5.1.4 Inicialización de la matriz

Esta fase consiste en rellenar la matriz al azar con bloques de piedra de los tipos 1, 2 y 3, como muestra la Figura 5.11.

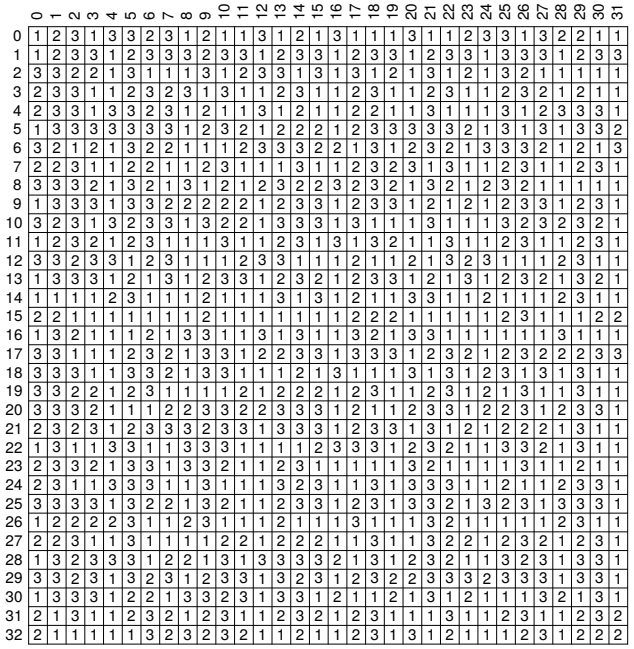


Figura 5.11. Ejemplo de inicialización de la matriz del mapa.

5.1.5 Generación del laberinto

El proceso de generación comienza en el centro del mapa, en la celda (15, 15), y avanza en pasos de dos celdas. En cada paso, se actualizan las celdas recorridas con bloques vacíos –empty, código 0–, se anotan las posibles bifurcaciones en una tabla y se elige una de ellas al azar. El algoritmo continúa hasta que el camino se cierra y no es posible avanzar. En ese momento, recupera de la tabla la última bifurcación registrada, la elimina e intenta continuar desde este punto. Si es posible avanzar, prosigue del mismo modo hasta agotar ese camino; en caso contrario, toma una nueva bifurcación de la tabla. Este procedimiento se repite hasta que no quedan bifurcaciones pendientes.

En resumen, se trata de un algoritmo de exploración en profundidad con retroceso: se desarrolla el árbol siguiendo un camino hasta agotarlo y, cuando no es posible continuar, se retrocede hasta una ramificación para seleccionar un nuevo camino.

Veamos los pasos con más detalle –Figura 5.12–:

1. Comenzar en la celda (15, 15).
2. Comprobar los movimientos que son posibles en pasos de dos celdas desde la posición actual, en las direcciones arriba, derecha, abajo, izquierda.
3. A la vista de las posibilidades de movimiento:

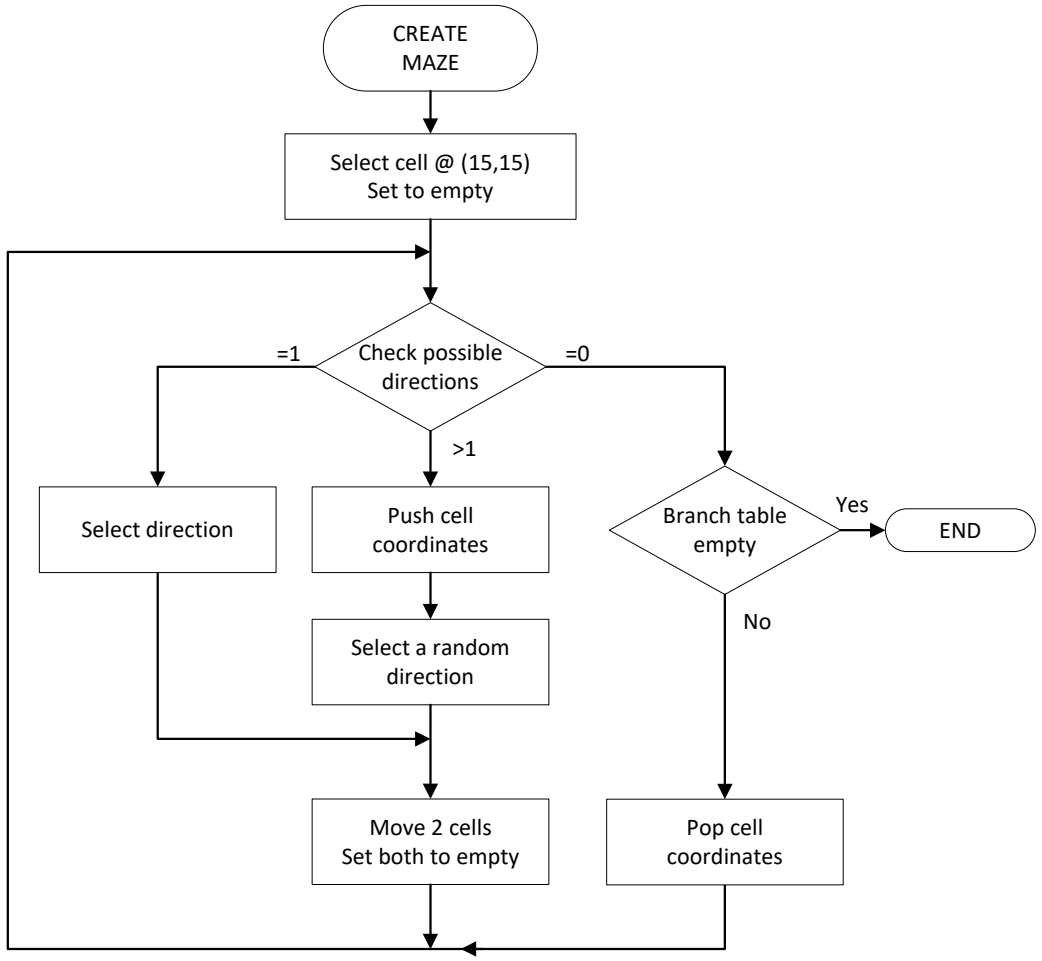
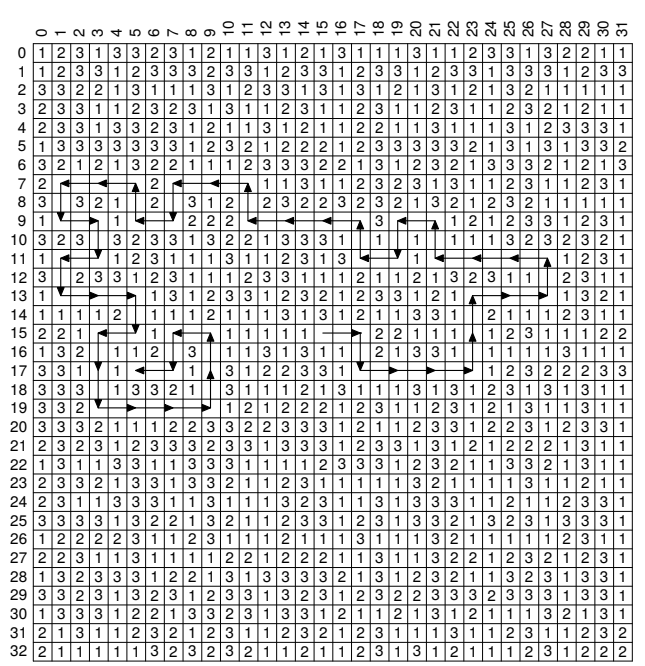


Figura 5.12. Diagrama de flujo del algoritmo de generación del laberinto.

a. Si solo hay una opción de movimiento, tomar esta opción.
b. Si es posible más de un movimiento, anotar las coordenadas de la celda en la tabla y tomar un movimiento al azar.
c. Si no es posible ningún movimiento, comprobar si quedan entradas en la tabla. Si es así, tomar las coordenadas de la última entrada, retirar la entrada de la tabla, seleccionar la celda y saltar al paso 2. En caso contrario, detener el proceso.

4. Ejecutar el movimiento, rellenando la matriz con el código 0 –bloque vacío– en las celdas afectadas. Saltar al paso 2.

Al construir el laberinto de este modo los corredores están formados por un número impar de celdas y se encuentran situados en coordenadas X impares. Para ilustrar el funcionamiento del algoritmo veamos cómo se genera uno de los corredores del laberinto de la Figura 5.1. Comenzamos en la celda (15,



1	(15,15)	24	(11,7)
2	(17,15)	25	(9,7)
3	(17,15)	26	(7,7)
4	(17,17)	27	(7,9)
5	(19,17)	28	(5,9)
6	(21,17)	29	(5,7)
7	(23,17)	30	(3,7)
8	(23,15)	31	(1,7)
9	(23,13)	32	(1,9)
10	(25,13)	33	(3,11)
11	(27,13)	34	(1,13)
12	(27,11)	35	(3,13)
13	(25,11)	36	(5,13)
14	(23,11)	37	(5,15)
15	(21,11)	38	(3,15)
16	(21,9)	39	(3,17)
17	(19,9)	40	(3,19)
18	(19,11)	41	(5,19)
19	(17,11)	42	(7,19)
20	(17,9)	43	(9,19)
21	(15,9)	44	(9,17)
22	(13,9)	45	(9,15)
23	(11,9)	46	(7,15)

Figura 5.13. A la izquierda, el proceso de creación de un corredor. A la derecha, la tabla asociada.

15). Como es posible el movimiento en varias direcciones (arriba, derecha, abajo e izquierda), anotamos las coordenadas (15, 15) en la tabla y seleccionamos un movimiento al azar, en este caso hacia la derecha. Ahora nos encontramos en la celda (17, 15). De nuevo, es posible el movimiento en varias direcciones (arriba, derecha y abajo), por lo que anotamos las coordenadas (17, 15) en la tabla y seleccionamos un movimiento, hacia abajo.

El proceso continúa del mismo modo hasta alcanzar la celda en las coordenadas (3, 9), donde solo es posible un movimiento –hacia abajo–, por lo que lo seleccionamos sin anotar sus coordenadas en la tabla. Por lo tanto, la siguiente celda es (3, 11).

Desde aquí el camino continúa del modo descrito, sucesivamente hasta alcanzar la celda (5, 17), donde acaba este camino. En este punto no es posible ningún otro movimiento. La última entrada en la tabla es la número 46, celda (7, 15). El procedimiento prosigue en esta celda: tomamos las coordenadas de la tabla y eliminamos la entrada.

Cabe señalar que a medida que el laberinto progresa, las opciones de movimiento desde cada celda cada vez son menores, de modo que los movimientos potenciales registrados en la tabla se irán reduciendo a un solo movimiento o incluso a ninguno.

Veamos el caso, por ejemplo, de la entrada número 9 en la tabla, celda (23, 13). Cuando fue insertada en la tabla eran posibles tres movimientos, derecha, izquierda y arriba, y en aquel momento seleccionamos el movimiento a la derecha, es decir, quedaban otros dos posibles. En el punto en el que nos encontramos ahora, una vez desarrollado el camino, solo es posible un movimiento, a la izquierda.

En resumen, muchos de los movimientos de la tabla no tendrán efecto y solo alguno de ellos dará lugar a nuevos corredores. Continuando de este modo, a través de la técnica descrita, el proceso llega

5.2.4 Inicialización y generación del laberinto

Para crear el mapa utilizaremos tres listas auxiliares. Dos de ellas se emplean para implementar la tabla en la que se guardan las coordenadas X e Y de las celdas con bifurcaciones durante la construcción del laberinto, se denominan slBranch_x y slBranch_y.

Por otro lado, necesitaremos una lista para guardar las direcciones posibles al avanzar un paso, lo que permitirá escoger una de ellas al azar. Para ello, utilizaremos la lista auxiliar slDirections. Cada dirección almacenada es un número entre 1 y 4, según la siguiente correspondencia:

1. Arriba.
2. Derecha.
3. Abajo.
4. Izquierda.

Por ejemplo, si la lista slDirections contiene los números 1, 3 y 4, desde la celda actual es posible el movimiento hacia arriba, hacia abajo y hacia la izquierda.

La Figura 5.25 muestra el script que recibe el mensaje Create Map. Si la variable gDebugMap toma el valor true, carga el mapa de depuración en la lista glMap mediante el script Load Debug Map, como veremos a continuación. Además, el script Set Trapdoor Position inicializa la variable gTrapdoor_x, que indica la coordenada X en la que se sitúa la trampilla de salida. Cabe señalar que, en el mapa de depuración, este elemento se encuentra en la columna 25.

Si gDebugMap es false, genera un nuevo mapa, llamando a los scripts Initialize Map, Create Maze, Set Up The Ropes y Set Hatch Position. Los nombres de estos scripts corresponden a las fases explicadas en los apartados anteriores.

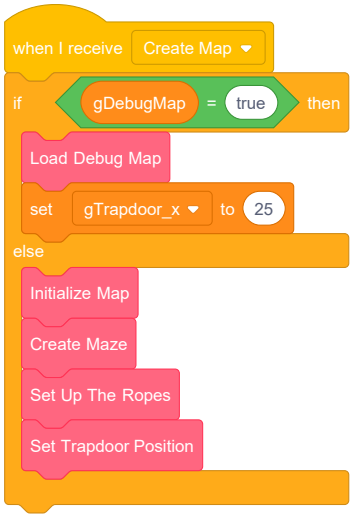


Figura 5.25. Script Create Map de Map.

El mapa de depuración, almacenado en la lista slDebugMap, corresponde a la Figura 5.1, utilizado durante el desarrollo del programa. Durante las pruebas, resulta conveniente trabajar con un mapa fijo y conocido en lugar de uno generado al azar.

La Figura 5.26 muestra el script Load Debug Map, encargado de copiar la lista que contiene el mapa de prueba, slDebugMap, en glMap. La carga se realiza elemento a elemento, ya que Scratch carece de un bloque para duplicar una lista. Observe que no es necesario utilizar una variable como índice: la longitud de glMap en cada iteración actúa como índice en curso.

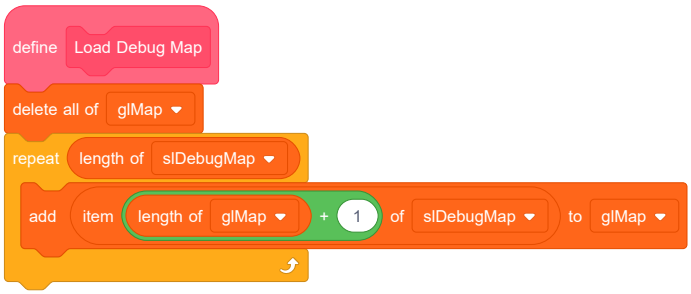


Figura 5.26. Script Load Debug Map de Map.

Script Initialize Map

El primer paso de la generación de un nuevo mapa consiste en inicializar la matriz. El script Initialize Map –Figura 5.27– borra todos los elementos de la lista glMap y la rellena al azar con bloques de tipo piedra 1, 2 y 3.

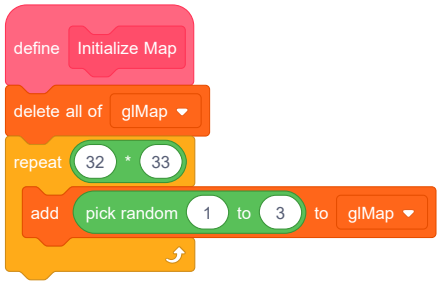


Figura 5.27. Script Initialize Map, de Map.

Script Create Maze

Este script genera el laberinto mediante iteraciones sucesivas –Figura 5.28–. Comienza borrando las listas auxiliares para la tabla, slBranch_x y slBranch_y, y la lista de movimientos posibles, slDirections. Luego, guarda las coordenadas de la celda en curso en las variables sCell_x y sCell_y, que inicialmente contienen las de la celda central (15, 15). Seguidamente, inicializa esta celda con el código de bloque vacío –empty– mediante una llamada a Set Block.

Seguidamente, entra en un bucle infinito que continúa mientras existan movimientos posibles. Cuando ya no puede realizar ninguno más, la construcción del laberinto ha finalizado y el script termina mediante stop this script.

En cada iteración realiza una llamada al script Check Cell, que comprueba las direcciones de movimiento posibles desde la celda actual. Estas direcciones se almacenan en la lista slDirections, como explicamos más arriba. Si no hay movimientos posibles, recurre a la tabla, con una llamada al script

Pop Position –Figura 5.29–, que extrae en (sCell_x, sCell_y) las coordenadas de la última celda donde existía una bifurcación y elimina dicha entrada de la tabla. Si no quedaran celdas en la tabla el proceso terminaría con la salida del script mediante stop this script.

A continuación, debe comprobar si existe algún movimiento posible desde la celda recuperada, ya que la situación del mapa puede haber cambiado desde el momento en que sus coordenadas fueron insertadas en la tabla. En caso de que haya más de una dirección posible, introduce las coordenadas de la celda en la tabla mediante el script Push Position –Figura 5.29–.

Por último, escoge un movimiento al azar entre los posibles y lo ejecuta mediante el script Move To Next Position. Para obtener la dirección que pasa como parámetro en la llamada, toma al azar uno de los valores almacenados en la lista slDirections.

La Figura 5.30 muestra la implementación de Move To Next Position. Este script actualiza las variables sCell_x y sCell_y según la dirección de movimiento indicada por el parámetro direction. Recordemos que las listas glDirection_x y glDirection_y contienen los desplazamientos asociados a cada dirección. Por lo tanto, basta con obtener los valores correspondientes usando direction como índice y sumarlos a ambas variables.

Script Check Cell

La búsqueda de los movimientos posibles desde la celda en curso comienza borrando la lista slDirections –Figura 5.31–. A continuación, comprueba si es posible el movimiento en las direcciones arriba, derecha, abajo e izquierda, en este orden. Para ello, utiliza la variable auxiliar sDirection que toma consecutivamente los valores 1 a 4.

Un movimiento es válido si se cumplen dos condiciones. En primer lugar, las coordenadas resultantes deben permanecer dentro del mapa; en segundo lugar, las dos celdas situadas en esa dirección deben contener una piedra –equivalentemente, no contener bloques vacíos–. El script Check Move, que recibe los incrementos en las direcciones X e Y, realiza las comprobaciones y devuelve el resultado en sIsMoveValid.

Si el movimiento es válido, añade la dirección sDirection a la lista de direcciones posibles, slDirections.

Script Check Move

La Figura 5.32 muestra el script Check Move, que realiza las comprobaciones indicadas utilizando lógica inversa.

Primero, inicializa sIsMoveValid a false. Si las coordenadas salen del laberinto o si las celdas contienen ya un bloque vacío, el script termina inmediatamente. En caso contrario, asigna true a sIsMoveValid antes de finalizar.

Observe que, como los desplazamientos son de amplitud 1, basta con multiplicarlos por 2 para obtener las coordenadas de la celda situada dos posiciones más allá de la celda actual.

5.2.5 Distribución de las cuerdas en el laberinto

Para desplegar las cuerdas en los corredores verticales utilizamos el script Set Up The Ropes –Figura 5.33–. La pareja (sCell_x, sCell_y) guarda las coordenadas de la celda en curso. Comienza en la columna sCell_x=1 y avanza hacia la derecha, recorriendo todas las columnas hasta que sCell_x sea mayor que 31.

La implementación se ajusta fielmente al algoritmo descrito más arriba –Figura 5.15–, por lo que resultará fácil seguir el flujo sin más detalles. Para acceder al tipo de bloque contenido en cada celda se utiliza el script Get Block, mientras que para inicializar la celda con un bloque específico empleamos Set Block, tal como hemos descrito en los apartados anteriores.

5.3 Notas finales sobre la generación del laberinto

El método de creación del laberinto descrito en el apartado 5.1.5 evalúa todas las direcciones posibles cada vez que examina una celda y selecciona una de ellas al azar.

Cabe preguntarse si bastaría con elegir la primera dirección válida encontrada siguiendo un orden aleatorio y sin repeticiones. Sin embargo, al desconocer si existen otras direcciones posibles, sería necesario guardar las coordenadas de cada celda en la tabla para volver a examinarla más adelante.

Comparando ambas posibilidades, el método alternativo podría requerir una tabla de bifurcaciones de mayor tamaño. Esto se debe a que las coordenadas de la celda se introducen en la tabla en cada paso, mientras que la versión original solo las almacena cuando existen varias direcciones posibles.

Veamos lo que sucede: al comienzo de la generación no hay pasillos aún formados, así que habrá más de una dirección posible al examinar cada celda. En esa situación ambos métodos consumen el mismo espacio en la tabla. A medida que el laberinto se va formando, la probabilidad de que una celda tenga varias direcciones posibles disminuye, hasta llegar a situaciones en las que solo exista una. En ese momento, el método alternativo seguirá introduciendo las coordenadas de la celda en la tabla, mientras que el método original no lo hará.

El método alternativo es más simple de implementar y consumiría menos tiempo de ejecución en cada paso. A cambio comprueba más celdas. No obstante, sin un análisis detallado o una medición experimental, resulta difícil determinar cuál de los dos métodos requeriría menos tiempo de ejecución.

Por otro lado, el método alternativo necesitaría en general reservar más memoria para la tabla, ya que su tamaño vendría determinado por la máxima longitud que puede tener un recorrido. En un caso extremo con un único camino que recorriera todo el mapa sin bifurcaciones, seguiría almacenando las coordenadas de cada celda incluso cuando solo existiera una dirección de avance. Esta limitación podría resultar importante en un ZX Spectrum, donde la memoria disponible es escasa. Por ello, resulta razonable pensar que los autores del juego original optaran por una implementación que redujera al mínimo las necesidades de memoria del algoritmo.

Más allá de las variantes iterativas descritas, este algoritmo presenta una característica interesante: su estructura se adapta de forma natural a una implementación recursiva.

Cada vez que el algoritmo avanza a una nueva celda, continúa el proceso desde ella hasta que ya no es posible avanzar. En ese momento regresa a la celda anterior para explorar las direcciones que aún no se han examinado. Este comportamiento coincide exactamente con el mecanismo de llamada y retorno propio de la recursión[11].

En una implementación recursiva convencional no sería necesaria una tabla de bifurcaciones explícita –como la implementada aquí con las listas slBranch_x y slBranch_y–, ya que la propia pila de llamadas del sistema conservaría automáticamente toda la información necesaria para continuar el algoritmo tras regresar de cada llamada.

Aunque Scratch permite que un script se invoque a sí mismo, no crea un contexto independiente con sus propias variables en cada invocación, sino que reutiliza el mismo contexto del objeto, a excepción de los parámetros de la llamada, que sí son independientes en cada invocación. Es decir, si el script se llama a sí mismo y modifica el valor de una variable local al objeto o global al programa, este cambio afectará a todas las invocaciones.

---

[11] La *recursividad* es una técnica que permite resolver un problema dividiéndolo en subproblemas más pequeños del mismo tipo. Un script recursivo se llama a sí mismo para resolver dichos subproblemas hasta alcanzar un caso base que puede resolverse directamente.

Esta limitación dificulta la implementación recursiva del algoritmo, aunque no la hace imposible. El apartado siguiente presenta una realización de este tipo basada en el método alternativo descrito al principio de este apartado.

5.4 Implementación recursiva de la generación del laberinto

La implementación recursiva del algoritmo de creación del laberinto resulta algo dificultosa al no poder contar con variables locales independientes en cada invocación.

Para optimizar la ejecución, precalculamos durante el arranque las 4!=24 permutaciones posibles de las cuatro direcciones de movimiento y las almacenamos en la lista sIPermutations[12]. Cada permutación se codifica como un número de cuatro dígitos, donde cada uno representa una dirección. Por ejemplo, la permutación 1234 indica que se deben examinar las direcciones arriba, derecha, abajo e izquierda, en ese orden.

Para ello utilizamos el script Generate Directions, llamado cuando se inicia el programa desde when clicked –Figura 5.35–. El procedimiento consiste en generar iterativamente una permutación aleatoria y comprobar si está presente en la lista. Si no lo está, se añade. El proceso termina cuando la lista alcanza los 24 elementos. Aunque existen métodos más sofisticados para obtener las permutaciones, esta solución es muy compacta en Scratch, y su coste en tiempo de ejecución es aceptable, ya que solo hay 24 posibilidades.

El script auxiliar Get Random Direction –Figura 5.36– obtiene una permutación al azar y la guarda en sDirection. Para ello, selecciona números al azar entre 1 y 4 y los incorpora a la variable únicamente si todavía no están en ella, hasta completar los cuatro elementos.

El script when I receive Create Map –Figura 5.37– asigna la celda (15, 15) a sCell_x y sCell_y como punto de partida del laberinto, introduce un bloque vacío en esa posición y realiza la primera llamada al script recursivo Create Maze. En la llamada pasa las coordenadas de la celda y una de las permutaciones elegida al azar.

El script Create Maze –Figura 5.38– comprueba si puede avanzar en cada una de las cuatro direcciones indicadas por la permutación recibida como parámetro, utilizando para ello el script Check Directions –Figura 5.39–. Cuando el avance en una dirección es posible, vuelve a llamarse a sí mismo, pasando como parámetros las nuevas coordenadas y una nueva permutación elegida al azar.

Observe que no es posible utilizar un bucle para recorrer las cuatro direcciones, ya que cualquier variable empleada como índice sería sobreescrita por las invocaciones posteriores del script. Por este motivo también recurrimos a la recursividad para esta tarea. Check Directions recibe como parámetros tanto la permutación –directions– como el índice que identifica cuál de las cuatro direcciones debe comprobar –dirnum–. Al finalizar la ejecución, el script se llama de nuevo a sí mismo incrementando el índice en una unidad para procesar la siguiente dirección. Para finalizar la recursión basta comprobar, al comienzo del script, si el índice supera el valor 4; en ese caso, la ejecución termina.

El script Check Move –Figura 5.40– verifica si es posible avanzar hasta la siguiente celda, formulando las condiciones en lógica inversa y guardando el resultado en la variable sIsMoveValid: si las coordenadas caen fuera del laberinto o si las celdas implicadas contienen ya un bloque vacío, termina con sIsMoveValid=false. Para este cometido utiliza una versión extendida de Get Block.

Si el movimiento es válido, inicializa las dos celdas con bloques vacíos mediante Set Corridor e inicia otra recursión de Create Maze pasando las coordenadas de la nueva celda y una nueva permutación tomada al azar.

---

[12] A una tabla de este tipo, con elementos precalculados, se la conoce como *lookup table*.

Finalmente, con independencia de si el movimiento fue válido o no, continúa verificando la siguiente dirección. Para ello, realiza una llamada recursiva a Check Directions en la que se incrementa el índice en una unidad.

El script Set Corridor –Figura 5.41– utiliza una versión extendida de Set Block, que recibe adicionalmente las coordenadas de la celda. Las Figuras 5.42 y 5.43 muestran los scripts adaptados para la ocasión Get Block Ex y Set Block Ex.

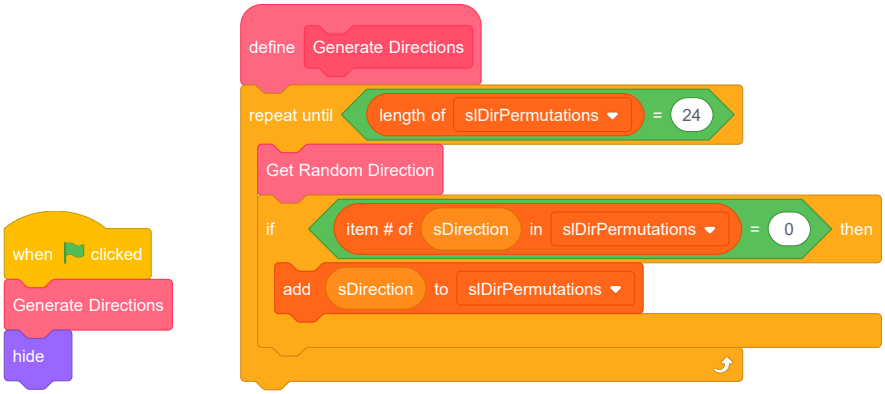


Figura 5.35. Scripts when clicked y Generate Directions de Map.

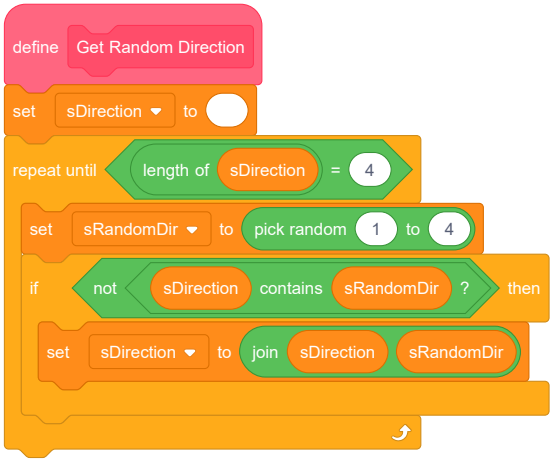


Figura 5.36. Script Get Random Direction de Map.

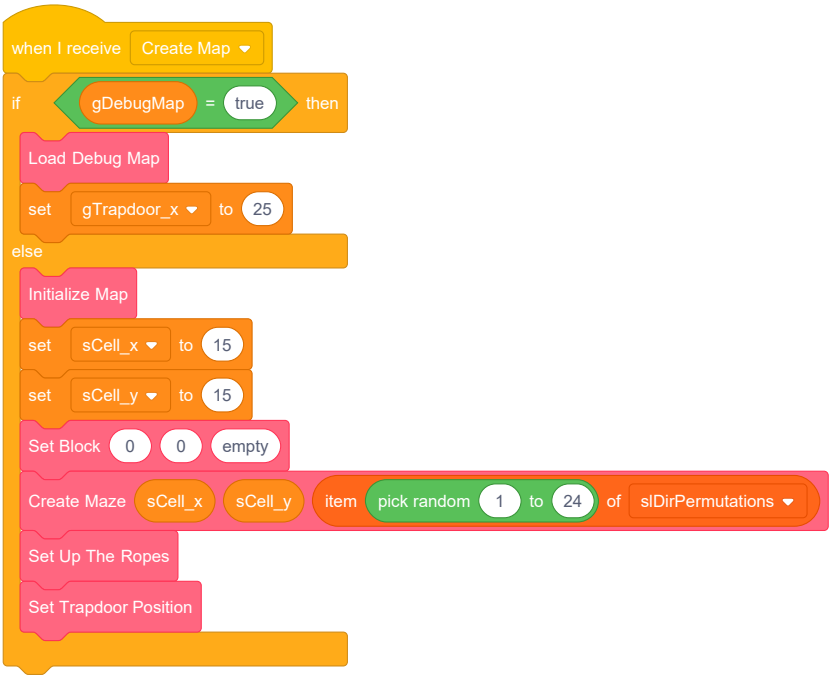


Figura 5.37. Script when I receive Create Map de Map.

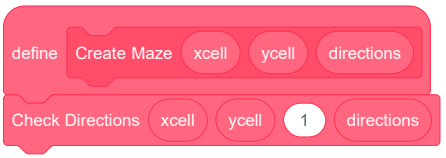


Figura 5.38. Script Create Maze de Map.

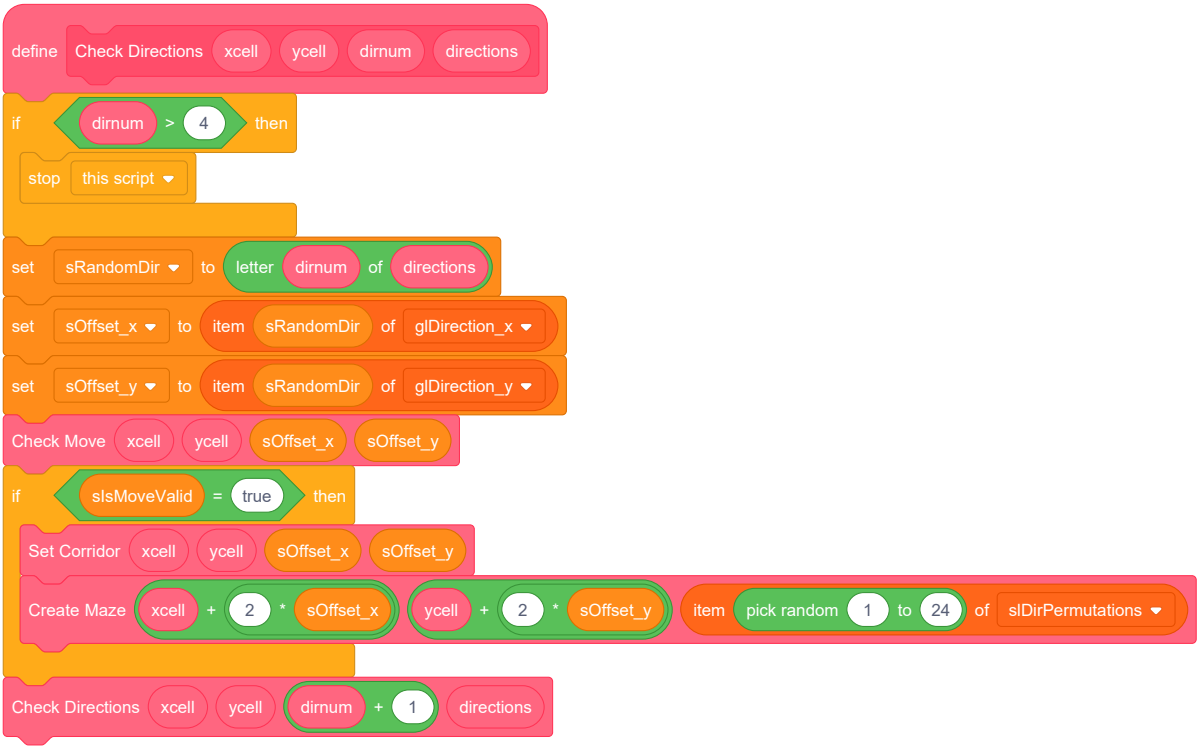


Figura 5.39. Script Check Directions de Map.

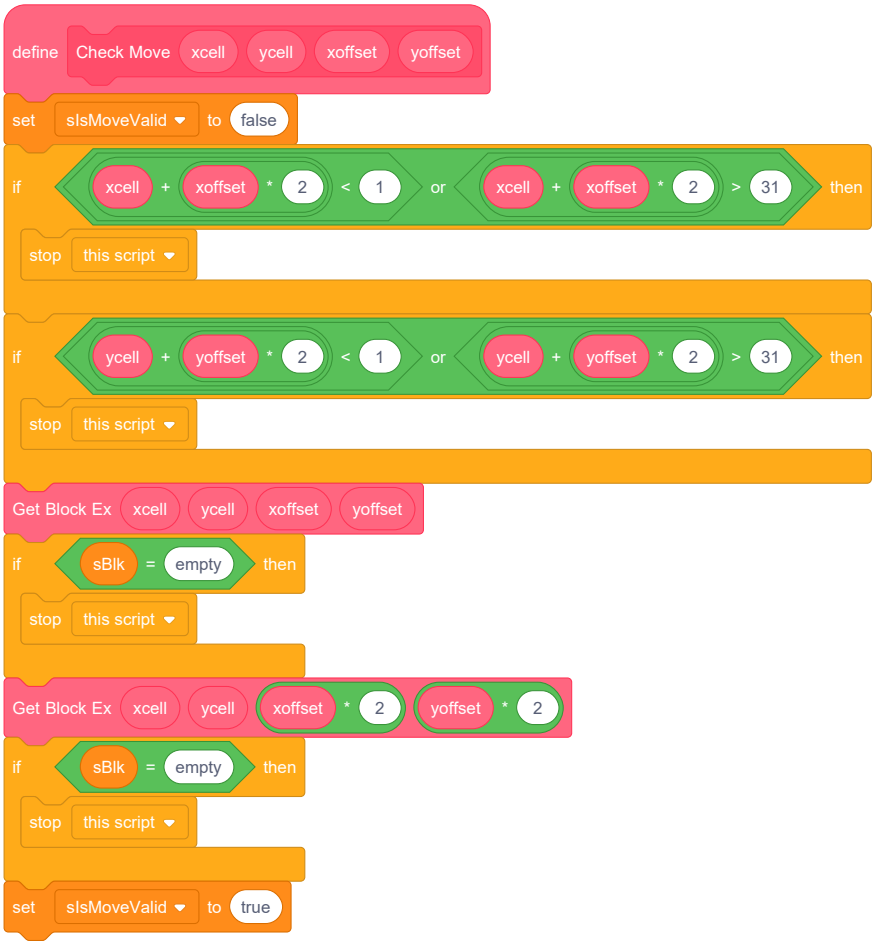


Figura 5.40. Script Check Move de Map.

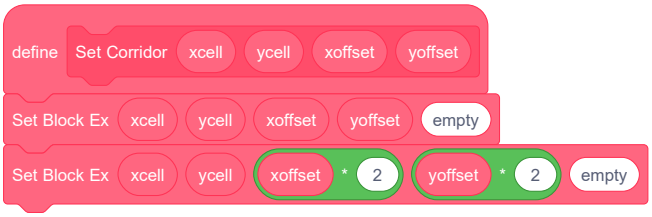


Figura 5.41. Script Set Corridor de Map.

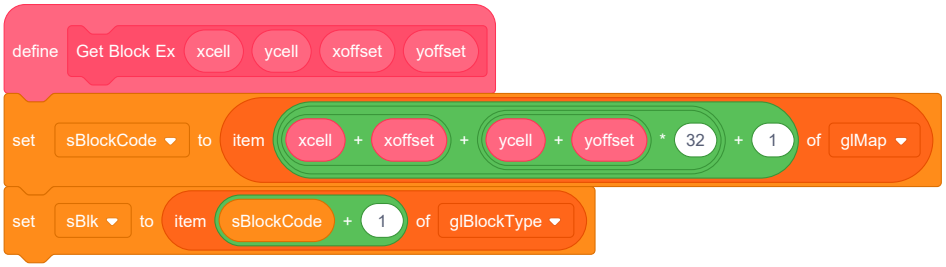


Figura 5.42. Script Get Block Ex de Map.

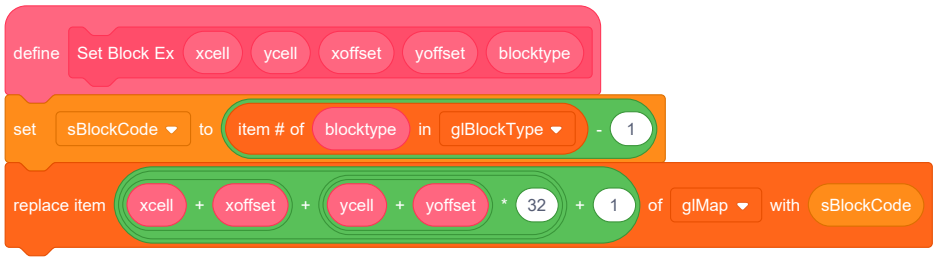


Figura 5.43. Script Set Block Ex de Map.