3 | Retractación *(Backtracking)*

Meta

Que el alumno utilice la estrategia *divide y vencerás* para diseñar un algoritmo recursivo.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Comprender el algoritmo de búsqueda con retractación (*backtrack*).
- Implementar dicho algoritmo utilizando recursión.
- Resolver un laberinto utilizando una estrategia recursiva.
- Visualizar el comportamiento del algoritmo y comprender la relación entre su funcionamiento y el manejo de la memoria.

Antecedentes

Retractación, o *backtracking* en inglés, es un algoritmo de búsqueda para la resolución de problemas, en el cual se van construyendo soluciones paso a paso, intentando varias posibilidades de forma sistemática, hasta encontrar una que funcione.

La estrategia consta de los pasos siguientes:

- Se construye una solución paso por paso, donde haya varias opciones se elige alguna y se sigue adelante, hasta que ocurra una de dos cosas:
  - ⋆ Si se llega a una solución completa el algoritmo termina y devuelve la solución.
  - ⋆ Si se terminan las acciones disponibles y no se obtuvo el resultado deseado, el algoritmo regresa sobre sus pasos hasta el último punto donde tuvo que elegir entre varias opciones posibles, cancela el camino seguido por no haber funcionado y elige otra de la opciones disponibles.

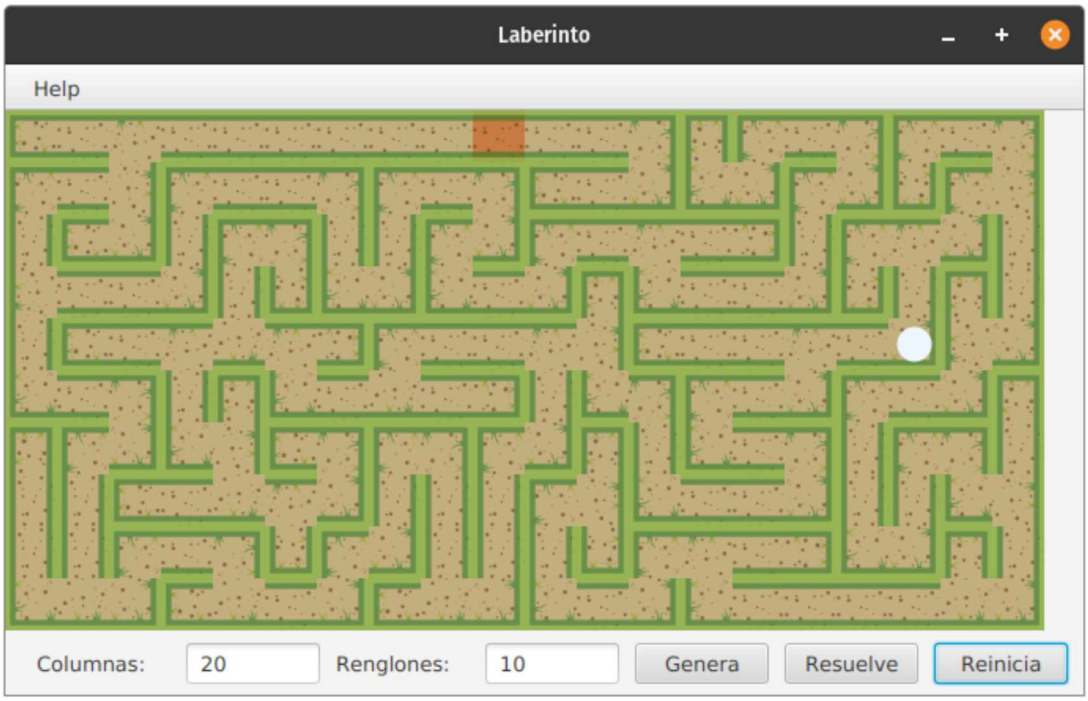

Figura 3.1 Agente en un laberinto generado aleatoriamente. Su objetivo es encontrar el camino que lo lleve a la meta.

- Si se han intentado todas las posibilidades y no se encontró solución, se declara que el problema no tenía solución.

Esta estretegia funciona bien para dominios finitos, donde la cantidad de pasos a intentar por cada camino está acotada, como en un laberinto. En otros escenarios este algoritmo puede no terminar.

Desarrollo

Se diseñará e implementará un algoritmo para que un agente resuelva laberintos que son generados aleatoriamente. Es posible ver el escenario en la Figura 3.1.

Herramientas

Para esta práctica se cuenta con un código auxiliar que ofrece los elementos siguientes:

- Interfaz gráfica de usuario programada con `JavaFX`.
- Generación automática de laberintos al presionar el botón *Genera*.

- Opciones para cambiar el número de columnas y renglones en el laberinto.
- Una clase agente con las acciones necesarias para implementar la solución al problema.
- Un botón auxiliar *Reinicia* que regresa al agente a su posición inicial en un laberinto dado y desmarca todas las celdas. Es útil para depurar el algoritmo de solución.

Se deberá implementar un algoritmo recursivo que le permita al agente alcanzar la meta. Para ello, el agente cuenta con los métodos siguientes:

- `public boolean avanza(Dirección dirección)`, donde la dirección es un `enum` con los valores NORTE, ESTE, SUR, OESTE. Este método mueve al agente en la dirección indicada. Nota, los objetos de esta enumeración tienen un método `public Dirección opuesta()` por si el agente necesita retroceder.

- `public LinkedList<Dirección> direccionesPasillos()`, devuelve una lista de direcciones en las cuales se puede mover el agente desde su posición actual. Para leer los elementos de la lista uno por uno, puedes utilizar el código siguiente en `Java`:

```
for(Dirección d : agente.direccionesPasillos()) {
  System.out.println("Hay un pasillo en la dirección " + d);
}
```

- `public boolean estáEnMeta()`, indica si la posición actual del agente es la meta a la que desea llegar.

- `public void marcaCelda()`, se utiliza para dejar una marca que le permita saber al agente que ya pasó por ahí.

- `public void desmarcaCelda()`, útil cuando se desea deshacer un movimiento pues ya se vió que no lleva a la solución. Cuidado, si se desmarcan celdas descuidadamente el agente podría atorarse visitando las mismas celdas una y otra vez.

- `public void ejecutaAnimación()`, se debe mandar llamar cuando el algoritmo haya encontrado la ruta hacia a la meta, para que el agente pueda seguirla y llegar ahí.

- `public boolean miraSiNoVisitada(Dirección d)`, permite al agente voltear en la dirección indicada para ver si ahí hay un pasillo que no haya sido marcado como visitado.

El objetivo de esta práctica es utilizar las acciones anteriores dentro del método `public boolean resuelveLaberinto()` de la clase `Control`, para que, al presionar el botón *Resuelve*, el algoritmo encuentre la ruta desde la posición actual del agente hasta la meta y ejecute la animación. Obsérvese que al inicio de este método el agente ya fue creado

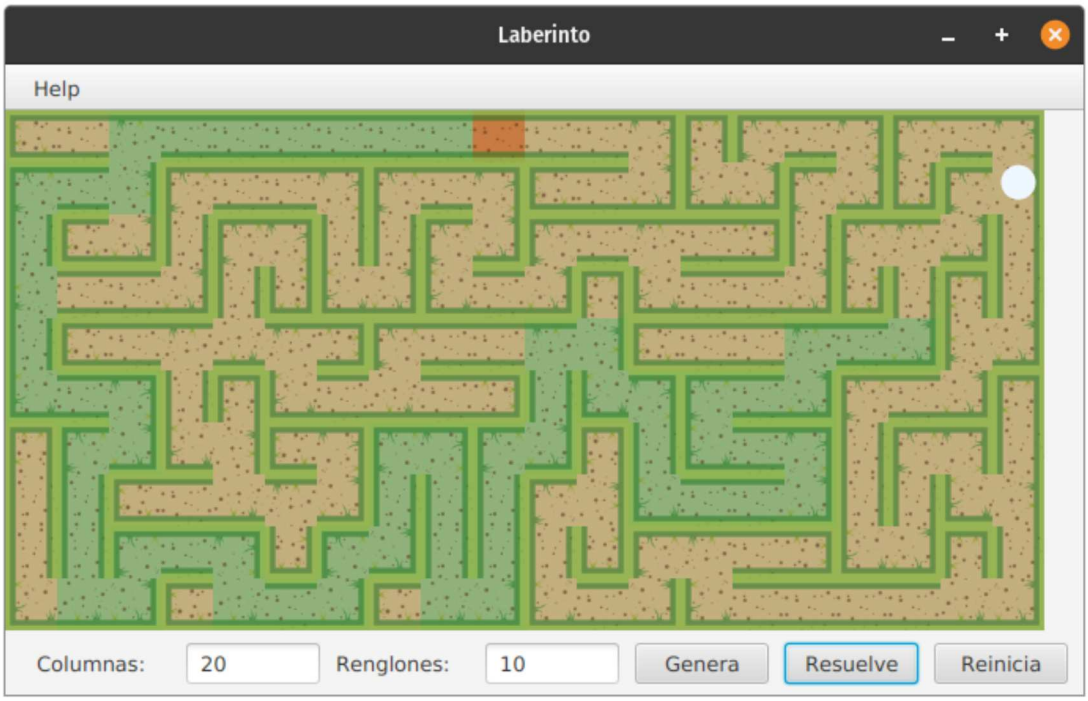

Figura 3.2 El agente recorre varias posibilidades antes de llegar a la ruta correcta, corrige cada vez que encuentra un camino cerrado regresándose por donde vino.

y se encuentra en algún lugar del laberinto, tal como se muestra en la Figura 3.1, por lo que no es necesario crear el objeto `agente` ni utilizar o modificar ningún otro método para programar la solución, sólo se necesita usar los métodos del agente.

La animación mostrará todas las rutas que intentó el agente antes de encontrar la correcta, pero las marcas en el piso deben corresponder sólo a la ruta correcta. Un ejemplo de esto se muestra en la Figura 3.2.

Diseño del algoritmo

Para diseñar el algoritmo de solución del laberinto se utiliza la estrategia *divide y vencerás*. Esto quiere decir que, en lugar de pensar en todo el laberinto, analizaremos las opciones que tiene el agente desde el lugar donde se encuentra para dividir su comportamiento en dos casos: **caso base** y **caso recursivo**. En cada caso, debemos pensar como si fuéramos el agente y sólo pudiéramos ver lo que ve él.

Caso base. Como en todo algoritmo recursivo, es el caso más sencillo: cuando ya terminamos. Se deben determinar dos cosas: ¿cómo saber si estamos en el caso base? ¿qué acciones corresponden a este caso? Para este problema las respuestas son las siguientes:

- ¿Cómo saber si estamos en el caso base? En este problema, terminamos cuando el agente se encuentra en la celda meta.
- ¿Qué acciones corresponden a este caso? Ejecutar la animación y devolver un `true` indicando que el agente ya encontró la meta. El agente tiene memoria, cada vez que se movió la acción correspondiente quedó guardada en su memoria, al ejecutar la animación veremos todos los movimientos que tuvo que realizar por los pasillos donde estuvo buscando.

Caso recursivo. El agente no está sobre la meta, se encuentra al inicio o a la mitad del pasillo y necesita decidir por dónde continuar su búsqueda. Tendrá que probar sistemáticamente todas las direcciones en las cuales se puede mover, excepto el pasillo por donde llegó. Si llega al final de un pasillo sin salida, tendrá que regresar por donde vino hasta el último punto donde le quedaban direcciones posibles.

Para implementar este algoritmo debes aprovechar la memoria de `Java` en dos formas:

- Marcas en el piso del laberinto. Asegúrate que, cada vez que el agente se desplace en el laberinto, deje una marca en la celda en la que estaba justo antes de moverse. Eso evitará que se atore yendo y viniendo sobre las mismas celdas. En la figura Figura 3.3 esta celda se marca con la flecha roja.
- En el registro de llamada a método, en la pila de `Java`. Recuerda que, al ejecutar un método recursivamente, cada llamada tiene su propio registro con los valores de las variables locales en esa llamada. Utiliza entonces las variables locales para que el agente sepa qué dirección debe visitar a continuación. **TIP:** te servirá el ejemplo del `foreach` con la lista de vecinos expuesto anteriormente.

Programa estos dos casos en el método `public boolean resuelveLaberinto()` de la clase `Control` y al presionar el botón de la GUI tu agente deberá llegar desde su posición actual hasta la celda marcada como meta.

Requerimientos

1. Para esta práctica sólo necesitas modificar la función indicada.
2. Adjunta un pdf con las respuestas a las preguntas de la sección siguiente.

Preguntas

1. ¿Es forzoso que el agente recorra todo el laberinto para encontrar la meta? ¿Por qué?

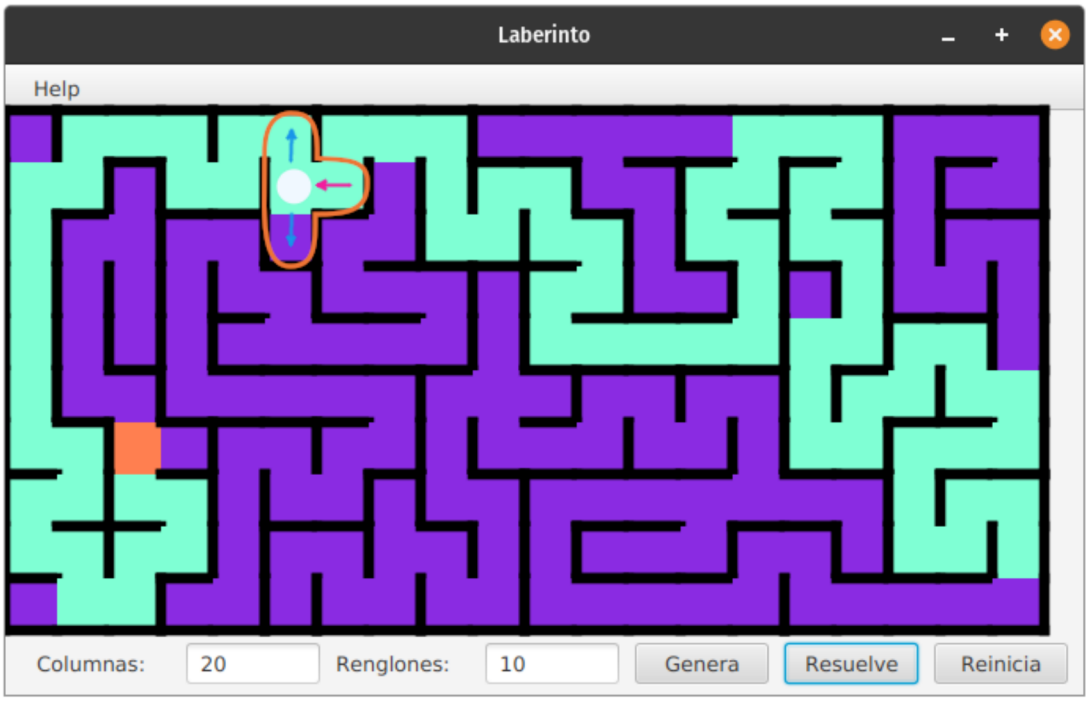

Figura 3.3 El agente sólo debe considerar lo que ocurre en su vecindad para decir qué dirección tomar a continuación. Importa: desde dónde viene, qué pasillos ya ha visitado y cuáles no.

2. ¿Es posible que el agente encuentre la ruta correcta en el primer intento? De ser así, ¿cuándo sucedería esto?