada, dando lugar a un nuevo tablero actual. Recordemos que no vamos a mantener un historial o registro de jugadas.

RYG: jugadas del ratón – aplicación de la jugada del usuario

Ya somos capaces de generar las jugadas válidas del ratón y de pedirle al usuario cuál es la que le interesa. El siguiente paso, por tanto, es aplicar la jugada seleccionada por el usuario sobre el tablero actual, el que mantiene la situación actual de la partida.

Si quisiéramos llevar un historial de jugadas, por ejemplo, para permitir rectificar una jugada o grabar y cargar una partida, tendríamos que sacar copia del tablero actual en un nuevo tablero, aplicar la jugada del usuario sobre el nuevo tablero, y enlazar ambos tableros entre sí:



Sin embargo, como no queremos hacer nada de esto (no sería complejo), es suficiente con que apliquemos la jugada seleccionada por el usuario directamente sobre el tablero actual, perdiendo la situación anterior.

Para aplicar una jugada o movimiento, da igual que hablemos del ratón o los gatos, lo que hay que hacer es:

- Poner vacío ($00) en la posición que actualmente ocupa la pieza.
- Poner la pieza, en este caso un ratón ($01), en la posición de destino.

Y esto es, precisamente, lo que hace la nueva rutina “aplicaJugadaRaton”, que ubicamos en el fichero “GenJugadas.asm”:

```
; Rutina para aplicar una jugada del ratón

ajTableroLo     byte $00
ajTableroHi     byte $00
ajFila          byte $00
ajColumna       byte $00
ajNuFila        byte $00
ajNuColumna     byte $00

aplicaJugadaRaton

        lda ajTableroLo
        sta fcTableroLo

        lda ajTableroHi
        sta fcTableroHi

        lda ajFila
        sta fcFila

        lda ajColumna
        sta fcColumna

        lda #Vacio
        sta fcFicha

        jsr fijaContenido

        lda ajNuFila
        sta fcFila

        lda ajNuColumna
        sta fcColumna

        lda #Raton
        sta fcFicha

        jsr fijaContenido

        rts
```

Como se puede observar, la rutina “aplicaJugadaRaton” recibe como parámetros el tablero sobre el que se quiere actuar, la posición (fila, columna) de origen, y la posición (fila, columna) de destino. Usando la rutina “fijaContenido”, la rutina vacía ($00) la posición de origen, y pone un ratón ($01) en el destino.

Ahora sólo queda mejorar el programa principal “RYG.asm” y aplicar sobre el tablero actual el movimiento seleccionado por el usuario. Esto se hace desde la nueva rutina “aplicaJugadaSolicitada”:

```
153 aplicaJugadaSolicitada
154
155         ldx byteLeido
156
157         lda #<tableroActual
158         sta ajTableroLo
159
160         lda #>tableroActual
161         sta ajTableroHi
162
163         lda gjvrFila
164         sta ajFila
165
166         lda gjvrColumna
167         sta ajColumna
168
169         lda gjvrNuFilas,x
170         sta ajNuFila
171
172         lda gjvrNuColumnas,x
173         sta ajNuColumna
174
175         jsr aplicaJugadaRaton
176
177         rts
178
```

Esta nueva rutina se llama desde el programa principal “RYG.asm” que, recordemos, había sido reestructurado en subrutinas para mayor claridad:

```
; Juego del ratón y el gato
; Fichero principal

; Aplica la jugada seleccionada por el usuario

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jmp actualiza

```

Como además el programa principal lo hemos cerrado con un “jmp actualiza” (bucle de juego), el usuario puede jugar sucesivamente con el ratón, moviéndolo ya por todo el tablero. De hecho, se recomienda al lector practicar con el juego, moviendo el ratón hacia adelante y hacia atrás y, sobre todo, acercándose a la fila de gatos y a los bordes del tablero para comprobar que las jugadas propuestas son correctas:

Parece mentira, pero en relativamente pocos pasos ya tenemos la base del juego. Mucha de esta base será de igual aplicación a los gatos, aunque las cosas se complicarán porque hay cuatro gatos (frente a un ratón) y, sobre todo, porque para que el C64 decida sus jugadas será necesario conformar y evaluar un árbol de juego de varios niveles de profundidad.

RYG: jugadas de los gatos – rutinas básicas

Los principios básicos que rigen la generación de jugadas para los gatos son los mismos que para el ratón:

- Los movimientos posibles se gestionan mediante una tabla doble que recoge los cambios posibles en filas y columnas.
- Los movimientos posibles hay que validarlos. Para que sean válidos, además de realizarse conforme a las reglas del juego (punto anterior), la posición de destino debe estar vacía y caer dentro del tablero.

Por otro lado, las principales diferencias entre gatos y ratón son:

- Ratón sólo hay uno, mientras que gatos hay cuatro. Esto complica un poco las rutinas, porque hay que indicarles sobre qué gato se trabaja. Y también complica los programas que usan esas rutinas, porque tendrán que hacer bucles desde 0 hasta 3 para trabajar con todos los gatos.
- Con el ratón juega el humano, y lo que él decida y comunique a través del teclado se aplicará al tablero actual. Sin embargo, con los gatos juega el C64 y, para que éste pueda decidir sus jugadas, hace falta conformar y evaluar un árbol de juego. Este árbol de juego tendrá varios niveles de profundidad (cuantos más tenga más potente será el juego), y para conformarlo habrá que alternar movimientos de gatos y ratón.

Por tanto, la diferencia fundamental entre ratón y gatos es que **para los gatos hay que conformar y evaluar un árbol de juego**. No es tarea fácil, pero vamos paso a paso. De momento vamos a empezar por generar, validar y aplicar los movimientos de los gatos:

Separación de “GenJugadasRaton.asm” y “GenJugadasGatos.asm”:

Aunque en la introducción teórica a los juegos de tablero hablamos genéricamente de un “generador de jugadas”, lo cierto es que por claridad y simplicidad es mejor separar el generador de jugadas en dos ficheros:

- “GenJugadasRaton.asm”, dedicado al ratón. Es lo que en versiones anteriores del proyecto llamábamos “GenJugadas.asm”.
- “GenJugadasGatos.asm”, dedicado a los gatos. Es nuevo.

Por tanto, en esta entrada nos centramos en el nuevo fichero “GenJugadasGatos.asm”.

Nueva rutina “generaJugadaGato”:

Esta rutina es totalmente análoga a la ya conocida “generaJugadaRaton”. La principal diferencia es que los gatos sólo se mueven hacia adelante, hacia abajo con la orientación del tablero que tenemos, por lo que la tabla doble que rige el movimiento es más sencilla:

```
3  ; Jugadas que pueden hacer los gatos
4
5  tblGatosF       byte $01, $01
6  tblGatosC       byte $ff, $01
7
```

Es decir, los gatos pueden moverse así (siempre avanzando la fila):

- Índice 0 → (+1, -1).
- Índice 1 → (+1, +1).

Por lo demás, las rutinas “generaJugadaGato” y “generaJugadaRaton” son un calco, por lo que no merece la pena comentar más la primera.

Nueva rutina “generaJugadasValidasGato”:

Para que un movimiento de un gato sea válido, además de estar generado conforme a las reglas del juego (punto anterior), el destino tiene que estar vacío y caer dentro de los márgenes del tablero.

Estas reglas de validación son las mismas que para el ratón, por lo que no hace falta disponer de dos rutinas de validación diferentes. Se puede reutilizar la ya conocida “validaJugada” de “GenJugadasRaton.asm”.

Lo que si hará falta será una rutina “generaJugadasValidasGato” que permita generar todas las jugadas válidas de un gato (y sólo las válidas), igual que en su momento desarrollamos la rutina “generaJugadasValidasRaton”.

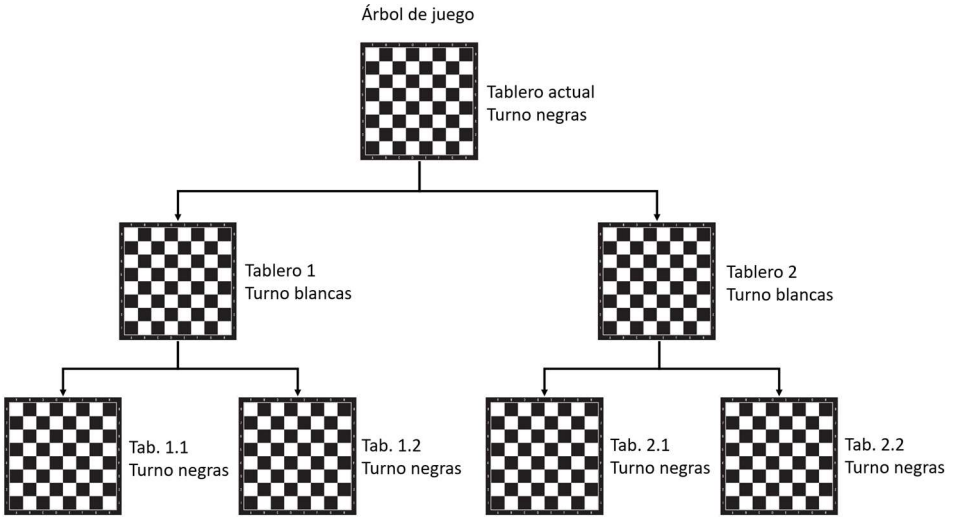
Ambas rutinas, “generaJugadasValidasGato” y “generaJugadasValidasRaton” son totalmente análogas, y básicamente consisten en generar las jugadas con “generaJugadaGato” o “generaJugadaRaton”, y luego validarlas con “validaJugada”. Por tanto, no abundaremos más en ellas aquí.

Nueva rutina “aplicaJugadaGato”:

Por último, igual que es necesario aplicar una jugada del ratón a un tablero, también será necesario aplicar una jugada de un gato. Por ello, desarrollamos la rutina “aplicaJugadaGato”, que es totalmente análoga a la ya conocida “aplicaJugadaRaton”.

En este punto, la principal diferencia entre ratón y gatos es que, con el primero, es el usuario quien decide la jugada. Y una vez decidida se aplica al tablero actual.

Con los gatos el tema es bastante más complejo porque, como ya hemos comentado, para que el C64 decida su jugada, hay que **conformar y evaluar un árbol de juego**. Para crear este árbol también habrá que generar jugadas (de gatos y ratón, por cierto) y aplicarlas a tableros. Pero ya no se aplicarán modificando un tablero concreto (la raíz del árbol), sino generando nuevos tableros hijo, nuevos tableros nieto, etc.:



Cuando todo este proceso de generación y evaluación del árbol haya terminado, y el C64 haya tomado una decisión sobre su jugada, entonces sí, la jugada elegida por el C64 habrá que aplicarla sobre el tablero actual, dando lugar a un nuevo tablero actual.

RYG: jugadas de los gatos – obtención de todas las jugadas válidas

Ahora que ya somos capaces de generar todas las jugadas válidas de un gato gracias a la rutina “generaJugadasValidasGato”, lo siguiente que podemos hacer es mejorar el programa principal “RYG.asm” para:

- Recorrer los cuatro gatos.

- Pedir las jugadas válidas de cada gato.
- Pintar por pantalla las 4 x 2 jugadas (podría haber alguna menos).

Esto lo vamos a hacer sólo de forma temporal, y con el objeto de depurar el programa. Es decir, lo vamos a hacer con la intención de comprobar que el programa genera bien las jugadas válidas de los gatos.

Más adelante dejaremos de hacerlo porque, al contrario que con el ratón, con los gatos no tenemos que comunicar nada ni pedir nada al usuario. Lo que tenemos que hacer es conformar y evaluar un árbol de juego a base de generar jugadas de gatos y ratón en turnos alternos.

Pero, de momento, para pintar las jugadas de los gatos ampliamos el programa "RYG.asm" con la nueva subrutina "pintaJugadasGatos" (recordemos que el programa principal, por claridad, ahora está estructurado en subrutinas):

```
; Juego del ratón y el gato
; Fichero principal

; Genera las jugadas de los gatos y las pinta

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaJugadasGatos

        rts ; jmp actualiza

```

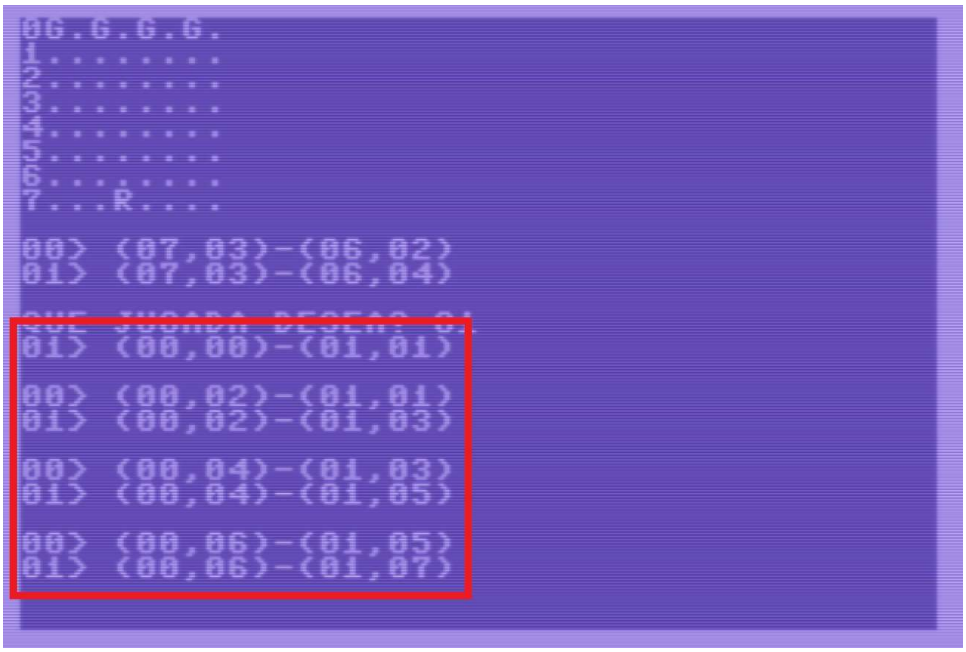
La rutina “pintaJugadasGatos” recorre los gatos desde el 0 hasta el 3 y, para cada uno, llama a la rutina “pintaJugadasGato” (que genera sus jugadas válidas y las pinta):

```
181 pintaJugadasGatos
182
183         ldx #$00
184
185 pjgBucle
186
187         stx pjgNumGato
188         jsr pintaJugadasGato
189
190         inx
191
192         cpx #$04
193         bne pjgBucle
194
195         rts
196
```

Por último, la rutina “pintaJugadasGato” es una mera combinación de rutinas ya conocidas:

- Localiza el gato con “dameGato”.
- Convierte la posición del gato de formato offset a formato (fila, columna) con “dameFilaCol”.
- Genera las jugadas válidas del gato con “generaJugadasValidasGato”.
- Pinta las jugadas válidas (0, 1 ó 2) con “pintaJugada”.

El resultado es algo así:



Como se puede observar, el programa genera siete movimientos válidos para los gatos, en vez de ocho, ya que el primer gato, el de la posición (0, 0), sólo puede moverse a la posición (1, 1). Los demás gatos pueden moverse a izquierda y derecha.

Por tanto, parece que el programa genera bien las jugadas del ratón (entradas anteriores) y de los gatos (esta entrada). El lector puede probar a cambiar los gatos de posición en la inicialización (rutina “inicializaTablero”), y comprobar que efectivamente la lista de jugadas cambia y es correcta.

Con estos mimbres ya podemos empezar a generar el árbol de juego, lo cual haremos a partir de la próxima entrada.

RYG: árbol de juego – memoria

Hasta ahora sólo teníamos el “tablero actual”, es decir, la estructura de datos que lleva control de en qué situación (en qué tablero) se encuentra el juego. Esto lo hacíamos así en el fichero “Arbol.asm”:

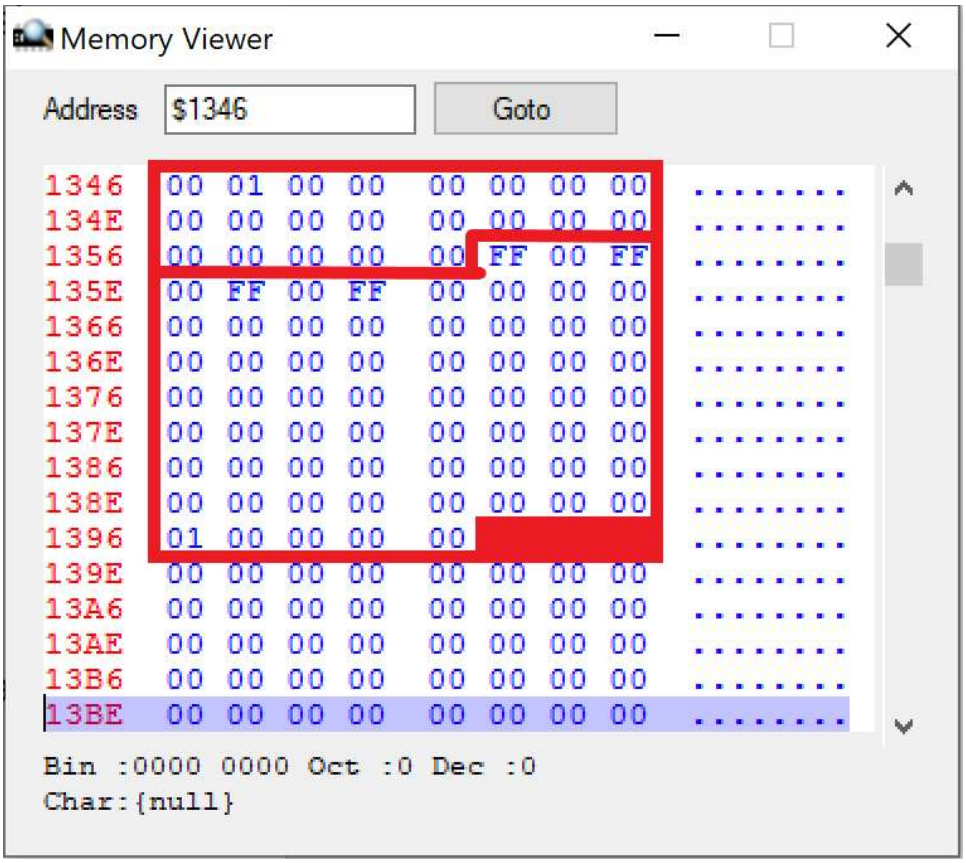
```
; Fichero para el árbol de jugadas
; Debe ir al final de los programas

tableroActual

```

Puede parecer un poco tonto definir una etiqueta “tableroActual” y no hacer nada más, pero no lo es. Usando <tableroActual (parte “lo”) y >tableroActual (parte “hi”) se puede definir un puntero a esa zona de la memoria y, mediante rutinas como “inicializaTablero” se puede configurar ahí un tablero (85 bytes) en su situación inicial:

```
inicializaTableroActual

        ; Inicializa el tablero
        lda #<tableroActual
        sta itTableroLo

        lda #>tableroActual
        sta itTableroHi

        jsr inicializaTablero

        rts

```

A partir de ese momento, se puede trabajar con ese “tablero actual”, generando los movimientos del ratón, preguntando al usuario cuál desea, aplicando ese movimiento al tablero, generando los movimientos para los gatos, etc.

Aquí podemos ver con el debugger el contenido de la memoria a partir de la etiqueta “tableroActual”, que en la versión 6 del proyecto es la dirección $1346:



Mirando con atención se pueden identificar el nivel ($00), el turno ($01), el valor ($00), el padre ($0000), los hijos ($0000, …, $0000) y, finalmente, el tablero propiamente dicho, donde están los gatos ($ff) y el ratón ($01).

Y todo ello con el objetivo de llevar el control de la situación actual de la partida.

Pero ahora tenemos otro objetivo: generar un árbol de juego. Y ese árbol de juego va a consistir en una serie de tableros (bloques de 85 bytes) ubicados uno a continuación del otro, y enlazados entre sí usando los punteros de padre e hijos. Esquemáticamente sería así:

Por tanto, necesitamos varias cosas:

- Reservar 85 bytes para el tablero actual, asegurándonos de que no son pisados por nadie.
- Una zona de memoria en la que meter tableros (bloques de 85 bytes) enlazados entre sí mediante punteros y conformando un árbol.
- La zona de memoria del árbol debe poder crecer según se vayan generando más tableros. Y no debe pisar zonas del sistema como el intérprete de BASIC (a partir de $a000).
- Y, una vez tomada una decisión de juego por el C64, la zona de memoria del árbol debe poder reutilizarse para las jugadas siguientes.

Esto lo logramos con un nuevo fichero “Arbol.asm” así:

```
; Fichero para el tablero actual y el árbol de jugadas
; Debe ir al final de los programas

; Tablero actual (85 bytes)

tableroActual

        ; Nivel, turno y valor
        byte $00
        byte $00
        byte $00

        ; Padre
        byte $00
        byte $00

        ; Hijos
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00

        ; Tablero propiamente dicho
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00

; Árbol de decisión para los gatos

libreLo byte $00
libreHi byte $00

raizArbol

```

Es decir:

- Reservamos 85 bytes de forma expresa para el tablero actual. Antes no había una reserva expresa; estaba implícita.
- Reservamos dos bytes para un puntero libreLo – libreHi, que nos va a indicar cuál es el primer byte de memoria que está libre para almacenar nuevos tableros que se vayan generando.
- Tras el puntero libreLo – libreHi viene la raíz del árbol, es decir, el primer tablero del árbol de juego. La raíz la copiaremos desde el tablero actual al comienzo de generar el árbol.

Por tanto, la memoria irá evolucionando así:

Es decir:

- Primero el código del juego, que empieza en $801.
- Luego el tablero actual con la situación de la partida.
- Luego el puntero a la memoria libre, que se irá actualizando según se generan tableros.
- Luego la raíz del árbol de juego, que se copiará desde el tablero actual al empezar a conformar el árbol.
- Luego los tableros hijo, los tableros nieto, etc., así hasta el nivel de profundidad establecido.
- Por último, la memoria libre, que irá decreciendo según se van generando tableros, y volverá a crecer cuando el C64 tome una decisión de juego.

Por esta estructura de tableros se puede navegar de padres a hijos, siguiendo cualquiera de los ocho punteros a los hijos que tiene cada tablero. También se puede navegar de hijos a padres, siguiendo el puntero al padre que tiene cada tablero. De hecho, tendremos que navegar por ella para aplicar el procedimiento minimax.

RYG: árbol de juego – generar tableros hijo

Ya sabemos generar las jugadas válidas del ratón y los gatos. También sabemos dónde y cómo vamos a almacenar el árbol de juego. Por tanto, el siguiente paso es empezar a construirlo.

Supongamos que acaba de jugar el humano. El humano ha seleccionado una jugada del ratón y ésta se ha aplicado al tablero actual dando lugar a un tablero actual modificado.

Ahora le toca mover al C64 y, para decidir su jugada, hay que conformar y evaluar el árbol de juego. El árbol va a tener varios niveles de profundidad, pero no vamos a abordar todos los niveles de golpe. Vamos paso a paso; empecemos por el primero.

Para generar el primer nivel del árbol de juego, los hijos de la raíz, hay que:

- Copiar el tablero actual a la raíz del árbol de juego.
- Recorrer los cuatros gatos, desde el 0 hasta el 3.
- Para cada gato, localizarlo sobre la raíz (rutina “dameGato”) y generar sus movimientos válidos (rutina “generaJugadasValidasGato”).
- Y, en vez de pintar las jugadas de los gatos, como en una entrada anterior, aplicar (*) cada jugada o movimiento al tablero raíz.

Pero cuidado con una cosa (*). Ahora no se trata de “aplicar la jugada” en el sentido de modificar la raíz del árbol o el tablero actual. Eso lo haremos más adelante, cuando el C64 haya decidido qué jugada le interesa más. Ahora estamos en un momento previo, construyendo el árbol que le permita tomar esa decisión.

Por tanto, ahora no queremos modificar la raíz del árbol, sino que queremos generar un tablero hijo por cada jugada posible. Por ello, vamos a desarrollar una nueva rutina “aplicaJugadaGatoHijo” que será similar a la rutina “aplicaJugadaGato” pero que, en vez de modificar el tablero de entrada, lo que hace es todo lo que sigue:

- Saca una copia del tablero de entrada (ahora la raíz del árbol) en un nuevo tablero que será el hijo. Para esto hace falta una nueva rutina de apoyo “copiaTablero” en el fichero “Tableros.asm”.
- Actualiza el puntero a la memoria libre (libreLo – libreHi), puesto que hemos creado un nuevo tablero en el árbol.
- **Sobre el tablero hijo, aplica el movimiento.**
- Más adelante, enlazará el tablero padre con el tablero hijo, y viceversa, aunque en esta fase del proyecto esto todavía no está implementado.

- A modo de depuración, y de forma temporal, imprime el tablero hijo.

Los parámetros de la rutina son así (el tablero padre, el tablero hijo, la posición actual del gato y su nueva posición):

```
154 ; Rutina para aplicar una jugada de un gato, pero generando un tablero hijo
155
156 ajghPadreLo     byte $00
157 ajghPadreHi     byte $00
158 ajghHijoLo      byte $00
159 ajghHijoHi      byte $00
160 ajghFila        byte $00
161 ajghColumna     byte $00
162 ajghNuFila      byte $00
163 ajghNuColumna   byte $00
164
165 aplicaJugadaGatoHijo
166
```

La copia de la raíz al tablero hijo es así (se apoya en la nueva rutina de apoyo “copiaTablero”):

```
165 aplicaJugadaGatoHijo
166
167         ; Copia el tablero padre en el hijo
168
169         lda ajghPadreLo
170         sta ctOrigenLo
171
172         lda ajghPadreHi
173         sta ctOrigenHi
174
175         lda ajghHijoLo
176         sta ctDestinoLo
177
178         lda ajghHijoHi
179         sta ctDestinoHi
180
181         jsr copiaTablero
182
```

La actualización del puntero a memoria libre es así (básicamente suma 85 bytes):

```
183         ; Actualiza el puntero a memoria libre
184
185         lda libreLo
186         clc
187         adc #85
188         sta libreLo
189
190         lda libreHi
191         adc #0
192         sta libreHi
193
```

La aplicación del movimiento es así (se apoya en “aplicaJugadaGato”, pero actuando sobre el tablero hijo):

```
194         ; Aplica la jugada al hijo
195
196         lda ajghHijoLo
197         sta ajgTableroLo
198
199         lda ajghHijoHi
200         sta ajgTableroHi
201
202         lda ajghFila
203         sta ajgFila
204
205         lda ajghColumna
206         sta ajgColumna
207
208         lda ajghNuFila
209         sta ajgNuFila
210
211         lda ajghNuColumna
212         sta ajgNuColumna
213
214         jsr aplicaJugadaGato
215
```

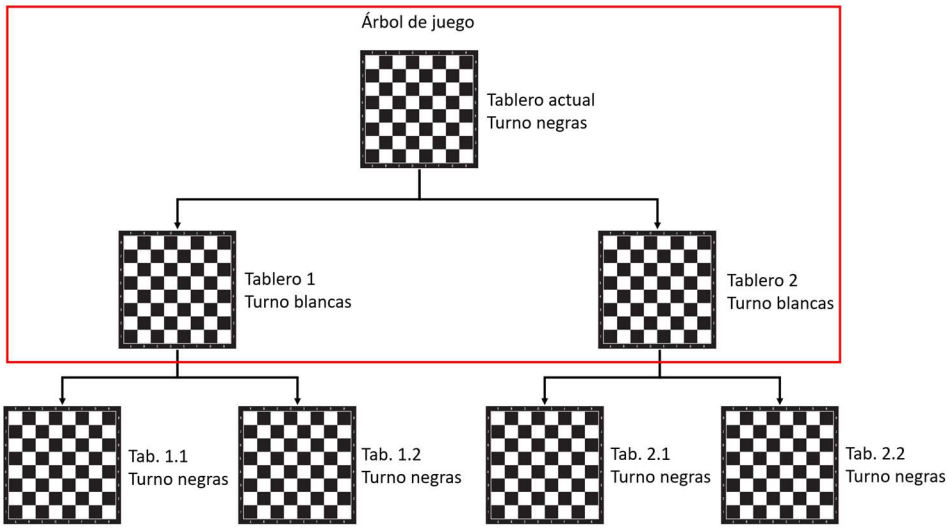
Las previsiones pendientes de implementar son así:

```
216         ; Incrementa el nivel y cambia el turno del hijo
217
218         ; Vincula el padre con el hijo
219
220         ; Vincula el hijo con el padre
221
```

Y, por último, la impresión del tablero hijo (para depurar) es así:

```
222         ; Pinta el hijo
223
224         lda ajghHijoLo
225         sta ptTableroLo
226
227         lda ajghHijoHi
228         sta ptTableroHi
229
230         jsr pintaTablero
231
232         rts
233
```

Con esta nueva rutina “aplicaJugadaGatoHijo” ya deberíamos ser capaces de generar el primer nivel del árbol, es decir, esta parte (ver rojo):



Para ello, lógicamente, habrá que invocar esta nueva rutina desde algún sitio del programa principal (fichero “RYG.asm”). Pero como esta entrada ya ha sido demasiado larga, lo dejamos para la siguiente.

RYG: árbol de juego – generación del primer nivel

Ahora que ya tenemos la capacidad de generar los tableros hijo de un tablero dado, al menos cuando mueven los gatos (rutina “aplicaJugadaGatoHijo”), ya estamos en disposición de generar el primer nivel del árbol de juego. Para ello, lo que vamos a hacer es modificar el programa principal “RYG.asm” con un paso adicional:

```
; Juego del ratón y el gato
; Fichero principal

; Para que los gatos decidan su jugada, crea un árbol de profundidad N

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

        jsr arbolJugadasGatos

        rts ; jmp actualiza

```

Este nuevo paso es la instrucción “jsr arbolJugadasGatos”, y la nueva rutina “arbolJugadasGatos” hace esto:

```
arbolJugadasGatos

        jsr copiaActualRaizArbol

        lda #<raizArbol
        clc
        adc #85
        sta libreLo

        lda #>raizArbol
        adc #0
        sta librehi

        ldx #$00

ajgBucle

        stx ajgNumGato
        jsr arbolJugadasGato

        inx

        cpx #$04
        bne ajgBucle

        rts
```

Es decir:

- Copia el tablero actual a la raíz del árbol (“jsr copiaActualRaizArbol”).
- Incrementa el puntero a la memoria libre en 85 bytes, porque la raíz del árbol ya pasa a estar ocupada por la copia del tablero actual.
- Recorre los cuatro gatos y, para cada uno de ellos, llama a “arbolJugadasGato”.

Por su parte, la rutina “arbolJugadasGato” tiene dos partes. La primera es así:

```
ajgNumGato      byte $00
ajgTempX        byte $00

arbolJugadasGato

        stx ajgTempX

        ; Localiza el gato
        lda #<raizArbol
        sta dgTableroLo

        lda #>raizArbol
        sta dgTableroHi

        lda ajgNumGato
        sta dgNumGato

        jsr dameGato

        ; Convierte a fila y columna
        lda dgOffset
        sta dfcOffset

        jsr dameFilaCol

        ; Genera las jugadas válidas del gato
        lda dfcFila
        sta gjvgFila

        lda dfcColumna
        sta gjvgColumna

        jsr generaJugadasValidasGato

```

Es decir:

- Localiza el gato en el tablero.
- Convierte su posición en formato offset a (fila, columna).
- Y genera las jugadas válidas de ese gato.

Y la segunda parte es así:

```
261
262         ldx #$00
263
264 ajgBucle2
265
266         lda gjvgNuFilas,x
267
268         cmp #$ff
269         beq ajgCont
270
271         lda #<raizArbol
272         sta ajghPadreLo
273
274         lda #>raizArbol
275         sta ajghPadreHi
276
277         lda libreLo
278         sta ajghHijoLo
279
280         lda libreHi
281         sta ajghHijoHi
282
283         lda gjvgFila
284         sta ajghFila
285
286         lda gjvgColumna
287         sta ajghColumna
288
289         lda gjvgNuFilas,x
290         sta ajghNuFila
291
292         lda gjvgNuColumnas,x
293         sta ajghNuColumna
294
295         jsr aplicaJugadaGatoHijo
296
297 ajgCont
298
299         inx
300
301         cpx #$02
302         bne ajgBucle2
303
304         ;lda #13
305         ;jsr chrout
306
307         ldx ajgTempX
308
309         rts
```

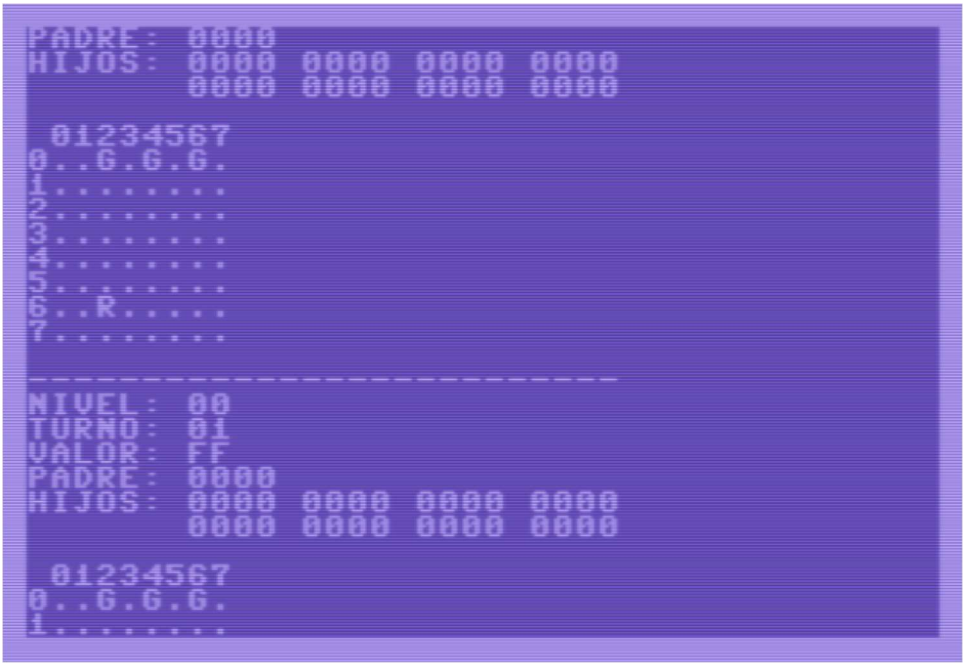
Es decir:

- Recorre las jugadas válidas del gato (0, 1 o 2 jugadas como mucho).
- Y genera un tablero hijo al que se aplica esa jugada.

Al final, al llamar cuatro veces a “arbolJugadasGato”, una vez para cada gato, lo que deberíamos tener es el primer nivel del árbol de juego. Y como, además, según vamos generando los tableros hijos los imprimimos (sólo temporalmente

y para depurar), el resultado es que el primer nivel del árbol debería verse por la pantalla.

Ensamblamos y ejecutamos en VICE. Se pinta el tablero inicial y, como humanos, se nos pide la jugada del ratón. Elegimos una cualquiera (ej. la 00), vemos que efectivamente se aplica y el ratón avanza, y, oh sorpresa, el programa se mete en un bucle infinito del que no sale. ¡¡Y encima desaparece un gato!!



¿Qué pensabais? ¿Que iba a funcionar bien a la primera? Ya empiezan los típicos problemas. A ver quién depura esto ahora… ☺

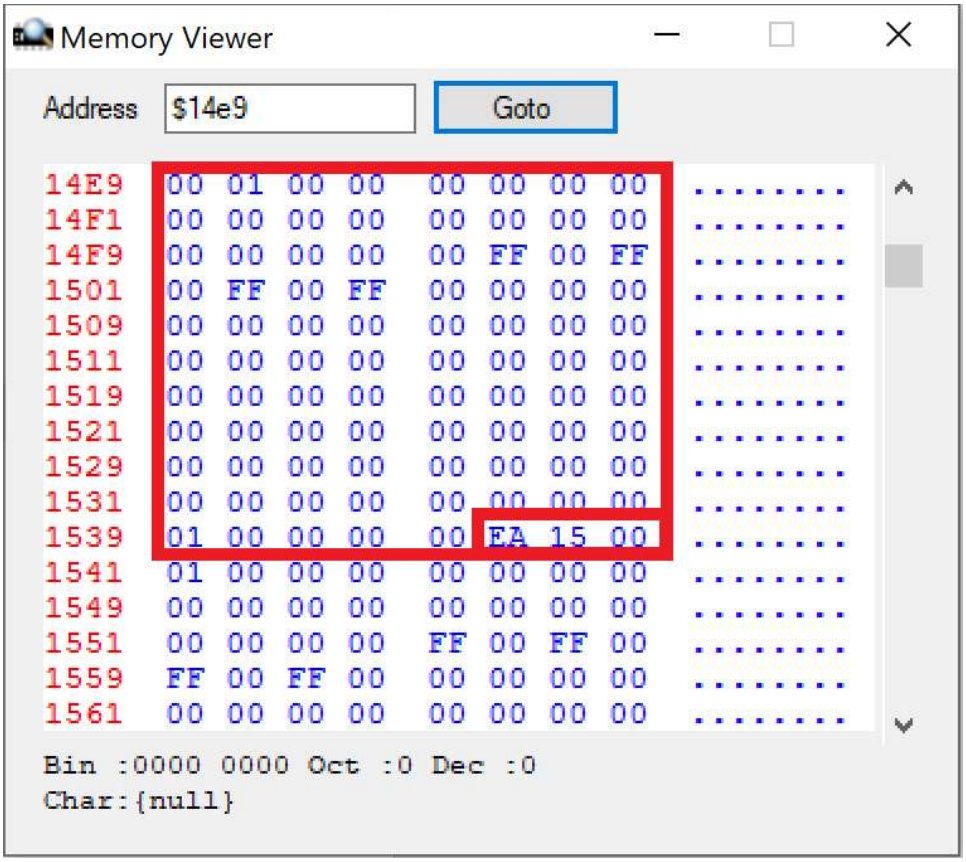
RYG: árbol de juego – tableros de 88 bytes

El primer intento de depurar el programa debe consistir en ir a la zona de memoria donde están el tablero actual y el árbol de juego, y revisar su contenido. Más concretamente, en la versión 7 del proyecto:

- El tablero actual está en $14e9.
- El puntero a la memoria libre está en $153e – $153f.

- La raíz del árbol está en $1540.

Pues bien, si depuramos el programa con el debugger de CMB prg Studio (menú Debugger > Debug Project), y más concretamente abrimos el visor de memoria y nos vamos al tablero actual (dirección $14e9), veremos algo así:



Es decir:

- En la dirección $14e9 comienza el tablero actual, que ocupa 85 bytes.
- En las direcciones $153e – $153f está el puntero a la memoria libre, que toma el valor $15ea, es decir, a partir de la dirección $15ea ya no hay tableros.

- En la dirección $1540, que actualmente toma el valor $00, empieza la raíz del árbol.

La primera conclusión al ver esto es que es muy difícil depurar nada. Si los tableros ocuparan 88 bytes, en vez de 85, ocuparían un número entero de filas en el visor de memoria (88 = 11 x 8), lo que facilitaría mucho la depuración de la memoria.

Por ello, la primera medida que vamos a tomar es ampliar la estructura de datos del tablero para hacerla pasar de 85 a 88 bytes. De este modo los tableros nos saldrán alineados en el visor.

No necesitamos guardar más variables. Se trata, simplemente, de alinear los datos. Por ello, si además metemos un patrón fácilmente reconocible, por ejemplo, el prefijo $aa – $aa – $aa, no sólo conseguimos alinear los datos en el visor, sino que además metemos un patrón que es fácilmente reconocible visualmente, lo que nos facilita identificar donde termina un tablero y empieza el siguiente.

Por todo lo dicho, finalmente la estructura de datos para manejar tableros queda así (ver prefijo $aa – $aa – $aa):

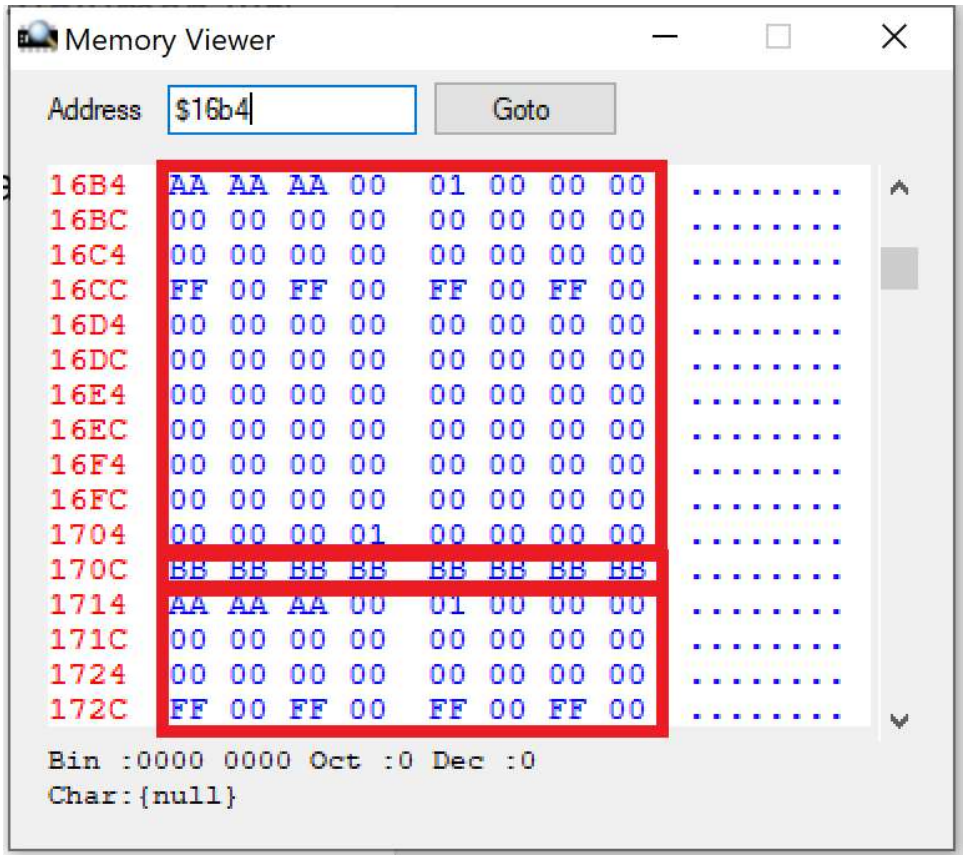
```
; Estructura de datos (88 bytes)

;relleno byte $aa
;relleno byte $aa
;relleno byte $aa

;nivel   byte $00
;turno   byte $00
;valor   byte $00

;padreLo byte $00
;padreHi byte $00

;hijosLo byte $00,$00,$00,$00,$00,$00,$00,$00
;hijosHi byte $00,$00,$00,$00,$00,$00,$00,$00

;tablero byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00,
;        byte $00,$00,$00,$00,$00,$00,$00,$00
```

Lógicamente, también hay que actualizar todas las rutinas que manejan tableros (fichero “Tableros.asm”), ya que las posiciones (offsets) de las variables respecto del comienzo del tablero se incrementan en 3 bytes.

Para que el puntero a la memoria libre (libreLo – libreHi) no estropee el alineamiento entre el tablero actual y los tableros del árbol, también se introducen algunos bytes extra en “Arbol.asm”:

```
; Fichero para el tablero actual y el árbol de jugadas
; Debe ir al final de los programas

; Tablero actual (88 bytes)

tableroActual

        ; Relleno para completar hasta 88 bytes (11x8)
        byte $00
        byte $00
        byte $00

        ; Nivel, turno y valor
        byte $00
        byte $00
        byte $00

        ; Padre
tableroActualPadreLo

        byte $00
        byte $00

        ; Hijos
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00

        ; Tablero propiamente dicho
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00
        byte $00,$00,$00,$00,$00,$00,$00,$00

; Puntero para control de la memoria libre

libreLo byte $bb
libreHi byte $bb

        byte $bb,$bb,$bb,$bb,$bb,$bb

; Árbol de decisión para los gatos

raizArbol

```

De este modo, en la versión 8 del proyecto los tableros ya aparecen alineados en el visor de memoria, lo que facilita mucho la depuración:

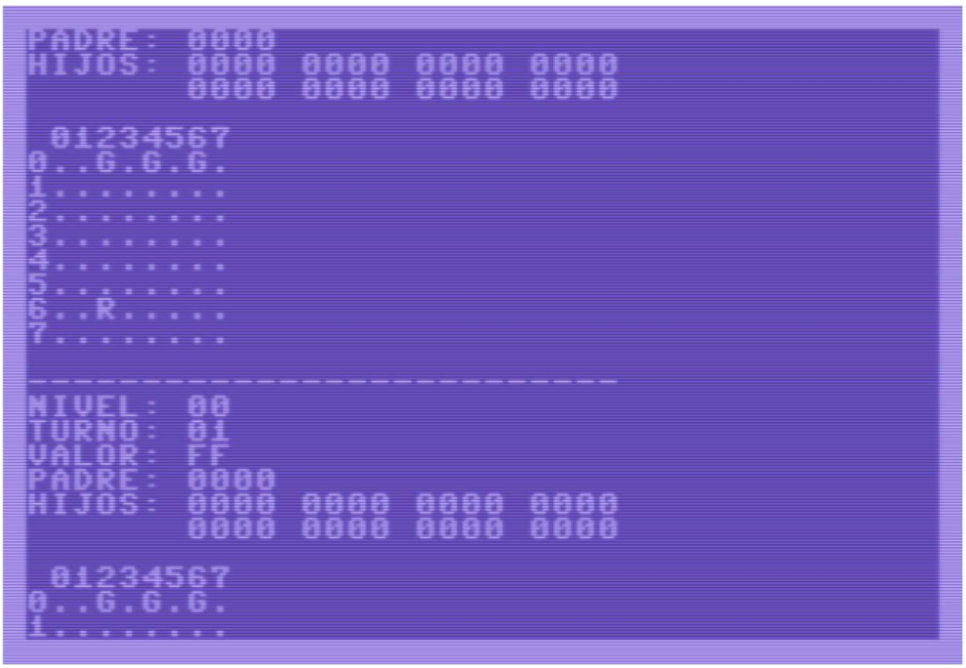


En esta imagen ya es más fácil identificar el tablero actual ($16b4), el puntero a la memoria libre ($170c - $170d), seis bytes $bb de relleno ($170e - $1713), y la primera parte de la raíz del árbol de juego ($1714), que es copia del tablero actual.

RYG: árbol de juego – nueva validación de jugadas

Una vez ampliados los tableros a 88 bytes, y alineados los datos en el visor de memoria, hay que seguir depurando el programa.

Una cosa que llama la atención de la versión 7 del proyecto, aparte de que se mete en un bucle, es que uno de los gatos desparece del tablero:



Esto nos puede llevar a pensar que la validación de las jugadas no es correcta. Recordemos que básicamente validábamos dos cosas (aparte de que el movimiento de la pieza era conforme a las reglas del juego):

- Que la casilla de destino caía dentro del tablero.
- Que la casilla de destino estaba vacía.

Quizás lo que ocurre no es tanto que ha desparecido un gato, sino que hay dos superpuestos en la misma casilla. Es decir, quizás está fallando la validación de las jugadas.

Y efectivamente, cuando hicimos la validación de jugadas ya adelantamos un presagio que de momento no hemos cumplido:

<<La rutina anterior tiene una limitación que de momento es aceptable: sólo trabaja sobre el tablero actual (que está implícito). Pero cuando construyamos el

árbol de juego para que el C64 juegue por los gatos hará falta validar jugadas sobre un tablero arbitrario del árbol. Llegados ese momento habrá que mejorarla.>>

Por tanto, tenemos que mejorar la rutina de validación de jugadas que, recordemos, es común para ratón y gatos (fichero "GenJugadasRaton.asm"). La principal mejora es hacerle llegar el tablero contra el que hacer la validación, es decir, el tablero contra el que mirar si la posición de destino está vacía u ocupada. La validación de límites es independiente del tablero.

La nueva rutina "validaJugada" pasa a tener esta definición (recibe un tablero como parámetro de entrada):

```
38  ; Rutina para validar una jugada
39
40  ; Para que una jugada sea válida el destino tiene que caer dentro
41  ; del tablero y estar libre; vale tanto para ratón como para gatos
42
43  vjTableroLo     byte $00
44  vjTableroHi     byte $00
45  vjNuFila        byte $00
46  vjNuColumna     byte $00
47  vjValida        byte $00
48
49  validaJugada
50
```

Y lo que cambia es la parte en que valida si el destino está vacío u ocupado, que lógicamente pasa a utilizar el tablero que recibe como parámetro:

```
vjOcu

        lda vjTableroLo
        sta dcTableroLo

        lda vjTableroHi
        sta dcTableroHi

        lda vjNuFila
        sta dcFila

        lda vjNuColumna
        sta dcColumna

        jsr dameContenido

        lda dcFicha
        cmp #Vacio

        beq vjOK

        rts

vjOK

        lda #$00
        sta vjValida

        rts

```

Por supuesto, no llega con cambiar la rutina “validaJugada” para que reciba un tablero. También hay que cambiar las rutinas que, bien de forma directa o indirecta, llaman a aquella. Ahora deberán pasar un tablero como parámetro.

Con este cambio, la versión 8 del proyecto ya debería funcionar y generar bien el primer nivel del árbol. No obstante, la versión 8 recoge algunos otros cambios variopintos que, por no alargar esta entrada, se recogerán en la siguiente.

RYG: árbol de juego – algunas mejoras

Hace unas pocas entradas, en la versión 7 del proyecto, desarrollamos la rutina “aplicaJugadaGatoHijo”. El ratón tendrá una rutina equivalente un poco más adelante.

Esta rutina permite aplicar una jugada de un gato a un tablero. Pero, en vez de modificar directamente ese tablero, genera un tablero hijo. De este modo, permite ir generando el árbol de juego que hace falta para que el C64 decida su jugada.

Pues bien, entonces decíamos que esta rutina hacía:

<<

- *Saca una copia del tablero de entrada (ahora la raíz del árbol) en un nuevo tablero que será el hijo. Para esto hace falta una nueva rutina de apoyo "copiaTablero" en el fichero "Tableros.asm".*
- *Actualiza el puntero a la memoria libre (libreLo – libreHi), puesto que hemos creado un nuevo tablero en el árbol.*
- *Sobre el tablero hijo, aplica el movimiento.*
- *Más adelante, enlazará el tablero padre con el tablero hijo, y viceversa, aunque en esta fase del proyecto esto todavía no está implementado.*
- *A modo de depuración, y de forma temporal, imprime el tablero hijo.*

>>

Lo anterior admite varias mejoras, a saber:

Copia del tablero padre en el hijo:

La rutina "copiaTablero" copia el tablero origen en el destino tal cual. Ahora bien, hay campos del tablero origen, como los hijos, que no tienen sentido en el tablero de destino.

Podemos esperar a que el programa vaya "machacando" esos campos, y dándoles nuevos valores con sentido o, para no crear confusión, inicializarlos en el hijo desde el comienzo.

Por esto último se desarrolla la rutina de apoyo "copiaTableroSinHijos" (fichero "Tableros.asm"), que en el fondo es igual que "copiaTablero" pero inicializando los hijos del destino.

Incrementar nivel y turno:

Más ejemplos de campos que no tiene sentido copiar tal cual del padre al hijo son los campos "nivel" y "turno". El nivel hay que incrementarlo en uno del padre al hijo. Y el turno hay que ir alternándolo entre ratón y gatos.

Estas modificaciones se podrían hacer en la de rutina de copia (ya sea la original "copiaTablero" o la nueva "copiaTableroSinHijos"). Pero como también hay que hacerlas cuando se aplica una jugada directamente a un tablero, sin dar lugar a un tablero hijo (por ejemplo, cuando el humano decide su jugada y hay que aplicarla al tablero actual), estos cambios se llevan a las rutinas "aplicaJugadaGato" y "aplicaJugadaRaton".

Ahora estas dos rutinas incluyen una llamada a la nueva rutina de apoyo "incrementaNivelYTurno" (fichero "Tableros.asm"). Como esta rutina no tiene mucho misterio, no abundaremos más en ella.

Vincular padre e hijos:

Para construir un árbol de verdad no sólo hay que generar tableros. Además, hay que vincular el padre con los hijos y al revés.

Vincular un hijo con su padre es fácil porque, como dicen, "padre no hay más que uno" (en realidad, se dice de las madres que es con las que no hay duda ;-)). Simplemente, se trata de poner la dirección del padre (de momento la raíz) en el campo "padre" de todos los hijos. Para ello se utiliza la rutina de apoyo "fijaPadre".

Vincular a un padre con sus hijos ya es más complicado, puesto que un padre puede tener varios hijos (hasta ocho). Y algunos hijos estarán "ocupados" y otros no. Por tanto, tampoco resulta muy práctico usar la rutina "fijaHijo", puesto que esta rutina exige saber qué número de hijo (primero, segundo, …) es el que se quiere fijar.

Por ello, resulta de utilidad desarrollar una nueva rutina "fijaHijoLibre" que, simplemente, va analizando los hijos de un padre hasta dar con el primero libre (valor $0000) y, una vez localizado su número de orden, guarda en él la información del hijo con "fijaHijo". Ver fichero "Tableros.asm".

Pintado de tableros con dirección:

Para depurar el árbol vamos a hacer dos cosas:

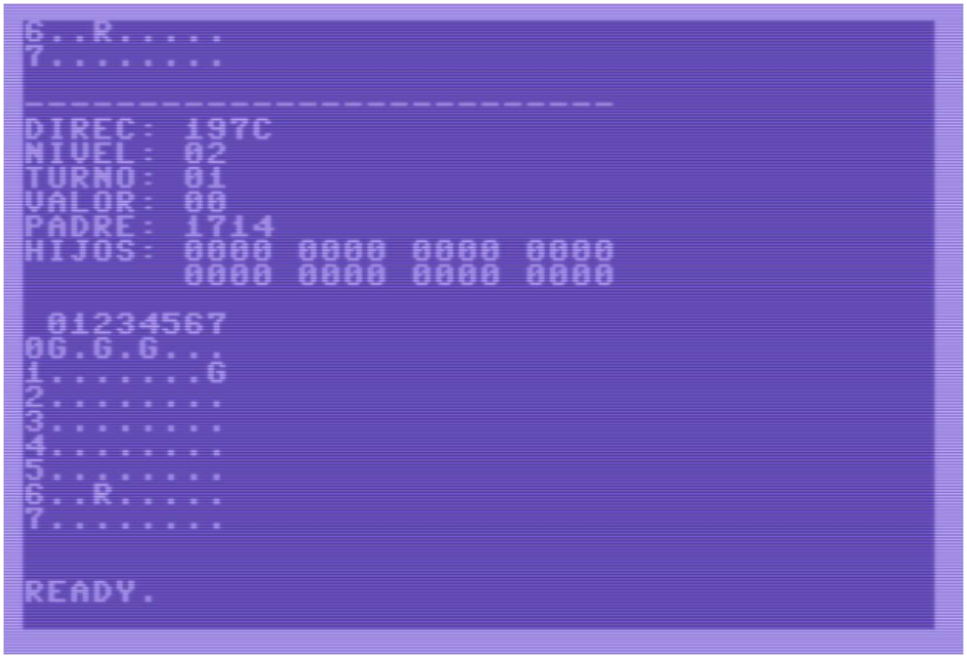
- Revisar la memoria del C64 con el depurador y el visor de memoria. Para esto alineamos los tableros a un múltiplo de 8 bytes.

- Simplemente, pintar el árbol por pantalla.

Y puesto que los padres identifican a sus hijos por sus direcciones, y al revés también, no hay nada más lógico que empezar pintando los tableros por su dirección. Por ello, se mejora la rutina “pintaTablero” (fichero “PintaTableros.asm”) con un nuevo paso “jsr pintaDireccion”.

Resultado final:

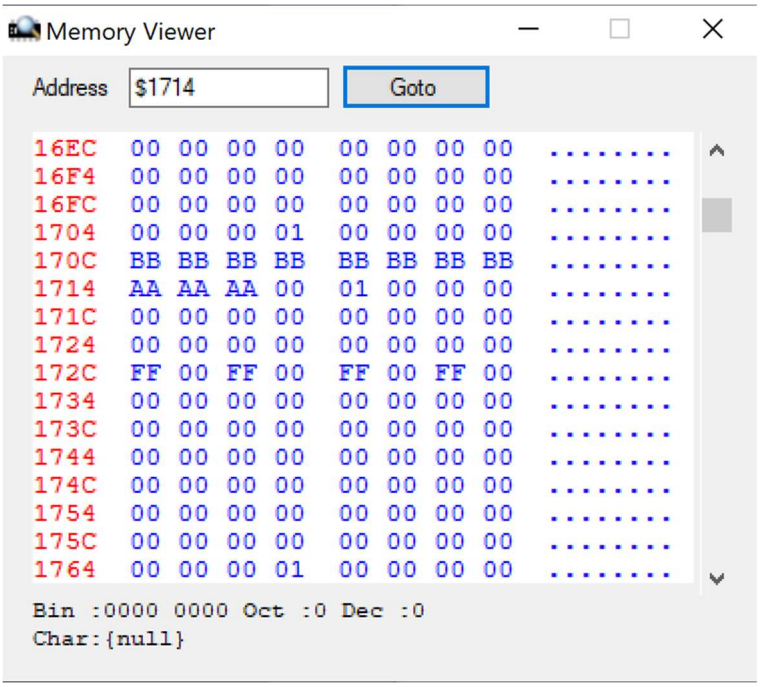
Tras todas estas mejoras, si ejecutamos la versión 8 del proyecto, el resultado es:



Es decir, se van generando las siete jugadas válidas de los gatos, y cada una de ellas se aplica a un tablero hijo, que queda identificado mediante su dirección (ej. $197c), y que queda enlazado con su tablero padre ($1714) que, de momento, y al tener sólo un nivel, es la raíz del árbol.

La ejecución en VICE se puede ir parando si se va pinchando con el ratón sobre la barra de menú de VICE, y se pueden ir comprobando todos los tableros hijo.

También es posible usar el depurador de CBM prg Studio, ejecutar el programa, irse con el visor de memoria a la zona de memoria del entorno de la raíz ($1714), y comprobar que están los tableros:



Si se hace esto, hay que tener cuidado con las impresiones por pantalla porque el depurador de CBM prg Studio no incluye por defecto en su mapa de memoria las rutinas del Kernal como “chrout”. Por ejemplo, se pueden comentar o saltar.

RYG: árbol de juego – selección de la profundidad

Una vez que hemos sido capaces de generar el primer nivel del árbol de juego, el siguiente paso ya es generar N niveles, tantos como la profundidad de análisis que se pretenda.

Y como la profundidad de análisis suele ser un parámetro de configuración del juego, se le suele pedir al usuario al arrancar. Y ya de paso le vamos a poner un titulillo, aunque sea cosa sencilla.

Por todo ello, en el programa principal, que está en el fichero “RYG.asm” aparecen estas dos llamadas nuevas (“jsr pintaTitulo” y “jsr solicitaProfundidad”):

```
; Juego del ratón y el gato
; Fichero principal

; Esta versión hace:
;
; - El arbol de jugadas no se limita a un nivel, se extiende a N niveles de
;   profundidad.

; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

        jsr desarrollaArbolJugadas;arbolJugadasGatos

        rts ; jmp actualiza

```

La primera llamada (“jsr pintaTitulo”) es bien sencilla, y se limita a pintar un título:

```
45  pintaTitulo
46
47          lda #13
48          jsr chrout
49
50          lda #<ptTitulo
51          sta cadenaLo
52
53          lda #>ptTitulo
54          sta cadenaHi
55
56          jsr pintaCadena
57
58          lda #13
59          jsr chrout
60
61          rts
62
63  ptTitulo        text "*raton y gatos - home vic software 2020*"
64                  byte $00
65
```

La segunda llamada (“jsr solicitaProfundidad”) es un poco más compleja, pero tampoco mucho:

```
66  solicitaProfundidad
67
68          lda #<spProfundidad
69          sta cadenaLo
70
71          lda #>spProfundidad
72          sta cadenaHi
73
74          jsr pintaCadena
75
76          jsr leeTeclado
77
78          lda #13
79          jsr chrout
80
81          lda byteLeido
82
83          cmp #$01
84          bcc solicitaProfundidad
85
86          cmp #$06
87          bcs solicitaProfundidad
88
89          sta prof
90
91          rts
92
93  spProfundidad   text "profundidad (01-05)? "
94                  byte $00
95
```

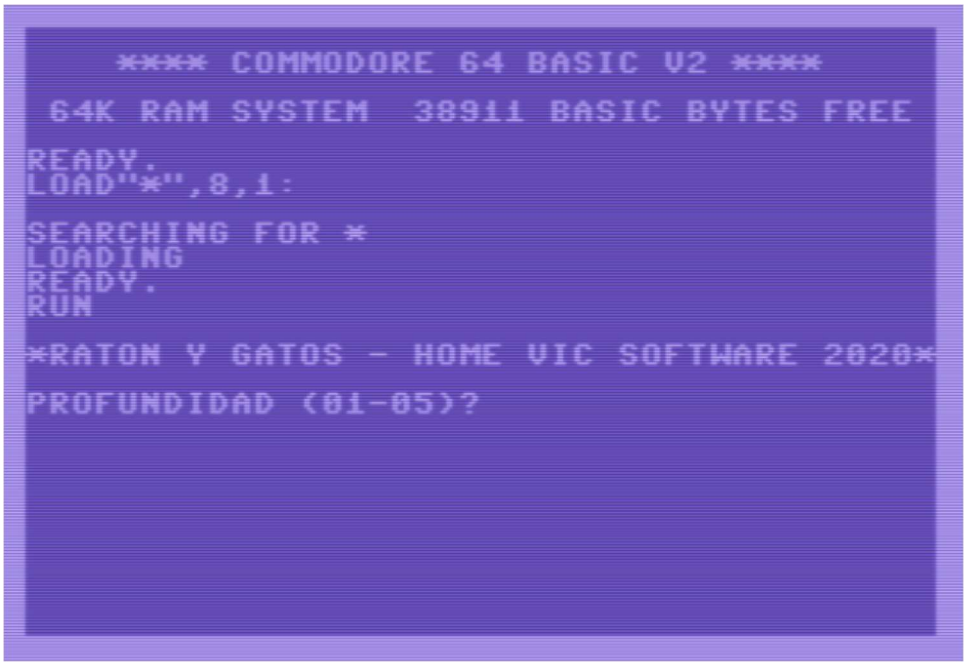
La rutina pinta una cadena, lee un byte del teclado, pinta un retorno de carro (13), verifica que el byte leído sea mayor o igual que 1 y menor que 6 y, en caso

afirmativo, lo deposita en la dirección “prof”. En caso negativo, vuelve a pedir la profundidad.

A estas alturas de la programación todavía no había echado las cuentas de memoria que vimos en la entrada “RYG: programa principal” así que todavía no era consciente de que con la codificación de 88 bytes por tablero el máximo de niveles era tres. Por eso en el programa veréis que se pide una profundidad entre 1 y 5.

Más adelante, cuando veamos la función de evaluación y cómo mejorarla, veremos que otra alternativa –o más bien otra medida complementaria– es meter más niveles de profundidad de análisis, lo que en nuestro caso pasaría inevitablemente por “comprimir” los tableros, porque el C64 tiene muy poca memoria.

Con estas dos pequeñas mejoras del programa principal, el arranque queda así:



En la siguiente entrada, ya con la profundidad seleccionada por el usuario en "prof", intentaremos generar un árbol de juego de esa profundidad, no de un solo nivel.

RYG: árbol de juego – recursividad

En matemáticas una función recursiva es una función que se define en términos de sí misma. Hay muchos ejemplos, pero uno típico es la secuencia de Fibonacci, que se define así:

$$F(N) = F(N-1) + F(N-2)$$

Es decir, el número de Fibonacci de orden N es la suma de los números de órdenes N-1 y N-2. Como se puede observar, la secuencia se define en términos de sí misma.

Para que la definición esté completa hace falta una condición inicial, es decir, hace falta indicar el valor de los dos primeros números de Fibonacci, que son:

$$F(0) = 0$$

$$F(1) = 1$$

De este modo, es fácil calcular:

$$F(2) = F(1) + F(0) = 1 + 0 = 1$$

$$F(3) = F(2) + F(1) = 1 + 1 = 2$$

$$F(4) = F(3) + F(2) = 2 + 1 = 3$$

$$F(5) = F(4) + F(3) = 3 + 2 = 5$$

$$F(6) = F(5) + F(4) = 5 + 3 = 8$$

$$F(7) = F(6) + F(5) = 8 + 5 = 13$$

…

En programación, igualmente, una función recursiva es una función que se llama a sí misma. Por ejemplo, el siguiente programa en C incluye la función recursiva int fibonacci(int n) que permite calcular la serie de Fibonacci:

```c
#include <stdio.h>
#include <conio.h>
#include <c64.h>
#include <cbm_petscii_charmap.h>

int fibonacci(int n)
{
  if (n==0)
    return 0;

  if (n==1)
    return 1;

  return fibonacci(n-1) + fibonacci(n-2);
}

void main(void) {
  int i;

  for (i=0; i<10; i++)
  {
        printf("%i", fibonacci(i));
        printf("\n");
  }
}
```

Es importante empezar la función recursiva verificando las condiciones de terminación (¿n==0? ¿n==1?), porque si no se corre el riesgo de que el programa se meta en un bucle infinito.

Cuando un problema es de naturaleza recursiva, su programación de forma recursiva también suele resultar natural. No obstante, casi todo lo que se puede programar de forma recursiva también se puede programar de forma iterativa, es decir, usando bucles. Por ejemplo, la versión iterativa del mismo programa en C sería así:

```c
#include <stdio.h>
#include <conio.h>
#include <c64.h>
#include <cbm_petscii_charmap.h>

void main(void) {
  int fib[10];
  int i;

  fib[0]=0;
  fib[1]=1;

  for (i=2; i<10; i++)
  {
        fib[i] = fib[i-1] + fib[i-2];
  }

  for (i=0; i<10; i++)
  {
        printf("%i", fib[i]);
        printf("\n");
  }
}
```

¿Y todo esto a santo de qué viene? Pues que un árbol es una estructura de datos recursiva. Así que vamos a usar funciones recursivas para generar el árbol de juego.

¿Y se pueden hacer rutinas recursivas en ensamblador? Pues no es muy habitual, pero poderse se puede, y lo vamos a demostrar en las entradas que siguen.

RYG: árbol de juego – resto de niveles

Desde la entrada “RYG: árbol de juego – generación del primer nivel” ya somos esencialmente capaces de generar el primer nivel del árbol de juego, aunque luego hemos tenido que depurar y meter algunas correcciones y mejoras.

La principal novedad de aquella entrada era la rutina “arbolJugadasGatos” que:

- Copiaba el tablero actual a la raíz del árbol.
- Recorría los cuatro gatos y, para cada uno de ellos, llamaba a la rutina “arbolJugadasGato”.

- La rutina “arbolJugadasGato”, a su vez, generaba tableros hijo con las jugadas válidas de cada gato.

Ahora toca generar el segundo nivel del árbol (y los siguientes). En el segundo nivel vuelve a ser turno del ratón. Por tanto, necesitaremos una rutina “arbolJugadasRaton” equivalente a “arbolJugadasGato”, es decir, una rutina que genere tableros hijo con las jugadas válidas del ratón.

Y ahora se trata de repetir el proceso de generación de jugadas / tableros hijo tantas veces como indique la profundidad de análisis seleccionada. Y lógicamente hay que ir alternando quién mueve, gatos o ratón:

- Copiar el tablero actual en la raíz del árbol.
- Partiendo de la raíz, generar las jugadas / tableros hijo de los cuatro gatos.
- Partiendo de los tableros hijo, generar las jugadas / tableros hijo del ratón.
- Partiendo de los tableros nieto, generar las jugadas / tableros hijo de los cuatro gatos.
- Partiendo de los tableros bisnieto, generar las jugadas / tableros hijo del ratón.
- …
- Y así hasta la profundidad indicada.

Y esto, precisamente, podemos conseguirlo con una rutina recursiva. Algo así:

- Copiar el tablero actual en la raíz del árbol.
- RutinaRecursiva(tablero, profundidad):
    - Si la profundidad ha llegado a cero, termina.
    - Si es turno de los gatos:
        - Genera las jugadas / tableros hijo de los cuatro gatos.
        - Recorre todos los tableros hijo.
        - Para cada tablero hijo, llama a RutinaRecursiva(tablero-hijo, profundidad-1).
    - Si es turno del ratón:
        - Genera las jugadas / tableros hijo del ratón.

- Recorre todos los tableros hijo.
- Para cada tablero hijo, llama a RutinaRecursiva(tablero-hijo, profundidad-1).

Como se puede observar, “RutinaRecursiva” efectivamente es recursiva, puesto que se llama a sí misma. Se va llamando a sí misma cada vez con menos profundidad (profundidad, profundidad-1, profundidad-2, profundidad-3, ...). Y cuando la profundidad llega a cero, termina. De este modo, es capaz de generar todo el árbol de juego hasta la profundidad seleccionada por el usuario, y alternando los turnos de movimiento.

En la práctica, en la versión 9 del proyecto esto se plasma en las rutinas:

Rutina “desarrollaArbolJugadas”:

Esta rutina es así:

```
247 desarrollaArbolJugadas
248
249         jsr copiaActualRaizArbol
250
251  lda #<raizArbol
252  sta ptTableroLo
253
254  lda #>raizArbol
255  sta ptTableroHi
256
257  jsr pintaTablero
258
259         lda #<raizArbol
260         clc
261         adc #88
262         sta libreLo
263
264         lda #>raizArbol
265         adc #0
266         sta librehi
267
268         lda #<raizArbol
269         sta dunTableroLo
270
271         lda #>raizArbol
272         sta dunTableroHi
273
274         lda prof
275         sta dunProf
276
277         lda #$00
278         sta dunNumHijo
279
280         jsr desarrollaUnNivel
281
282         rts
283
```

Es decir, trazas de depuración aparte (llamada a “pintaTablero”), esta rutina:

- Copia el tablero actual en la raíz del árbol.
- Lógicamente, incrementa el puntero a la memoria libre, puesto que la raíz del árbol ocupa memoria.
- Llama a la rutina recursiva “desarrollaUnNivel” pasando como tablero de partida la raíz y como profundidad la elegida por el usuario (“prof”).

<u>Rutina recursiva “desarrollaUnNivel”:</u>

Esta rutina, nuevamente contiene trazas de depuración (llamadas a “pintaHex”). Al margen de eso, primero comprueba si la profundidad ha llegado a cero:

```
dunTableroLo    byte $00
dunTableroHi    byte $00
dunProf         byte $00
dunNumHijo      byte $00

dunTempX        byte $00

desarrollaUnNivel

        stx dunTempX

 lda dunProf
 sta numeroHex
 jsr pintaHex

 lda dunNumHijo
 sta numeroHex
 jsr pintaHex

 lda #13
 jsr chrout

        lda dunProf
        cmp #$00
        bne dunOtroNivel

        ldx dunTempX

        rts

dunOtroNivel
```

Si la profundidad ha llegado a cero termina (instrucción “rts”). Si no ha llegado a cero, genera otro nivel del árbol (etiqueta “dunOtroNivel”).

Para generar otro nivel, lo primero es determinar el turno de juego con la rutina “dameDatosBasicos”:

```
dunOtroNivel

        lda dunTableroLo
        sta ddbTableroLo

        lda dunTableroHi
        sta ddbTableroHi

        jsr dameDatosBasicos

        lda ddbTurno
        cmp #Raton
        ;beq dunRaton
        bne dunGatos
        jmp dunRaton

dunGatos
```

Si el turno es de los gatos se ejecuta el código bajo la etiqueta “dunGatos”; si el turno es del ratón se ejecuta el código bajo la etiqueta “dunRaton”. En ambos casos el código es muy parecido, así que sólo veremos el de los gatos:

```
330 dunGatos
331
332         lda dunTableroLo
333         sta ajgTableroLo3
334
335         lda dunTableroHi
336         sta ajgTableroHi3
337
338         jsr arbolJugadasGatos
339
340         ldx #$00
341
342         lda dunTableroLo
343         sta dhTableroLo
344
345         lda dunTableroHi
346         sta dhTableroHi
347
348 dunGatosBucle
349
350         stx dhNumHijo
351
352         jsr dameHijo
353
354         lda dhHijoLo
355         sta dunTableroLo
356
357         lda dhHijoHi
358         sta dunTableroHi
359
360         beq dunGatosSgteHijo
361
362         lda dunProf
363         sec
364         sbc #$01
365         sta dunProf
366
367         stx dunNumHijo
368
369         jsr desarrollaUnNivel
370
371         lda dunProf
372         clc
373         adc #$01
374         sta dunProf
375
376 dunGatosSgteHijo
377
378         inx
379
380         cpx #$08
381         bne dunGatosBucle
382
383         ldx dunTempX
384         rts
```

Es decir:

- Genera las jugadas válidas / tableros hijo con “arbolJugadasGatos”.
- Recorre los tableros hijo (hasta un máximo de ocho).
- Se llama a sí misma pasando como parámetros el tablero hijo y profundidad – 1.

Si se revisa la parte que sigue a la etiqueta “dunRaton” es totalmente análoga:

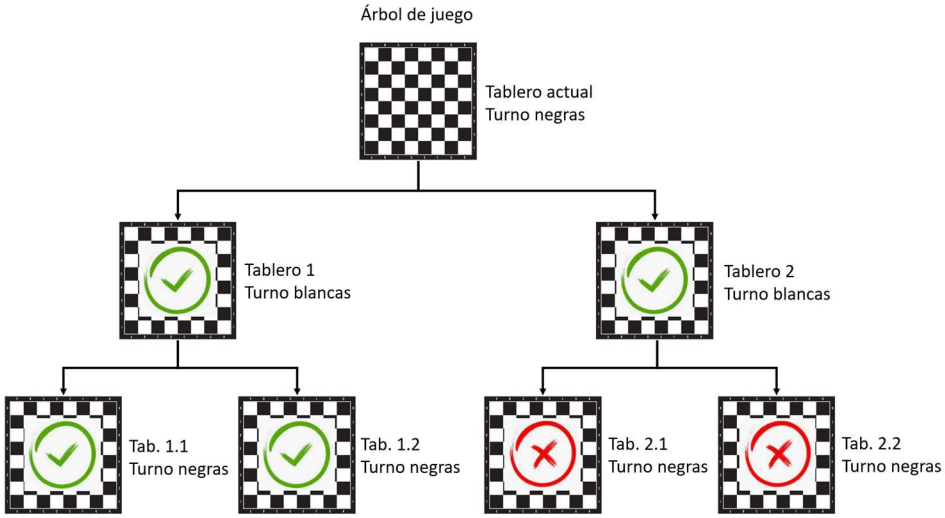
- Genera las jugadas válidas / tableros hijo con “arbolJugadasRaton”.
- Recorre los tableros hijo.
- Se llama a sí misma pasando como parámetros el tablero hijo y profundidad – 1.

Al llamarse a sí misma sobre cada tablero hijo (y con profundidad – 1) lo que se consigue es seguir poblando el árbol recursivamente.

Resultado:

Si se prueba la versión 9 del proyecto se verá que, con profundidad uno, el programa funciona perfectamente: genera bien los 7 tableros con los movimientos válidos de los gatos (nivel 1).

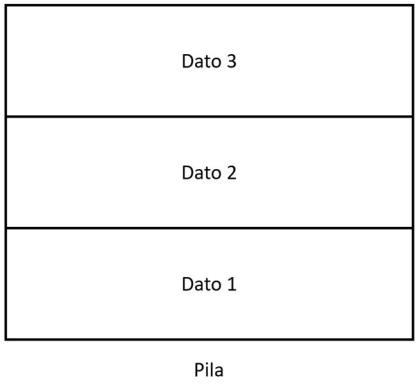
Sin embargo, si se prueba con profundidad 2 (o superiores), se verá que se generan bien los 7 tableros con los movimientos de los gatos (nivel 1), pero los movimientos subsiguientes del ratón (nivel 2) sólo se generan para el primer movimiento de los gatos. Es decir, algo así:



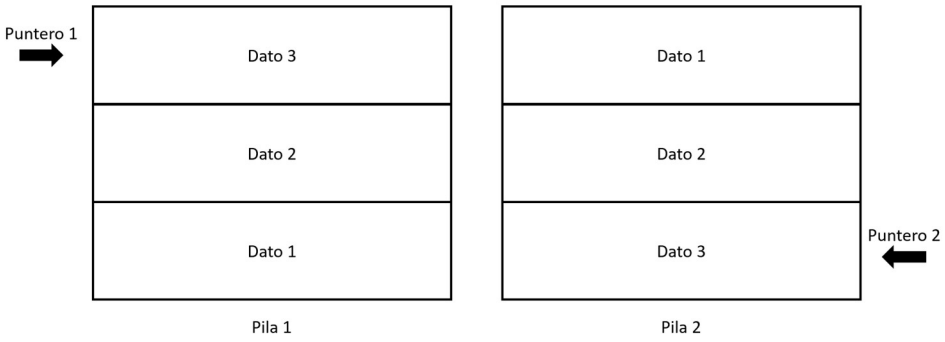
El motivo radica en que hacer funciones recursivas en lenguajes de alto nivel como C o Java es algo más fácil, pero en ensamblador hay que tomar precauciones adicionales. En la entrada siguiente veremos por qué.

RYG: árbol de juego – recursividad y ensamblador

En los lenguajes de programación de alto nivel (C, Java, etc.) las llamadas a rutinas, funciones, procedimientos, métodos, o como queramos llamarlo, se implementan mediante una pila de llamadas. Una pila es una estructura de datos LIFO – List In First Out, es decir, el último dato que se mete en la pila tiene que ser el primero en salir:



Las pilas las podemos pintar creciendo o apilando hacia arriba o hacia abajo. Lo más natural resulta pintarlas creciendo hacia arriba, ya que en la naturaleza las cosas se apilan así (la gravedad tira hacia abajo ;-)), pero también se pueden pintar al revés. De hecho, hay pilas como la del C64 (página uno) que crecen desde direcciones más altas (que se suelen pintar arriba) hacia direcciones más bajas (que se suelen pintar abajo). Pero da igual cómo las pintemos, creciendo hacia arriba o hacia abajo, lo realmente importante es su funcionamiento LIFO:

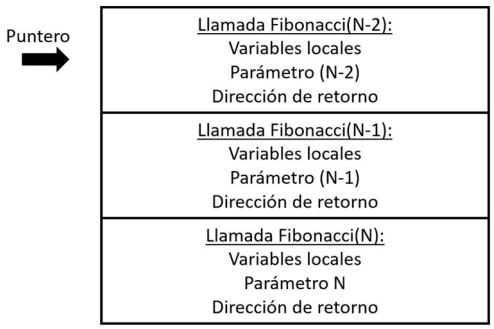


En cualquier caso, en la pila de llamadas se mete, por cada función llamada:

- La dirección de retorno: Esto permite continuar la ejecución del programa en el sitio correcto cuando termine la función.

- Los parámetros con que fue llamada la función.
- Variables y constantes de carácter local a la función.

Si ahora aplicamos esto a una función recursiva, es decir una función que se llama a sí misma, la pila tendría una apariencia así:



Pila de llamadas

Lo que es importante observar es que cada llamada a la función recursiva tiene su propia copia de los parámetros de entrada. Por tanto, el programa no se pierde: sabe qué función está ejecutando, con qué parámetros, y dónde volver o continuar.

Sin embargo, si llevamos esto al C64, su pila (página uno), y las llamadas a rutinas en ensamblador con “jsr”, lo primero que observamos es que los parámetros no se pasan a las rutinas mediante la pila (aunque se podría hacer). Lo normal es pasar los parámetros:

- Mediante los registros del microprocesador (acumulador, registro X y registro Y).
- O mediante posiciones de memoria, ya sean de la página cero o de cualquier otra página.

Las rutinas del Kernal suelen funcionar pasando parámetros en los registros del microprocesador. A mí me parece más claro, y por tanto me gusta más, usar posiciones de memoria.

Lo importante es observar que, si se usan posiciones de memoria (o registros del microprocesador), no hay una copia de los parámetros por cada llamada a la rutina recursiva. Todas las llamadas comparten las mismas posiciones de memoria y, por tanto, según se van produciendo las llamadas recursivas unos valores de entrada van "machacando" a los anteriores. Esto puede hacer que las rutinas recursivas "se pierdan".

Y esto es, precisamente, lo que ocurría en la entrada anterior. Y por este motivo el árbol de juego no se generaba de forma completa, sino que sólo se generaba en su máxima profundidad (la elegida por el usuario) en la primera rama, quedándose incompletas el resto de ramas.

La solución pasa porque los parámetros de entrada a la rutina "desarrollaUnNivel" no sean posiciones de memoria simples, sino tablas. De este modo, si usamos la profundidad (que es la variable que sirve para controlar las sucesivas llamadas recursivas y cuándo se ha llegado al final) como índice para acceder a estas tablas, cada ejecución de la rutina puede acceder a los parámetros de entrada que le corresponden:

```
290 dunProf         byte $00
291 dunTableroLo    byte $00,$00,$00,$00,$00,$00,$00,$00
292 dunTableroHi    byte $00,$00,$00,$00,$00,$00,$00,$00
293 dunNumHijo      byte $00,$00,$00,$00,$00,$00,$00,$00
294
295 ;dunTempX        byte $00
296
297 desarrollaUnNivel
298
299         lda dunProf
300         tay
301
302         cmp #$00
303         bne dunOtroNivel
304
305         rts
306
```

Al cambiar la definición de la rutina (sus parámetros de entrada) lógicamente hay que adaptar las llamadas a la misma, tanto desde el programa principal "RYG.asm" como las llamadas recursivas que se hace a sí misma.

Y con esta solución la versión 10 del proyecto ya no "se pierde" y el árbol de juego se genera de forma completa independientemente del nivel de profundidad elegido por el usuario (1, 2 ó 3):

El siguiente paso ya será evaluar el árbol de juego, para que el C64 pueda tomar una decisión sobre qué jugada realizar. Pero antes de eso haremos algunas reflexiones sobre la memoria que ocupa el árbol, cuántos niveles de profundidad admite, cómo se podrían admitir más niveles, etc.

RYG: árbol de juego – memoria revisitada

En la entrada "RYG: programa principal" ya hicimos una estimación de la profundidad de análisis que puede tener el juego. Si actualizamos aquellos cálculos para tableros de 88 bytes quedan así:

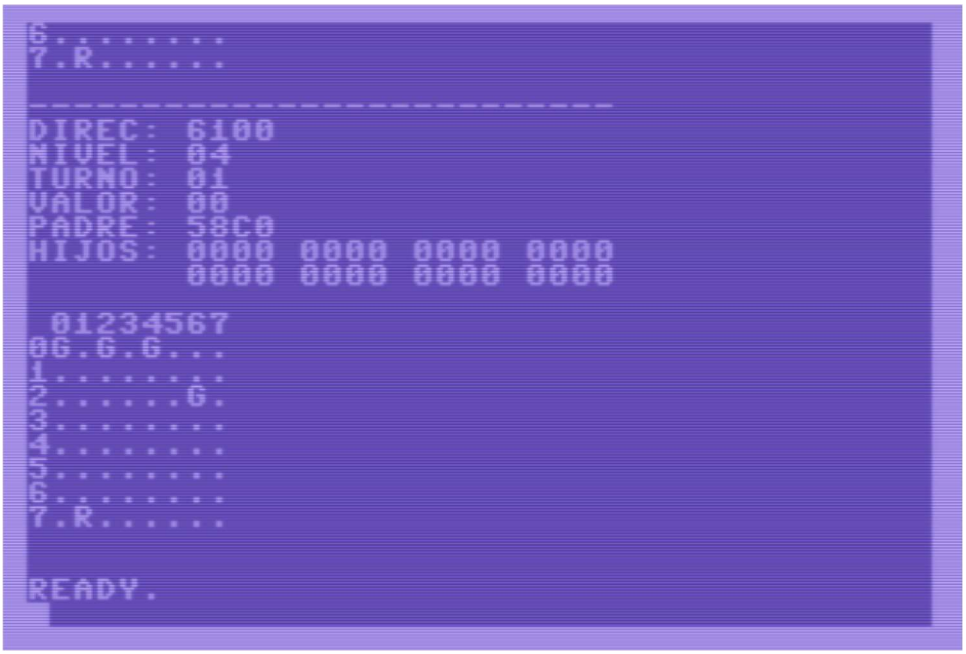
- 1 nivel → 88 bytes x 8 movimientos = 704 bytes.
- 2 niveles → 88 bytes x 8 x 4 = 2816 bytes.
- 3 niveles → 88 bytes x 8 x 4 x 8 = 22528 bytes.
- 4 niveles → 88 bytes x 8 x 4 x 8 x 4 = 90112 bytes.

Por tanto, para trabajar con 4 niveles de profundidad necesitaríamos unos 90 KB. Y lógicamente el C64 no los tiene.

Pero es que, además, como la versión 10 del proyecto ya es capaz de generar el árbol de juego completo, podemos comprobarlo empíricamente:

Árbol de profundidad 3:

Si el usuario pide el árbol de profundidad 3, el último tablero que se genera después de unos minutos de “pensar” es éste:



La dirección $6100 todavía está por debajo de la dirección de base del intérprete de BASIC, es decir, $a000.

De hecho, si de $6100 restamos $19d8, que es el valor que toma la etiqueta “raizArbol” en esta versión del proyecto, el resultado es $6100 - $19d8 = $4728 = 18216 bytes. Es decir, 18216/88 = 207 tableros. En realidad 208, si incluimos también el último tablero, el que empieza en $6100.

Esto viene a ser casi lo mismo que 8 movimientos de los gatos x 4 movimientos del ratón x 8 movimientos de los gatos = 256 tableros, si no fuera porque no

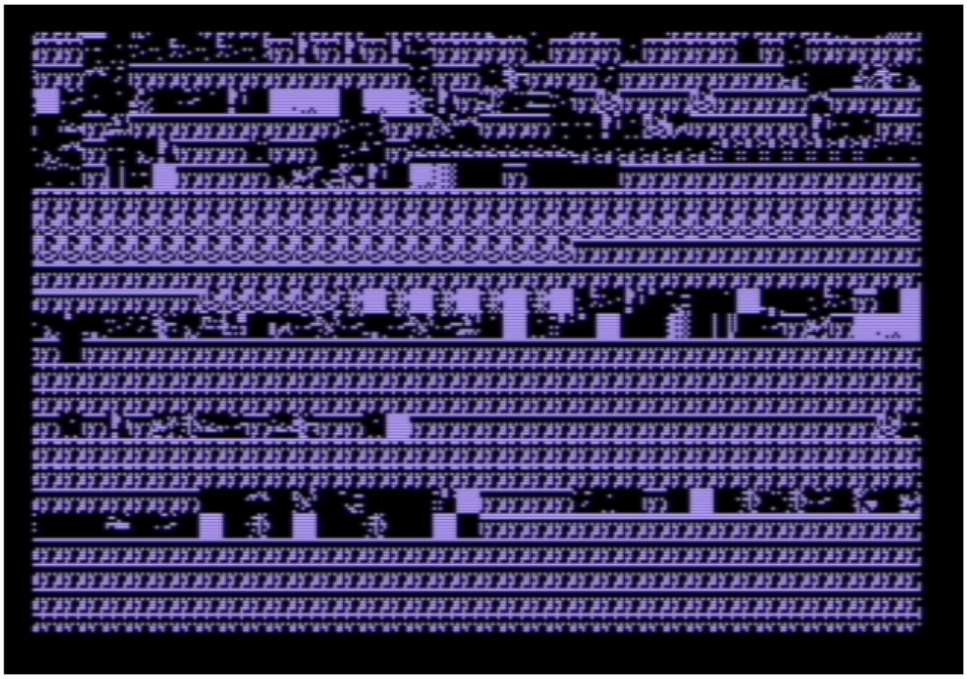
siempre los gatos tienen ocho movimientos ni el ratón cuatro. A veces tienen menos. Todo encaja.

Árbol de profundidad 4:

Sin embargo, si el usuario pide el árbol de profundidad 4, a partir de $a000 se generan tableros así:

Obsérvese que el tablero no está bien generado, pero el programa tampoco "casca" de momento. El motivo es el modelo de "RAM bajo ROM" del C64. Por encima de la dirección $a000, los "sta" operan sobre RAM, pero los "lda" leen de ROM. El resultado es que los tableros no están bien, pero el programa como tal tampoco falla.

Y a partir de $d000, es decir, cuando ya llegamos al VIC, se empieza a ver esto:



Esto es porque los “sta” que vamos ejecutando para generar el árbol van modificando los registros del VIC y, en particular, cambian los modos gráficos.

Por último, el VICE se acaba quedando “tostado”.

Alternativas:

Ante esta situación, tenemos varias alternativas:

- Conformarnos con tres niveles de profundidad.
- Ampliar la memoria disponible desactivando la ROM del intérprete de BASIC y la del Kernal.
- Aprovechar más la memoria disponible diseñando estructuras de datos más compactas.

Estas medidas no son necesariamente excluyentes; pueden ser complementarias.

Conformarse con tres niveles de profundidad es perfectamente posible, especialmente si la función de evaluación es buena. Como ya se comentó en la

entrada “La función de evaluación” el árbol de juego y la función de evaluación conforman un compromiso. Si el árbol de juego pudiera ser ilimitado, la función de evaluación podría ser muy tonta (llegaría con que detectara que un bando gana). Contrariamente, dado que el árbol normalmente tendrá que ser limitado, hará falta una función de evaluación “fuerte” que sirva para valorar cómo de prometedora es una línea de juego.

Por otro lado, uno de los fuertes del C64 es su capacidad para configurar su memoria. Efectivamente, actuando sobre los bits del registro R6510 = $0001 es posible ampliar la RAM disponible desactivando el intérprete de BASIC ($a000 - $bfff) y el Kernal ($e000 - $ffff). En nuestro caso, es posible desactivar el intérprete de BASIC, puesto que no lo usamos, pero sí usamos rutinas del Kernal como “chrout”.

Por último, es posible diseñar estructuras de datos más compactas. Por ejemplo, en vez de usar una matriz de 8 x 8 = 64 bytes para guardar el tablero, es posible guardar información equivalente con sólo 5 bytes:

- 1 byte = offset respecto a la casilla (0, 0) del ratón.
- 1 byte = offset respecto a la casilla (0, 0) del gato 1.
- …
- 1 byte = offset respecto a la casilla (0, 0) del gato 4.

Con sólo 5 bytes has resuelto la misma papeleta que antes resolvías con 64. Y se pueden tomar medidas similares con el resto de campos que acompañan a un tablero (nivel, turno, valor, padre e hijos).

El problema de hacer esto es que se te complica la programación, por ejemplo, la generación de jugadas. Generar jugadas sobre un tablero de 8 x 8 es relativamente fácil; basta con sumar los incrementos / decrementos permitidos por las reglas de juego en las coordenadas (X, Y) de las piezas. Pero si en vez de tener las posiciones (X, Y) tienes el offset respecto al origen…

Una posibilidad es guardar los tableros en RAM en formato comprimido, pero trabajar con ellos (ej. generar jugadas) en formato descomprimido. Ganas espacio, pero pierdes capacidad de cómputo en comprimir / descomprimir tableros.

Conclusiones:

Sea como fuere, las medidas que vamos a aplicar a nuestro caso son dos:

- Vamos a limitar el número de niveles a tres.
- Vamos a ampliar la RAM disponible desactivando el intérprete de BASIC.

La primera medida la tomamos ya en la versión 11 del proyecto.

La segunda media la aplicamos en alguna versión posterior, casi más por ponerla en práctica que por su efectividad, puesto que ganar los $bfff - $a000 = 8 KBytes del intérprete de BASIC tampoco es suficiente para alcanzar un nivel extra de profundidad.

El siguiente paso será ya la función de evaluación. ¿Conseguiremos que sea lo suficientemente fuerte como para compensar un árbol de profundidad limitada?

RYG: función de evaluación – cuestiones previas

De las cuatro piezas fundamentales de todo juego de tablero:

- El árbol de juego.
- El generador de jugadas.
- La función de evaluación.
- Y el procedimiento minimax.
- (El procedimiento de poda es opcional).

ya tenemos las dos primeras. Yo creo que con esto ya hemos alcanzado la cima del proyecto y a partir de ahora ya es todo un poco más cuesta abajo.

El siguiente paso, por tanto, es la función de evaluación. La función de evaluación es una rutina que recibe un tablero y devuelve cómo de bueno o malo es para ratón y gatos.

Criterios de evaluación:

Las funciones de evaluación, como ya comentamos, suelen tener en cuenta:

- Criterios materiales.
- Y criterios posicionales.

Los criterios materiales tienen que ver con las piezas que quedan sobre el tablero y las que ya se han "comido". Pero como en el juego del ratón y los gatos no hay capturas, no tiene sentido aplicar criterios materiales. Siempre habrá las mismas piezas sobre el tablero.

Y los criterios posicionales tienen que ver con la posición de las piezas: si tienen más movilidad, menos movilidad, si han llegado o están cerca del objetivo, si su posición es más ofensiva, más defensiva, etc. Todos los criterios que aplicaremos aquí serán posicionales.

Empezaremos haciendo una función de evaluación sencilla, con pocos criterios posicionales, y la iremos mejorando con sucesivas versiones.

Evaluación y procedimiento minimax:

En principio, la función de evaluación se aplica sólo sobre las hojas del árbol de juego, es decir, sobre los tableros de máxima profundidad. Y, a partir de ahí, las evaluaciones se van haciendo subir por el árbol de juego mediante el procedimiento minimax.

Sin embargo, debemos ir poco a poco. Así que en una primera versión haremos una función de evaluación sencilla y la aplicaremos sobre todos los nodos (tableros) del árbol de juego. Esto nos permitirá dos cosas:

- Probar bien la función de evaluación, ya que tendremos muchos tableros de ejemplo sobre los que probarla (todos los del árbol).
- Y preparar la base del procedimiento minimax, que sí tiene que trabajar sobre todos los tableros del árbol.

Cuándo evaluar:

Respecto a cuándo evaluar, básicamente hay dos opciones:

- O generar todo el árbol completo y luego evaluar.
- O evaluar los tableros según se van generando.

La primera opción es más sencilla y es la que vamos a aplicar. Es más sencilla porque el tipo de evaluación depende de que el tablero sea una hoja del árbol o un nodo intermedio. En el primer caso se aplica la función de evaluación

propiamente dicha; en el segundo caso se aplica el procedimiento minimax. Y esta distinción (hoja o nodo intermedio) es más fácil de hacer teniendo el árbol completo que sobre la marcha.

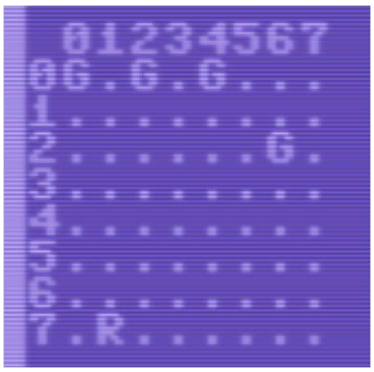
La segunda opción es un poco más compleja, pero es necesaria si se quiere aplicar un procedimiento de poda. No tendría sentido generar el árbol completo para luego podar algunas ramas, ya que ya se habría incurrido en el coste de generarlas.

Valores de las evaluaciones:

El objetivo de la función de evaluación es devolver un número que diga cómo de bueno o malo es un tablero para el ratón y los gatos. El valor absoluto es lo de menos; lo importante es que, si se compara la evaluación de un tablero T1 con la de otro tablero T2, quede claro qué tablero es más favorable para el ratón (o los gatos) y qué tablero es más desfavorable.

Por ello, tenemos bastante libertad para elegir el rango de valores para las evaluaciones. Supongamos que decidimos usar un byte (256 valores distintos).

Puesto que el ratón se señaliza sobre el tablero con el valor $01 (positivo) y los gatos con el valor $ff = -1 (negativo), una primera opción intuitiva sería asignar valoraciones positivas para lo que favorece al ratón y perjudica a los gatos, y valoraciones negativas para lo que favorece a los gatos y perjudica al ratón.

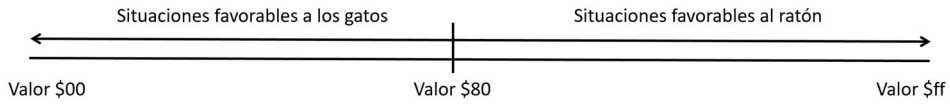


Por ejemplo, algo así (recordemos que el objetivo del ratón es llegar a la fila cero, y el objetivo de los gatos es acorralar al ratón):

- Situaciones favorables al ratón:
  - Ratón en fila 7 → +0 puntos
  - Ratón en fila 6 → +1 puntos
  - Ratón en fila 5 → +2 puntos
  - Ratón en fila 4 → +3 puntos
  - Ratón en fila 3 → +4 puntos
  - Ratón en fila 2 → +5 puntos
  - Ratón en fila 1 → +6 puntos
  - Ratón en fila 0 (gana el ratón) → +32 puntos = $20
- Situaciones favorables a los gatos:
  - Ratón con 4 movimientos → -0 puntos
  - Ratón con 3 movimientos → -1 puntos = $ff
  - Ratón con 2 movimientos → -2 puntos = $fe
  - Ratón con 1 movimientos → -3 puntos = $fd
  - Ratón con 0 movimientos (ganan los gatos) → -32 puntos = $e0

Sin embargo, manejar números negativos en ensamblador no es fácil. No es fácil compararlos, no es fácil calcular el máximo ni el mínimo (procedimiento minimax), al operar con ellos se puede producir desbordamiento, es decir, al sumar dos números positivos el resultado puede ser aparentemente negativo (situación señalizada con el flag V), etc.

Por todo ello, y dado que los valores absolutos de las evaluaciones son lo de menos, simplifica bastante la vida tomar un valor intermedio en el rango 0 – 256 como valor neutro y, a partir de ahí, tomar valores por debajo para situaciones que favorecen a los gatos y valores por encima para situaciones que favorecen al ratón. En nuestro caso en particular vamos a tomar $80 = 128 como ese valor neutro y, a partir de ahí, sumaremos o restaremos, pero siempre manejaremos números positivos.



Con todos estos comentarios previos ya estamos listos para abordar la primera versión de la función de evaluación, cosa que ya dejamos para la siguiente entrada.

RYG: función de evaluación – primera versión

La función de evaluación es una funcionalidad nueva. Por tanto, vamos de dotarla en un fichero nuevo: “EvalTableros.asm”. En este fichero encontramos:

Rutina “evaluaTablero”:

La rutina principal de este nuevo fichero es “evaluaTablero”:

```
etTableroLo     byte $00
etTableroHi     byte $00
etValor         byte $00

evaluaTablero

        ; Inicializa el valor
        lda #$80
        sta etValor

        ; Evalúa la fila del ratón
        jsr evaluaFilaRaton

        ; Evalúa los movimientos del ratón
        jsr evaluaMovsRaton

        ; Mete el valor en el propio tablero evaluado
        jsr meteValorTablero

        rts
```

Esta rutina recibe un tablero y devuelve su valor. Para ello:

- Inicializa la evaluación al valor neutro $80.
- Evalúa la posición del ratón con la rutina “evaluaFilaRaton”.
- Evalúa la movilidad del ratón con la rutina “evaluaMovsRaton”.

La rutina no sólo devuelve el valor en la variable “etValor”. Además, lo almacena en el propio tablero evaluado llamando a “meteValorTablero”. Recordemos que uno de los campos de todo tablero es el campo “valor”.

Rutina “evaluaFilaRaton”:

La fila del ratón es el primer criterio posicional. Como ya adelantamos, vamos a evaluarla así:

- Situaciones favorables al ratón:
    - Ratón en fila 7 → +0 puntos
    - Ratón en fila 6 → +1 puntos
    - Ratón en fila 5 → +2 puntos
    - Ratón en fila 4 → +3 puntos
    - Ratón en fila 3 → +4 puntos
    - Ratón en fila 2 → +5 puntos
    - Ratón en fila 1 → +6 puntos
    - Ratón en fila 0 (gana el ratón) → +32 puntos = $20

Esto es fácil de implementar con una tabla de datos que, en función de la fila ocupada por el ratón, devuelve la evaluación:

```
; Rutina para evaluar la fila del ratón

efrTempX        byte $00
tablaFilas      byte $20,$06,$05,$04,$03,$02,$01,$00

evaluaFilaRaton

        stx efrTempX

        ; Localiza el ratón
        lda etTableroLo
        sta drTableroLo

        lda etTableroHi
        sta drTableroHi

        jsr dameRaton

        ; Convierte a fila y columna
        lda drOffset
        sta dfcOffset

        jsr dameFilaCol

        ldx dfcFila

        ; Evalúa la fila del ratón
        lda tablaFilas,x

        ; Y la suma a la valoración de partida ($80)
        clc
        adc etValor
        sta etValor

        ldx efrTempX

        rts

```

Para hacer esto, la rutina:

- Localiza el ratón sobre el tablero con “dameRaton”.
- Convierte el offset del ratón en (fila, columna).
- Usando la fila como índice, accede a la tabla con las valoraciones.
- Suma la nueva valoración a la de partida ($80).

Rutina “evaluaMovsRaton”:

La movilidad del ratón es el segundo criterio posicional. Como ya dijimos, también, vamos a evaluarlo así:

- Situaciones favorables a los gatos:
    - Ratón con 4 movimientos → -0 puntos

- Ratón con 3 movimientos → -1 puntos = $ff
- Ratón con 2 movimientos → -2 puntos = $fe
- Ratón con 1 movimientos → -3 puntos = $fd
- Ratón con 0 movimientos (ganan los gatos) → -32 puntos = $e0

Esto también se puede resolver con una tabla de datos, pero requiere un poco más de trabajo, porque previamente hay que calcular cuántos movimientos admite el ratón en el tablero evaluado.

Por ello, la rutina es así (primera parte):

```
; Rutina para evaluar los movimientos del ratón

emrTempX        byte $00
emrTempY        byte $00
tablaMovs       byte $e0,$fd,$fe,$ff,$00

evaluaMovsRaton

        stx emrTempX
        sty emrTempY

        ; Localiza el ratón
        lda etTableroLo
        sta drTableroLo

        lda etTableroHi
        sta drTableroHi

        jsr dameRaton

        ; Convierte a fila y columna
        lda drOffset
        sta dfcOffset

        jsr dameFilaCol

        ; Genera las jugadas válidas del ratón
        lda etTableroLo
        sta gjvrTableroLo

        lda etTableroHi
        sta gjvrTableroHi

        lda dfcFila
        sta gjvrFila

        lda dfcColumna
        sta gjvrColumna

        jsr generaJugadasValidasRaton
```

Es decir:

- Localiza el ratón sobre el tablero.
- Convierte el offset del ratón en (fila, columna).
- Y genera las jugadas válidas del ratón.

Posteriormente, la rutina sigue así (parte 2):

```
135         jsr generaJugadasValidasRaton
136
137         ; Cuenta las jugadas válidas con Y
138         ldx #$00
139         ldy #$00
140
141 emrBucle
142
143         lda gjvrNuFilas,x
144
145         cmp #$ff
146         beq emrCont
147
148         iny
149
150 emrCont
151
152         inx
153
154         cpx #$04
155         bne emrBucle
156
157         ; En función de Y, obtiene la valoración
158         lda tablaMovs,y
159
160         ; Y la suma a la valoración por fila
161         clc
162         adc etValor
163         sta etValor
164
165         ldx emrTempX
166         ldy emrTempY
167
168         rts
169
```

Es decir:

- Cuenta las jugadas válidas del ratón con el registro Y.
- Usando Y como índice, accede a la tabla con las valoraciones.
- Suma la nueva valoración a la anterior ($80 + fila del ratón).

De este modo, ya tenemos la valoración completa, al menos en lo que respecta a esta primera versión de la función de evaluación.

Pero, además de devolver el valor con “etValor”, resulta cómodo tenerlo a mano en el propio tablero evaluado, motivo por el que “evaluaTablero” termina llamando a “meteValorTablero”.

Rutina “meteValorTablero”:

La rutina que almacena el valor en el tablero evaluado es así:

```
; Rutina para meter el valor en el tablero

meteValorTablero

        ; Obtiene los datos básicos del tablero
        lda etTableroLo
        sta ddbTableroLo

        lda etTableroHi
        sta ddbTableroHi

        jsr dameDatosBasicos

        ; Y los vuelve a fijar, cambiando el valor
        lda etTableroLo
        sta fdbTableroLo

        lda etTableroHi
        sta fdbTableroHi

        lda ddbNivel
        sta fdbNivel

        lda ddbTurno
        sta fdbTurno

        lda etValor ; Obsérvese que no se usa ddbValor sino etValor
        sta fdbValor

        jsr fijaDatosBasicos

        rts
```

Y como el valor forma parte de la terna que hemos llamado “datos básicos” (nivel, turno y valor), la rutina:

- Utiliza “dameDatosBasicos” para obtener la terna.
- Y utiliza “fijaDatosBasicos” para volver a fijar la terna, previo cambio del valor original (“ddbValor”) por el nuevo valor (“etValor”).

En alguna versión posterior del proyecto haremos una rutina más directa (“fijaValor”) que permita cambiar sólo el valor sin tanto trasiego de datos.

Y con esto la versión 12 del proyecto ya tiene su primera versión de la función de evaluación. Pero falta aplicarla a un tablero o conjunto de tableros, cosa que ya haremos en la siguiente entrada.

RYG: función de evaluación – evaluación del árbol completo

Ahora que ya tenemos la función de evaluación, vamos a aplicarla.

Lo suyo sería aplicarla sólo sobre las hojas del árbol, y hacer subir las valoraciones mediante el procedimiento minimax. Sin embargo, de momento, vamos a evaluar todos y cada uno de los tableros del árbol. Así podemos comprobar si la rutina funciona bien con un montón de tableros y, de paso, vamos preparando el procedimiento minimax.

Por otro lado, ya comentamos que la evaluación puede hacerse sobre la marcha, según se van generando los tableros, o ya al final, con todo el árbol de juego generado. Y como no pensamos aplicar un procedimiento de poda, vamos a hacerlo al final. Así es más claro.

Rutina “evaluaArbol”:

Para conseguir lo anterior dotamos una nueva rutina “evaluaArbol” en el fichero “EvalTableros.asm”. Esta rutina es recursiva, igual que la que generaba el árbol de juego completo. Recibe un tablero (con sus hijos enlazados) y una profundidad, y hace lo siguiente:

- Si la profundidad ya es menor que cero, termina.
- Si la profundidad es cero o mayor:
    - Evalúa el tablero actual.
    - Recorre los tableros hijo y, de forma recursiva, los evalúa (reduciendo previamente la profundidad en uno).

De este modo se consigue evaluar de forma recursiva todo el árbol. Cuando la evaluación llega a las hojas del árbol termina, porque ya no hay más hijos que evaluar.

Esta es la primera parte de “evaluaArbol”, donde se decide si seguir evaluando o terminar ya (profundidad = $ff = -1):

```
; Rutina para evaluar un árbol completo (considerando cada tablero de forma
; aislada de los demás)

eaProf          byte $00
eaTableroLo     byte $00,$00,$00,$00,$00,$00,$00,$00
eaTableroHi     byte $00,$00,$00,$00,$00,$00,$00,$00
eaNumHijo       byte $00,$00,$00,$00,$00,$00,$00,$00

evaluaArbol

        lda eaProf
        tay

        cmp #$ff
        bne eaOtroNivel

        rts

eaOtroNivel

```

Esta es la segunda parte de la rutina, donde se evalúa el tablero actual con la rutina ya conocida “evaluaTablero”:

```
eaOtroNivel

        ; Evalúa el tablero actual
        lda eaTableroLo,y
        sta etTableroLo

        lda eaTableroHi,y
        sta etTableroHi

        jsr evaluaTablero

```

Y en la tercera parte se recorren los hijos del tablero actual (“jsr dameHijo”), se decrementa la profundidad en uno (“sbc #$01”), y se continúa recursivamente con todos los hijos (“jsr evaluaArbol”):

```
232         ; Evalúa los hijos
233         ldx #$00
234
235 eaBucle
236
237         lda eaTableroLo,y
238         sta dhTableroLo
239
240         lda eaTableroHi,y
241         sta dhTableroHi
242
243         stx dhNumHijo
244
245         jsr dameHijo
246
247         lda dhHijoHi
248         beq eaSgteHijo
249
250         lda eaProf
251         sec
252         sbc #$01
253         sta eaProf
254         tay
255
256         lda dhHijoLo
257         sta eaTableroLo,y
258
259         lda dhHijoHi
260         sta eaTableroHi,y
261
262         txa
263         sta eaNumHijo,y
264
265         jsr evaluaArbol
266
267         lda eaNumHijo,y
268         tax
269
270         lda eaProf
271         clc
272         adc #$01
273         sta eaProf
274         tay
275
276 eaSgteHijo
277
278         inx
279
280         cpx #$08
281         bne eaBucle
282
283         rts
284
```

Llamada a la rutina “evaluaArbol”:

La rutina “evaluaArbol”, como toda rutina, hay que llamarla para que en la práctica haga algo. Y ya hemos dicho que vamos a llamarla tras generar el árbol completo, es decir, aquí:

```
*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

        jsr desarrollaArbolJugadas

        jsr evaluaArbolJugadas

        jsr pintaArbolJugadas

        rts ; jmp actualiza

```

La llamada a “evaluaArbolJugadas” se produce justo después de generar el árbol de juego (“jsr desarrollaArbolJugadas”). Y “evaluaArbolJugadas” es básicamente una simple llamada a la recién presentada “evaluaArbol”:

```
717 evaluaArbolJugadas
718
719         lda prof
720         sta eaProf
721         tay
722
723         lda #<raizArbol
724         sta eaTableroLo,y
725
726         lda #>raizArbol
727         sta eaTableroHi,y
728
729         lda #$00
730         sta eaNumHijo,y
731
732         jsr evaluaArbol
733
734         rts
735
```

Puede que llame la atención también la llamada “jsr pintaArbolJugadas”, pero su propósito lo comentaremos ya en la entrada siguiente.

RYG: función de evaluación – pintado del árbol

Ya tenemos una función de evaluación (rutina “evaluaTablero”) y una rutina que evalúa recursivamente todos los tableros del árbol (rutina “evaluaArbol”).

Todo esto hay que probarlo, y la mejor manera de probarlo es pintando el árbol de juego con sus evaluaciones, y comprobando que éstas efectivamente responden a los criterios definidos.

Para ello vamos a desarrollar una nueva rutina, llamada “pintaArbol”, que se encargará de pintar el árbol de juego completo. Esta rutina nuevamente será recursiva (se llamará a sí misma), y estará definida en el fichero “pintaTableros.asm”. La llamaremos justo después de construir y evaluar el árbol:

```
*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

        jsr desarrollaArbolJugadas

        jsr evaluaArbolJugadas

        jsr pintaArbolJugadas

        rts ; jmp actualiza
```

```
pintaArbolJugadas

        lda prof
        sta paProf
        tay

        lda #<raizArbol
        sta paTableroLo,y

        lda #>raizArbol
        sta paTableroHi,y

        lda #$00
        sta paNumHijo,y

        jsr pintaArbol

        rts
```

Hasta ahora teníamos una rutina para pintar tableros (“pintaTablero”), pero no una rutina para pintar el árbol completo (“pintaArbol”). Además, la rutina para

pintar tableros la llamábamos sobre la marcha, según se iban generando los tableros (rutinas “aplicaJugadaGatoHijo” y “aplicaJugadaRatonHijo”), lo que tenía como efectos secundarios:

- No se podían pintar las direcciones de los hijos, porque recién creado un tablero sus hijos todavía no están creados.
- No se podía pintar la evaluación del tablero, porque primero creamos el árbol completo (sus tableros) y luego lo evaluamos también completo.

En realidad, hasta ahora veníamos llamando a la rutina “pintaTablero” desde “aplicaJugadaGatoHijo” y desde “aplicaJugadaRatonHijo” más como una forma de depurar que los tableros o jugadas se generan bien, que por el interés que tuviera pintar esos tableros.

Sea como fuere, hay que cambiar el enfoque, porque ahora queremos pintar las evaluaciones, lo que nos lleva a:

- Dejar de llamar a “pintaTablero” desde “aplicaJugadaGatoHijo” y “aplicaJugadaRatonHijo”. En vez de esto pintamos un punto (“.”), lo que pretende transmitir la idea de que el C64 está pensando. Y en realidad lo está haciendo, porque está generando tableros.
- Empezar a llamar a “pintaArbol” tras construir y evaluar el árbol completo, lo que nos permitirá visualizar las evaluaciones y, ya de paso, también las direcciones de los hijos.

Finalmente, aprovechamos para hacer un pequeño cambio estético, y en vez de pintar los turnos con $01 (ratón) y $ff (gatos), definimos la rutina “pintaTurno”, que pinta los literales “RATON” y “GATOS” respectivamente.

La rutina “pintaArbol” tiene una estructura muy similar a las otras rutinas recursivas que ya hemos visto (“evaluaArbol” y “desarrollaUnNivel”):

- Primero evalúa si se ha terminado de pintar el árbol.
- Si no se ha terminado, pinta la raíz del árbol con “pintaTablero”.
- Y luego recorre los hijos pintándolos también llamándose de forma recursiva.

En esta primera parte evalúa si se ha terminado o continuar:

```
559 ; Rutina para pintar un árbol completo
560
561 paProf          byte $00
562 paTableroLo     byte $00,$00,$00,$00,$00,$00,$00,$00
563 paTableroHi     byte $00,$00,$00,$00,$00,$00,$00,$00
564 paNumHijo       byte $00,$00,$00,$00,$00,$00,$00,$00
565
566 pintaArbol
567
568         lda paProf
569         tay
570
571         cmp #$ff
572         bne paOtroNivel
573
574         rts
575
576 paOtroNivel
577
```

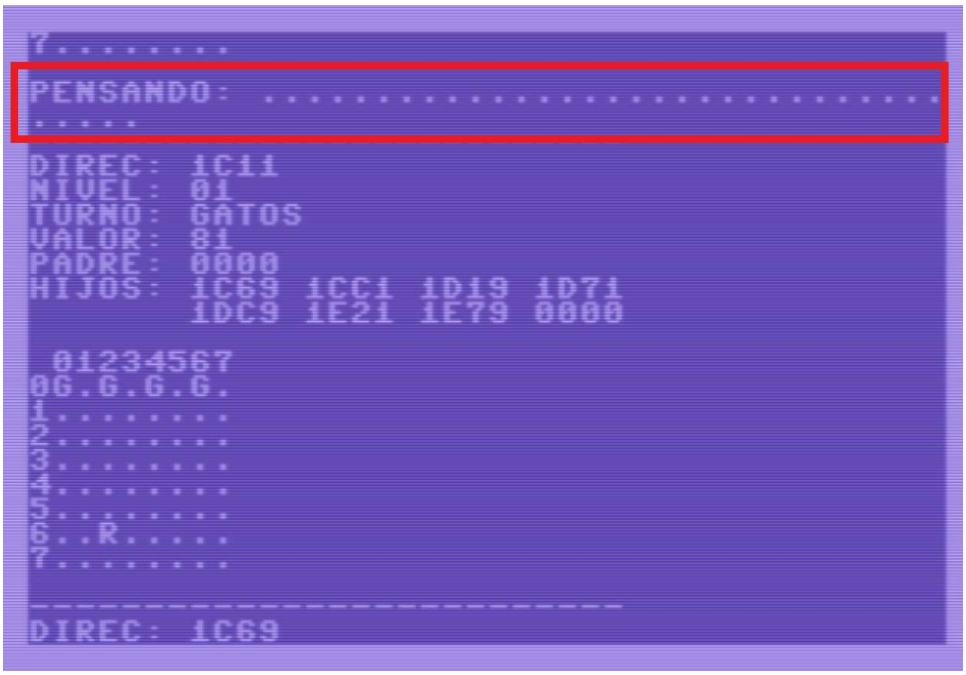
En esta segunda parte pinta la raíz del árbol actual. Para ello usa “pintaTablero”:

```
576 paOtroNivel
577
578         ; Pinta el tablero actual
579         lda paTableroLo,y
580         sta ptTableroLo
581
582         lda paTableroHi,y
583         sta ptTableroHi
584
585         jsr pintaTablero
586
```

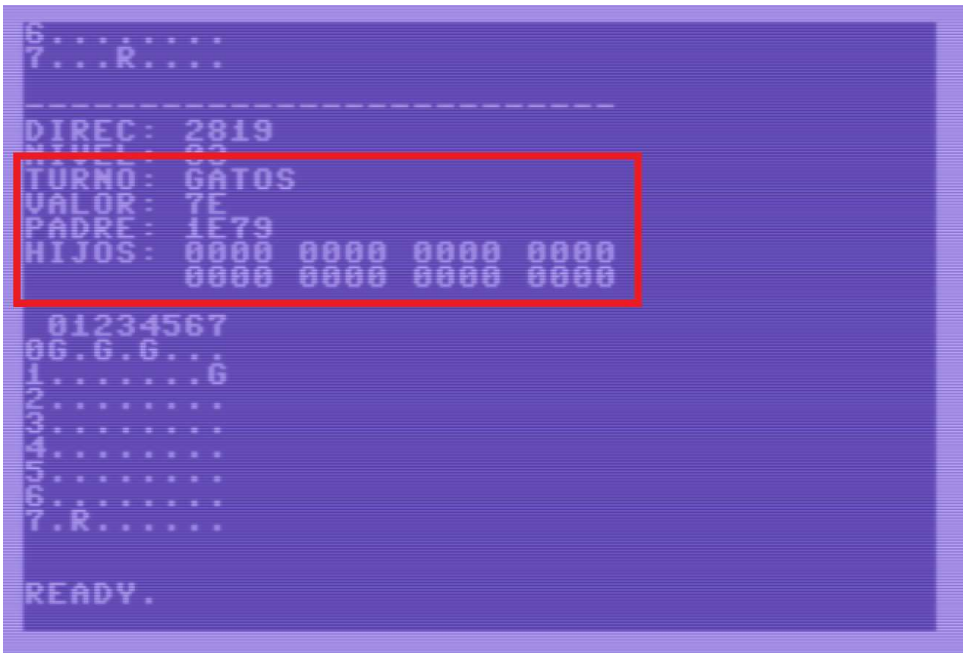
Y, por último, en la tercera parte recorre los hijos y los pinta recursivamente:

```
587         ; Pinta los hijos
588         ldx #$00
589
590 paBucle
591
592         lda paTableroLo,y
593         sta dhTableroLo
594
595         lda paTableroHi,y
596         sta dhTableroHi
597
598         stx dhNumHijo
599
600         jsr dameHijo
601
602         lda dhHijoHi
603         beq paSgteHijo
604
605         lda paProf
606         sec
607         sbc #$01
608         sta paProf
609         tay
610
611         lda dhHijoLo
612         sta paTableroLo,y
613
614         lda dhHijoHi
615         sta paTableroHi,y
616
617         txa
618         sta paNumHijo,y
619
620         jsr pintaArbol
621
622         lda paNumHijo,y
623         tax
624
625         lda paProf
626         clc
627         adc #$01
628         sta paProf
629         tay
630
631 paSgteHijo
632
633         inx
634
635         cpx #$08
636         bne paBucle
637
638         rts
```

Lo importante es el resultado final. Y si ahora ejecutamos el juego con un nivel de profundidad dos, primero aparece “PENSANDO: ……..” mientras se genera el árbol de juego:



Y luego, tras generarlo, se pinta el árbol completo (lo que sigue es la pantalla final tras todo el scroll vertical):



En esta imagen se puede observar que:

- En el turno ya aparece “RATON” o “GATOS” y no $01 o $ff.
- Ya aparece la evaluación del tablero, que en el caso anterior es de $7e.
- Ya aparecen las direcciones de los hijos, salvo que el tablero en cuestión sea una hoja del árbol (como el de la imagen anterior), en cuyo caso no tiene hijos y aparecen todos inicializados a $0000.

Lo importante ahora es **validar si $7e es la evaluación correcta del tablero conforme a los criterios definidos**:

- Fila del ratón = 7 → +0 puntos.
- Movimientos del ratón = 2 → -2 puntos.
- Evaluación de partida (punto neutro) → $80.

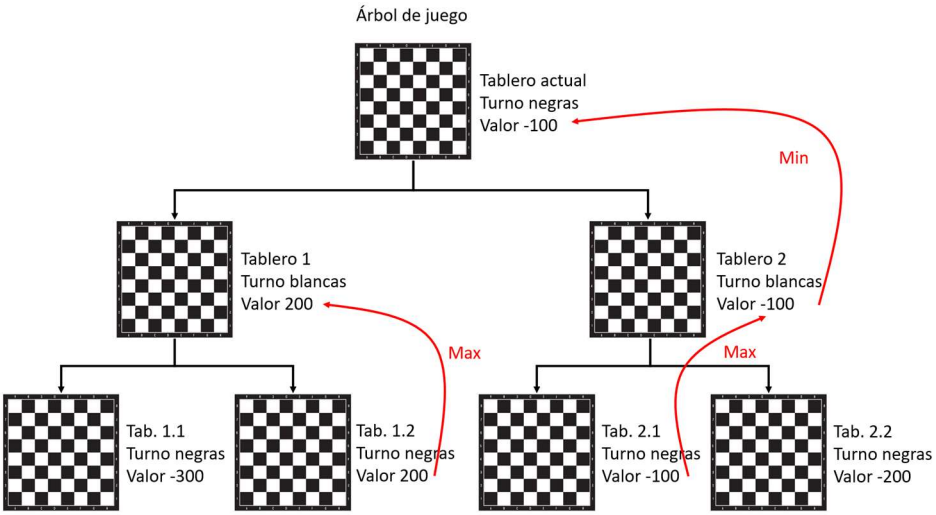
Total = $80 – $02 = $7e. Por tanto, la evaluación es correcta. Y esto mismo se puede comprobar para todos los tableros del árbol generado y pintado.

Y puesto que parece que ya conseguimos evaluar bien, el siguiente paso será el procedimiento minimax.

RYG: procedimiento minimax – rutinas previas

En su momento ya introdujimos el procedimiento minimax. Como dijimos, este procedimiento es pura lógica, es asumir que igual que tú (ratón) quieres la mejor jugada para ti (la de mayor puntuación), tu contrincante (los gatos) querrán la mejor jugada para sí (la de menor puntuación). De ahí el nombre “minimax”.

Por tanto, no tenemos que evaluar todos los tableros del árbol aisladamente unos de otros, como hemos hecho. Sólo tenemos que evaluar las hojas del árbol y, a partir de ahí, aplicar el procedimiento minimax (obtener el máximo o el mínimo, según el turno de juego) para llevar estas evaluaciones hasta la raíz del árbol.



Aplicando este procedimiento evaluaremos todo el árbol, es decir, todos sus tableros, pero poniendo en relación unos con otros. De poco valdría que el C64 eligiera una siguiente jugada en apariencia muy buena, si N jugadas más allá finalmente resulta ser muy mala. Al contrario, si hemos generado el árbol completo (hasta una profundidad N) es para poder “mirar más allá” de la siguiente jugada.

La suerte es que ya tenemos la base del procedimiento minimax: la rutina "evaluaArbol". Con algunas pequeñas modificaciones será suficiente para construir una nueva rutina "miniMax". Pero vayamos por partes:

Rutina "dameNumHijos":

Lo primero que vamos a hacer es desarrollar es una rutina "dameNumHijos". Esta rutina nos resultará útil para saber si estamos ante una hoja del árbol (número de hijos = 0) o ante un tablero intermedio (número de hijos > 0). Es decir, nos servirá para saber si tenemos que evaluar el tablero con la función de evaluación o con el procedimiento minimax.

La nueva rutina "dameNumHijos" la meteremos en el fichero "Tableros.asm" y, como ya tenemos una rutina "dameHijo" que nos permite recuperar el hijo enésimo, nos apoyaremos en ella:

```
; Rutina para recuperar el números de hijos de un tablero

dnhTableroLo    byte $00
dnhTableroHi    byte $00
dnhNumHijos     byte $00

dnhTempY        byte $00

dameNumHijos

        sty dnhTempY

        ldy #0

dnhBucle

        lda dnhTableroLo
        sta dhTableroLo

        lda dnhTableroHi
        sta dhTableroHi

        sty dhNumHijo

        jsr dameHijo

        ;lda dhHijoLo
        ;bne dhlCont

        lda dhHijoHi
        cmp #$00
        beq dnhFin

        iny

        cpy #8
        bne dnhBucle

dnhFin

        sty dnhNumHijos

        ldy dnhTempY

        rts
```

La rutina es básicamente un bucle desde Y = 0 hasta Y = 7. Para cada valor del registro Y se pide el hijo Y-ésimo llamando a “dameHijo”. Si el resultado es la dirección $0000 hemos terminado, porque los hijos de un tablero se van rellenando por orden (0, 1, 2, …). Si el resultado no es $0000 incrementamos Y, pasando al siguiente hijo e indirectamente incrementando la cuenta de hijos (que llevamos en Y). Si llegamos a Y = 8 necesariamente hemos terminado porque el máximo de hijos de un tablero es ocho.

Una cosa curiosa es que la dirección del hijo Y-ésimo tendrá dos partes, la parte "hi" y la parte "lo". En teoría, deberíamos contrastar ambas contra $00 para saber si existe el hijo (<> $0000) o si no existe (= $0000). En la práctica, llega con comparar la parte "hi" contra $00 puesto que, como no usamos la página cero para almacenar el árbol, si hi = $00 es indicio suficiente de que el hijo no existe.

Rutina "minValorHijos":

El procedimiento minimax se basa en elegir el hijo de mínima puntuación cuando juegan los gatos o el de máxima puntuación cuando juega el ratón. Por tanto, vamos a necesitar una rutina que identifique al hijo de mínima puntuación (este apartado) y otra que identifique al hijo de máxima puntuación (apartado siguiente).

En realidad, ni siquiera es necesario identificar el hijo de mínima / máxima puntuación, en el sentido de saber cuál es ese hijo o qué posición ocupa en la tabla de hijos de un tablero. En realidad, es suficiente con ser capaces de obtener esa puntuación mínima / máxima y asignarla al tablero padre. Así que esto es lo que vamos a hacer.

Cuando tienes una lista de valores, por ejemplo, $80, $87 y $7c, una forma de obtener el mínimo es partir del valor máximo posible, que en el caso de un byte sería $ff, e ir comparando los valores contra ese mínimo. Si el valor actual es menor que el mínimo hasta ahora, te quedas con el nuevo mínimo; si el valor actual es mayor que el mínimo hasta ahora, no haces nada. Veamos:

- Partimos de mínimo = $ff.
- ¿Es $80 menor que $ff? Sí, por tanto, mínimo = $80.
- ¿Es $87 menor que $80? No, por tanto, no hacemos nada.
- ¿Es $7c menor que $80? Sí, por tanto, mínimo = $7c.

De este modo, terminamos con el valor mínimo ($7c). Y ahora en versión rutina "minValorHijos" del fichero "Tableros.asm":

Partimos del máximo valor posible:

```
; Rutina para fijar como valor de un tablero el valor mínimo de sus hijos

minvTableroLo   byte $00
minvTableroHi   byte $00
minvValor       byte $00

minvTempY       byte $00

minValorHijos

        sty minvTempY

        ; Inicializa el valor al máximo posible

        lda #$ff
        sta minvValor

```

Recorremos los hijos con “dameHijo”, obtenemos su valor con “dameValor”, y vamos comparando el valor de los hijos contra el mínimo hasta ahora:

```
430         ; Recorre los hijos y va comparando valores
431
432         ldy #$00
433
434 minvBucle
435
436         lda minvTableroLo
437         sta dhTableroLo
438
439         lda minvTableroHi
440         sta dhTableroHi
441
442         sty dhNumHijo
443
444         jsr dameHijo
445
446         lda dhHijoHi
447         beq minvSgteHijo
448
449         lda dhHijoLo
450         sta dvTableroLo
451
452         lda dhHijoHi
453         sta dvTableroHi
454
455         jsr dameValor
456
457         ; Compara el valor del hijo con el mínimo hasta ahora
458         ; y si el valor del hijo es mayor no hace nada; si es menor
459         ; lo toma como nuevo mínimo
460
461         lda dvValor
462         cmp minvValor
463         bcs minvSgteHijo;bpl minvSgteHijo
464
465         sta minvValor
466
467 minvSgteHijo
468
469         iny
470
471         cpy #8
472         bne minvBucle
473
```

Finalmente, ya con el valor mínimo de los hijos identificado, lo fijamos como valor del padre con “fijaValor”:

```
473
474         ; Al llegar aquí, tenemos el mínimo en minvValor
475         ; Lo fijamos como valor del padre
476
477         lda minvTableroLo
478         sta fvTableroLo
479
480         lda minvTableroHi
481         sta fvTableroHi
482
483         lda minvValor
484         sta fvValor
485
486         jsr fijaValor
487
488         ldy minvTempY
489
490         rts
491
```

Las rutinas “dameValor” y “fijaValor” son nuevas también, y permiten obtener y fijar el valor de un tablero sin tener que obtener / fijar todos sus datos básicos (nivel, turno y valor). Son sencillas, y también están en “Tableros.asm”.

Gracias a que manejamos valoraciones que son siempre positivas (valor neutro $80) la comparación de valores puede hacerse fácilmente con las instrucciones “cmp” y “bcs”. La comparación de valores hubiera sido notablemente más complicada en caso de usar valoraciones positivas y negativas.

Rutina “maxValorHijos”:

Esta rutina es totalmente análoga a la “minValorHijos” ya vista. La principal diferencia es que ahora buscamos el valor máximo y, por tanto, partimos del valor mínimo posible que, en el caso de un byte, es $00:

```
492 ; Rutina para fijar como valor de un tablero el valor máximo de sus hijos
493
494 maxvTableroLo   byte $00
495 maxvTableroHi   byte $00
496 maxvValor       byte $00
497
498 maxvTempY       byte $00
499
500 maxValorHijos
501
502         sty maxvTempY
503
504         ; Inicializa el valor al mínimo posible
505
506         lda #$00
507         sta maxvValor
508
```

Otra diferencia es que ahora la comparación de valores la hacemos con las instrucciones “cmp” y “bcc”:

```
536         ; Compara el valor del hijo con el máximo hasta ahora
537         ; y si el valor del hijo es menor no hace nada; si es mayor
538         ; lo toma como nuevo máximo
539
540         lda dvValor
541         cmp maxvValor
542         bcc maxvSgteHijo;bmi maxvSgteHijo
543
544         sta maxvValor
545
```

Por lo demás, son rutinas casi idénticas. Nuevamente, el hecho de usar valoraciones siempre positivas simplifica la comparación y la obtención del máximo.

Con estos mimbres ya somos capaces de hacer el cesto (procedimiento minimax), pero como esta entrada ya ha sido demasiado larga lo dejamos para la siguiente.

RYG: procedimiento minimax

Ya somos capaces de saber si estamos ante una hoja del árbol o ante un tablero intermedio. En el primer caso evaluamos con la función de evaluación; en el segundo caso aplicamos el máximo o el mínimo, según el turno de juego.

Y esto es, precisamente, el procedimiento minimax, que se materializa en la nueva rutina “miniMax” del fichero “EvalTableros.asm”. Una vez más, será una rutina recursiva.

Rutina “miniMax”:

Empezamos analizando si el tablero tiene hijos o no con “dameNumHijos”:

```
; Rutina para evaluar un árbol completo (siguiendo el procedimiento minimax)

mmProf          byte $00
mmTableroLo     byte $00,$00,$00,$00,$00,$00,$00,$00
mmTableroHi     byte $00,$00,$00,$00,$00,$00,$00,$00
mmNumHijo       byte $00,$00,$00,$00,$00,$00,$00,$00

miniMax

        lda mmProf
        tay

        ; Mira si el tablero tiene hijos

        lda mmTableroLo,y
        sta dnhTableroLo

        lda mmTableroHi,y
        sta dnhTableroHi

        jsr dameNumHijos

        lda dnhNumHijos
        bne mmSiHijos
```

Si no tiene hijos, estamos ante una hoja y, por tanto, aplicamos la función de evaluación con “evaluaTablero”:

```
        ; Si no tiene hijos es hoja => evalúa el tablero directamente

mmNoHijos

        lda mmTableroLo,y
        sta etTableroLo

        lda mmTableroHi,y
        sta etTableroHi

        jsr evaluaTablero

        rts
```

Y si sí tiene hijos, estamos ante un nodo intermedio, lo que significa que tenemos que aplicar el valor máximo o mínimo en función del turno de juego:

```
269         ; Si sí tiene hijos => min o max de los hijos en función del turno
270
271 mmSiHijos
272
273         lda mmTableroLo,y
274         sta ddbTableroLo
275
276         lda mmTableroHi,y
277         sta ddbTableroHi
278
279         jsr dameDatosBasicos
280
281         lda ddbTurno
282         cmp #Gato
283         beq mmGatos
284
285 mmRaton
286
287         jsr maxHijos
288
289         rts
290
291 mmGatos
292
293         jsr minHijos
294
295         rts
296
```

El turno de juego lo obtenemos con la rutina “dameDatosBasicos” (aunque también podríamos hacer una nueva rutina “dameTurno”) y, en función del turno, llamamos a “maxHijos” si es el turno del ratón o “minHijos” si es el turno de los gatos.

Estas últimas rutinas no son más que una forma de estructurar un poco más la rutina “miniMax”, para que no salga demasiado larga o compleja. Se revisan a continuación.

Rutina “minHijos”:

Recordemos que estamos ante un tablero intermedio, no ante una hoja, y que es el turno de los gatos. Por tanto, los gatos elegirán el hijo con menor puntuación.

Para ello, recorremos los hijos con “dameHijo” y llamamos recursivamente a “minimax” para cada uno de ellos:

```
minHijos

        ; Recorre los hijos haciendo su minimax

        ldx #$00

minBucle

        lda mmTableroLo,y
        sta dhTableroLo

        lda mmTableroHi,y
        sta dhTableroHi

        stx dhNumHijo

        jsr dameHijo

        lda dhHijoHi
        beq minSgteHijo

        lda mmProf
        sec
        sbc #$01
        sta mmProf
        tay

        lda dhHijoLo
        sta mmTableroLo,y

        lda dhHijoHi
        sta mmTableroHi,y

        txa
        sta mmNumHijo,y

        jsr miniMax

        lda mmNumHijo,y
        tax

        lda mmProf
        clc
        adc #$01
        sta mmProf
        tay

minSgteHijo

        inx

        cpx #$08
        bne minBucle
```

Posteriormente, cuando todos los hijos están ya evaluados recursivamente, nos quedamos con la menor puntuación con “minValorHijos”:

```
350
351         ; Se queda el mínimo para el padre
352
353         lda mmTableroLo,y
354         sta minvTableroLo
355
356         lda mmTableroHi,y
357         sta minvTableroHi
358
359         jsr minValorHijos
360
361         rts
362
```

Rutina “maxHijos”:

Esta rutina es totalmente análoga a “minHijos”. Igualmente, es una rutina instrumental, cuyo objetivo no es tanto abstraer una funcionalidad autocontenida como simplificar “miniMax”.

Igual que “minHijos”, recorre los hijos del tablero intermedio, llama recursivamente a “miniMax” para cada uno de ellos y, cuando todos los hijos están ya evaluados, se queda con el máximo al ser ahora el turno del ratón.

Nuevo programa principal “RYG.asm”:

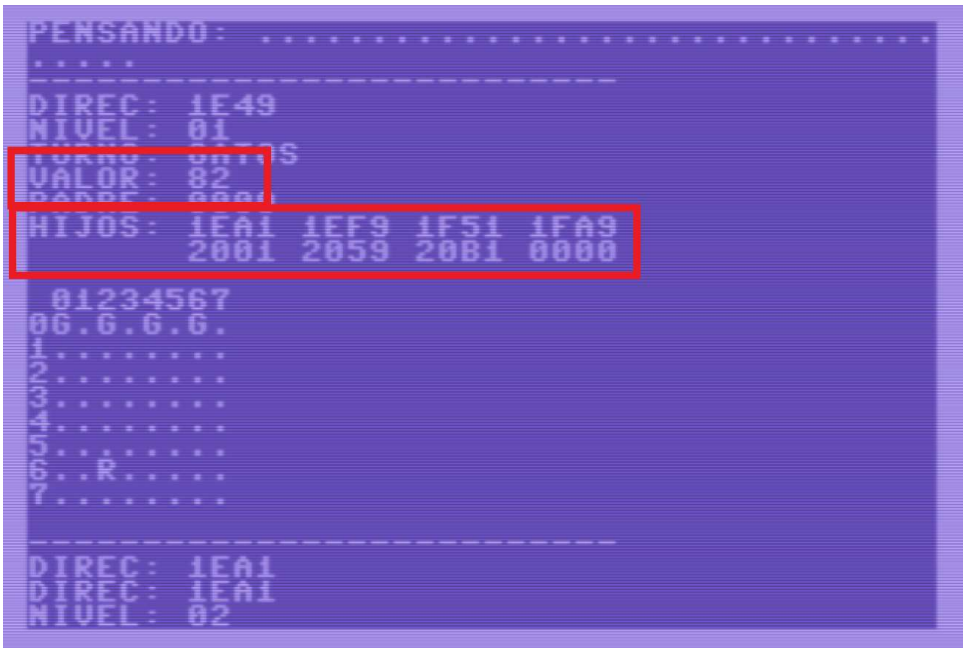
Para poner en funcionamiento el nuevo procedimiento minimax ya sólo queda modificar el programa principal “RYG.asm”, y más concretamente su rutina “evaluaArbolJugadas”, deja de llamar a “evaluaArbol”, que evaluaba todos los tableros de forma independiente, y pasar a llamar a “miniMax”.

Es decir, este cambio:

```
711
712 evaluaArbolJugadas
713
714         lda prof
715         sta mmProf
716         tay
717
718         lda #<raizArbol
719         sta mmTableroLo,y
720
721         lda #>raizArbol
722         sta mmTableroHi,y
723
724         lda #$00
725         sta mmNumHijo,y
726
727         jsr miniMax;evaluaArbol
728
729         rts
730
```

Si ahora probamos la versión 13 del proyecto, veremos que las hojas del árbol se siguen evaluando conforme a los criterios posicionales definidos (fila y número de movimientos del ratón), pero que los tableros intermedios se evalúan conforme a la puntuación de los hijos y de quién es el turno (ratón o gatos).

Por ejemplo, si probamos con un árbol de profundidad dos, tras construir, evaluar y pintar el árbol completo, nos sale esta raíz:



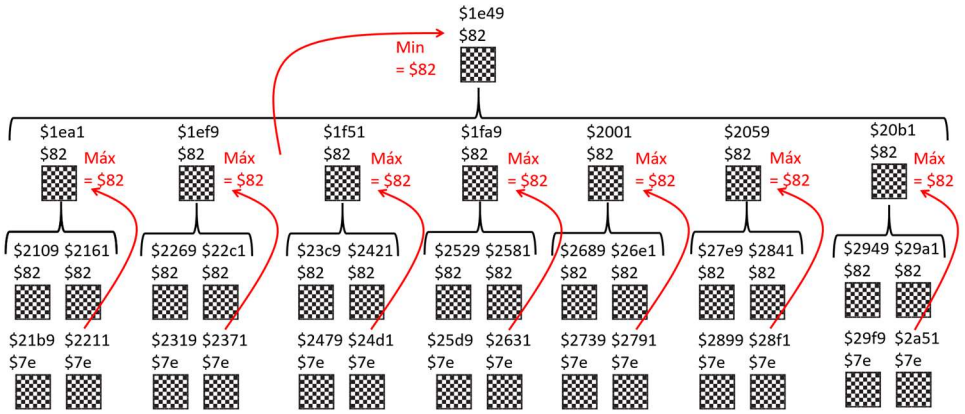
Es decir, el tablero raíz es el resultado de que el usuario haya movido el ratón desde (7, 3) hasta (6, 2). Este tablero, tiene siete hijos correspondientes a los siete posibles movimientos de los gatos. Las direcciones de los hijos son $1ea1, $1ef9, …, $20b1.

Si evaluamos el tablero atendiendo a los criterios posicionales, saldría:

- Valor de partida → $80 puntos.
- Fila del ratón = 6 → +1 puntos.
- Movimientos del ratón = 4 → -0 puntos.

Es decir, el valor del tablero sería $81. De hecho, este es el valor que salía con la versión 12 del proyecto. Entonces… ¿por qué ahora sale el valor $82? Pues porque ahora no estamos evaluando ese tablero atendiendo a la función de evaluación o los criterios posicionales, sino en función del procedimiento minimax.

De hecho, el árbol de juego de profundidad dos es así:



Partiendo de la raíz $1e49 se observan los siete hijos en $1ea1, $1ef9, …, $20b1. Estos siete hijos son los siete movimientos posibles de los gatos. A su vez, cada uno de los hijos tiene otros cuatro nietos, correspondientes a los cuatro movimientos del ratón. En total, 1 raíz + 7 hijos + 7x4 nietos = 36 tableros.

En el nivel más bajo, el ratón elegiría su mejor movimiento, es decir, el que maximiza el valor. Por eso entre $82 y $7e se elige $82. Y luego los gatos también elegirán su mejor movimiento, que en su caso será el minimice el valor. En este caso particular todos los hijos valen $82, así que el mínimo también es $82. Esto es habitual al comienzo de las partidas, donde todos los tableros analizados son más o menos parecidos. En un caso más general los hijos tomarán valores distintos e igualmente habrá que elegir entre ellos.

Total, al final la valoración que el procedimiento minimax lleva hasta la raíz es $82, en vez de $81 que sería la valoración posicional, y el movimiento que eligen los gatos (el C64) es el que lleva al hijo que toma ese valor. Pero esta decisión la tomaremos ya en la entrada siguiente.

RYG: el C64 por fin decide su jugada

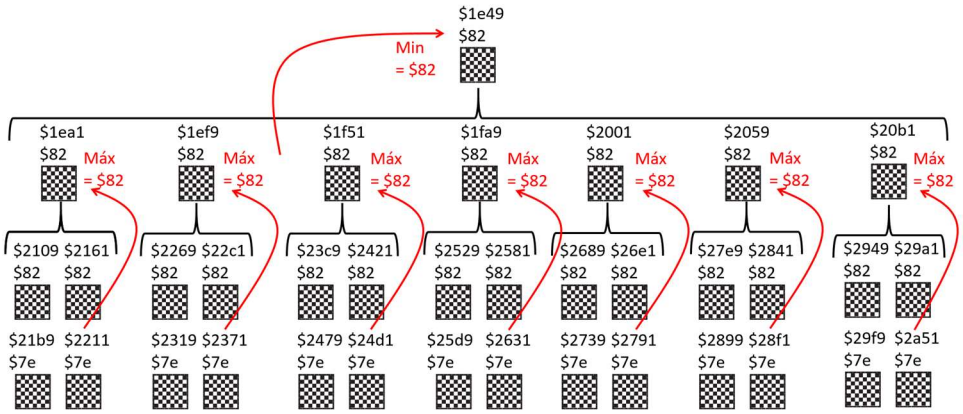
¡¡Por fin el C64 va a decidir su jugada!! ¿¿No es emocionante?? ☺

No sé si seréis conscientes, pero para llegar hasta aquí hemos tenido que recorrer todo este camino:

- Generar las jugadas o movimientos básicos de los gatos.
- Generar tableros hijo a partir de un tablero padre, aplicando un movimiento.
- Generar el primer nivel del árbol de juego.
- Corregir la validación de jugadas, que tenía errores.
- Generar un árbol de juego de profundidad N aplicando recursividad.
- Evaluar tableros aplicando criterios posicionales.
- Evaluar el árbol de juego completo con la función de evaluación.
- Pintar el árbol de juego completo con sus evaluaciones y vinculaciones entre tableros.
- Evaluar las hojas del árbol con la función de evaluación, y llevar estas evaluaciones hasta la raíz con el procedimiento minimax.

Todo este camino es para eso: ¡¡para que el C64 decida su jugada!! El camino ha sido tan largo que es fácil perder la perspectiva…

Gráficamente, la situación en la que nos encontramos es ésta:



Es decir, tenemos un tablero de partida (raíz del árbol), este tablero tiene una serie de hijos (posibles movimientos de los gatos) y, tras desarrollar el árbol de juego hasta una profundidad N y evaluarlo con minimax, hemos llegado a unas valoraciones para los hijos e, incluso, nos hemos quedado con la menor de ellas

(la más beneficiosa para los gatos: $82). Por tanto, ya sólo se trata de algo tan sencillo como **optar por el hijo que tenga esa valoración mínima**.

En este caso particular se ha producido un **empate**, porque todos los tableros hijo han sido valorados con $82. Por tanto, podemos optar por el primero de ellos, el último de ellos, elegir uno aleatoriamente, o tomar el que más rabia nos dé.

Si queremos darle emoción al asunto, y que el C64 juegue de una forma más imprevisible, podemos apoyarnos en el Jiffy Clock (posiciones $a0 – $a1 – $a2) para elegir uno de los tableros empatados de forma aleatoria. A estas alturas, y con el objeto de simplificar, optaremos por el primero de los tableros en empate.

Lo importante es no tomar el caso particular por el general. En un caso general, especialmente cuando las partidas estén más avanzadas, el C64 tendrá que elegir entre puntuaciones que serán **dispares**.

Para optar por el hijo de puntuación mínima hacemos lo siguiente:

Rutina “decideJugadaMin”:

En el fichero “EvalTableros.asm” dotamos una nueva rutina “decideJugadaMin” que, dado un tablero padre evaluado con minimax, determina el hijo que ha dado lugar a esa evaluación, es decir, determina el hijo con menor puntuación.

Primero recuperamos el valor del padre:

```
; Rutina para decidir la jugada de los gatos tras pasar miniMax
; Se queda con la primera jugada que tiene la puntuación mínima

djmTableroLo    byte $00
djmTableroHi    byte $00
djmValorPadre   byte $00
djmNumHijo      byte $00
djmHijoLo       byte $00
djmHijoHi       byte $00

djmTempX        byte $00

decideJugadaMin

        ; Obtiene el valor del tablero padre
        lda djmTableroLo
        sta dvTableroLo

        lda djmTableroHi
        sta dvTableroHi

        jsr dameValor
        sta djmValorPadre
```

Y luego recorremos los hijos buscando al que aporta esa valoración:

```
453         ; Recorre los hijos buscando el valor
454         ldx #$00
455
456 djmBucle
457
458         lda djmTableroLo
459         sta dhTableroLo
460
461         lda djmTableroHi
462         sta dhTableroHi
463
464         stx dhNumHijo
465
466         jsr dameHijo
467
468         lda dhHijoHi
469         beq djmSgteHijo
470
471         lda dhHijoLo
472         sta dvTableroLo
473
474         lda dhHijoHi
475         sta dvTableroHi
476
477         jsr dameValor
478
479         lda dvValor
480         cmp djmValorPadre
481         beq djmFin
482
483 djmSgteHijo
484
485         inx
486
487         cpx #$08
488         bne djmBucle
489
```

Por último, cuando damos con el hijo que aporta esa valoración (“beq djmFin”), nos quedamos con los datos de ese hijo (número de hijo y dirección):

```
489
490 djmFin
491
492         stx djmNumHijo
493
494         lda dhHijoLo
495         sta djmHijoLo
496
497         lda dhHijoHi
498         sta djmHijoHi
499
500         ldx djmTempX
501
502         rts
503
```

Obsérvese que en caso de empate entre varios hijos estaríamos optando por el primero, puesto que en cuanto encontramos un hijo con la valoración correcta

nos quedamos con él. Aquí es donde se podría meter aleatoriedad para darle más emoción al asunto.

Nuevo programa principal “RYG.asm”:

Ahora que ya sabemos localizar al mejor hijo, al predilecto, vamos a optar por él. Para ello, volvemos a modificar el programa principal “RYG.asm”:

```
; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

actualiza

        jsr pintaTableroActual

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

        jsr desarrollaArbolJugadas

        jsr evaluaArbolJugadas

        ;jsr pintaArbolJugadas

        jsr decideJugadaGatos

        jsr aplicaJugadaDecidida

        jmp actualiza
```

Es decir, tras la evaluación del árbol con minimax:

- Dejamos de pintar el árbol. Hasta ahora veníamos pintando el árbol para depurar la función de evaluación y el procedimiento minimax.
- Optamos por la jugada de menor puntuación con “decideJugadaGatos”.
- Aplicamos esa jugada con “aplicaJugadaDecidida”, igual que en su momento aplicamos la jugada elegida por el usuario humano con “aplicaJugadaSolicitada”.
- Y cerramos el bucle de juego con “jmp actualiza”, que vuelve a pintar el tablero actual (ya actualizado) y vuelve a pedir la jugada del humano.

Respecto a la rutina “decideJugadaGatos” básicamente es una llamada a la ya presentada “decideJugadaMin”:

```
739
740 decideJugadaGatos
741
742         lda #<raizArbol
743         sta djmTableroLo
744
745         lda #>raizArbol
746         sta djmTableroHi
747
748         jsr decideJugadaMin
749
750         rts
751
```

Y respecto a la rutina “aplicaJugadaDecidida”, consiste en copiar el hijo elegido sobre el tablero actual, el que controla la situación actual de la partida, y hacer alguna labor menor de inicialización:

```
751
752 aplicaJugadaDecidida
753
754         ; Copia el tablero decidido en el actual
755
756         lda djmHijoLo
757         sta ctshOrigenLo
758
759         lda djmHijoHi
760         sta ctshOrigenHi
761
762         lda #<tableroActual
763         sta ctshDestinoLo
764
765         lda #>tableroActual
766         sta ctshDestinoHi
767
768         jsr copiaTableroSinHijos
769
770         ; Inicializa el padre del tablero actual a $0000
771         ; Si no se inicializa, aparece como padre la raíz del árbol
772
773         lda #<tableroActual
774         sta fpTableroLo
775
776         lda #>tableroActual
777         sta fpTableroHi
778
779         lda #$00
780         sta fpPadreLo
781
782         lda #$00
783         sta fpPadreHi
784
785         jsr fijaPadre
786
787         rts
```

Resultado:

El resultado es la versión 14 del proyecto, que prácticamente ya está terminado. El juego está casi completo, ya que permite jugar al humano y al C64 de forma continuada, una jugada tras otra.

Queda detectar si la partida ha terminado, es decir, si tras un movimiento han ganado el ratón o los gatos. Pero esto ya lo haremos en la entrada siguiente.

RYG: condiciones de fin de partida

El C64 ya es capaz de generar el árbol de juego y decidir la jugada que más le interesa. De hecho, el humano y el C64 ya pueden jugar de forma continuada, alternando turnos.

Por tanto, el siguiente paso es detectar si la partida ha terminado o no, si han ganado el ratón o los gatos. En principio, la partida termina:

- Cuando el ratón llega a la fila cero. En este caso gana el ratón.
- Cuando los gatos acorralan al ratón. En este caso ganan los gatos.

Ahora bien, hay otra situación que, aunque improbable, interesa considerar. Supongamos que el ratón rebasa a los gatos, de modo que estos ya no pueden acorralarlo. Lo lógico sería que el ratón continuara directo hasta la fila cero para ganar la partida. Pero si el ratón es juguetón, y se dedica a dar un paso adelante y otro atrás, puede acabar ocurriendo que los gatos, que sólo pueden avanzar, se queden sin capacidad de moverse. En este caso, se considera que también gana el ratón porque ha escapado de los gatos.

Por tanto, son tres las condiciones de fin de partida:

- Si el ratón ha llegado a la fila cero, gana el ratón.
- Si los gatos no pueden moverse, también gana el ratón.
- Si el ratón no puede moverse, ganan los gatos.

Las dos primeras condiciones las valoraremos justo después de mover el ratón. La última condición la valoraremos justo después de mover los gatos.

<u>El ratón ha llegado a la fila cero:</u>

Tras mover el ratón, es decir, tras solicitar al usuario el movimiento que desea, aplicarlo y pintar el tablero, llamamos a la rutina “verificaRatonFila0”:

```
; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

        jsr pintaTableroActual

actualiza

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

ratonFila0

        jsr verificaRatonFila0

        lda vrf0
        bne gatos0Jugs

        jsr ganaRaton
        jmp inicializa

gatos0Jugs

        jsr verificaGatos0Jugs

        lda vg0j
        bne noGanaRaton

ganaRaton

        jsr ganaRaton2
        jmp inicializa
```

La rutina “verificaRatonFila0” verifica en qué fila se encuentra el ratón y, en caso de estar en la fila cero, lo señaliza con el valor $00 en el flag “vrf0”:

```
827 vrf0    byte $00
828
829 verificaRatonFila0
830
831         ; Inicializa
832
833         lda #$ff
834         sta vrf0
835
836         ; Ratón en fila 0?
837
838         lda #<tableroActual
839         sta drTableroLo
840
841         lda #>tableroActual
842         sta drTableroHi
843
844         jsr dameRaton
845
846         lda drOffset
847         sta dfcOffset
848
849         jsr dameFilaCol
850
851         lda dfcFila
852         bne vrf0False
853
854 vrf0True
855
856         lda #$00
857         sta vrf0
858
859 vrf0False
860
861         rts
862
```

Al final, si el ratón está en la fila cero se acaba ejecutando “ganaRaton2”, que hace esto:

```
863 ganaRaton2
864
865         lda #<grFelicitacion
866         sta cadenaLo
867
868         lda #>grFelicitacion
869         sta cadenaHi
870
871         jsr pintaCadena
872
873         lda #13
874         jsr chrout
875
876         lda #<grPulsaTecla
877         sta cadenaLo
878
879         lda #>grPulsaTecla
880         sta cadenaHi
881
882         jsr pintaCadena
883
884         lda #13
885         jsr chrout
886
887 grTecla
888
889         jsr getin
890         beq grTecla
891
892         rts
893
894 grFelicitacion  text "ha ganado el raton!! enhorabuena!!"
895                 byte $00
896
897 grPulsaTecla    text "pulsa una tecla para continuar..."
898                 byte $00
899
```

Es decir, pinta una felicitación, pide la pulsación de una tecla, espera a que se pulse, y termina, tras lo cual vuelve a inicializarse el juego.

Los gatos no pueden moverse:

Si el ratón no ha llegado a la fila cero, entonces llamamos a la rutina “verificaGatos0Jugs”:

```
; 10 SYS2064

*=$0801

        BYTE    $0B, $08, $0A, $00, $9E, $32, $30, $36, $34, $00, $00, $00

; El programa empieza aquí

*=$0810

RYG

inicializa

        jsr pintaTitulo

        jsr solicitaProfundidad

        jsr inicializaTableroActual

        jsr pintaTableroActual

actualiza

        jsr pintaJugadasRaton

        jsr solicitaJugadaRaton

        jsr aplicaJugadaSolicitada

        jsr pintaTableroActual

ratonFila0

        jsr verificaRatonFila0

        lda vrf0
        bne gatos0Jugs

        jsr ganaRaton
        jmp inicializa

gatos0Jugs

        jsr verificaGatos0Jugs

        lda vg0j
        bne noGanaRaton

ganaRaton

        jsr ganaRaton2
        jmp inicializa
```

La rutina “verificaGatos0Jugs” va obteniendo el número de jugadas posibles del ratón 0, el ratón 1, el ratón 2 y el ratón 3. Para ello utiliza la nueva rutina “cuentaJugadasGato”, que cuenta las jugadas del gato indicado (0, 1, 2 o 3).

Si en algún caso el número de jugadas es superior a cero, entonces directamente devuelve falso ($ff en el flag “vg0j”). Si el número de jugadas es cero en todos los casos, entonces devuelve cierto ($00 en el flag “vg0j”):

```
968  vg0j    byte $00
969
970  verificaGatos0Jugs
971
972          ; Inicializa
973
974          lda #$ff
975          sta vg0j
976
977          ; Número de jugadas del ratón 0
978
979          lda #<tableroActual
980          sta cjgTableroLo
981
982          lda #>tableroActual
983          sta cjgTableroHi
984
985          lda #$00
986          sta cjgNumGato
987
988          jsr cuentaJugadasGato
989
990          lda cjgNumJugadas
991          bne vg0jFalse
992
993          ; Número de jugadas del ratón 1
994
995          lda #<tableroActual
996          sta cjgTableroLo
997
998          lda #>tableroActual
999          sta cjgTableroHi
1000
1001         lda #$01
1002         sta cjgNumGato
1003
1004         jsr cuentaJugadasGato
1005
1006         lda cjgNumJugadas
1007         bne vg0jFalse
1008
1009         ; Número de jugadas del ratón 2
1010
```

```
1008
1009            ; Número de jugadas del ratón 2
1010
1011            lda #<tableroActual
1012            sta cjgTableroLo
1013
1014            lda #>tableroActual
1015            sta cjgTableroHi
1016
1017            lda #$02
1018            sta cjgNumGato
1019
1020            jsr cuentaJugadasGato
1021
1022            lda cjgNumJugadas
1023            bne vg0jFalse
1024
1025            ; Número de jugadas del ratón 3
1026
1027            lda #<tableroActual
1028            sta cjgTableroLo
1029
1030            lda #>tableroActual
1031            sta cjgTableroHi
1032
1033            lda #$03
1034            sta cjgNumGato
1035
1036            jsr cuentaJugadasGato
1037
1038            lda cjgNumJugadas
1039            bne vg0jFalse
1040
1041    vg0jTrue
1042
1043            lda #$00
1044            sta vg0j
1045
1046    vg0jFalse
1047
1048            rts
1049
```

Al final, igual que cuando el ratón estaba en la fila cero, si el número de movimientos de los gatos es cero se acaba ejecutando “ganaRaton2”, que lógicamente hace lo ya mencionado (felicita al ratón, pide una tecla, e inicializa el juego).

<u>El ratón no puede moverse:</u>

Esta condición es similar a la anterior, pero aplicada al ratón. Si el ratón no ha ganado, el C64 desarrolla y evalúa el árbol de jugadas, decide la jugada que más le interesa, la aplica, pinta el tablero, y llama a “verificaRaton0Jugs”:

```
71  noGanaRaton
72
73          jsr desarrollaArbolJugadas
74
75          jsr evaluaArbolJugadas
76
77          ;jsr pintaArbolJugadas
78
79          jsr decideJugadaGatos
80
81          jsr aplicaJugadaDecidida
82
83          jsr pintaTableroActual
84
85  raton0Jugs
86
87          jsr verificaRaton0Jugs
88
89          lda vr0j
90          bne noGananGatos
91
92  gananGatos
93
94          jsr gananGatos2
95          jmp inicializa
96
97  noGananGatos
98
99          jmp actualiza
100
```

La rutina “verificaRaton0Jugs” es muy parecida a “verificaGatos0Jugs”, siendo la principal diferencia que sólo hay un ratón frente a cuatro gatos:

```
899
900 vr0j    byte $00
901
902 verificaRaton0Jugs
903
904         ; Inicializa
905
906         lda #$ff
907         sta vr0j
908
909         ; Número de jugadas del ratón?
910
911         lda #<tableroActual
912         sta cjrTableroLo
913
914         lda #>tableroActual
915         sta cjrTableroHi
916
917         jsr cuentaJugadasRaton
918
919         lda cjrNumJugadas
920         bne vr0jFalse
921
922 vr0jTrue
923
924         lda #$00
925         sta vr0j
926
927 vr0jFalse
928
929         rts
930
```

Usando la nueva rutina “cuentaJugadasRaton” contamos el número de jugadas del ratón y, caso de ser cero, lo señalizamos con $00 en el flag “vr0j”.

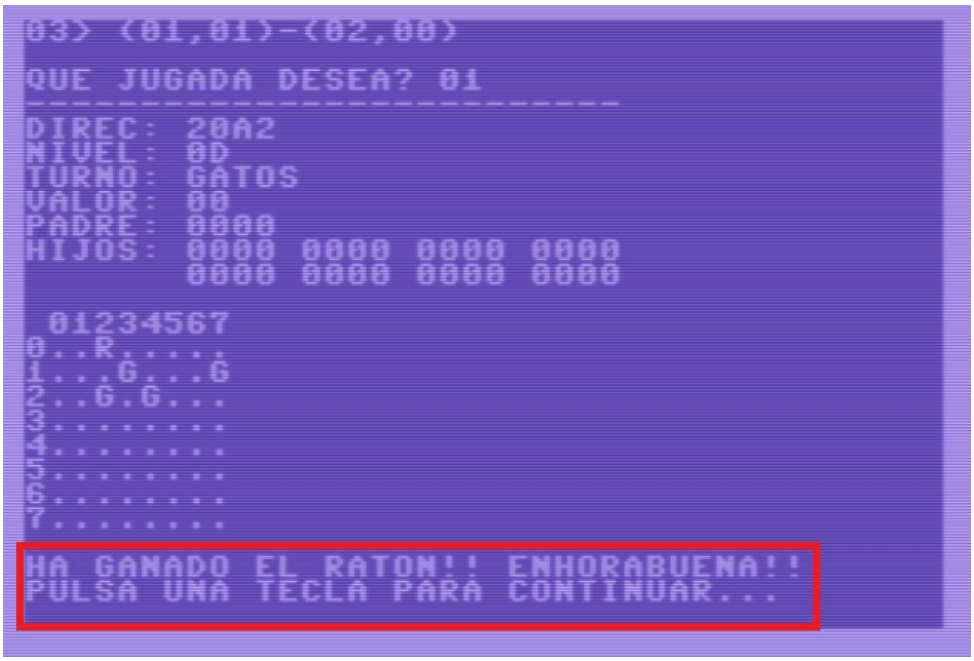
Al final, si el número de jugadas del ratón es cero se ejecuta “gananGatos2”:

```
931 gananGatos2
932
933         lda #<ggPesame
934         sta cadenaLo
935
936         lda #>ggPesame
937         sta cadenaHi
938
939         jsr pintaCadena
940
941         lda #13
942         jsr chrout
943
944         lda #<ggPulsaTecla
945         sta cadenaLo
946
947         lda #>ggPulsaTecla
948         sta cadenaHi
949
950         jsr pintaCadena
951
952         lda #13
953         jsr chrout
954
955 ggTecla
956
957         jsr getin
958         beq ggTecla
959
960         rts
961
962 ggPesame        text "oh...!! han ganado los gatos!!"
963                 byte $00
964
965 ggPulsaTecla    text "pulsa una tecla para continuar..."
966                 byte $00
967
```

Esta rutina, de forma análoga a “ganaRaton2”, pinta un mensaje (ahora es un lamento en vez de una felicitación ☹), pide una tecla, espera a que se pulse, y vuelve a inicializar el juego.

Resultado:

El resultado es que se permite el juego continuado de ratón y gatos y, además, ya se detectan las condiciones de final de partida. Por ejemplo, en este caso ha ganado el ratón:



RYG: ampliar la RAM disponible

El juego ya esté esencialmente completo: permite que el humano y el C64 jueguen de forma continuada, alternando turnos, y detecta las condiciones de fin de partida.

Ahora bien, si probamos a jugar con la versión 15 del proyecto, da igual que sea con una profundidad de análisis de 1, 2 o 3 niveles, se verá que es relativamente fácil ganar al C64. Por tanto, tenemos que conseguir que el C64 juegue mejor.

Básicamente hay dos formas de conseguir que le C64 juegue mejor:

- O aumentando la profundidad de análisis.
- O mejorando la función de evaluación.

Ya hemos comentado varias veces que la profundidad del árbol de juego y la función de evaluación conforman una especie de compromiso. Si el árbol de juego pudiera generarse completo, bastaría con una función de evaluación muy tonta. Pero como el árbol no puede ser completo, porque la memoria es limitada,

la función de evaluación tiene que ser más lista, tiene que permitir identificar aquellas ramas del juego que son más prometedoras.

Aumentar la profundidad de análisis:

Para aumentar la profundidad de análisis:

- O aumentamos la memoria disponible.
- O compactamos las estructuras de datos.
- O ambas cosas.

De todo esto ya hablamos largo y tendido en la entrada "RYG: árbol de juego – memoria revisitada", y ya decidimos entonces no compactar las estructuras de datos (complicaba mucho la programación) pero sí ampliar la RAM disponible. De hecho, quedó pospuesto hasta esta entrada.

Para ampliar la RAM disponible nos aprovechamos de las funciones de configuración del mapa de memoria del C64. Concretamente, actuando sobre el registro R6510 = $0001 podemos activar y desactivar diferentes partes de la memoria del C64:

Bit	**Nombre del bit**	**Si vale 0**	**Si vale 1**
0	LORAM	Las direcciones $a000 – $bfff direccionan RAM	Las direcciones $a000 – $bfff direccionan la ROM con el intérprete de BASIC
1	HIRAM	Las direcciones $e000 – $ffff direccionan RAM	Las direcciones $e000 – $ffff direccionan la ROM con el Kernal
2	CHAREN	Las direcciones $d000 – $dfff direccionan la ROM con el mapa de caracteres	Las direcciones $d000 – $dfff direccionan los chips especiales y la RAM de color

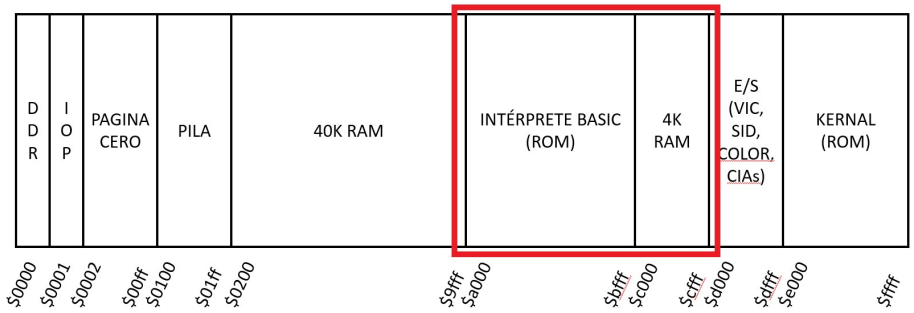
Podemos desactivar el intérprete de BASIC, porque no lo utilizamos. Sin embargo, no podemos desactivar el Kernal, porque sí utilizamos rutinas como “chrout”.

Todo esto lo hacemos con la nueva rutina “ampliaRam”, que llamamos al arrancar el programa, y que básicamente pone a 0 el bit 0 (LORAM) del registro R6510 = $0001:

```
119
120 ampliaRam
121
122         ; Ponemos LORAM a 0
123         ; Esto permite aprovechar $a000 - $bfff para tableros
124         ; $c000 - $cfff también son aprovechables
125
126         lda $0001
127         and #%11111110
128         sta $0001
129
130         rts
131
```

De este modo, ganamos como RAM disponible el rango de direcciones del intérprete de BASIC ($a000 – $bfff = 8 KB), así como el rango $c000 – $cfff = 4 KB que, si bien siempre es RAM, al no estar contiguo hasta ahora con el espacio de trabajo del programa ($0801 – $9fff), no podíamos aprovecharlo fácilmente.

En total, ganamos estos 12 KB:



Sin embargo, incluso así es insuficiente para ganar otro nivel de análisis (pasar de tres a cuatro niveles) porque, como ya calculamos en su momento, para tres niveles hacían falta 22 KB y para cuatro 90 KB.

Total, nuestra principal esperanza es mejorar la función de evaluación, cosa que haremos en las entradas que siguen.

RYG: función de evaluación – segunda versión

Bueno, pues vamos a intentar mejorar la función de evaluación para que el C64 juegue mejor. Hasta ahora veníamos considerando estos dos criterios posicionales:

- La fila del ratón.
- El número de movimientos del ratón.

Esto se puede ver en la rutina “evaluaTablero” del fichero “EvalTableros.asm”:

```
37  etTableroLo     byte $00
38  etTableroHi     byte $00
39  etValor         byte $00
40
41  evaluaTablero
42
43          ; Inicializa el valor
44          lda #$80
45          sta etValor
46
47          ; Evalúa la fila del ratón
48          jsr evaluaFilaRaton
49
50          ; Evalúa los movimientos del ratón
51          jsr evaluaMovsRaton
52
53          ; Mete el valor en el propio tablero evaluado
54          jsr meteValorTablero
55
56          rts
57
```

Parecen criterios un poco pobres, insuficientes para saber si un tablero es prometedor para el ratón o los gatos, que en el fondo es la información que usa el C64 para decidir su jugada.

Pero podemos observar que, si los gatos consiguieran guardar una fila cerrada, o al menos lo intentaran, sería mucho más difícil para el ratón rebasarlos. Mantener una fila cerrada no será siempre posible, ya que el ratón puede forzar que los gatos rompan la formación acercándose a ellos todo lo posible. Recordemos que en este juego no hay capturas.

Podemos intentar que los gatos guarden una fila cerrada obteniendo la fila mínima de los gatos, la máxima, y calculando la diferencia entre ellos. Si esa diferencia es cero, los gatos tienen que estar en fila; si es uno, no será una fila, pero al menos los gatos no estarán muy dispersos por el tablero; y así sucesivamente.

Total, añadimos la llamada a la nueva rutina “evaluaDifFilasGatos” en “evaluaTablero”:

```
37
38  etTableroLo     byte $00
39  etTableroHi     byte $00
40  etValor         byte $00
41
42  evaluaTablero
43
44          ; Inicializa el valor
45          lda #$80
46          sta etValor
47
48          ; Evalúa la fila del ratón
49          jsr evaluaFilaRaton
50
51          ; Evalúa las jugadas del ratón
52          jsr evaluaJugsRaton
53
54          ; Evalúa la diferencia de filas de los gatos
55          jsr evaluaDifFilasGatos
56
57          ; Mete el valor en el propio tablero evaluado
58          jsr meteValorTablero
59
60          rts
61
```

La nueva rutina “evaluaDifFilasGatos” es así:

```
; Rutina para evaluar la diferencia de filas de los gatos

edfTempX        byte $00
tablaDifs       byte $f0,$f8,$00,$00,$00,$00,$00,$00 ; Números negativos

evaluaDifFilasGatos

        stx edfTempX

        ; Calcula la max fila
        jsr maxFilaGatos

        ; Calcula la min fila
        jsr minFilaGatos

        ; Obtiene la diferencia entre max fila y min fila
        lda maxfFila
        sec
        sbc minfFila

        ; Obtiene la valoración en función de esa diferencia
        tax
        lda tablaDifs,x

        ; Y la suma a la valoración por jugadas del ratón
        clc
        adc etValor
        sta etValor

        ldx edfTempX

        rts

```

Es decir, calcula la fila máxima con “maxFilaGatos”, la fila mínima con “minFilaGatos”, calcula la diferencia entre máximo y mínimo con la instrucción “sbc” y, el resultado (0, 1, 2, …, 7) lo usa como índice para acceder a la tabla con las evaluaciones parciales. Si la diferencia es 0 la evaluación parcial será $f0 = -16, si la diferencia es 1 será $f8 = -8 y, a partir de ahí, cero. Estas evaluaciones parciales son negativas porque se supone que guardar una fila cerrada, o casi, es una situación que favorece a los gatos.

Para calcular la fila máxima y la fila mínima las rutinas “maxFilaGatos” y “minFilaGatos” hacen básicamente lo mismo que las rutinas equivalentes del procedimiento minimax (máximo valor y mínimo valor), es decir, partir del valor más bajo ($00) o más alto ($ff), ir comparando las filas de los gatos contra ese valor, y quedarse al final con el valor más alto o más bajo de los comparados. Nada nuevo que merezca la pena detallar más.

Si tras este cambio de la versión 16 ensamblamos y jugamos, veremos que el C64 ya juega mejor. El juego no es tan naif como en la versión 15. Parece que el C64

ya juega como con más intención, como con más “mala leche”, buscando tapar las vías de escape. Aun así, todavía es posible ganarle con relativa facilidad, por lo que tendremos que seguir mejorando la función de evaluación.

RYG: función de evaluación – tercera versión

Hay otro criterio que sería muy interesante introducir en la función de evaluación. En teoría, el ratón gana el juego cuando llega a la primera línea del tablero. Sin embargo, en cuanto el ratón rebase la línea de gatos “ya estará todo el pescado vendido”.

Por tanto, si en la función de evaluación metemos un criterio del tipo “si el ratón rebasa la línea de gatos sumar X puntos” estamos consiguiendo que el C64 identifique las victorias del ratón varios movimientos antes de que se produzcan de forma efectiva.

Para conseguir esto mejoramos la función de evaluación con una nueva llamada “jsr evaluaRatonRebasaGatos”:

```
37  ; Para evitar manejar números positivos y negativos, que siempre es más
38  ; complejo trabajar con ellos (ej. calcular el máximo o el mínimo en la
39  ; función minimax) partimos de un valor neutral de $80. A este valor le
40  ; sumaremos y restaremos los puntos anteriores, manejando siempre números
41  ; positivos.
42
43  etTableroLo     byte $00
44  etTableroHi     byte $00
45  etValor         byte $00
46
47  evaluaTablero
48
49          ; Inicializa el valor
50          lda #$80
51          sta etValor
52
53          ; Evalúa la fila del ratón
54          jsr evaluaFilaRaton
55
56          ; Evalúa las jugadas del ratón
57          jsr evaluaJugsRaton
58
59          ; Evalúa la diferencia de filas de los gatos
60          jsr evaluaDifFilasGatos
61
62          ; Evalúa si ratón rebasa a los gatos
63          jsr evaluaRatonRebasaGatos
64
65          ; Mete el valor en el propio tablero evaluado
66          jsr meteValorTablero
67
68          rts
```

A su vez, la nueva rutina “evaluaRatonRebasaGatos” es así:

```
; Rutina para evaluar si el ratón ha rebasado la fila de gatos

evaluaRatonRebasaGatos

        ; Obtiene la fila mímina de los gatos
        jsr minFilaGatos

        ; Obtiene la fila del ratón
        lda etTableroLo
        sta drTableroLo

        lda etTableroHi
        sta drTableroHi

        jsr dameRaton

        lda drOffset
        sta dfcOffset

        jsr dameFilaCol

        ; Compara la fila del ratón y la mínima de los gatos
        lda minfFila
        cmp dfcFila
        bcc errgNoRebasa

errgSiRebasa

        ;inc $d020 ; Para hacer una señal visual si hay riesgo de rebase

        lda #$20
        clc
        adc etValor
        sta etValor

errgNoRebasa

        rts
```

Es decir, obtiene la fila mínima de los gatos con “minFilaGatos” (rutina ya conocida), la fila del ratón con “dameRaton” y “dameFilaCol” (rutinas ya conocidas), compara ambas (instrucción “cmp”) y, si la fila mínima de los gatos es estrictamente menor (instrucción “bcc”) que la fila del ratón, se decide que no hay rebase. En caso contrario, es decir, si la fila mínima de los gatos es mayor o igual que la fila del ratón, entonces se decide que el ratón ha rebasado a los gatos, y se suman $20 = 32 puntos a la evaluación del tablero.

Esta mejora de la función de evaluación puede verse en la versión 17 del proyecto, que todavía juega un poquito mejor.

RYG: función de evaluación – más mejoras

Como decíamos, el C64 ya juega mejor. Tras mejorar la formación de los gatos y detectar los rebases, ya es más difícil ganarle. Sin embargo, todavía es posible hacerlo, por lo que el proceso de mejora de la función de evaluación debería continuar.

De hecho, el proceso de mejora de la función de evaluación puede continuar casi indefinidamente. Iterativamente se pueden identificar nuevos criterios, introducirlos en el juego, probarlos y, en función de su contribución a la fortaleza del juego, mantenerlos o retirarlos.

Otra forma de mejorar la función de evaluación sería jugando muchas partidas, ya sea contra un humano o incluso contra sí mismo. Para esto último habría que adaptar un poco el juego, pero tampoco tanto, puesto que el C64 ya "sabe" mover el ratón. En vez de pedir la jugada del ratón al humano, debería decidirla el C64 en base a un árbol de juego, igual que ya hace en el caso de los gatos.

Sea como fuere, cuando el C64 pierda, se deberá identificar el movimiento que ha sido clave en la derrota y, más en particular, el criterio que ha hecho que el C64 se haya decantado por ese movimiento, reduciendo su peso o su puntuación para evitar que se repita. Y si esto (aislar el criterio clave en la derrota y ajustar su peso) fuéramos capaces de hacerlo automáticamente, sin intervención de un programador, ya casi estaríamos hablando de *machine learning*.

En cualquier caso, la mejora de la función de evaluación es un proceso iterativo y largo. Y si después del proceso no se han conseguido los resultados esperados habría que plantearse alternativas, como ampliar la profundidad de análisis compactando las estructuras de datos.

RYG: tableros más compactos – mayor profundidad de análisis

Ya llevamos tres versiones de la función de evaluación que permite al C64, junto con el procedimiento minimax, decidir sus jugadas.

En la primera versión, hemos considerado como criterios la fila del ratón y el número de jugadas que puede hacer (incluyendo las situaciones particulares de que la fila del ratón sea cero – en cuyo caso gana el ratón – y de que su número de jugadas sea cero – en cuyo caso ganan los gatos –). En la segunda versión, hemos añadido como criterio que los gatos guarden una formación (una fila o

dos). Y en la tercera versión, hemos añadido como criterio detectar si el ratón rebasa a los gatos. Y así deberíamos continuar añadiendo y revisando criterios hasta conseguir que el juego sea lo suficientemente fuerte.

Si a pesar de añadir muchos criterios no se consigue un juego fuerte, una medida complementaria (no sustitutiva) es ampliar la profundidad de análisis. Para ello ya sabemos que necesitamos más memoria y, si no disponemos de ella, hay que aprovechar mejor la disponible. Dicho de otro modo, hay que compactar las estructuras de datos.

Esto ya lo habíamos comentado muchas veces, y hasta ahora nos habíamos resistido por no complicar la programación, pero por fin ha llegado el momento.

Nueva estructura de datos compactada:

Los tableros usados hasta ahora eran así:

- Cabecera: 3 bytes.
- Nivel, turno y valor: 3 bytes.
- Dirección del padre: 2 bytes.
- Direcciones de los hijos: 2 x 8 = 16 bytes.
- Tablero propiamente dicho: 8 x 8 = 64 bytes.
- Total: 88 bytes.

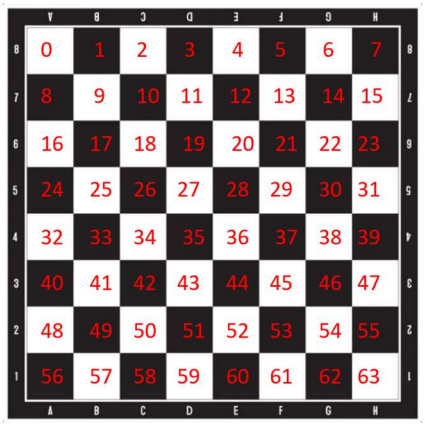
De esos 88 bytes, la gran mayoría (64 bytes, el 73%) son el tablero propiamente dicho, es decir, el escaqueado de 8 x 8 casillas. Por tanto, éste es el gran candidato a una fuerte reducción.

Teniendo en cuenta que sólo tenemos cinco piezas, y que no hay capturas, esos 64 bytes se puede reducir a sólo cinco:

- Un byte para guardar la posición del ratón.
- Cuatro bytes para guardar las posiciones de los gatos.

Todas las demás posiciones del tablero, se consideran vacías.

Las posiciones del ratón y los gatos se podrían guardar en formato (fila, columna) pero, puestos a ahorrar memoria, casi mejor guardar los offsets (desplazamientos) respecto a la casilla (0, 0) del tablero, es decir, esto:



Es decir, en la posición inicial del juego los offsets serían:

- Gatos = 0, 2, 4 y 6.
- Ratón = 59.

De este modo, el tablero compactado queda así:

- Cabecera: 3 bytes.
- Nivel, turno y valor: 3 bytes.
- Dirección del padre: 2 bytes.
- Direcciones de los hijos: 2 x 8 = 16 bytes.
- Tablero propiamente dicho: 5 bytes.
- Total: 29 bytes.

Es decir, hemos conseguido una reducción de 88 – 29 = 59 bytes, del 67%.

Reducciones adicionales:

El tablero se puede seguir compactando. El siguiente candidato por tamaño sería la tabla con las direcciones de los hijos (16 bytes), ya que no todos los tableros tendrán ocho hijos.

Se podría hacer una tabla de tamaño variable, marcando el final de la misma con algún marcador. O se podrían diseñar dos estructuras de datos, una con un

máximo de ocho hijos para cuando muevan los gatos y otra con un máximo de cuatro hijos para cuando mueva el ratón.

En cualquier caso, es improbable que ahorrar 4 hijos / 8 bytes, y sólo en algunos tableros, permita ganar niveles adicionales en la profundidad de análisis, especialmente si se tiene en cuenta que el crecimiento del árbol de juego (número de tableros) es exponencial con la profundidad.

También se podría quitar la cabecera (tres bytes), aunque facilita la depuración de los datos y los programas, porque facilita identificar dónde empiezan los tableros en memoria.

Como no parecen opciones muy prometedoras, de momento, nos conformaremos con los tableros de 29 bytes.

Nueva profundidad de análisis:

Con este nuevo tamaño de tablero, las previsiones de consumo de memoria en función de los niveles de profundidad quedan así:

- 1 nivel → 29 bytes x 8 movimientos = 232 bytes.
- 2 niveles → 29 bytes x 8 x 4 = 928 bytes.
- 3 niveles → 29 bytes x 8 x 4 x 8 = 7424 bytes.
- 4 niveles → 29 bytes x 8 x 4 x 8 x 4 = 29696 bytes.
- 5 niveles → 29 bytes x 8 x 4 x 8 x 4 x 8 = 237568 bytes.

Por tanto, al compactar los tableros de 88 a 29 bytes conseguimos subir de tres a cuatro niveles en la profundidad de análisis (29 KB). No podemos subir a cinco niveles porque el C64 no tiene esos 237 KB.

Es decir, a la hora de decidir su jugada, ahora el C64 podrá tener en cuenta los efectos de su jugada inmediata, la siguiente jugada del humano, nuevamente la jugada del C64 y, por último, la siguiente jugada del humano.

El número máximo de tableros a analizar por cada jugada será de 8 x 4 x 8 x 4 = 1024 tableros, aunque con frecuencia serán menos porque no siempre serán posibles todas las jugadas que en principio permiten las reglas de movimiento.

Todos estos cambios tienen su impacto en el código. Esto lo veremos ya en la entrada que sigue, aunque se adelanta la versión 18 del proyecto.

RYG: tableros más compactos – cambios en el código

Los cambios en el diseño del tablero tienen su impacto en el código del juego. Además de una reducción obvia en el tamaño de la estructura de datos (de 88 a 29 bytes) hay un cambio en el enfoque de los datos: antes teníamos una matriz de 8x8 casillas que podían estar vacías, ocupadas por un ratón o por gatos; ahora tenemos directamente la posición de los cinco protagonistas y todo lo demás se presupone vacío.

Lógicamente tienen que cambiar cosas en el código como la forma de generar las jugadas o la forma de evaluar los tableros. Sin embargo, como veremos más adelante, y sorprendentemente, los cambios son mucho más acotados de lo que podría parecer en un principio.

Nuevo tablero actual:

Si la estructura de datos del tablero cambia, lo primero que cambia es el tablero actual, el que controla la situación actual de la partida (ver fichero “Arbol.asm”). En vez de tener 88 bytes, tendrá 29.

Nuevas rutinas para el manejo de tableros:

Más importante que lo anterior, también cambian las rutinas que permiten manejar tableros, es decir, las rutinas del fichero “Tableros.asm”. Sorprendentemente, muchas rutinas no cambian porque sólo manejan los datos de los tableros que no hemos tocado (nivel, turno, valor, padre o hijos).

Sin embargo, otras rutinas sí tienen que cambiar:

- La rutina “inicializaTablero” ya no tiene que rellenar un tablero con 64 casillas vacías. Ahora sólo tiene que fijar las posiciones iniciales del ratón (offset = 59) y los gatos (offsets = 0, 2, 4 y 6), ya que todo lo demás se presupone vacío.
- Las rutinas “dameContenido” y “fijaContenido” dejan de tener sentido. Ya no tenemos 64 casillas que rellenar con un contenido (#Raton, #Gato

o #Vacio). Ahora tenemos cinco bytes con las posiciones (offsets) de ratón y gatos. Estas posiciones se especifican con las nuevas rutinas “fijaPosicionRaton” y “fijaPosicionGato”.

- Las rutinas “dameOffset” y “dameFilaCol” antes empezaban a contar los offsets desde el comienzo de la estructura de datos. Por tanto, la casilla (0, 0) del tablero recibía el offset 24. Ahora cambiamos el criterio y a la casilla (0, 0) le asignamos el offset 0.
- Las rutinas “dameRaton” y “dameGato” antes recorrían la matriz de 8x8 casillas para buscar y localizar el ratón o el gato enésimo (su offset). Ahora no hace falta buscar nada; tenemos la información a tiro hecho en los cinco bytes que sustituyen a la matriz de 8x8. Por tanto, esas rutinas se sustituyen por las nuevas rutinas “damePosicionRaton” y “damePosicionGato”.

Y poco más hasta aquí.

Nueva generación de jugadas:

La generación de jugadas del ratón está en el fichero “GenJugadasRaton.asm”. La pieza clave anteriormente eran las tablas con los movimientos permitidos:

```
; Fichero para generar jugadas, es decir, desarrollar el árbol

; Jugadas que puede hacer el ratón

tblRatonF       byte $ff, $ff, $01, $01
tblRatonC       byte $ff, $01, $01, $ff

```

Es decir, los cambios en la posición (fila, columna) podían ser:

- ($ff, $ff) = (-1, -1).
- ($ff, $01) = (-1, 1).
- ($01, $01) = (1, 1).
- ($01, $ff) = (1, -1).

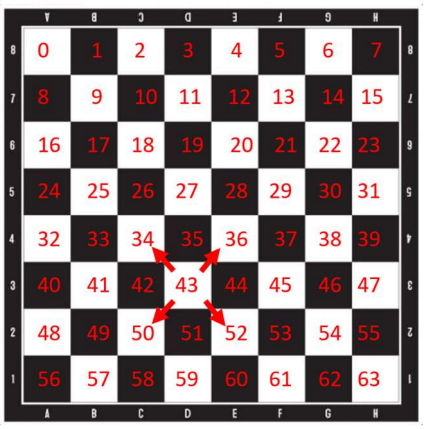
Como ahora no tenemos una matriz de 8x8 posiciones, ni tampoco gestionamos las posiciones en formato (fila, columna), sino en formato offset, esa pieza clave se convierte en esta otra:

```
3  ; Jugadas que puede hacer el ratón (-9, -7, +9, +7)
4
5  tblRaton        byte $f7, $f9, $09, $07
6
```

Es decir, el offset del ratón puede variar en:

- $f7 = -9.
- $f9 = -7.
- $09 = +9.
- $07 = +7.

Estas variaciones corresponden a estos cuatro movimientos (suponiendo que el ratón estuviera en el offset 43):



De este modo, generar los movimientos posibles consiste en tomar el offset actual (43), recorrer las cuatro posiciones de la tabla, e ir sumando:

- 43-9=34.
- 43-7=36.
- 43+9=52.
- 43+7=50.

En el fondo, muy parecido a lo que ya hacíamos antes, pero manejando variaciones en el offset (-9, -7, +9, +7) en vez de variaciones en filas y columnas.

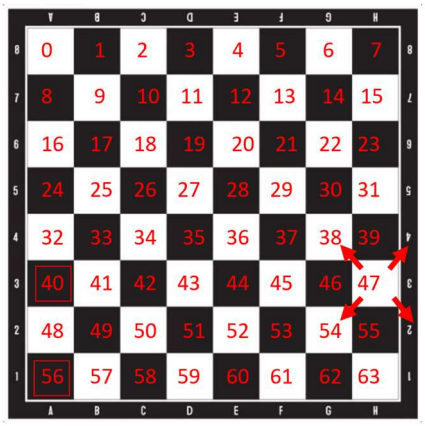
Otra pieza importante es la validación de las jugadas. Para que éstas sean válidas las reglas son (antes y ahora):

- Tienen que generarse conforme a las reglas de movimiento de las piezas.
- El destino tiene que estar vacío.
- El destino tiene que caer dentro de los márgenes del tablero.

La primera condición se garantiza por la forma de generar las jugadas, partiendo de una tabla (antes tablas) que recogen los movimientos válidos.

La segunda es igual de fácil de comprobar ahora que antes. Antes teníamos la (fila, columna) destino; ahora tenemos el offset destino. Si el offset destino está ocupado por el ratón o algún gato, no será un destino válido. En caso contrario, sí lo será.

Y la tercera condición cambia un poco. Antes comprobábamos si la fila o la columna eran menores que cero o mayores que siete. Ahora tenemos un offset entre 0 y 63, y sumar o restar 7 o 9 puede hacer que se traspasen los bordes del tablero sin detectarlo, si no hacemos nada especial. Por ejemplo, 47-7=40 o 47+9=56:



Una forma inteligente de impedir estos traspasos es dándose cuenta de que ratón y gatos siempre se mueven por las casillas blancas (offsets en blanco) y, cuando tiene lugar un traspaso ilegal de este tipo, acaban en una casilla negra (ver 40 o

56). Por tanto, llega con validar que el offset de destino es blanco (0, 2, 4, 6, 9, 11, 13, 15, …, 61, 63). Esto se puede hacer con una tabla de offsets válidos:

```
42  vjOffsets       byte 0,2,4,6
43                  byte 9,11,13,15
44                  byte 16,18,20,22
45                  byte 25,27,29,31
46                  byte 32,34,36,38
47                  byte 41,43,45,47
48                  byte 48,50,52,54
49                  byte 57,59,61,63
50
51  validaJugada
52
```

Por lo demás, todas las rutinas de “GenJugadasRaton.asm” (generar jugadas, validar jugadas, generar jugadas válidas, etc.) son esencialmente iguales a las que ya había antes, con la salvedad de que las posiciones de origen y destino se especifican mediante offsets, y no mediante parejas (fila, columna).

Y lo mismo se puede decir de “GenJugadasGatos.asm”.

Otros cambios:

Los cambios principales son los ya comentados: rutinas para manejar tableros y generación de jugadas. En el código surgen más cambios (evaluación de tableros, etc.) pero son cambios derivados de los anteriores.

En particular, el hecho de cambiar las rutinas “dameRaton” y “dameGato”, que buscaban en la matriz 8x8, por las nuevas rutinas “damePosicionRaton” y “damePosicionGato”, que devuelven las posiciones a tiro hecho, implica bastantes cambios en la evaluación de tableros (“EvalTableros.asm”), pero son más estéticos (por el cambio de nombre en las rutinas) que otra cosa.

También hay cambios menores en “PintaTableros.asm”. Antes los tableros tenían una matriz 8x8 que pintar (con “G”, “R” o un “.”). Ahora gráficamente hay que pintar la misma matriz 8x8, porque la representación gráfica del tablero no ha cambiado, pero la información de partida es muy distinta (cinco offsets).

Los cambios en el programa principal “RYG.asm” también se limitan a manejar las posiciones mediante offsets, en vez de mediante filas y columnas, o bien cambios en los nombres de algunas rutinas llamadas. Además, está el cambio obvio de

pasar de tres a cuatro niveles de profundidad que, no olvidemos, es el motivo por el que hemos montado todo este lío ☺ .

Por último, un cambio de presentación. Como ahora podemos analizar hasta cuatro niveles de profundidad, es decir, hasta 1024 tableros, pintar "PENSANDO:" y un punto por cada tablero son muchos puntos. Por ello, se prefiere pintar la dirección del tablero que se está generando. De este modo, consumimos menos pantalla, hacemos menos scroll, tenemos igual idea del avance, y encima sabemos si el árbol consume más o menos memoria:

Todos estos cambios pueden verse en la versión 18 del proyecto. Lo bueno de esta versión es que ya permite hasta cuatro niveles de profundidad.

Ah, y una lección que ganamos: lo más intuitivo para el programador (ej. tener un tablero de 8x8) no es siempre lo más intuitivo ni lo mejor para la máquina.

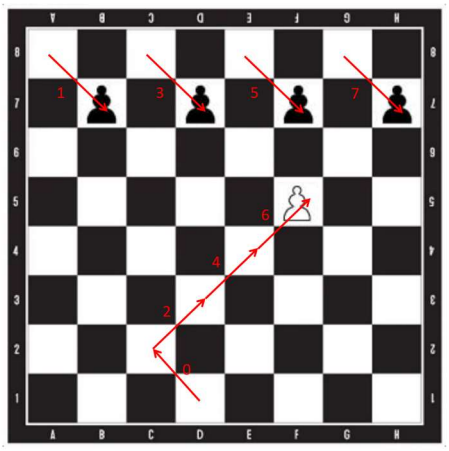
RYG: inicios y finales de partida

Los aficionados al ajedrez sabrán que es muy habitual estudiar aperturas y finales de partida, porque son situaciones especiales en las que conviene saber cómo actuar. Pues bien, una funcionalidad interesante para el juego consiste en añadir una “base de datos” de inicios y/o finales de partida.

Los inicios de partida son fáciles de añadir, porque es fácil identificar cuándo una partida se encuentra al inicio. Por ejemplo, si el número de movimientos aplicados es menor que X está claro que estamos ante un inicio de partida.

Por el contrario, los finales de partida son mucho más difíciles de identificar y, por tanto, de aplicar. No hay un umbral de movimientos a partir del cual podemos suponer que estamos ante un final de partida. Por tanto, se trata más bien de reconocer situaciones o patrones (ej. número de piezas, posiciones, etc.). En consecuencia, es algo bastante más complejo de implementar.

En el caso que nos ocupa, siempre que el número de movimientos esté por debajo de 8, el ratón no habrá podido alcanzar la formación de los gatos y, por tanto, no podrá condicionar ni limitar los movimientos de estos. Por tanto, por debajo de 8 movimientos podremos aplicar una secuencia de movimientos fija (o varias) que sea de interés para los gatos. A partir del movimiento 9, en cambio, el ratón ya habrá podido alcanzar a los gatos y, por tanto, los movimientos posibles para los gatos ya dependerán de cómo se haya movido el ratón.



Como a los gatos les interesa mantener una formación cerrada, una posible “base de datos” de movimientos al inicio de la partida consiste hacer los movimientos 1, 3, 5 y 7 como se muestra en la imagen anterior.

Implementar lo anterior es fácil. En vez de construir un árbol de jugadas, evaluarlo, aplicar minimax y elegir la jugada que minimiza la evaluación en todo caso, haremos lo siguiente:

- Por debajo de la jugada 8, aplicaremos la “base de datos” de movimientos. Da igual lo que haya movido el ratón.
- Por encima de la jugada 8, actuaremos como hasta ahora, es decir, construiremos el árbol de jugadas, lo evaluaremos, aplicaremos minimax, y optaremos por la jugada que minimiza el valor (porque el C64 mueve a los gatos).

Para ello, en el fichero principal “RYG.asm” dotamos la nueva rutina “decideBDvsArbol” que lo primero que hace es mirar en qué número de movimiento estamos (nivel):

```
274 tablaNumGatos   byte 00,00,00,01,00,02,00,03
275 tablaOffsets    byte 00,09,00,11,00,13,00,15
276
277 decideBDvsArbol
278
279         lda #<tableroActual
280         sta ddbTableroLo
281
282         lda #>tableroActual
283         sta ddbTableroHi
284
285         jsr dameDatosBasicos
286
287         lda ddbNivel
288         cmp #$08
289         bcc dbdaBaseDatos
290
291 dbdaArbol
292
```

Por debajo de 8 (bcc), saltamos a la etiqueta “dbdaBaseDatos”, que se encarga de aplicar la “base de datos” de jugadas de inicio. Por encima de 8, continuamos con la etiqueta “dbdaArbol”, es decir, continuamos con la operativa normal hasta ahora:

```
dbdaArbol

        jsr desarrollaArbolJugadas

        jsr evaluaArbolJugadas

        ;jsr pintaArbolJugadas

        jsr decideJugadaGatos

        jsr aplicaJugadaDecidida

        rts

dbdaBaseDatos
```

En cambio, cuando se trata de aplicar la “base de datos” de jugadas la novedad es como sigue:

```
        ---

dbdaBaseDatos

        tax

        lda #<tableroActual
        sta ajgTableroLo

        lda #>tableroActual
        sta ajgTableroHi

        lda tablaNumGatos,x
        sta ajgNumGato

        lda tablaOffsets,x
        sta ajgNuOffset

        jsr aplicaJugadaGato

        rts
```

Es decir, movemos el nivel o número de jugada (que en ese momento está en el acumulador) al registro X y, usando X como índice, accedemos a una tabla doble de gatos y offsets:

```
tablaNumGatos   byte 00,00 00,01 00,02 00,03
tablaOffsets    byte 00,09 00,11 00,13 00,15
```

Esta tabla doble es propiamente la “base de datos de jugadas” ya que, para un número de jugada (valor del índice X), nos dice qué gato hay que mover (tabla “tablaNumGatos”) y a qué offset hay que moverlo (tabla “tablaOffsets”).

A los gatos les corresponden los movimientos 1, 3, 5 y 7. En el movimiento 1 hay que mover el gato 0 al offset 9, en el movimiento 3 hay que mover el gato 1 al offset 11, en el movimiento 5 hay que mover el gato 2 al offset 13 y, por último, en el movimiento 7 hay que mover el gato 3 al offset 15. Todo lo demás son ceros de relleno que corresponden a los movimientos del ratón.

De este modo, al arrancar la partida veremos que el C64, es decir, los gatos, mueve siempre la secuencia anterior. Ni siquiera veremos que aparece "PENSANDO: XXXX", ya que el C64 no está pensando nada, no está generando un árbol de jugadas, sino que juega "a tiro hecho".

Todo esto da lugar a la versión 19 del proyecto.

RYG: otra forma mejor de generar el árbol de juego

Hacía tiempo que quería retomar el proyecto del ratón y los gatos. Tengo algún avance interesante…

La verdad es que a pesar de nuestros muchos esfuerzos (ampliar la RAM disponible, sucesivas versiones de la función de evaluación, tableros más compactos que permiten aprovechar mejor la memoria, e, incluso, inicios de partida), la versión 19 del proyecto sigue siendo fácil de derrotar. Le he dado muchas vueltas: nuevos criterios de evaluación, etc. Pero nada, con cuatro niveles de profundidad no resulta fácil conseguir nada mucho mejor.

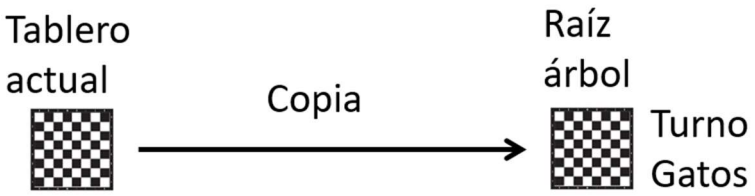
Incluso he hecho una versión del proyecto en Java para PC que, con la misma función de evaluación que la versión en ensamblador para C64, es casi invencible con ocho o diez niveles de profundidad (obviamente un PC tiene mucha más memoria que un C64). Esto me hace pensar que el problema no es la función de evaluación, sino la escasa profundidad que alcanzamos (cuatro niveles).

Y de repente he caído en un detalle bastante evidente por otro lado… ¿necesitamos todo el árbol completo en memoria? Hombre, con el enfoque que le hemos dado sí, porque primero construimos el árbol y luego evaluamos. Pero… ¿no podríamos ir evaluando sobre la marcha y, según evaluamos, ir liberando y reutilizar memoria? ¡¡Pues claro que sí!! ¡¡Esa es la clave!!

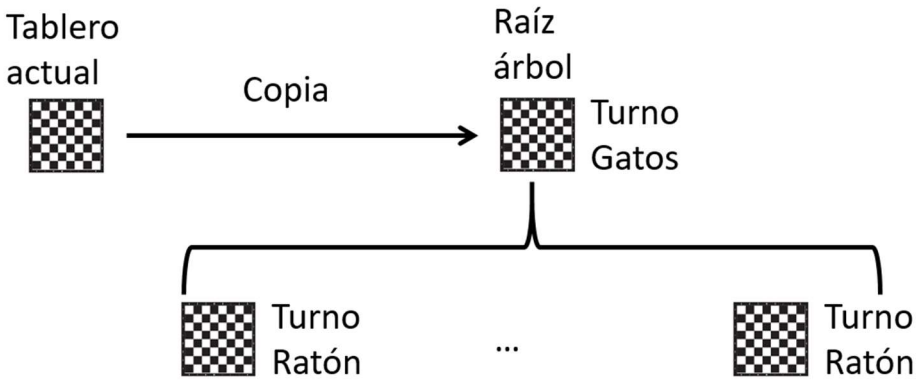
<u>Forma anterior de construir el árbol de juego:</u>

Suponiendo que la profundidad elegida fuera dos (para simplificar), ahora mismo construimos y evaluamos el árbol así:

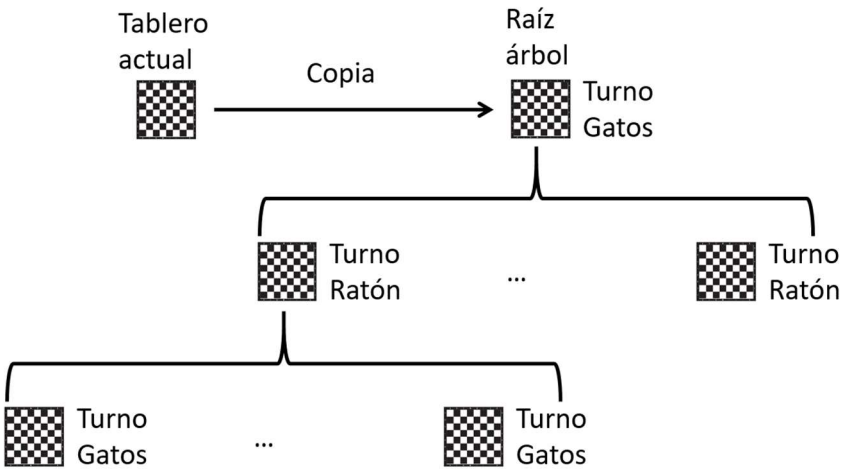
1) Empezamos copiando el tablero actual a la raíz del árbol:



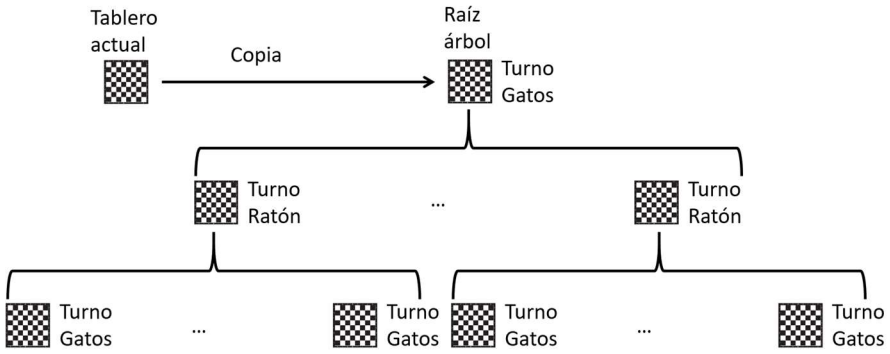
2) Desarrollamos el primer nivel por debajo de la raíz:



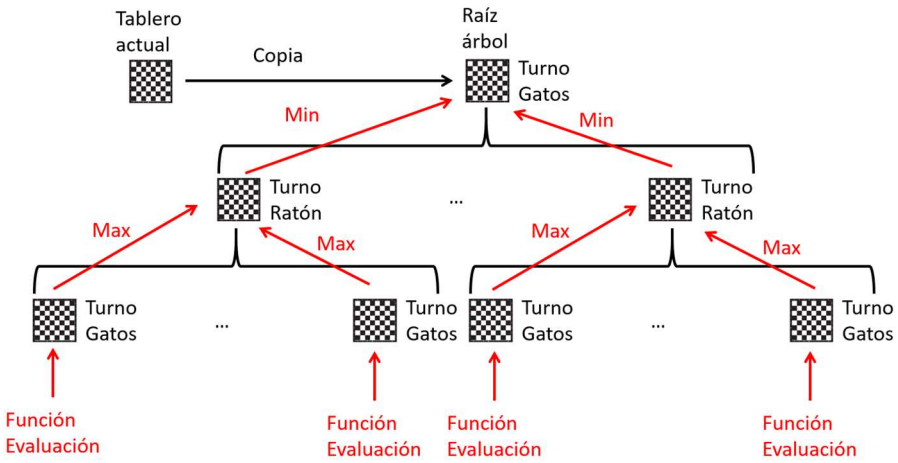
3) Desarrollamos el segundo nivel para el primer tablero del primer nivel:



4) Y así sucesivamente hasta que…
5) Desarrollamos el segundo nivel para el último tablero del primer nivel:



6) Cuando el árbol está completo, lo evaluamos con minimax, lo que significa que:
   - Si el nodo es una hoja, lo evaluamos con la función de evaluación.
   - Si el nodo no es una hoja, seleccionamos el máximo o el mínimo de los hijos, según el turno. Si es el turno de los gatos, seleccionamos el mínimo; si es el turno del ratón el máximo.

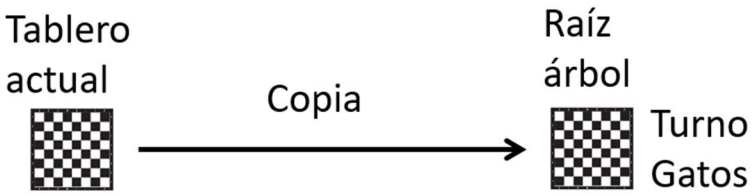


Con esta forma de trabajar necesitamos el árbol completo en memoria y, por muy compactos que sean los tableros (unos 30 bytes), con 40 KB de memoria disponible ($d000 - $3000 = 40.960), eso da para unos 1.300 tableros, es decir, para cuatro niveles de profundidad.

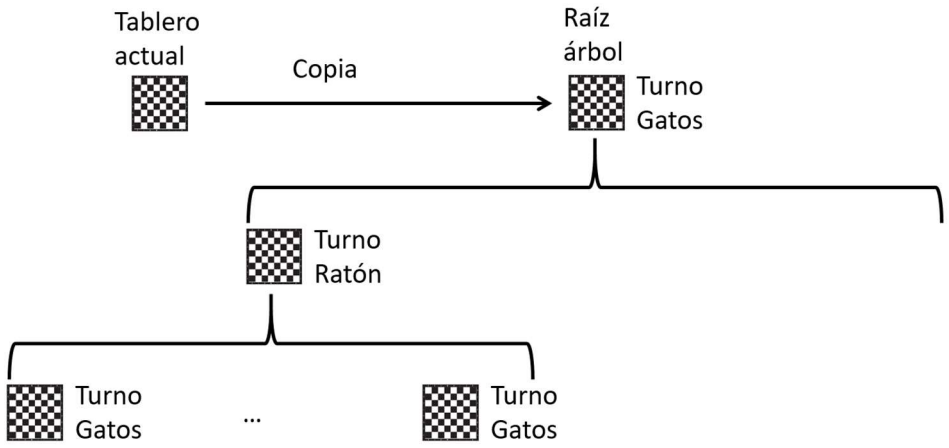
Nueva forma de construir el árbol de juego:

Pero ahora observemos esta nueva forma de construir y evaluar el árbol (nuevamente con una profundidad de dos para simplificar):

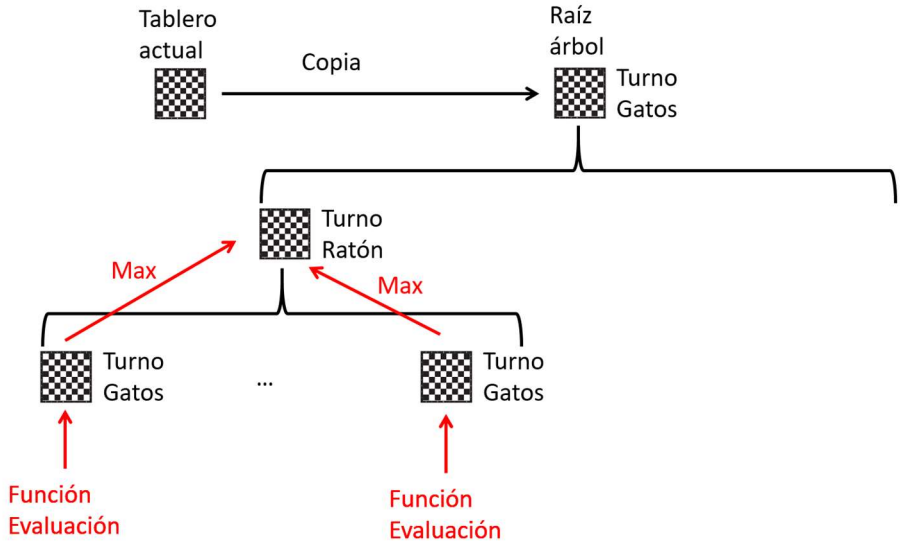
1) Nuevamente, empezamos copiando el tablero actual a la raíz:



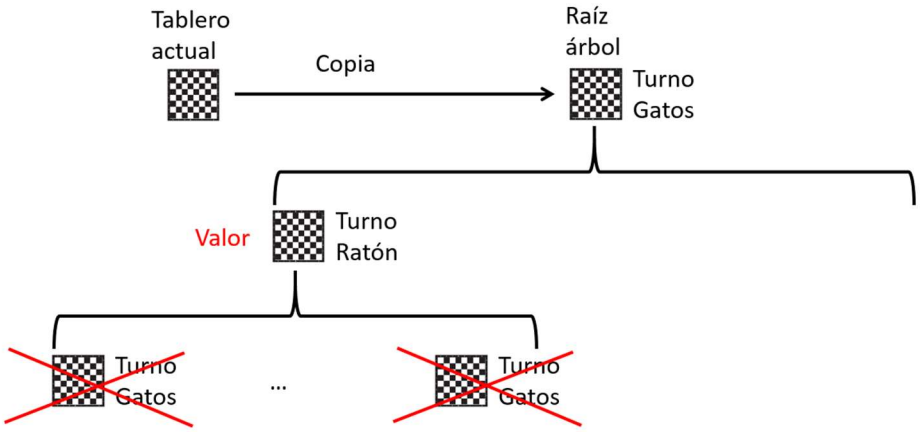
2) Ahora desarrollamos la primera rama hasta su máxima profundidad:



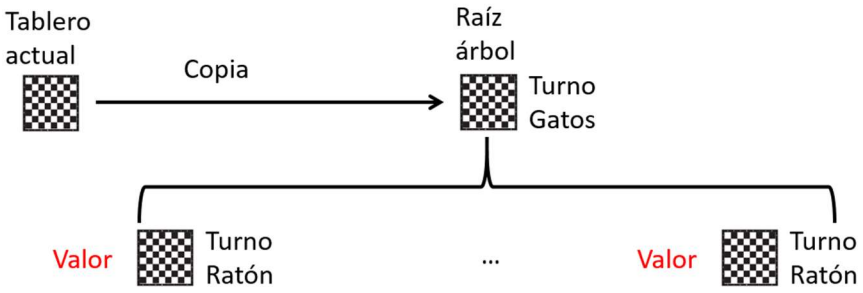
3) Aplicamos minimax a la primera rama:



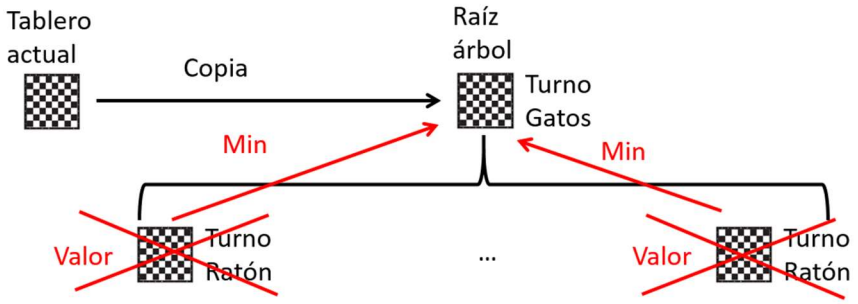
4) Puesto que la primera rama ya está evaluada, esos nodos ya no son necesarios, de modo que podamos liberar su memoria y reutilizarla para la rama dos:



5) Y hacemos lo mismo con la rama dos (generarla hasta su máxima profundidad, evaluarla y liberar su memoria), y con la rama tres, etc., hasta terminar todas las ramas.



6) Finalmente, repetimos el proceso también con la raíz, es decir, le aplicamos minimax y liberamos la memoria de sus hijos. En el fondo es un proceso recursivo.



De este modo, el consumo de memoria va creciendo mientras se desarrolla una rama, pero una vez evaluada, al liberar su memoria, el consumo de memoria disminuye, y esa memoria puede reutilizarse para la rama siguiente.

El máximo consumo de memoria se da cuando la última rama está desarrollada hasta su máxima profundidad. Pero, aunque estuviéramos hablando de una profundidad de 10 niveles, eso sería:

- Nodo raíz.
- Nivel 1: Un máximo de 8 hijos.
- Nivel 2: Un máximo de 4 hijos.
- …
- Nivel 10: Un máximo de 4 hijos.

Por tanto, en total, y como máximo, estamos hablando 61 nodos que, a 30 bytes por nodo (en realidad 29 bytes), nos da… ¡¡1.830 bytes para 10 niveles!!

Conclusiones:

¡¡La diferencia es abrumadora!! Antes con cuatro niveles consumíamos toda la memoria disponible (40 KB). Ahora con 10 niveles ni siquiera llegamos a 2 KB.

En la práctica esto significa que podemos hacer que el C64 juegue prácticamente a **cualquier profundidad que se nos antoje: 10 niveles, 20 niveles, 30 niveles, … ¡¡La memoria ya no nos limita!!**

RYG: otra forma mejor de generar el árbol de juego – cambios en el código

Implementar todos los cambios descritos en la entrada anterior no es fácil. Vayamos paso a paso:

Generar el árbol de juego "a lo profundo":

El árbol de juego se genera en la rutina "desarrollaUnNivel" del fichero principal ("RYG.asm"). Esta rutina tiene dos ramas principales, pero son muy parecidas:

- La rama "dunGatos", cuando el turno es de los gatos.
- La rama "dunRaton", cuando el turno es del ratón.

Por simplicidad, revisaremos sólo la segunda. La primera es muy parecida.

En la versión 19 del proyecto el código era así:

- Primero, se generaban todas las jugadas del ratón, dando lugar a tableros hijo (rutina "arbolJugadasRaton"):

```
470 dunRaton
471
472         lda dunTableroLo,y
473         sta ajrTableroLo2
474
475         lda dunTableroHi,y
476         sta ajrTableroHi2
477
478         jsr arbolJugadasRaton
479
```

- Después, se recorrían los tableros hijos llamando recursivamente a "desarrollaUnNivel":

```
480         ldx #$00
481
482 dunRatonBucle
483
484         lda dunTableroLo,y
485         sta dhTableroLo
486
487         lda dunTableroHi,y
488         sta dhTableroHi
489
490         stx dhNumHijo
491
492         jsr dameHijo
493
494         lda dhHijoHi
495         beq dunRatonSgteHijo
496
497         lda dunProf
498         sec
499         sbc #$01
500         sta dunProf
501         tay
502
503         lda dhHijoLo
504         sta dunTableroLo,y
505
506         lda dhHijoHi
507         sta dunTableroHi,y
508
509         txa
510         sta dunNumHijo,y
511
512         jsr desarrollaUnNivel
513
514         lda dunNumHijo,y
515         tax
516
517         lda dunProf
518         clc
519         adc #$01
520         sta dunProf
521         tay
522
523 dunRatonSgteHijo
524
525         inx
526
527         cpx #$08
528         bne dunRatonBucle
529
530         rts
531
```

Es decir, que efectivamente el árbol se estaba generando “a lo ancho”, no “a lo profundo”.

En la versión 20 del proyecto ese código queda así:

- Primero se genera una jugada, dando lugar a un tablero hijo:

```
dunRatonJugsBucle

        ; Localiza el ratón
        lda dunTableroLo,y
        sta dprTableroLo

        lda dunTableroHi,y
        sta dprTableroHi

        jsr damePosicionRaton

        ; Pide la jugada x-esima del ratón
        lda dprOffsetRaton
        sta gjrOffset

        stx gjrNumJugada

        jsr generaJugadaRaton

        ; Valida la jugada
        lda dunTableroLo,y
        sta vjTableroLo

        lda dunTableroHi,y
        sta vjTableroHi

        lda gjrNuOffset
        sta vjNuOffset

        jsr validaJugada

        ; Si es válida, aplícala sobre un tablero hijo
        lda vjValida
        cmp #$ff
        beq dunRatonSgteJug

        lda dunTableroLo,y
        sta ajrhPadreLo

        lda dunTableroHi,y
        sta ajrhPadreHi

        lda libreLo
        sta ajrhHijoLo

        lda libreHi
        sta ajrhHijoHi

        lda gjrNuOffset
        sta ajrhNuOffset

        jsr aplicaJugadaRatonHijo
```

- Después, antes de pasar a generar el siguiente hijo, se llama recursivamente a “desarrollaUnNivel” sobre el hijo recién generado:

```
573         ; Llama recursivamente sobre el hijo
574         lda dunProf
575         sec
576         sbc #$01
577         sta dunProf
578         tay
579
580         lda ajrhHijoLo
581         sta dunTableroLo,y
582
583         lda ajrhHijoHi
584         sta dunTableroHi,y
585
586         txa
587         sta dunNumJug,y
588
589         jsr desarrollaUnNivel
590
591         lda dunNumJug,y
592         tax
593
594         lda dunProf
595         clc
596         adc #$01
597         sta dunProf
598         tay
599
600 dunRatonSgteJug
601
602         inx
603
604         cpx #$04
605         beq dunRatonFin
606
607         jmp dunRatonJugsBucle
608
609 dunRatonFin
610
611         rts
612
```

De este modo, el árbol ya se genera “a lo profundo”. Es decir, primero se genera la primera rama hasta su máxima profundidad, luego la segunda rama hasta su máxima profundidad, etc. Y así hasta completar el árbol.

Todavía no hemos cambiado la forma de evaluar el árbol. Ese es el siguiente paso.

Evaluar el árbol según se genera:

En las versiones 19 y 20 del proyecto todavía esperamos a tener todo el árbol completo para evaluarlo. Esto se ve aquí (rutina “decideBDvsArbol”):

```
275 tablaNumGatos   byte 00,00,00,01,00,02,00,03
276 tablaOffsets    byte 00,09,00,11,00,13,00,15
277
278 decideBDvsArbol
279
280         lda #<tableroActual
281         sta ddbTableroLo
282
283         lda #>tableroActual
284         sta ddbTableroHi
285
286         jsr dameDatosBasicos
287
288         lda ddbNivel
289         cmp #$08
290         bcc dbdaBaseDatos
291
292 dbdaArbol
293
294         jsr desarrollaArbolJugadas
295
296         jsr evaluaArbolJugadas
297
298         ;jsr pintaArbolJugadas
299
300         jsr decideJugadaGatos
301
302         jsr aplicaJugadaDecidida
303
304         rts
305
306 dbdaBaseDatos
307
308         tax
309
310         lda #<tableroActual
311         sta ajgTableroLo
312
313         lda #>tableroActual
314         sta ajgTableroHi
315
316         lda tablaNumGatos,x
317         sta ajgNumGato
318
319         lda tablaOffsets,x
320         sta ajgNuOffset
321
322         jsr aplicaJugadaGato
323
324         rts
325
```

Sin embargo, como nuestro objetivo es liberar memoria en cuanto podamos, ya no debemos hacerlo así. Tenemos que evaluar sobre la marcha.

Estos cambios se ven en la versión 21 del proyecto:

- Si estamos ante un nodo intermedio del árbol, dependiendo de que el turno sea de los gatos o del ratón, hay que obtener el mínimo de los hijos

(gatos) o el máximo (ratón). Para el caso del ratón / máximo esto se ve aquí:

```
dunRatonSgteJug

        inx

        cpx #$04
        beq dunRatonMax

        jmp dunRatonJugsBucle

dunRatonMax

        ; HAY QUE OBTENER EL MÁXIMO DE LOS HIJOS

        lda dunTableroLo,y
        sta maxvTableroLo

        lda dunTableroHi,y
        sta maxvTableroHi

        jsr maxValorHijos

        rts
```

- Y si estamos ante una hoja del árbol (ya sin más hijos), aplicamos la función de evaluación:

```
384 dunProf         byte $00
385 dunTableroLo    byte $00,$00,$00,$00,$00,$00,$00,$00
386 dunTableroHi    byte $00,$00,$00,$00,$00,$00,$00,$00
387 dunNumJug       byte $00,$00,$00,$00,$00,$00,$00,$00
388
389 dunNumsGato     byte $00,$00,$01,$01,$02,$02,$03,$03
390 dunNumsJug      byte $00,$01,$00,$01,$00,$01,$00,$01
391
392 ;dunTempX        byte $00
393
394 desarrollaUnNivel
395
396         lda dunProf
397         tay
398
399         cmp #$00
400         bne dunOtroNivel
401
402         ; Estamos ante una hoja del árbol (dunProf = 0)
403         ; No hay que desarrollar más el árbol, pero sí evaluar el tablero
404
405         lda dunTableroLo,y
406         sta etTableroLo
407
408         lda dunTableroHi,y
409         sta etTableroHi
410
411         jsr evaluaTablero
412
413         rts
414
```

De este modo, ya vamos evaluando el árbol sobre la marcha, según se va construyendo, lo que nos permitirá liberar memoria en el siguiente paso.

Tras evaluar un nodo, liberar la memoria de sus hijos:

En las versiones 19, 20 y 21 construimos el árbol completo en memoria. La memoria sólo se libera tras generar el árbol completo, evaluarlo (al final o sobre la marcha), y decidir la jugada de los gatos.

Esto puede verse aquí. Cada vez que empieza a generarse el árbol para decidir una nueva jugada, se copia el tablero actual a la raíz y el puntero a la memoria libre (libreLo – libreHi) se pone a la raíz más 29 bytes (29 bytes es lo que ocupa la raíz, el tablero de partida). Por tanto, se está “pisando” la memoria del árbol anterior:

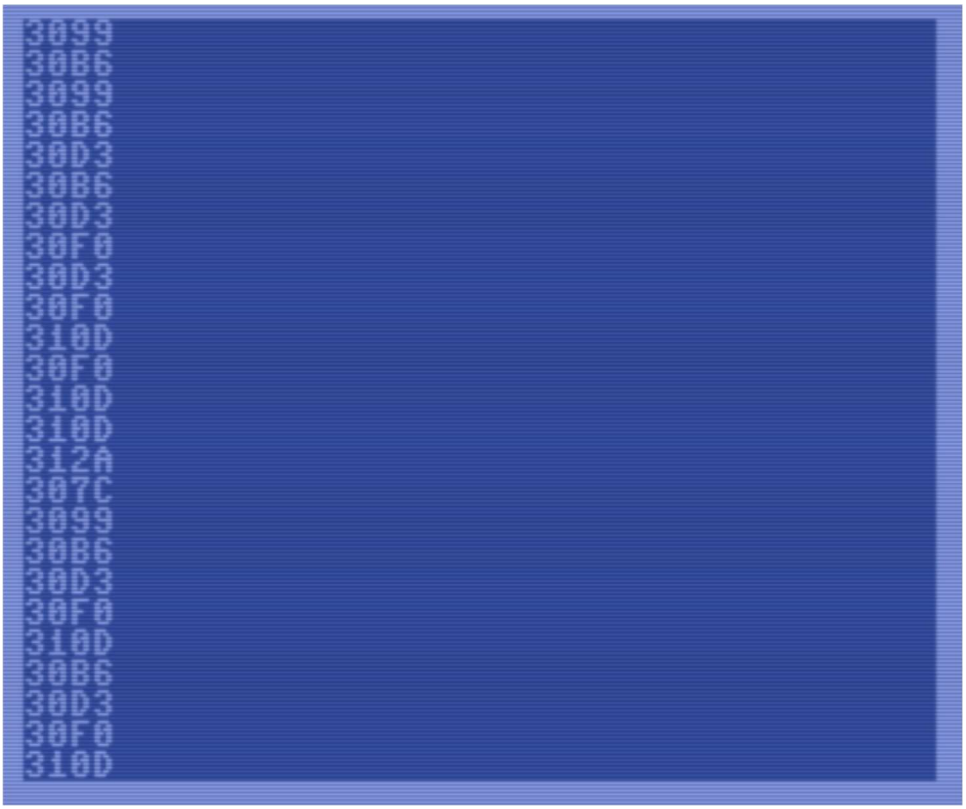
```
334 desarrollaArbolJugadas
335
336         jsr copiaActualRaizArbol
337
338         lda #<raizArbol
339         clc
340         adc #29;88
341         sta libreLo
342
343         lda #>raizArbol
344         adc #0
345         sta librehi
346
347         jsr pintaPensando
348
349         lda prof
350         sta dunProf
351         tay
352
353         lda #<raizArbol
354         sta dunTableroLo,y
355
356         lda #>raizArbol
357         sta dunTableroHi,y
358
359         lda #$00
360         sta dunNumJug,y
361
362         jsr desarrollaUnNivel
363
364         lda #13
365         jsr chrout
366
367         rts
368
```

Pero la gracia de todos estos cambios es generar el árbol de otra manera, evaluarlo según se va generando, y, tras evaluar un nodo a partir de sus hijos, liberar la memoria consumida por ellos. De este modo, la memoria consumida por el árbol de juego crece, decrece, y se reutiliza, lo que nos permite llegar a muchos más niveles de profundidad y, por tanto, a un juego más fuerte.

En la versión 22 del proyecto ya liberamos memoria tras evaluar un nodo a partir de sus hijos. Por ejemplo, si es el turno del ratón:

```
dunRatonMax

        ; HAY QUE OBTENER EL MÁXIMO DE LOS HIJOS

        lda dunTableroLo,y
        sta maxvTableroLo

        lda dunTableroHi,y
        sta maxvTableroHi

        jsr maxValorHijos

        ; Y LIBERAMOS LA MEMORIA DE LOS HIJOS

        lda dunTableroLo,y
        clc
        adc #29
        sta libreLo

        lda dunTableroHi,y
        adc #0
        sta libreHi

        rts
```

De hecho, una cosa muy curiosa de la versión 22 es ver cómo ahora crece y decrece la memoria utilizada según se va generando y evaluando ramas del árbol. Para facilitar el seguimiento del uso creciente y decreciente de la memoria modificamos temporalmente la traza “PENSANDO: XXXX”, de modo que vemos qué posiciones de memoria se van ocupando sin borrar las anteriores:



Obsérvese cómo la memoria crece según se va profundizando en el árbol, y luego decrece según se va evaluando y liberando. Y luego se reutiliza, porque aparecen repetidas las mismas posiciones. La diferencia entre dos posiciones siempre es de 29 bytes, es decir, lo que ocupa un tablero (ej. 310d – 30f0 = 29 bytes).

Pero todo esto lo hemos hecho con un objetivo final, que es el siguiente…

Mayor profundidad de análisis:

Puesto que ahora consumimos mucha menos memoria (consumimos, liberamos y reutilizamos), podemos llegar a muchos más niveles de profundidad y conseguir un juego más fuerte.

Virtualmente, casi podríamos decir que ahora la memoria no nos limita. Cuando más ocupa el árbol es cuando se está desarrollando y evaluando su última rama,

porque tiene que conservar en memoria los hijos de primer nivel de las ramas anteriores. Pero ya vimos que para 10 niveles de profundidad estamos hablando de menos de 2 KB. Y tenemos 40 KB disponibles ($d000 - $3000 = 40.960).

Pero como todo en la vida tiene que tener un tope, vamos a limitarlo a 15 niveles ($0f), cosa que ya hacemos en la versión 23 del proyecto:

```
117 solicitaProfundidad
118
119         lda #<spProfundidad
120         sta cadenaLo
121
122         lda #>spProfundidad
123         sta cadenaHi
124
125         jsr pintaCadena
126
127         jsr leeTeclado
128
129         lda #13
130         jsr chrout
131
132         lda byteLeido
133
134         cmp #$01
135         bcc solicitaProfundidad
136
137         cmp #$10
138         bcs solicitaProfundidad
139
140         sta prof
141
142         rts
143
144 spProfundidad   text "profundidad (01-0f)? "
145                 byte $00
146
```

Pero con esto no es suficiente porque, al admitir más profundidad de análisis, ahora hay más llamadas recursivas para generar y evaluar el árbol, así que también hay que ampliar la tabla de parámetros de “desarrollaUnNivel”:

```
382 dunProf         byte $00
383 dunTableroLo    byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
384 dunTableroHi    byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
385 dunNumJug       byte $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
386
387 dunNumsGato     byte $00,$00,$01,$01,$02,$02,$03,$03
388 dunNumsJug      byte $00,$01,$00,$01,$00,$01,$00,$01
389
390 ;dunTempX        byte $00
391
392 desarrollaUnNivel
393
```

Y eso es todo, que no es poco…

Conclusiones:

Os aconsejo jugar con la versión 23 con profundidades crecientes (2, 4, 6, 8, …). Iréis viendo que, cuanto más profundo analiza el C64, mejor juega. **De hecho, jugando a una profundidad de 10 yo no he conseguido ganarle, lo cual me parece una magnífica noticia.**

Pero la vida es puñetera, y ahora que nos hemos sacudido la limitación de la memoria, podemos analizar muchos más niveles, y jugar mejor, nos surge otra limitación... ¿Alguna idea de cuál podría ser?

Baste decir que a partir del nivel 8 de profundidad sugiero usar la función "Ward mode" de VICE… (Settings > Warp mode).

RYG: memoria vs tiempo

Efectivamente, ya no tenemos limitaciones de memoria. Esto nos permite analizar muchos niveles de profundidad y, por tanto, que el C64 juegue mejor.

Sin embargo, es cierto que la vida es puñetera y ahora tenemos un nuevo límite: **el tiempo**. Lógicamente, se trata de no aburrir al personal y de no tirarse media hora en cada jugada. ¿Cuánto puede esperar un jugador al siguiente movimiento del C64? ¿Cinco minutos como máximo?

A modo de ejemplo se muestra la siguiente tabla de datos. Téngase en cuenta que cuando es el turno de los gatos hay un máximo de 8 movimientos (4 gatos x 2 movimientos) y cuando es el turno del ratón hay un máximo de 4 movimientos (1 ratón x 4 movimientos):

Nivel de profundidad	**Número de tableros a analizar**	**Tiempo para mover**	**Tiempo con Warp mode**
2	8x4 = 32	Instantáneo	No hace falta
4	8x4x8x4 = 1.024	Pocos segundos	No hace falta
6	8x4x8x4x8x4 = 32.768	1 o 2 minutos	No hace falta
8	8x4x8x4x8x4x8x4 = 1 millón	Unos 15 minutos	Unos 4 minutos
10	8x4x8x4x8x4x8x4x8x4 = 33,5 millones	Unas 2 horas (*)	Unos 30 minutos

(*) Aunque 2 horas puede parecer mucho, y sin duda lo es para un jugador esperando el siguiente movimiento, hay que tener en cuenta que estamos hablando de 33,5 millones de tableros para un ordenador que funciona a 1 MHz.

En definitiva, es fantástico que el C64 juegue fuerte a 10 niveles de profundidad, y que sea imbatible o muy difícil ganarle a ese nivel, pero sería muy conveniente agilizar el proceso. Para ello hay varias estrategias:

Podar el árbol de juego:

Podar el árbol de juego consiste en eliminar aquellas ramas que no tiene sentido generar y evaluar.

Por ejemplo, si en un tablero del árbol de juego el ratón está a la espalda de los gatos, la partida ya está ganada para el ratón (salvo que el humano juegue a perder intencionadamente), por lo que no se hace necesario seguir desarrollando esa rama. Se puede podar.

En este sentido, el procedimiento minimax admite una mejora importante que se llama **poda alfa – beta**.

Poner un límite de tiempo a cada movimiento:

Si el C64 empieza a tener un juego fuerte a partir del nivel 10, pero en ese nivel decidir cada jugada lleva mucho tiempo, simplemente ha llegado el momento de asumir que no es posible generar y evaluar todos los movimientos. Por tanto, nos ponemos un límite de tiempo razonable (ej. X minutos por movimiento) y, cuando se haya agotado ese tiempo, optamos por el mejor movimiento descubierto hasta ese momento.

La forma habitual de implementar esto es mediante una técnica llamada **“iterative deepening” o “profundización progresiva”**. Es decir, aunque el usuario haya decidido jugar a profundidad 10, la máquina juega a profundidades 1, 2, 3, 4, 5, …, y así sucesivamente hasta que se agote el tiempo disponible. Una vez que se agota el tiempo, sea en el nivel que sea, por ejemplo 7, se elige la mejor jugada descubierta hasta ese momento.

Esta forma de funcionar parte de la base de que no vas a poder desarrollar el árbol completo hasta el nivel N, y que te vas a tener que conformar con lo mejor que encuentres en un tiempo limitado, por lo que se vuelve especialmente interesante **ordenar los movimientos y desarrollar primero los que sean más prometedores**.

Hasta ahora no hemos hecho esto. Hemos generado todos los movimientos y en un orden un tanto arbitrario. El orden de los movimientos ha venido fijado por las matrices que los generan:

```
; Fichero para generar jugadas, es decir, desarrollar el árbol

; Jugadas que puede hacer el ratón

tblRatonF       byte $ff, $ff, $01, $01
tblRatonC       byte $ff, $01, $01, $ff

```

```
; Jugadas que pueden hacer los gatos

tblGatosF       byte $01, $01
tblGatosC       byte $ff, $01

```

Se podrían ordenar y desarrollar primero, por ejemplo, aquellos movimientos que mantienen la formación de los gatos, es decir, aquellos que no generan huecos. De este modo, la probabilidad de que el “deadline” nos pille habiendo generado y analizado jugadas malas se reduce.

Otras técnicas:

Hay más técnicas para acelerar el proceso de generar y evaluar el árbol de juego. Por ejemplo, si dos secuencias de jugadas distintas llevan al mismo tablero, no hay necesidad de desarrollar ese tablero dos veces. La segunda vez que se presente el mismo tablero en el árbol de juego, se puede detectar que es **una repetición** y copiar su evaluación previa o podarlo, sin necesidad de desarrollarlo más.

Para que dos tableros sean idénticos las piezas deben ocupar las mismas casillas y, además, el turno de juego debe ser el mismo.

Y como recorrerse todo el árbol generado previamente a la caza y captura de un tablero igual que el actual sería inviable, puesto que costaría más hacerlo que lo

que se ahorraría, lo que se suele hacer es definir una especie de “función hash” que, dado un tablero, genera un número (ej. la suma de los offsets de los gatos menos el offset del ratón). Luego estos números se almacenan en una tabla ordenada y, si al generar un nuevo tablero su “hash” ya está en la tabla, asumimos que el tablero es repetido y actuamos en consecuencia.

Bueno, iremos viendo hasta dónde llegamos…