

*Departamento de Gestión de Proyectos y Sistemas.*

*Estructuras de datos. BPTSP06*

*Trimestre 2223-1*

*Prof. Christian Guillén Drija.*

# **Proyecto 1:** Hacedor de laberintos

## Contexto p**reliminar**

La industria de los videojuegos es una de las que tienen mayor crecimiento a nivel global, empleando una amplia variedad de profesionales tales como programadores, diseñadores, ingenieros de sonido, ingenieros de software, analistas de pruebas, expertos en física, matemáticos de la computación, guionistas, artistas, entre otros. Uno de los aspectos fundamentales en el desarrollo de este tipo de programas es la creación de los ambientes en los cuales se muestra el juego, es decir, los mapas a través de los cuales los personajes o naves se mueven. Para la creación de tales ambientes o mapas, existen diversos procedimientos y en su forma más simple, pueden estar constituidos por laberintos que restringen el movimiento de los móviles en el mundo recreado en el videojuego. En este proyecto, usted y su equipo de trabajo deberán diseñar un sistema para la creación de laberintos sencillos que además encuentren su solución.

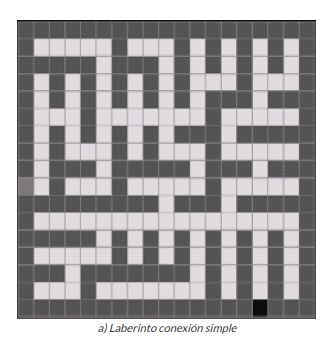
Como actividad preliminar, usted deberá realizar una investigación sobre el Tipo de Dato Abstracto Grafo. Se anexa un enlace en el que podrá conseguir información. Entiéndase tal recurso como introductorio y no único. La razón por la que se hace referencia a este TDA es que para el diseño del sistema Hacedor de Laberintos tendrán que implementarlo.

<https://drive.google.com/file/d/1Q65Rh-Tx6qwJODUlsmWVfw_-bzqttwiI/view?usp=sharing>

## Problema

Se requiere que su equipo de trabajo diseñe e implemente un sistema en Java que genere laberintos de conexión simple (LCS) en 2 dimensiones. Los laberintos de este tipo se caracterizan porque tienen puntos muertos o callejones sin salida, por lo tanto, solo tienen una solución entre la entrada y la salida. Además, el sistema deberá ser capaz de resolver el laberinto. A continuación se indican los requerimientos de manera más detallada.

## Requerimientos funcionales

1. Construir laberinto: generará el laberinto rectangular que deberá ser representado gráficamente. El usuario deberá indicar las dimensiones del mismo (n x m) y seguidamente generar el laberinto correspondiente. De forma aleatoria, el sistema escogerá alguna de las celdas que se encuentren en el borde del laberinto como la entrada y al finalizar la construcción del mismo, seleccionará otra celda, también en el borde, como salida. El algoritmo que se debe utilizar para generar el laberinto será el de Prim’s. En cualquier momento, el usuario podrá generar un nuevo laberinto LCS (ver figura a continuación).
2. Resolver laberinto: una vez construido el laberinto, el sistema deberá ser capaz de revolverlo, mostrando progresivamente los espacios visitados hasta encontrar la salida, tras lo cual, el sistema mostrará la ruta resultante. El usuario podrá seleccionar entre 3 métodos de solución: 
   1. Wall follower.
   2. Depth First Search (DFS).
   3. Breadth First Search (BFS).

En todos los casos, se deberá mostrar al usuario el tiempo empleado por el algoritmo seleccionado para resolver el laberinto. El usuario podrá aplicar distintos métodos en el laberinto activo.

## Requerimientos técnicos

1. La solución debe ser implementada con base en un grafo, que a su vez deberá ser implementado mediante una **lista de adyacencia**. Considere que los vértices representan los espacios y las aristas los pasillos entre los espacios. Si dos espacios están conectados, no habrá un muro entre ellos; en caso contrario, existirá un muro. Solo pueden establecerse conexiones entre espacios adyacentes.
2. Puede utilizar cualquier otra estructura auxiliar de ser necesario. Sin embargo, **NO podrá utilizar librerías para la implementación de las estructuras de datos**, solo podrá utilizar librerías para lo relativo a la representación gráfica del laberinto si fuese el caso.
3. El programa debe poder representar el laberinto de manera gráfica.
4. La aplicación debe ofrecer una interfaz gráfica al usuario.
5. Debe documentar el proyecto con [**Javadoc**](https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=646:documentar-proyectos-java-con-javadoc-comentarios-simbolos-tags-deprecated-param-etc-cu00680b&catid=68&Itemid=188).
6. Junto al programa, cada equipo deberá presentar un [**Diagrama de clases**](https://www.lucidchart.com/pages/es/tutorial-de-diagrama-de-clases-uml)*(arquitectura detallada)* que explique la solución obtenida.

## Consideraciones

* Los proyectos **podrán ser sometidos a defensa**, es decir, el facilitador y/o preparadores convocarán al equipo para una revisión oral.
* Los equipos de trabajo deberán utilizar [**GitHub**](https://github.com/) para el control de versiones y facilitar el trabajo en equipo de manera remota. De esta forma, podrán comenzar a crear su portafolio de trabajos, elemento que puede ser importante a la hora de buscar trabajo. En el registro se deberá reflejar la participación activa y significativa de los integrantes.
* Los proyectos que no tengan interfaz gráfica, serán calificados con **0 (cero)**.
* Los proyectos que sean iguales o parecidos, serán calificados con **0 (cero)**.
* Los programas que “no corran”, serán calificados con **0 (cero).**
* Los equipos pueden tener como **máximo 3 personas.**

## Criterios de evaluación

* *Funcionalidad:* Capacidad para proporcionar las funcionalidades que satisfacen las necesidades explícitas e implícitas bajo unas ciertas condiciones. **(50%)**
  + *Adecuación:* El programa ofrece todas las funcionalidades que respondan a las necesidades, tanto explícitas (contenidas en el documento descriptivo del proyecto) como implícitas; entendiendo como necesidades implícitas, aquellas que, no estando descritas en el documento, surgen como resultado de un concienzudo análisis del problema planteado y que aseguran el correcto funcionamiento del programa.
  + *Exactitud*: El programa genera los resultados o efectos correctos o acordados, con el grado necesario de precisión.
* *Fiabilidad*: Capacidad para mantener un nivel especificado de prestaciones cuando se usa bajo ciertas condiciones.
  + *Madurez*: El programa no presenta fallas originadas por errores de programación, análisis o diseño. **(10%)**
  + *Tolerancia a fallos*: El programa responde adecuadamente al manejo inadecuado del usuario; es decir, mantiene su correcto funcionamiento aun cuando el usuario introduzca datos erróneos o manipule inadecuadamente las interfaces de usuario. **(10%)**
* *Usabilidad*: Capacidad del proyecto para ser entendido, aprendido, usado y al mismo tiempo, ser atractivo para el usuario, cuando se usa bajo condiciones específicas.
  + *Comprensibilidad*: El programa ofrece una interfaz de fácil comprensión, facilitando su aprendizaje y correcta utilización. El programa emite mensajes de alerta cuando se introducen valores erróneos. Existen elementos informativos que indican al usuario como operar el programa. **(5%)**
  + *Capacidad de ser atractivo*: El diseño de la interfaz de usuario, esto es: disposición de controles, esquema de colores, utilización de cajas de diálogo y demás elementos gráficos; es atractivo para el usuario. **(5%)**
* *Eficiencia*: Capacidad para proporcionar prestaciones apropiadas, relativas a la cantidad de recursos usados, bajo condiciones determinadas.
  + *Estructuras de datos*: Utiliza eficientemente las estructuras de datos para la solución del problema. **(10%)**
  + Reutilización de Código: No debe haber redundancia de clases en el diseño de la solución. Deben utilizarse los mecanismos de reutilización de código, propios de la POO, tales como la herencia, implementación de interfaces, sobrecarga de métodos. **(10%)**