4 | Recursión y Retractación (*Backtrack*)

Rodrigo Eduardo Colín Rivera

Objetivo

Comprender el algoritmo de búsqueda con retractación (*backtrack*) y su relación con recursión. Llevar a cabo su implementación en la construcción de laberintos. Entender la forma de representar la lógica de la construcción del laberinto y su visualización.

Introducción

Retractación es un algoritmo para encontrar soluciones en problemas computacionales donde se encuentran varios candidatos parciales de solución que se van descartando conforme avanza el algoritmo o la búsqueda.

El problema más clásico de la aplicación de retractación es el problema de las **ocho reinas**, que consiste en colocar ocho reinas en un tablero de ajedrez convencional de manera que ninguna ataque a las demás. En cada paso del algoritmo se agrega una reina a una casilla y se verifica que no ataque a las **k** reinas del tablero. Cuando no se cumple esta condición, se vuelve a una solución válida previa y se continúa probando hasta que satisfaga que no se ataquen ninguna de las reinas del tablero.

La aplicación de la técnica de retractación sólo tiene sentido en problemas que involucran soluciones parciales. Hay que notar que es una manera más eficiente que el uso de fuerza bruta, ya que se descartan muchas posibilidades que no cumplen los requisitos para ser solución del problema.

Conceptualmente, retractación es similar a la búsqueda en profundidad en árboles. Considerando que cada nodo en el árbol de búsqueda es una posible solución parcial del problema, la forma de recorrerlo es mediante recursión; podando o descartando subárboles que no son válidos como solución.

Desarrollo e implementación

Se desarrollará una aplicación que genera laberintos usando una interfaz gráfica.

Algoritmo de construcción del laberinto

El objetivo de este algoritmo es construir un escenario que consiste en una serie de pasillos que cumplan con los requisitos siguientes:

- Es posible llegar a cualquier punto del laberinto desde cualquier otro punto. Es decir, no hay regiones cerradas inaccesibles.
- La ruta entre dos puntos cualquiera del laberinto es única.

Por lo tanto, es posible seleccionar un punto como *inicio* y cualquier otro como *destino* y existe un único camino entre ellos.

Para definir este problema se empleó el ejemplo de la siguiente página para ver la construcción (primer ejemplo: "*recursive backtracker*"): Recursive Backtracker [1] También hay una liga sobre el algoritmo y una implementación hecha en lenguaje Ruby: Maze generation [2]

A continuación se explica la construcción del laberinto, con un algoritmo que satisface los requisitos anteriores:

1. El algoritmo comienza con un tablero de celdas de tamaño N x M. Figura 4.1

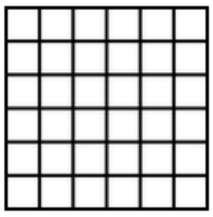

Figura 4.1 Ejemplo con N=6 y M=6.

2. Se elige una celda aleatoriamente como la celda *inicial*/actual, se marca como visitada y se agrega a la pila. Figura 4.2
3. Se elige de manera aleatoria una dirección hacia donde moverse, esto consiste en elegir una casilla adyacente que no haya sido visitada. Dependiendo de la dirección elegida se debe borrar la pared correspondiente. Se marca como visitada la celda, se empuja a una pila de celdas y se actualiza la celda actual. Figura 4.3

[1] http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap
[2] http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking

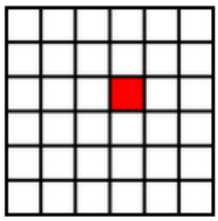

Figura 4.2 Celda inicial.

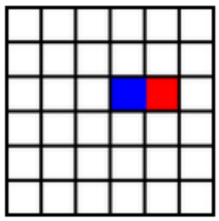

Figura 4.3 Ejemplo donde se elige la celda de la derecha. Se borra la pared entre las celdas y se marca la nueva celda actual (color rojo).

4. El paso anterior se repite hasta que ya no haya direcciones por elegir, es decir, la celda actual se queda encerrada. Figura 4.4

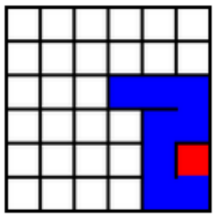

Figura 4.4 Tras algunos movimientos, la celda actual ya no puede seguir moviéndose.

5. Estando encerrados sin poder elegir una celda adyacente sin visitar, se procede a explusar una celda de la pila para cambiar la posición de la celda actual y repetir el algoritmo desde el paso 3 hasta recorrer todo el tablero de celdas.Figura 4.5 y Figura 4.6

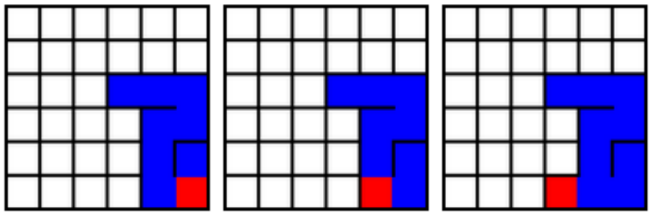

Figura 4.5 Ejemplo de un retroceso usando la pila y cambiando la celda actual.

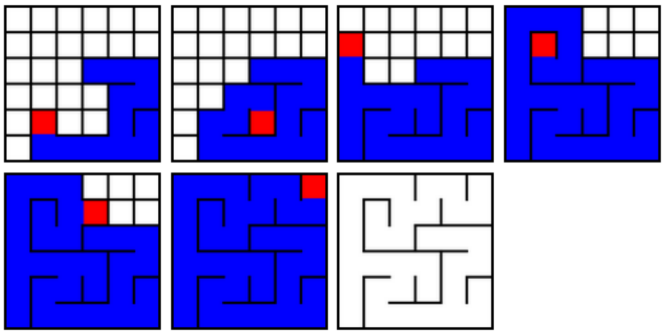

Figura 4.6 Ejemplo tras varios pasos del algoritmo hasta cubrir todo el tablero.

Implementación

Se debe implementar tanto la lógica de construcción del laberinto como su visualización. Utilicen `Processing/Java` ya que solo requiere usar las primitivas de dibujo: `stroke()` y `line()`. Adicionalmente también pueden usar: `fill()` y `rect()`. Es recomendable usar los siguientes métodos para una adecuada visualización: `background()` y `size()`. El material auxiliar para esta práctica ya incluye cómo dibujar el laberinto.

Requisitos y resultados

Para llevar a cabo la evaluación de esta práctica es necesario implementar la construcción del laberinto usando retractación. Empleen como estructura auxiliar durante la construcción una pila (*stack*).

La implementación debe ser lo más generalizada y robusta posible, es decir, se deben definir parámetros o valores para determinar el ancho y largo del laberinto.

No olviden documentar su código. Utilicen el estándar de `JavaDoc`, describan de manera breve, clara y concisa el funcionamiento de sus métodos. Si omiten documentación en su código les afectará negativamente en su calificación.

El ejemplo visto en laboratorio se construye en tiempo de ejecución. Es deseable que se muestre esta construcción pero no es necesaria. Es válido mostrar el resultado final aunque no se visualice la construcción.

Si todo se implementa correctamente debe ser posible generar diferentes tamaños de laberintos.