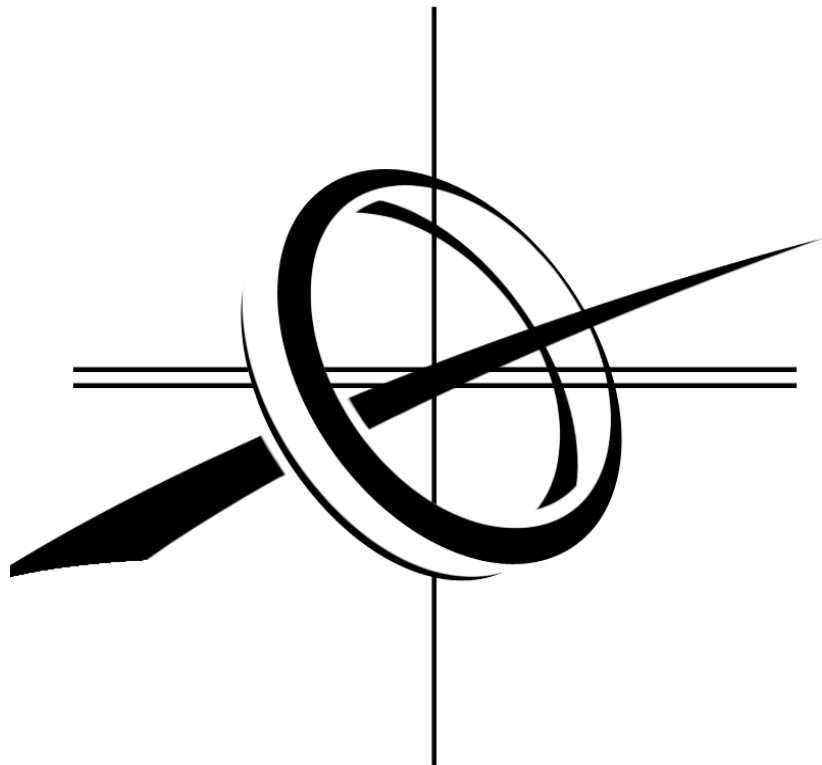


UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA UNIVERSITARIA DE INFORMÁTICA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN**

TRABAJO FIN DE CARRERA

Desarrollo y gestión de programas y juegos recreativos en
programación orientada a objetos.



Autor: Juan Carlos Recio Abad

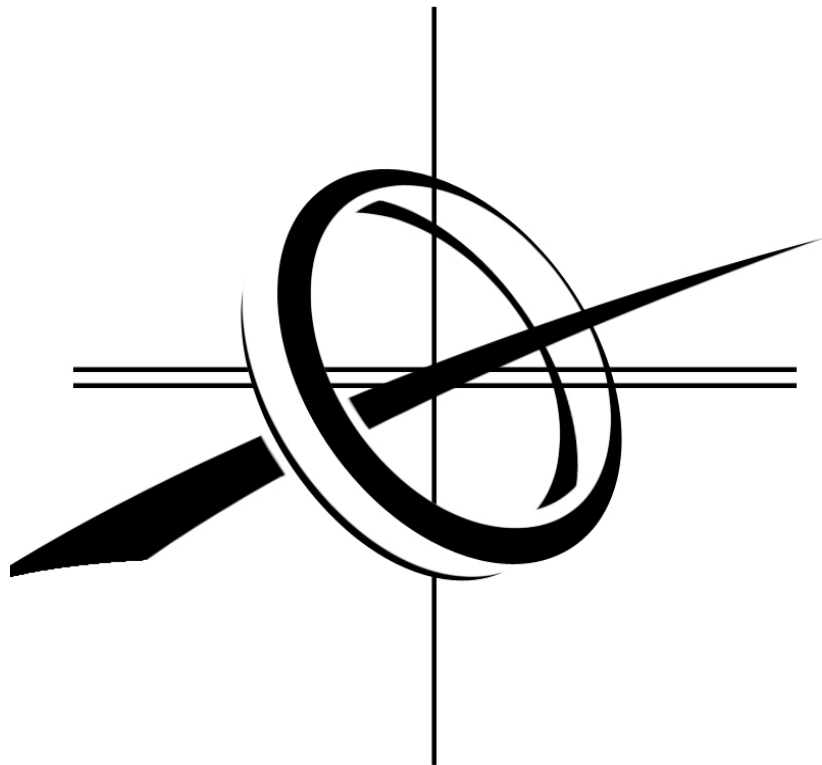
Curso Académico: 2012-13

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA UNIVERSITARIA DE INFORMÁTICA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN**

TRABAJO FIN DE CARRERA

Desarrollo y gestión de programas y juegos recreativos en
programación orientada a objetos.



Autor: Juan Carlos Recio Abad

Tutora: Rosa María Pinero

Curso Académico: 2012-13

Vuestro honor no lo constituirá vuestro origen, sino vuestro fin.

Friedrich Wilhelm Nietzsche

ÍNDICE

1. INTRODUCCIÓN	5
2. OBJETIVOS	8
2.1 Objetivos específicos	8
2.2. Estructura del documento	8
3. PROGRAMAS Y JUEGOS RECREATIVOS	9
3.1 Juegos recreativos como programas informáticos	9
3.2 Estrategia y táctica en los juegos recreativos	10
3.3 Estructura y entorno de los juegos recreativos	14
4. ANÁLISIS DE REQUISITOS DE LA APLICACIÓN	16
4.1 Juegos de la aplicación: Laberintos y sopas de letras	16
4.2 Formas del área de juego	17
4.3 Características y estructura	18
4.4 Representación de los elementos	20
4.5 Estado de los juegos	23
4.6 Movimientos y funcionalidad	24
4.7 Dificultad	32
4.8 Soluciones	34
4.9 Densidad de un laberinto	38
5. DISEÑO	39
5.1 Diseño de algoritmos	39
5.1.1 Generación de laberintos	40
5.1.1.1 Generación de números aleatorios	40
5.1.1.2 Generación de la estructura	41
5.1.1.3 Generación de las dimensiones	42
5.1.1.4 Generación de las posiciones	43
5.1.1.5 Generación de las soluciones	45
5.1.1.6 Generación de contenidos	47
5.1.1.7 Generación de movimientos	50
5.1.2 Resolución de laberintos	51
5.1.3 Generación de sopas de letras	56
5.1.3.1 Generación de la estructura	56
5.1.3.2 Generación de contenidos	57
5.1.3.3 Generación de palabras	59
5.1.4 Resolución de sopas de letras	59
5.2 Diseño e implementación	60
5.2.1 Diagrama de casos de uso	60

5.2.2 Diagrama de clases	67
5.2.3 Diagramas de secuencia	68
5.2.4 Definición e implementación de clases	70
5.2.1 Posiciones	71
5.2.1 Tableros	76
5.2.2 Caminos	80
5.2.3 Espacio de soluciones	84
5.2.4 Funcionalidad números aleatorios	87
5.2.5 Movimientos	88
5.2.6 Entorno completo	90
5.3 Almacenamiento externo	95
5.3.1 Almacenamiento en ficheros	100
5.3.2 Almacenamiento en XML y bases de datos	101
5.4 Interfaz gráfica de usuario	102
6. EJECUCIÓN Y RESULTADOS	106
7. CONCLUSIONES	116
8. BIBLIOGRAFÍA	117

1. INTRODUCCIÓN

La historia de la informática desde un punto de vista de los juegos comenzó realmente mucho antes de lo que la mayor parte de la gente cree. Los juegos interactivos arrancan en la década de los años setenta, pero es importante destacar aspectos lúdicos que tuvieron lugar al final de los años sesenta, en el Centro de Cálculo de la Universidad de Madrid. En esa época IBM entregó el equipo 7090 que había estado trabajando durante casi diez años en el CERN de Ginebra. Junto con el equipo llegaron, por ejemplo, diversas cintas con rutinas matemáticas escritas en FORTRAN o ALGOL y, también, un programa que era capaz de generar en una de las impresoras del sistema un determinado dibujo, del todo incomprensible, para cuya impresión los martillos de la impresora se movían de manera que el ruido que hacían reproducía a grosso modo, al himno de los Estados Unidos.

Puede parecer exagerado hacer pruebas y más pruebas para reproducir en una impresora el himno nacional de un país, pero hay otros ejemplos también espectaculares de lo que se llegó a hacer en aquellos tiempos.

En 1973, en el equipo G-58 de la empresa francesa Bull, existía un programa con unas características sorprendentes. Para ver sus efectos era necesario utilizar un transistor, un sencillo receptor de radio a transistores sintonizado en una emisora cualquiera. Mientras el programa se ejecutaba, sólo había que acercar el receptor de radio a un determinado lugar de la unidad central del sistema G-58 para que la sintonía radiofónica se perdiera y, con una serie de silbidos, el receptor de radio emitiera el himno francés de La Marsellesa.

Sencillos juegos de lógica como el NIM y otros parecidos son algunos de los primeros juegos pasados al ordenador, hasta llegar a los complejos juegos de hoy.

En 1958, William "Willy" A. Higginbotham consiguió simular una especie de juego elemental de tenis o ping-pong en la pantalla de un osciloscopio que fue utilizado como divertimento por parte de los visitantes del Museo Nacional de Brookhaven. A día de hoy se conoce como Tennis for two.

Desde incluso antes de la aparición de los primeros lenguajes de programación, ya se diseñaban en informática programas para resolver problemas y juegos relativamente simples, utilizando para ello las técnicas y recursos de los se disponían, que aunque a día de hoy están catalogados como demasiado primitivos, cumplían objetivamente bien su cometido. Un ejemplo de esto pueden ser las tarjetas perforadas. Para ello, previamente se tenía que haber escrito el código pertinente, por ejemplo en lenguaje ensamblador, para posteriormente poder procesar los algoritmos diseñados y obtener los resultados. La complejidad y tedio que esos procedimientos requerían, se fue simplificando progresivamente con la aparición de lenguajes de programación de más alto nivel más comprensibles para las personas.

Con todo esto, se podían elaborar situaciones en las que se disponían de ciertas condiciones que había que resolver, como encontrar la jugada ganadora en un juego, encontrar la salida en un laberinto, resolver una ecuación matemática, colocar varias damas en un tablero de ajedrez sin estar amenazadas, etc.

IMPORTANCIA DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

En un principio, la mayor parte de los esfuerzos se centró en el hecho de encontrar algoritmos más adecuados para procesar, analizar y resolver los distintos problemas que se pudiera, así como encontrar métodos más eficientes de hacer que una computadora pudiese jugar mejor o cumplir mejor funciones auxiliares y de soporte para trabajos que necesitaban mucha potencia de cálculo.

No sería exagerado decir que se habrán diseñado cientos de millones de ejemplos de programas y juegos de estas características, contruidos con lenguajes de programación tan conocidos como C, u orientados a la inteligencia artificial, como LISP.

No obstante, el paradigma de programación que realmente podría dar un enfoque realmente poderoso a todo esto, es la programación orientada a objetos.

La orientación a objetos es uno de los paradigmas más importantes y significativos de los últimos años. Sus técnicas pueden aplicarse durante el análisis, diseño e implementación de los programas, y permiten enfocar el problema que se quiere resolver en términos similares a los utilizados por la mente humana.

Es la enseñanza de la programación orientada a objetos como primer lenguaje la que desencadena el principal problema de éste proyecto. Los estudiantes antes de poder realizar un programa sencillo deben tener una clara noción de lo que es una clase, un objeto, un método, un paquete, etc. Y son estos precisamente los que generan dicha problemática.

Estudios realizados por el Instituto Científico Weizmann acerca de la enseñanza temprana de la orientación a objetos han demostrado que las principales causas que dificultan la comprensión de la programación orientada a objetos son los conceptos de clase y objeto, debido a que los estudiantes no lo diferencian. Así como el estado de un objeto, el paso de mensajes, el hecho de que un método puede cambiar el estado de un objeto, etc. [1]

Es por ello que existen numerosos manuales y herramientas creadas con el único fin de facilitar la comprensión de los conceptos básicos de la programación orientada a objetos, pues no es suficiente aprender la sintaxis de un lenguaje como es Java para poder resolver un problema, sino que se deben tener claros los conceptos para poder llevarlo a cabo.

Dichos conceptos, pueden ser comprendidos y asimilados de una forma bastante acertada, si los aplicamos al ámbito de los juegos recreativos de tablero, que representan de una forma muy clara cada propiedad y características de los objetos, debido a que su interpretación puede asociarse de una forma fácil y rápida mediante la visualización gráfica y geométrica que ofrecen dichos juegos.

Aunque programar juegos de computadora puede sonar divertido, para nada es trivial. Un buen juego es un excelente ejemplo de ingeniería: simple, fácil de usar, divertido, entretenido pero con mucho esfuerzo y trabajo bien pensado.

TECNOLOGÍAS Y RECURSOS UTILIZADOS EN EL PROYECTO

Para la elaboración del proyecto se ha utilizado el lenguaje de programación Java, tanto por su potencia actualmente, su versatilidad para poder realizar prácticamente cualquier tarea, y por ser realmente adecuado para ilustrar el uso de la programación orientada a objetos, haciendo transparente a usuarios y programadores el uso y gestión interna de la memoria, como por ejemplo es necesario hacer en lenguajes como C++, SmallTalk u Objective C.

Java, en este aspecto, brinda de una limpieza bastante considerable en cuanto a la lectura del código y su comprensibilidad, sin necesidad de mezclar punteros de memoria y otras características de bajo nivel con los aspectos pertinentes de la programación orientada a objetos.

Java también ofrece funcionalidades ricas en el uso de interfaces gráficas por lo que para el desarrollo de las mismas en este proyecto también se han realizado con el mismo lenguaje.

Para poder ejecutar Java, independientemente del sistema operativo, tan solo requiere tener instalada una máquina virtual Java que es donde se ejecutan los programas.

Para llevar a cabo el desarrollo del mismo se ha utilizado el IDE Eclipse, con la versión Juno.

2. OBJETIVOS

El objetivo principal del proyecto es construir una aplicación para crear algunos juegos basados en búsquedas de los cuales el proyecto se centra en laberintos y sopas de letras.

Para la consecución de estos objetivos es imprescindible estudiar las características y comportamiento de los laberintos y las sopas de letras, de este modo se consigue gestionar eficientemente el proyecto en sus diferentes fases.

2.1 OBJETIVOS ESPECÍFICOS

1. Crear una aplicación para diseñar laberintos y sopas de letras.
2. Conseguir automatizar la construcción y resolución de laberintos y sopas de letras.
3. Desarrollar las funcionalidades para extender juegos utilizando programación orientada a objetos.
4. Crear un modelo de objetos exportable a ficheros y bases de datos.
5. Implementar una interfaz gráfica de usuario para el manejo de la aplicación.

2.2 ESTRUCTURA DEL DOCUMENTO

La estructura del documento consta de una primera sección de estudio de los juegos recreativos, laberintos y sopas de letras. Con ello se realiza la fase de análisis de requisitos, características, conceptos y elementos que conforman la aplicación que se va a desarrollar.

Una segunda parte donde se realiza la fase de diseño y desarrollo de la aplicación, en la que se muestran a detalle los algoritmos utilizados, las clases y objetos involucrados, la relación entre todos ellos y el funcionamiento de todos los componentes.

En la siguiente parte, se muestran una serie de pruebas para mostrar los resultados de la aplicación.

Por último, el documento incluye las conclusiones de haber realizado este trabajo y la bibliografía y bibliografía web utilizada.

3. PROGRAMAS Y JUEGOS RECREATIVOS

A día de hoy, existen numerosos frameworks y herramientas dedicadas al desarrollo de juegos de múltiples tipos, lo cual hace que los desarrolladores puedan llevar a cabo gran cantidad de tareas de una calidad considerable. No obstante, el problema de este enfoque es la limitación que pueden sufrir dichos desarrolladores al verse inmersos en un determinado entorno que hace que solo puedan utilizar aquello que dichos frameworks les ofrezca. Para esquivar dichas dificultades, se pueden llegar a idear increíbles peripecias para realizar una simple acción por el mero hecho de que el diseño de dichas herramientas no da directamente la posibilidad de realizarlo. Puede ser una buena idea, desarrollar de forma independiente desde cero aquellos recursos que hagan posible crear aquello que se desea de una forma simple y orientada directamente a ello.

3.1 QUÉ SON LOS JUEGOS RECREATIVOS

Los juegos son una actividad típica no solamente del ser humano. Varias especies de animales enseñan a sus crías a comer, a moverse, a socializarse, a cazar, etc a través del juego. Dichas acciones, pueden verse de una forma más analítica como aquellas realizadas por distintos agentes o usuarios de un determinado tipo, que interactúan entre sí para conseguir objetivos de distinta índole entre los que puede categorizarse la victoria, la derrota, empate u otros resultados basados en puntuaciones, tanto colectivas como individuales.

Los juegos forman parte de nuestra vida cotidiana, los encontramos con frecuencia a nuestro alrededor: en el casino, las loterías, el mercado de valores, etc. Esto, desde un punto de vista más formal, puede entenderse como el entorno en el que los distintos agentes antes mencionados, se relacionan y efectúan las tareas y acciones para desarrollar dicha relación e interacción.

Hay juegos individuales y juegos colectivos. Los hay probabilísticos, más intelectuales o una mezcla de ambos. Juegos donde se juega “a ganar” y otros de tipo cooperativo. Están los crucigramas, el sudoku, el ajedrez, el cubo de Rubik, el “Go”, y un gran número de ellos con características en común que permiten clasificarlos como juegos basados u orientados a tablero.

En matemáticas hay un espacio dedicado al estudio de la teoría de juegos, no solo por sus aspectos combinatorios o lógicos, sino porque también nos sirven para explicar mejor fenómenos económicos y sociales que ocurren a diario a nuestro alrededor, y también porque este estudio nos sirve para tomar decisiones de manera más acertada.

Cuanto mejor comprendamos dicha forma de toma de decisiones, y mejoremos el proceso para llevarla a cabo, antes podremos desarrollar métodos para que las máquinas hagan lo propio de una forma mucho más eficiente.

No es casualidad que la teoría de juegos no se enseñe solamente en las facultades de matemáticas, sino que también forma parte del currículo obligatorio en academias militares y facultades de economía e informática, debido al fuerte carácter estratégico y táctico que pueden transmitir a distintas áreas del conocimiento.

3.2 ESTRATEGIA Y TÁCTICA EN LOS JUEGOS RECREATIVOS

Es debido al carácter estratégico y táctico de los juegos, a sus características, así como el entorno en el que se desarrollan (orientados a estructuras que pueden representarse mediante un tablero), lo que determina el estudio central del proyecto.

Un juego es cualquier situación en la cual un jugador, o varios de ellos buscan unas condiciones particulares o colectivas que ayuden a conseguir sus objetivos, ya sea de forma individual o en conjunto. También puede verse un juego como cualquier situación en la que compiten dos o más jugadores. El Ajedrez y el “Hundir la Flota” son buenos ejemplos, pero también lo son el duopolio y el oligopolio en los negocios si encontramos una forma de representarlos a través de matrices o estructuras gráficas adecuadas. La extensión con que un jugador alcanza sus objetivos en un juego depende del azar, de sus recursos físicos y mentales (o recursos de los que disponga un procesador) y de los de sus rivales, de las reglas del juego y de los cursos de acciones que siguen los jugadores individuales, es decir, sus estrategias.

Una estrategia es una especificación de la acción que ha de emprender un jugador en cada contingencia posible del juego, en la que los detalles de como realizar las acciones dictadas serán finalmente establecidas o llevadas a cabo según una táctica puntual, inherente al propio jugador (su habilidad, sus reflejos, capacidad de procesamiento, etc).

Para desarrollar juegos en computación, debe suponerse que, todos los jugadores son racionales, inteligentes y están bien informados. Incluso, si los jugadores compiten entre sí, se supone que cada uno de ellos conoce todo el conjunto de estrategias existentes, no sólo para él, sino también para sus rivales, y que cada jugador conoce los resultados de todas las combinaciones posibles de las estrategias. Este planteamiento puede facilitar el hecho de crear situaciones intermedias en el que un jugador controlado por la computadora pueda comportarse como un peor o mejor jugador (simulando que no conoce todas las combinaciones, estrategias, etc, aunque es posible que simplemente no pueda calcularlas y no necesite simular nada).

Si dada una situación determinada, un problema o un juego que no esté basado en un tablero, puede encontrarse la forma de expresarlo mediante un tablero o plantearlo de tal

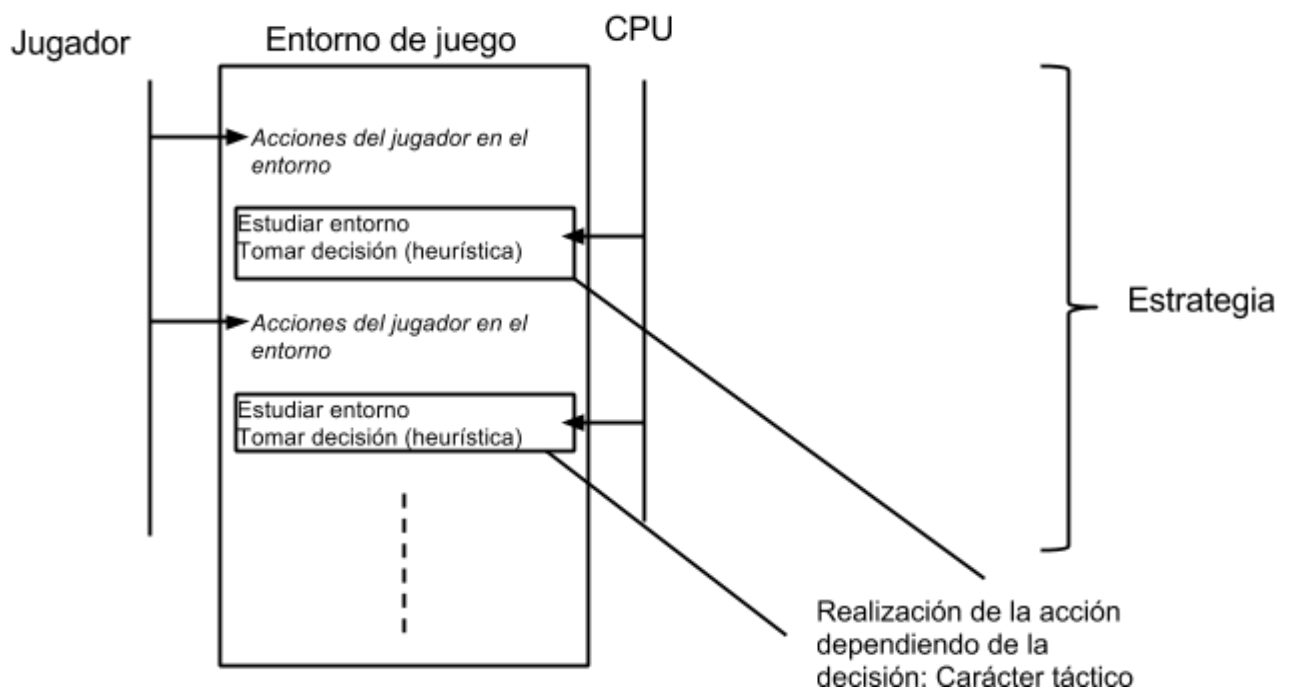
forma que permita resolverse como uno de ellos y con las características adecuadas establecidas, tal vez pudiera encontrarse la solución igual que se haría con un juego, como por ejemplo mediante árboles de búsqueda de soluciones.

El desarrollo de juegos recreativos implica en muchas ocasiones el uso de técnicas de inteligencia artificial, dependiendo de qué acciones tendrán que llevar a cabo los jugadores dependientes de la computadora.

Si ese es el caso, es prácticamente inevitable el uso de una o varias funciones heurísticas para decidir qué es lo más conveniente en cada caso.

Una función heurística es aquella que valorará cuán buena (o mala, según el enfoque) es la situación en la que se encuentra un jugador en un juego, ya sea para resolver ciertas dificultades o en competición con otros jugadores.

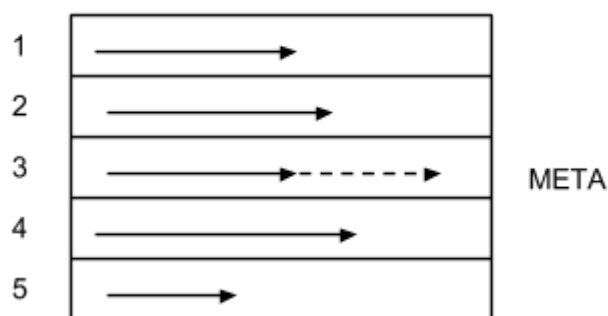
La calidad de la función heurística diseñada para el juego determinará en gran parte la estrategia a largo plazo que se sigue en el desarrollo de un juego por parte de los jugadores.



Dependiendo de la calidad de las decisiones tomadas por cada jugador (para los jugadores cpu la calidad será la de la función heurística utilizada) el entorno de juego será favorable para un jugador u otro (dentro de un conjunto de ellos), o el estado del juego será calificado de malo o bueno dependiendo de lo que se requiere en un juego en concreto sin necesidad de que esté involucrado más de un jugador. Es decir, es la propia actitud del jugador la que mejora o empeora su situación y no la de otros jugadores.

En un juego en el que compiten varios jugadores puede darse el caso de que una decisión tomada por uno de ellos basada en una determinada estrategia que está dada por el estado actual del juego, no solo mejore su situación respecto de la de los demás, sino que empeore directamente la del resto.

Juego de carreras. $E(i)$ valoración de posición de cada jugador



$E(i)$ no varía para $i \neq 3$, $E(3)$ aumenta.

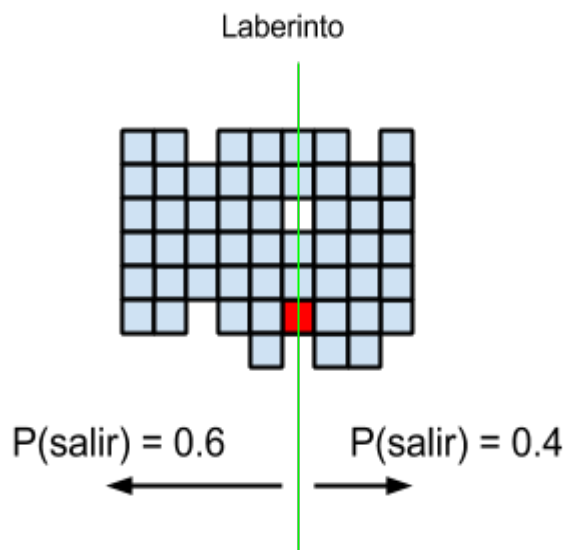
La posición del jugador(3) mejora con su último movimiento pero no influye en el estado actual de cada jugador particular.



La posición del jugador de las blancas mejora considerablemente con la decisión tomada, y además sí influye de forma directa en el estado del jugador de las negras. $E(\text{blancas})$ aumenta y $E(\text{negras})$ disminuye

Independientemente del juego del que se trate habrá que actuar en función a algo, ya sea con una función evaluadora, una función de búsqueda, etc. que lleve al jugador a la mejor situación posible a la que esté capacitado a alcanzar en su estado actual. Por ejemplo, en el

caso de un laberinto, se puede evaluar el entorno más próximo, ya que para que el jugador pueda moverse necesita explorar los caminos disponibles. El problema de estos casos es que se cae en la secuencialidad, y tampoco hay un conocimiento más profundo que permita establecer rutas más acertadas que otras, por lo que se pueden estimar rutas probables, es en los casos en los que la función heurística es probabilística (de forma explícita).



Si se conoce el nº de posiciones a cada lado de la franja verde, será más probable salir del laberinto si se va hacia el lado izquierdo, ya que hay más elementos en esa parte.

La probabilidad forma parte de una enorme cantidad de las decisiones que se toman día a día, así como las estrategias que se implantan a largo plazo para obtener determinados objetivos en los que incluso se estiman riesgos.

Un ejemplo de juego que ilustra el hecho de toma de decisiones basadas en la probabilidad es el llamado dilema de Monty Hall.

El dilema de Monty Hall es una situación en la que el presentador de un programa de televisión ofrece al concursante elegir un premio que se encuentra tras una de las tres puertas. Dos de ellas no contienen nada y una de ellas un automóvil.

El jugador elige una de ellas, supongamos la primera y el presentador (Monty) abre la puerta número tres enseñando que no hay nada en su interior. Acto seguido nos ofrece cambiar la puerta ¿qué es mejor teniendo en cuenta que el presentador sabe que hay detrás de cada puerta? La respuesta es que es mejor cambiar de puerta. Guiándonos por la estadística el presentador al abrir una puerta cerrada ha incrementado las posibilidades que tenemos de llevarnos el premio, pasamos de jugar con 33% de posibilidades al 66%

porque en realidad el presentador aumenta nuestras posibilidades al 66% si cambiamos de puerta. Si permanecemos con la elegida nuestras posibilidades se mantienen en un 33%

PROFUNDIDAD Y CONCEPCIÓN DE ESTRATEGIAS AVANZADAS

Aunque ciertas situaciones, bajo la metodología correcta y siguiendo unos procedimientos y forma de actuar bien organizados, puede requerir decisiones y esquemas de actuación bastante complejos.

Si tenemos un determinado juego, puede que tener una función heurística que valore lo que hay que hacer en un momento dado no sea suficiente, e incluso que varias de ellas no lo sean. Puede ser necesario tener una estructura jerárquica formada por un conjunto de funciones heurísticas que decidan qué función heurística es mejor usar, es decir: funciones heurísticas de funciones heurísticas. Existen motores informáticos para jugar a las Damas o al Ajedrez tan complejos que utilizan este tipo de jerarquías, y en décimas de segundos pueden obtener una situación con una valoración altísima, que incluso a expertos les llevaría un tiempo considerable alcanzar.

3.3 ESTRUCTURA Y ENTORNO DE LOS JUEGOS RECREATIVOS

Los juegos recreativos orientados a tablero necesitan de una estructura o espacio de representación de su entorno y de sus elementos.

ELEMENTOS DE LOS JUEGOS RECREATIVOS

El espacio de representación será obviamente un tablero, que no es más que un espacio de 2 o 3 dimensiones (dependiendo de los requerimientos del juego). No obstante, es raro encontrar juegos orientados a tableros y que no sean bidimensionales. Por éste hecho el proyecto se centrará en estos últimos.

Los tableros tienen que estar compuestos a su vez, de elementos que recubran su área, que comúnmente se denominan posiciones o casillas. A su vez, tanto el tablero como estas casillas tendrán propiedades determinadas que constituirán características específicas del juego y de su situación en un momento determinado.

También disponen de una serie de elementos que interactúan con los elementos anteriores: piezas movidas por los jugadores (controladores), que son los agentes que participan.

JUGADORES O AGENTES DE LOS JUEGOS RECREATIVOS

Cada juego está formado por uno o varios jugadores que interactúan entre ellos, directa o indirectamente a través de una serie de acciones realizadas sobre el tablero (y en consecuencia utilizando las casillas o posiciones) utilizando para ello las piezas asociadas a ellos. Los jugadores pueden ser humanos, procesos de la CPU, jugadores ficticios existentes bajo condiciones específicas del juego, etc. Puede hablarse de ellos como agentes que sigan un plan de operaciones concreto a lo largo del juego para conseguir unos objetivos dados.

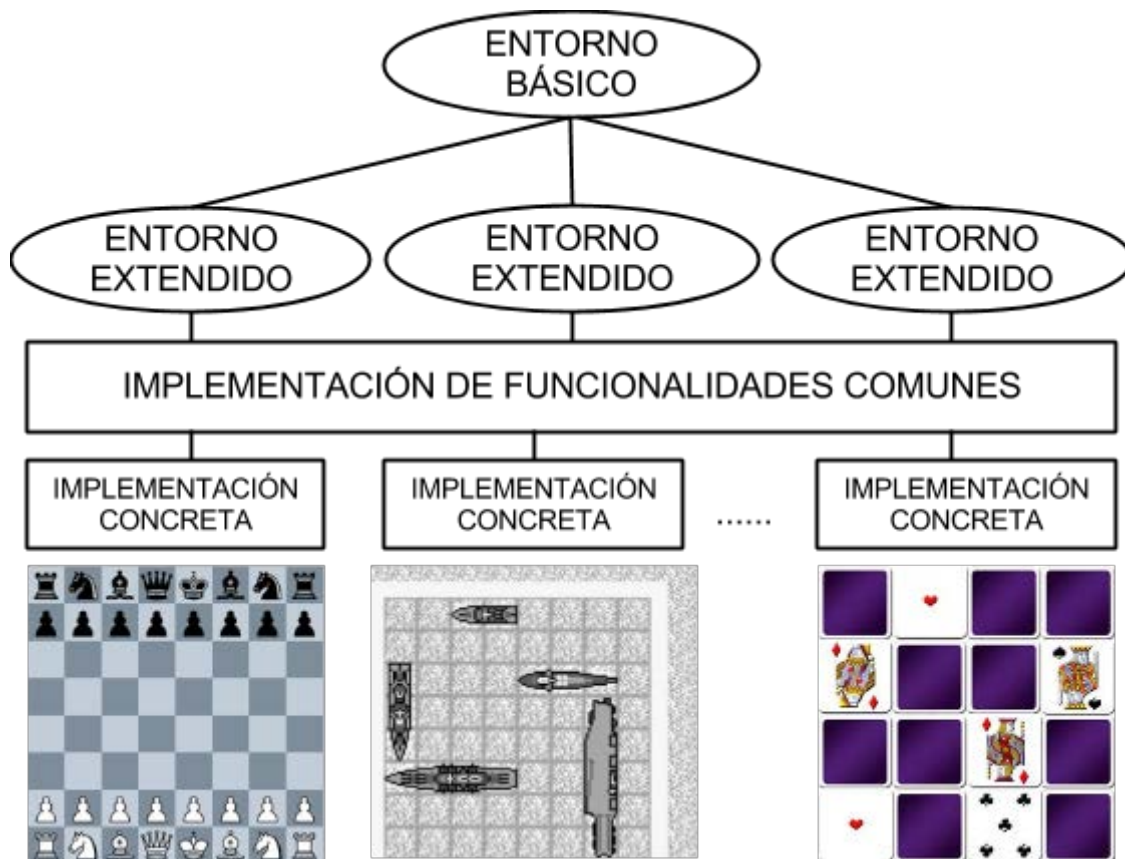
Para que esta interacción entre los jugadores y los elementos de un juego tenga lugar, tiene que haber definida una serie de reglas y condiciones que respetar y cumplir, para jugar de una forma en cierto modo equitativa y que pueda ser juzgada y valorada para obtener puntuaciones y estados determinantes del juego.

REGLAS DE LOS JUEGOS RECREATIVOS

Los juegos recreativos tendrán unas reglas definidas, que impondrán restricciones y formas de llevar a cabo las acciones del juego. Dichas reglas pueden estar establecidas en el juego de forma implícita, o incluso puede existir otro agente (en este caso no es un jugador) que regule el cumplimiento de dichas reglas, es decir, un juez.

4. ANÁLISIS DE LA APLICACIÓN

Los juegos que se tratan en el proyecto tienen todos una característica común, la base donde se desarrolla su actividad, se ha definido previamente como el tablero, un área plano (2 dimensiones) o espacial (3 dimensiones) donde ciertas piezas o elementos interactúan entre sí siendo utilizadas por jugadores externos.



Una de las razones por las cuales la programación orientada a objetos es especialmente adecuada para aplicarse al desarrollo de este tipo de juegos, es debido a que permite crear una serie de capas intermedias que vayan agrupando en cada nivel características comunes, tanto funcionales como estructurales.

La simplicidad y claridad de realizar un proyecto bajo esas directrices, permite añadir sin una dificultad demasiado elevada, componentes y módulos que hagan mucho más dinámicos los juegos, así como elementos interactivos, efectos visuales, etc.

La aplicación que se realiza para el estudio del proyecto, consta de un juego de laberintos, y un juego de sopas de letras, por lo que se realizará un análisis de requisitos y de las características de los laberintos y las sopas de letras para desarrollar la funcionalidad asociada, mediante una estructura común para extender juegos orientados a tablero

El análisis y el diseño de la aplicación muestra la esencia o concepto de cómo se realizaría para cualquier juego recreativo orientado a tablero.

Tanto el análisis, como el diseño y desarrollo de la aplicación se realizará de forma paralela para los laberintos y las sopas de letras para verificar que aunque son completamente diferentes, están basados en una estructura “padre” semejante y comparten propiedades que facilitará la realización de ambos gracias a la programación orientada a objetos.

4.2 JUEGOS DE LA APLICACIÓN

LABERINTOS

La definición más básica de un laberinto puede ser la de *una estructura formada por caminos entrelazados que tiene como objetivo dificultar el hecho de encontrar uno (o varios de ellos) que lleven a una posible salida*.

Obviamente podemos complementar y ajustar esta definición de una forma mucho más técnica y precisa, pero con lo mencionado es más que suficiente para entender el concepto.

Un laberinto no es más que un conjunto de caminos interconectados de forma más o menos compleja de tal modo que una vez dentro de él, encontrar la salida constituya un reto.

Es preciso indicar que dicha definición no está dando por hecho que siempre exista una salida, sino que el objetivo de la existencia de un laberinto es complicar su ubicación, si no existiese ninguna salida podría considerarse que dicha complicación es infinita. No se podrá salir nunca. En cuanto a la entrada, se considera que es tan solo la posición inicial y no constituirá jamás una salida (a no ser que el estudio de un problema en concreto haga que coincidan).

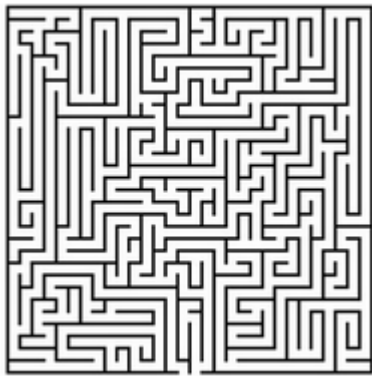
SOPAS DE LETRAS

Por otro lado, una sopa de letras es un pasatiempo que consiste en una cuadrícula u otra forma geométrica rellena con diferentes letras y sin sentido aparente. El juego consiste en descubrir un número determinado de palabras enlazando estas letras de forma horizontal, vertical o diagonal y en cualquier sentido, tanto de derecha a izquierda como de izquierda a derecha, y tanto de arriba a abajo, como de abajo a arriba.

4.3 FORMAS DEL ÁREA DE JUEGO

MÚLTIPLES FORMAS DE LABERINTOS

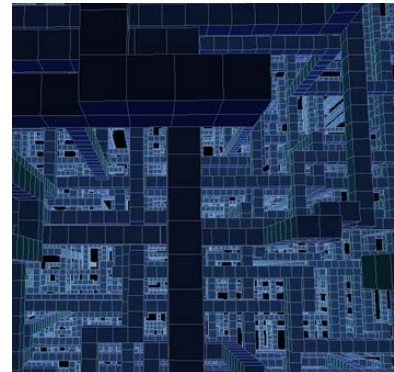
Los laberintos pueden tener infinidad de formas, cuadradas, circulares, polígonos regulares, o contornos completamente irregulares. Los laberintos también pueden darse en espacios dimensionales distintos, dos dimensiones (el plano), tres dimensiones (el espacio) o si nos adentrásemos en espacios de más dimensiones estaríamos incurriendo al estudio teórico de los mismos mediante los objetos matemáticos necesarios para ello.



Laberinto cuadrado en dos dimensiones



Laberinto en una superficie esférica.



Laberinto de tres dimensiones.

En nuestro caso nos centraremos en los laberintos rectangulares de dos dimensiones.

SIMPLICIDAD DE FORMAS DE LAS SOPAS DE LETRAS

La forma de una sopa de letras, es en principio más sencilla. Una sopa de letras puede definirse como una matriz de dos dimensiones cuyos elementos son letras, y en la cual la combinación consecutiva de algunas de esas letras forman palabras que constituyen la solución o soluciones de la misma de la sopa de letras. No obstante, podrían definirse formas extravagantes donde albergar las sopas de letras.

4.4 CARACTERÍSTICAS Y ESTRUCTURA

ESTRUCTURA DE CAMINOS DE LOS LABERINTOS

Los laberintos, al estar formados por caminos, necesariamente tienen que tener algún tipo de contorno que los distinga y separe del resto. Normalmente suelen ser paredes, pero la separación podría ser de cualquier tipo. Nos referiremos a ellos como separadores, como lo opuesto a los caminos.

Un caso extremo a contemplar podría ser aquel en el que no existan separadores, y por lo tanto no haya un “diferenciador” de los caminos. En ese caso, podrían definirse los separadores de otra forma, pero en realidad no sería necesario porque en ese caso la existencia de una solución es segura. Llamamos al laberinto que no tiene ningún obstáculo, *laberinto trivial*.

Al laberinto en el cual no se pueda alcanzar la salida lo denominamos *laberinto imposible*.

Para poder representar un laberinto de forma analítica, se hace necesario el hecho de definir la unidad mínima que lo conforma, podríamos decir que dicha unidad mínima será una posición.

A su vez, una posición por sí misma tendrá que ser "autosuficiente" para saber si forma parte de un camino del laberinto (o de varios, si hablamos de una intersección), o si por otro lado forma parte de un separador (o varios, por la misma razón que antes).

Respectivamente podemos hablar de *huecos* u *obstáculos*.

La diferencia entre un hueco y un obstáculo es simple, los huecos pueden ser ocupados o "pisados" mientras que los obstáculos no.

Para contemplar todos los elementos existentes en un laberinto hay que señalar que una posición hueco podrán contener a su vez al individuo que está dentro del laberinto.

Las posiciones además, podrá contener una entrada o una salida, entendiendo como tales, posiciones huecas con una propiedad especial (la de entrar o salir).

De este modo podemos entender un laberinto como un espacio totalmente recubierto por un conjunto de posiciones. Como nuestro caso se contempla dos dimensiones rectangulares, un laberinto será un área rectangular totalmente recubierta de posiciones.

También podríamos hablar de aquel área rectangular que esté totalmente recubierto por caminos y separadores.

Puede comprobarse entonces, que los separadores son estructuras similares a los caminos, pero formadas por un tipo de posición diferente.

Las posiciones de entrada y salida siempre formarán parte de los caminos y no de los separadores, debido a que éstas sólo pueden estar contenidas en posiciones hueco.

Para referirnos al individuo o "ente inteligente" que intentará salir de un laberinto, usaremos el término *jugador*.

ESTRUCTURA MATRICIAL DE LAS SOPAS DE LETRAS

Las sopas de letras en cambio, incluso estando también formada por elementos posición semejantes a los laberintos, no contiene ni caminos ni separadores, al menos "físicamente" hablando. El comportamiento, algoritmos de juego y búsqueda serán completamente independientes, pero la estructura donde se desarrolla todo tendrá un antecesor común.

4.5 REPRESENTACIÓN DE LOS ELEMENTOS

ELEMENTOS REPRESENTATIVOS DE UN LABERINTO

Para representar un laberinto, se utiliza una matriz de elementos que tendrán ciertos valores que hagan referencia a huecos, obstáculos, al jugador o a las entradas y salidas. Los elementos de la matriz representan a las posiciones del laberinto.

El hecho de utilizar una matriz para la representación tiene una clara ventaja, la posición de un elemento en la matriz determina exactamente la misma posición que ocupa en el laberinto, al contrario de lo que ocurriría si se utilizasen otras estructuras de datos como listas, tablas unidimensionales, vectores ordenados, etc

Sea L un laberinto, y M la matriz de valores enteros que lo representa.

M tendrá las mismas dimensiones que el laberinto, dos enteros mayores que 0 que definan la estructura rectangular del laberinto, así que la matriz M será de la forma

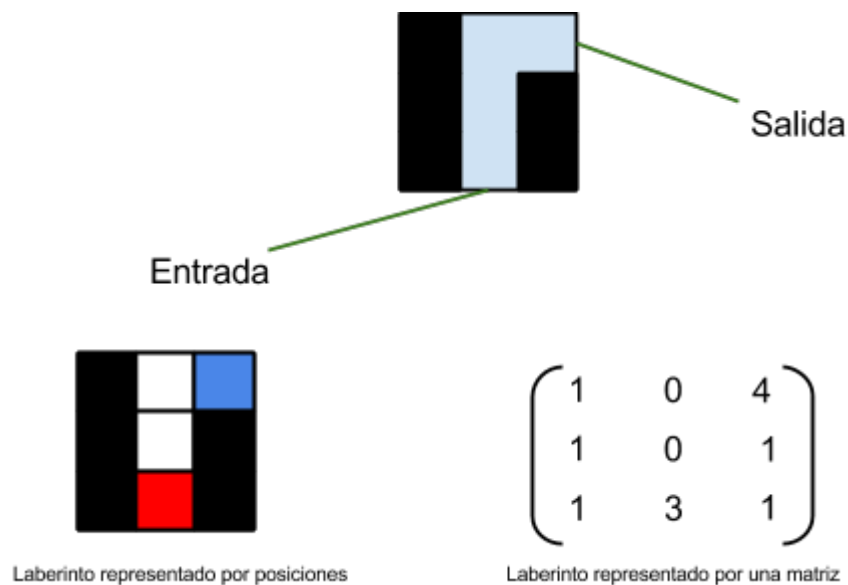
$$M_{i,j} = \begin{pmatrix} m_{1,1} & \dots & m_{1,j} \\ \vdots & \ddots & \vdots \\ m_{i,1} & \dots & m_{i,j} \end{pmatrix}$$

Cada posición del laberinto puede representarse con un número entero en la matriz cuyo valor determinará si dicha posición es un hueco, un obstáculo u otro elemento.

Si asignamos a partir del 0 valores consecutivos a los distintos tipos de posición de laberinto obtenemos la siguiente tabla:

Posición	Valor
Hueco	0
Obstáculo	1
Jugador	2
Entrada	3
Salida	4

Con esto podemos representar laberintos mediante matrices de enteros, por ejemplo:



En el ejemplo se trata de una situación inicial, y la posición del jugador coincide con la posición de entrada, en este caso seguimos el siguiente criterio, en caso de coincidencia: el mayor valor tiene prioridad sobre el menor en cuanto a aparecer en la misma posición de una matriz.

Para hacer referencia a una posición P , basta con referenciar a un elemento de la matriz a través de sus coordenadas.

$$P(i,j) = M_{i,j}$$

Para representar un camino C , basta con usar un vector o matriz de una fila cuyos elementos son las posiciones ordenadas con el mismo criterio en que se recorran en el laberinto para ir de una posición a otra, sin repetir la misma posición. Es decir, sin bucles.

$$C = \{ P(i), P(i+1), \dots, P(i+n) \}$$

Con el ejemplo de la imagen anterior, tendríamos que:

$$M_{3,1} = 3 \Rightarrow P(3,1) \text{ es una entrada}$$

$C(P(3,1), P(1,3))$ es un camino, y además es solución.

Una condición necesaria para que en un laberinto se pueda encontrar la salida es que al menos haya 2 posiciones hueco (entrada y salida).

Sean x e y las dimensiones de un laberinto L , entonces una condición necesaria para alcanzar la salida es

$$n^{\circ} \text{ obstáculos}(L) \leq x \cdot y - 2$$

ELEMENTOS REPRESENTATIVOS DE UNA SOPA DE LETRAS

La representación de una sopa de letras es más sencilla que la de un laberinto, ya que puede representarse con una matriz $N \times M$ en la que todos los elementos son caracteres o letras del alfabeto.

Las posiciones de una sopa de letras se referencian de la misma forma que se hace con los laberintos, ya que es una forma de referenciar matrices de cualquier tipo.

Aparte de dicha matriz existe una lista de posibles palabras-solución.

Dicha lista podría considerarse como un superconjunto que contiene al espacio de soluciones, esto es así porque el usuario podría definir situaciones en las que palabras que estén en la lista, no estén en el tablero.



La palabra QEZ es solución

4.6 ESTADOS DE LOS JUEGOS

ESTADOS AL RECORRER UN LABERINTO

Con la matriz M podemos expresar cualquier estado de un laberinto o una sopa de letras, ya que tenemos ubicados los elementos que están en él y definen su situación por completo en un instante determinado.

Por lo tanto, en un laberinto, tenemos que un estado *E* es la configuración de la información en un momento dado.

Un estado dependerá del valor que tengan los elementos de la matriz M, y a partir de ello podemos definir tres tipos de estado:

- Estado inicial *E_o* : Es la situación inicial de un laberinto, justo cuando el jugador aparece en la entrada, es decir, la posición de entrada y la posición del jugador coinciden.
- Estado intermedio *E_i* : Es aquel en la que la posición del jugador no coincide ni con la posición de salida ni con la posición de entrada, o lo que es lo mismo, es aquella situación en la que el jugador está buscando la posición de salida.
- Estado final *E_f* : Se ha logrado encontrar la salida, y en consecuencia la posición del jugador coincide con la posición de salida, o bien se han recorrido todos los caminos del laberinto sin poder encontrar la salida.

Una vez definidos los elementos de un laberinto cuando se encuentra en diferentes estados, pasamos a estudiar cómo se realizan las transiciones que producen cambios entre los estados, para construir posteriormente el diagrama de estados.

BI-ESTADOS DE LAS SOPAS DE LETRAS

En el caso de las sopas de letras, se definen dos estados:

- Resuelta: Se ha procesado la lista de palabras-solución.
- No resuelta: Se ha modificado la lista de palabras-solución o bien se acaba de crear la sopa de letras.

Una sopa de letras cuyo contenido incluye una serie de palabras que puedan estar o no en el diccionario, no da información acerca de si se ha procesado su resolución y por lo tanto de si ha sido resuelta, debido a que la sopa ha podido ser modificada después del proceso. En las sopas de letras no se contemplan los estados intermedios.

4.7 MOVIMIENTOS Y FUNCIONALIDAD

MOVIMIENTOS EN UN LABERINTO

Los elementos encargados de realizar transiciones en los estados en un laberinto son los movimientos, que denotamos con *mov*.

Los movimientos son los responsables de que el jugador se pueda mover por las posiciones hueco de un laberinto, es decir, un movimiento “intercambia” al jugador entre las diferentes posiciones del laberinto, o lo que es lo mismo, intercambia el valor 2 por todos aquellos elementos de M que así lo permitan. Así que para definir la estructura de un movimiento hay que estudiar que tipo de cambios de posición pueden permitirse al jugador.

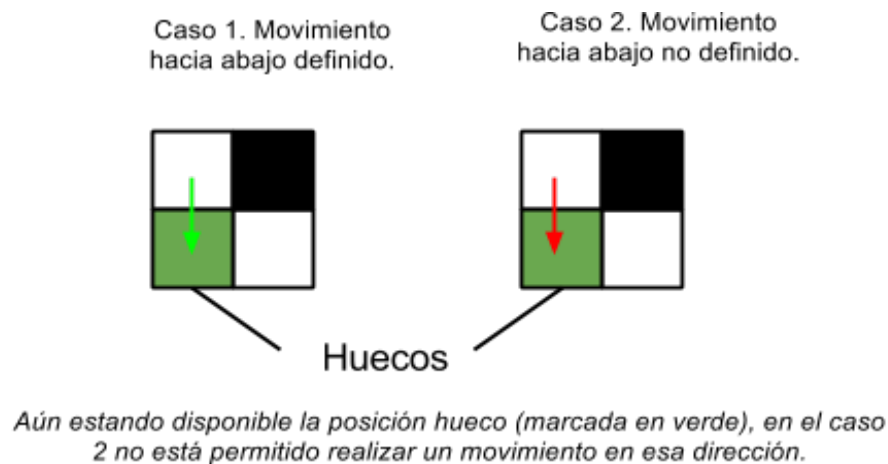
DEPENDENCIA DE LOS MOVIMIENTOS DE UN JUGADOR

En un principio, un jugador podrá moverse libremente por un laberinto siempre que no haya un obstáculo por donde quiera pasar, así, el jugador se moverá por donde no haya obstáculos, del modo en que quiera hasta encontrar la solución.

Pues bien, esto es sólo así hasta cierto punto, un jugador podrá moverse por donde no haya obstáculos, sí, pero sólo si es posible para él realizar ese movimiento, independientemente del lugar en que se encuentre, así como otro laberinto o un espacio completamente libre de obstáculos, siempre que una dirección este libre para moverse, también tendrá que tener definido el movimiento en dicha dirección.

De este modo se puede afirmar que un movimiento es en primera instancia, dependiente del jugador, y después dependiente de su entorno.

En términos de funcionamiento, daría igual si un movimiento depende del jugador o del propio laberinto aludiendo a la posibilidad de realizar un movimiento en una posición hueco.

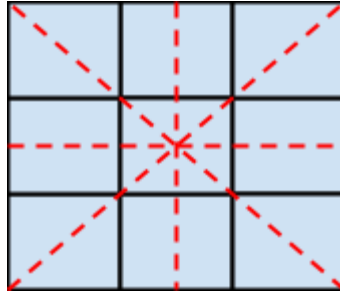


DIRECCIONES EN UN LABERINTO

Para poder definir completamente los movimientos, primero tenemos que clasificar los distintos tipos de direcciones que se contemplan en un laberinto L .

La respuesta viene dada en su mayor parte por la estructura que almacena su información, hablamos de la matriz de valores M .

Dada una posición determinada, las direcciones posibles que pueden trazarse para un laberinto coinciden con las posiciones adyacentes, como se ilustra a continuación:



Como se puede observar, las líneas remarcadas en rojo representan todas las direcciones posibles que pueden existir en un laberinto.

Esto es en el sentido “clásico” del concepto de dirección, que dice:

El rumbo que sigue un cuerpo en su movimiento y la línea sobre la que se mueve un punto se conocen como dirección.

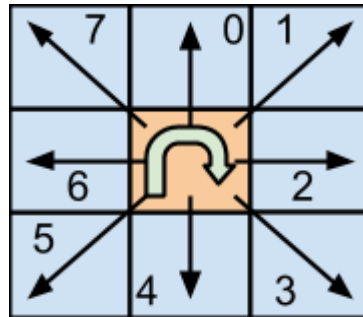
En este caso, nuestro concepto de dirección variará en algunos aspectos. Para el caso que nos ocupa, una dirección le dirá al movimiento hacia dónde tiene que mover al jugador. Luego, definimos una dirección d en un laberinto L , como *la propiedad de un movimiento que le indica al mismo hacia qué posición adyacente debe mover al jugador al aplicarse dicho movimiento sobre la posición que lo contiene.*

Con esta definición, pasamos de un máximo de 4 direcciones a 8. (Cada sentido de la “dirección antigua” se convierte ahora en una “dirección nueva”)

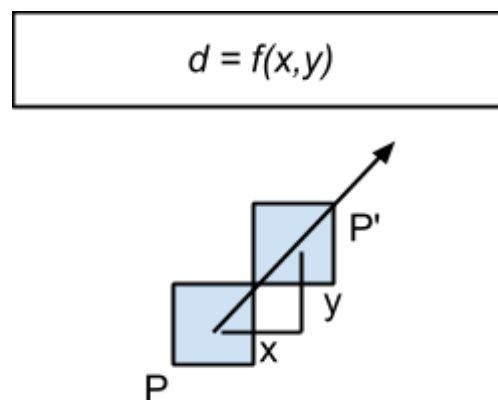
Así, una dirección ya no está definida como una línea y se convierte en una especie de apuntador. (Se pueden considerar los puntos cardinales)

Para identificar las direcciones, las enumeramos de 0 a 7 siguiendo el sentido de las agujas del reloj. Así, cada dirección apunta a cada una de las posiciones adyacentes dada una posición central que se toma como referencia.

Gráficamente es trivial y se puede ver en la siguiente imagen:

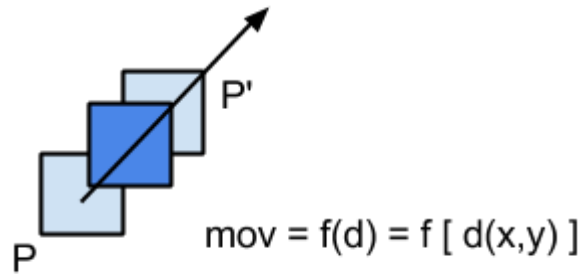


Como ya se ha comentado, las direcciones le dicen a un movimiento hacia “donde” debe mover al jugador, es decir, a qué posición cambiará una dada. Como se está trabajando en un plano, para cambiar de una posición a otra, se requieren de 2 componentes. Así, tenemos que una dirección d es una función que depende de dos componentes en el laberinto L .



Tenemos entonces que un movimiento mov tiene como propiedad una dirección, y dicho movimiento se aplicará sobre una posición para obtener otra. Es claro que mov depende de d , ya que es quién determina por completo como se aplica sobre una posición.

$$\text{mov} = f(d)$$

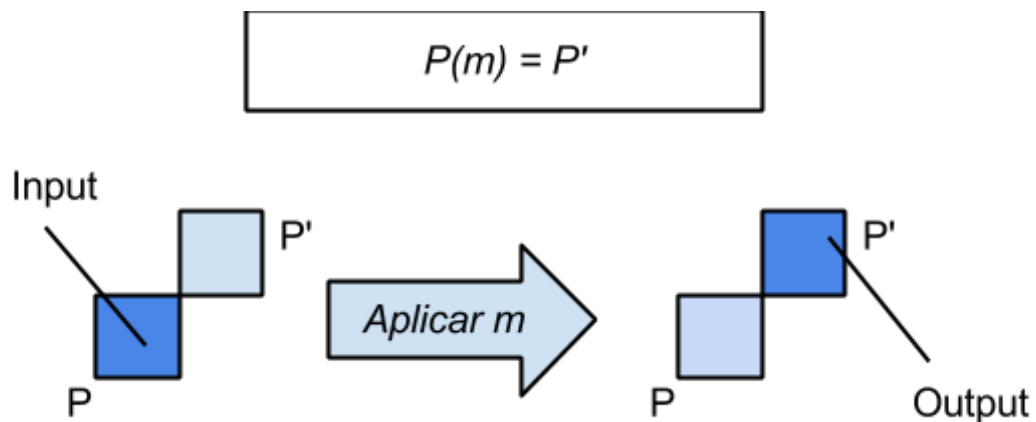


El movimiento es un vector que solo existe mientras se produce el cambio de posición, es el cambio en sí.

Para ser más concretos, un movimiento puede depender directamente de las coordenadas de la dirección, o bien comportarse como un vector cartesiano que modificará los valores de la matriz, lo que en el laberinto se traduce a que desplazará al jugador de una posición a otra.

$$\text{mov}(d) = \vec{m}(x,y) \text{ siendo } m \text{ un vector}$$

Dada una posición P y un movimiento, la nueva posición P' será el resultado de aplicar el vector a la posición dada:



Los movimientos sólo pueden aplicar el cambio de una posición a otra adyacente, con lo cual la distancia entre P y P' siempre será de una unidad, lo que definirá el cambio como tal no será nunca la distancia, sino la dirección. De hecho, en los laberintos que se tratan se considera la misma distancia entre una posición y cualquiera de las adyacentes. Esto no es así en la geometría euclídea, en la que la distancia en diagonal es mayor.

Se están considerando laberintos homogéneos, en los que sus caminos son “estables” y los movimientos que puedan producirse sobre ellos son equivalentes, de unidad en unidad, pero incluso si hablamos de un peso diferente de las posiciones, y un coste al ir de una a otra, la aplicación del movimiento seguiría manteniéndose como si se tratará solo de una separación de una unidad, debido a la adyacencia, y el peso solo cambiaría el hecho de medir distancias o longitud de caminos, y no de moverse en sí.

La adyacencia, hace que el dominio de x e y (coordenadas de una dirección) sea discreto y se defina como sigue:

$$x, y \in [-1, 1] \text{ con dominio discreto.}$$

x e y podrán tomar los valores $-1, 0$ y 1 , y el conjunto de todas sus combinaciones definen el dominio de las direcciones.

Como sabemos todos los posibles valores de las coordenadas de una dirección, podemos definir absolutamente cualquier movimiento.

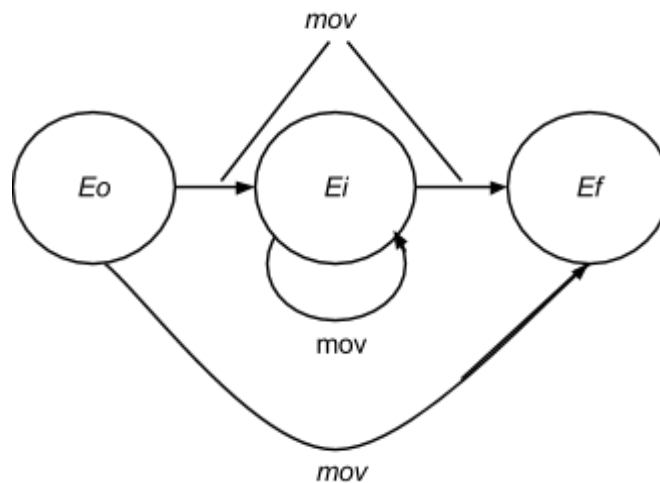
En la siguiente tabla se recogen los pares de valores que cubren todo el dominio posible de las direcciones de un laberinto.

Dirección	X	Y	
0	0	-1	↑
1	1	-1	↗
2	1	0	→
3	1	1	↘
4	0	1	↓
5	-1	1	↖
6	-1	0	←
7	-1	-1	↙

La insistencia en esto viene dada por el hecho de que una vez definido un movimiento con su dirección correspondiente, las operaciones de coordenadas en las posiciones no se contemplan, se consigue la transparencia de esto, simplemente tenemos en cuenta el movimiento y la posición, *aplicamos*, y como resultado obtenemos la nueva posición.

Una vez aplicado un movimiento sobre una posición, ya se puede hablar de transiciones o cambios de estado.

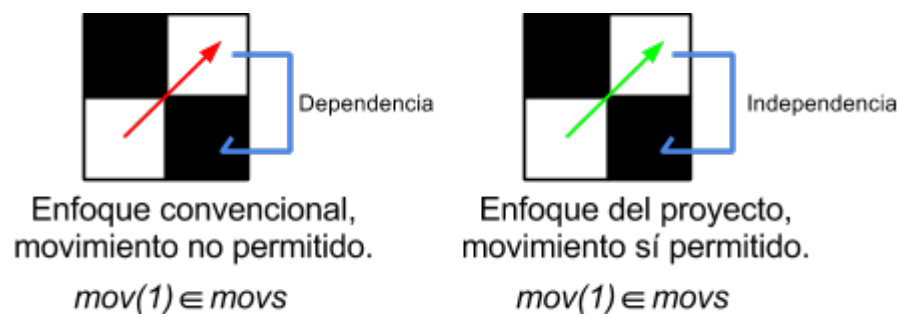
Flujograma de un laberinto



INDEPENDENCIA DE LOS MOVIMIENTOS EN UN LABERINTO

La forma más convencional de moverse por un laberinto, suele ser hacia arriba, abajo, derecha e izquierda, y en algunos casos en diagonal si está libre la casilla correspondiente de al lado.

En nuestro caso, todo movimiento será independiente de los demás. Por ejemplo, si en un momento dado no es posible realizar un movimiento a la derecha porque en esa posición existe un obstáculo, pero en la diagonal superior derecha hay un hueco, entonces sí se podrá realizar dicho movimiento, al igual que lo realizaría un alfil en un tablero de ajedrez. Este hecho condiciona en gran medida la posibilidad de nuevos estados.



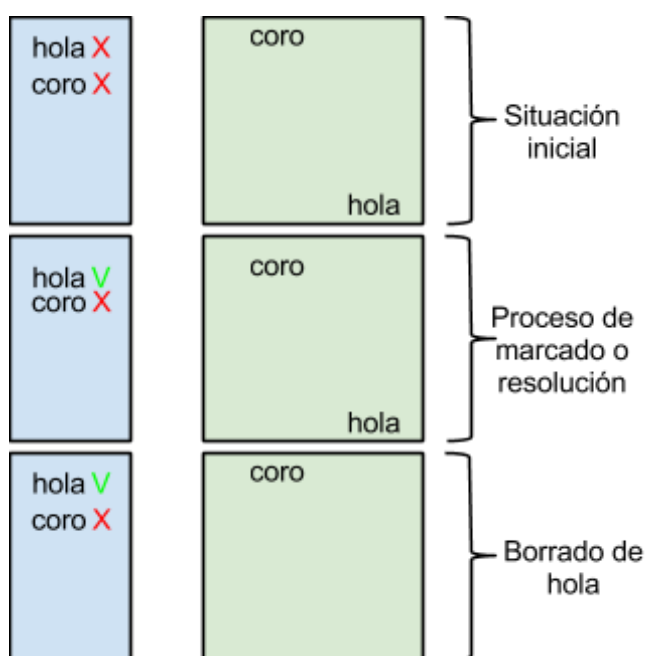
En ambos casos el jugador sí tiene definido el movimiento diagonal.

FUNCIONALIDAD ESTÁTICA EN LAS SOPAS DE LETRAS

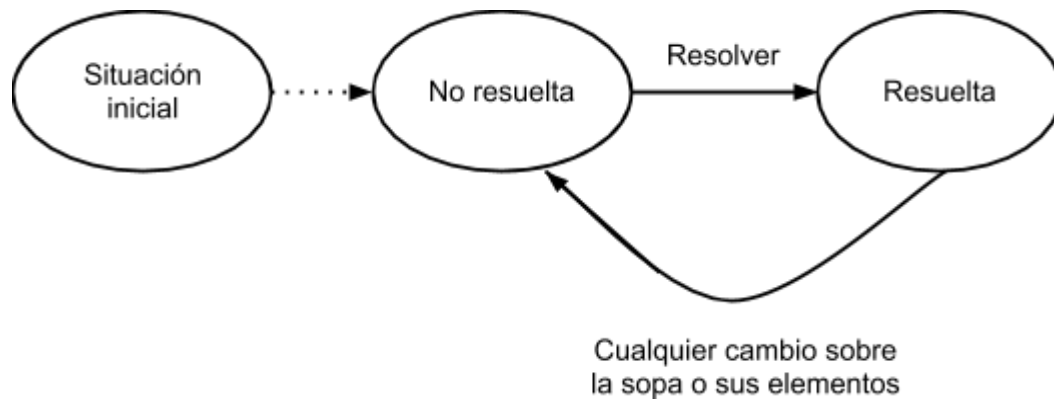
En una sopa de letras no hay movimiento como tal, en lugar de eso hay un proceso secuencial y ordenado de resolución.

Para evitar la “dualidad” entre resuelta y no resuelta que se mencionó en los estados de una sopa de letras, habría que almacenar la información referente a cada proceso de marcado, siendo un estado resuelto aquel en el que después de haber realizado un proceso no se hayan realizado modificaciones sobre los elementos.

En la siguiente figura se ilustra un ejemplo de esta situación:



En realidad, esto puede evitarse desmarcando del diccionario o bien aquellas palabras que se han modificado o eliminado de la sopa de letras, o bien desmarcando todo el diccionario cuando se realice algún cambio. Sin embargo, el marcado, se utiliza tan sólo como parte visual para el usuario, con objeto de enseñar la ubicación de las palabras del diccionario en la sopa de letras. Con esto, siempre que el usuario decida ver la resolución de una sopa de letras, o lo que es lo mismo, ver la ubicación de dichas palabras, se realizará una nueva búsqueda y un marcado posterior, de esta forma siempre se mostrará la información más actualizada posible sin necesidad de utilizar algunos mecanismos de control. Esto puede permitirse ya que tanto los algoritmos de búsqueda como de marcado son computacionalmente muy ligeros.



4.8 DIFICULTAD

VALORACIÓN DE LA DIFICULTAD EN LABERINTOS

Cuando hablamos de dificultad de un laberinto, nos referimos de forma natural a cuánto cuesta poder encontrar una salida una vez dentro de él. Esto puede estar condicionado por varios factores, como pueden ser sus dimensiones, cuán entrelazados están sus caminos, la distancia que hay entre la entrada y la salida o la cantidad de obstáculos que hay dentro. Ésos sólo son algunos ejemplos de factores que pueden influir en que un laberinto sea más difícil de resolver.

Sin embargo, dichos factores no determinan que necesariamente deba ser difícil, ni siquiera cuando varios de ellos se dan en conjunto.

Puede existir un laberinto con una cantidad astronómica de caminos y obstáculos, en el cual la entrada esté justo al lado de la salida y su solución sea trivial.

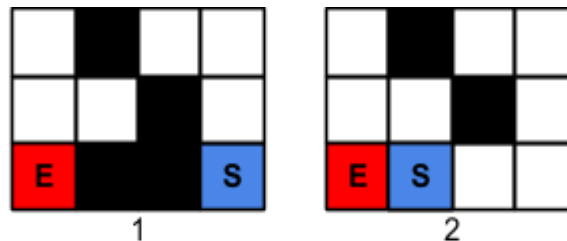
Entonces, para definir de la mejor forma la dificultad de un laberinto, podemos hablar de la dificultad como la unión de todos los factores que impliquen posible dificultad en su mayor grado posible en un laberinto.

Sea la dificultad *dif* y *X* el conjunto de todos los elementos que puedan ocasionar dificultad, siendo $h(E)$ el valor de la función heurística en un momento dado:

$$\begin{aligned}
 dif &= fD1 + fD2 + fD3 \dots + fDN \\
 &\text{con } fDi \text{ factor de dificultad} \\
 dif &= h(E) = fD1.. + fDN
 \end{aligned}$$

La dificultad, desde un punto de vista computacional, tal vez sea una de las características más difíciles de evaluar, sin recorrer el laberinto.

Podríamos hablar de ella como una función heurística h evaluando una situación dada E .



Si suponemos como factores de dificultad la distancia entre salida y entrada y el número de obstáculos, el laberinto 1 es más difícil que el 2.

$$\begin{aligned} \text{dif}(\text{lab } 1) &= h(1) = \text{núm. de obst.} + \text{distancia entr.} - \text{salida} = 4 + 3 = 7 \\ \text{dif}(\text{lab } 2) &= h(2) = \text{núm. de obst.} + \text{distancia entr.} - \text{salida} = 2 + 1 = 3 \\ h(1) &> h(2) \end{aligned}$$

PSEUDO-VALORACIÓN EN SOPAS DE LETRAS

En cuanto a una forma de medir la dificultad en las sopas de letras, es un tema que compete incluso a la lingüística del lenguaje/s utilizados.

Se necesitaría disponer de una función h que dada una palabra p y una posición x tuviese como salida una valoración que indicase la dificultad de encontrar la palabra p en una sopa de letra S .

O	X	X	X	A
X	I	X	I	X
X	X	C	X	X
X	E	X	E	X
R	X	X	X	R

$$h(\text{"recio"}, p(4,4)) > h(\text{"recia"}, p(4,0))$$

Suponemos que la palabra leída al revés (en ambas direcciones) supone mayor dificultad. La dificultad total será la suma de las dificultades de cada palabra.

4.9 SOLUCIONES

SALIDAS Y SOLUCIONES DE UN LABERINTO

Las soluciones de un laberinto son los caminos que nos llevan desde la entrada a la salida. Así que definir una solución es definir un camino cuya primera posición es la entrada, y la última posición la salida (sin repetir ninguna claro, aunque esto es una condición de camino y no de la solución en sí).

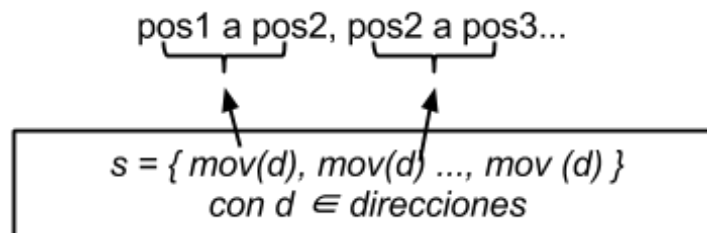
Una solución s será un determinado camino del laberinto que cumpla:

$$s = C_i = \{ entrada, pos_1, pos_2, \dots, salida \}$$

Un laberinto puede tener una, varias o ninguna solución. Cada una de ellas tendrá un número determinado de posiciones recorridas.

Cabe señalar que una solución puede definirse como un vector de movimientos y sería igualmente válido, de hecho el número de elementos que contendrían los vectores de movimientos solución coincidiría con el número de posiciones en un camino solución menos una unidad.

Se expresa de la siguiente forma:



Se considerará que una solución es mejor que otra si contiene un número menor de elementos.

Sean s_1, s_2 soluciones de L , entonces s_1 es mejor que s_2 si $n^\circ posiciones(s_1) < n^\circ posiciones(s_2)$

La solución o soluciones óptimas de un laberinto son aquellos que contienen el menor número de elementos.

Algo importante a señalar: En este caso las soluciones ya no solo dependen de cómo es un laberinto, ahora también dependen de los movimientos definidos para un jugador en un laberinto, esto es debido a la *dependencia de los movimientos de un jugador*, tratado más arriba en este documento.



$$P_i [\text{mov}(2)] = P_{i+1}$$

Caso 1. Movimientos definidos: { mov(2), mov(7) }



En este caso se alcanza la salida aplicando el mov(2) a las posiciones.

$$s = \{ p0, p1, p2, p3, p4, p5 \}$$



$$P_i [\text{mov}(7)] = \emptyset$$

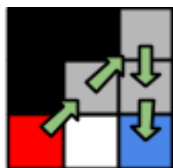
Caso 2. Movimientos definidos: { mov(7) }



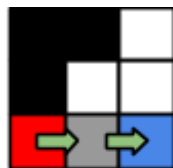
En este caso sólo se podría aplicar mov(7), pero ni siquiera hay posiciones que lo permitan.

Sin solución.

La dependencia de los movimientos de un jugador también influye sobre la calidad de las soluciones.



Movs: { mov(1), mov(4) }



Movs: { mov(1), mov(4), mov(2) }

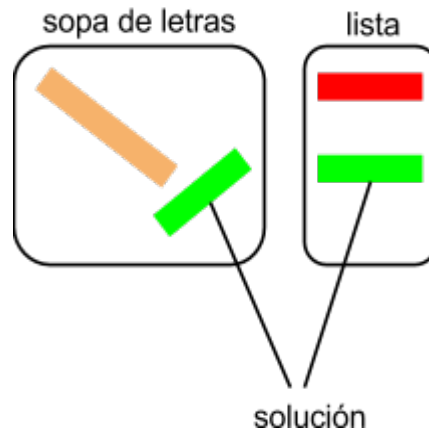


En un mismo laberinto, mov(2) determina una mejor solución, en este caso la óptima.

SOLUCIONES DE UNA SOPA DE LETRAS

Para las sopas de letras el concepto de solución se convierte en algo trivial:

Toda palabra colocada en cualquier posición, dirección y sentido que se encuentre dentro de la sopa de letras y a su vez en un listado de palabras externo a la sopa.



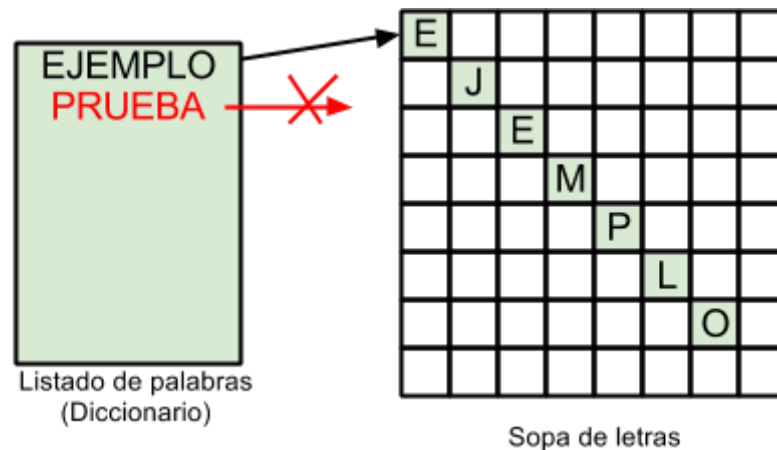
La lista que contiene todas las palabras entre las cuales se encuentran las palabras-solución se denomina diccionario.

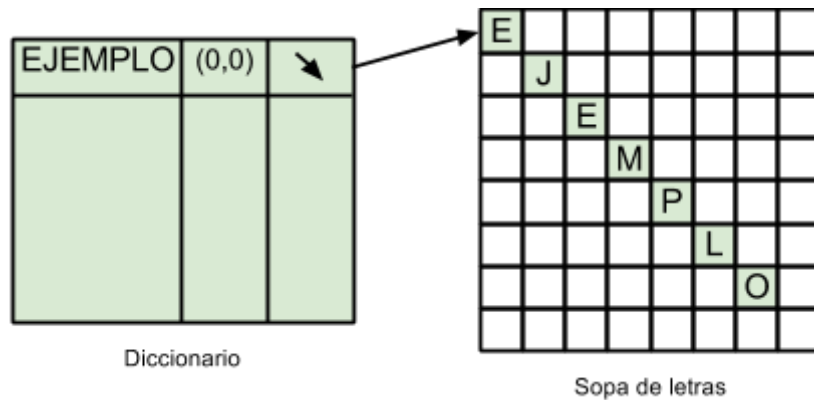
Espacio soluciones

idiccionar

Los elementos de la lista y que están en la sopa de letras se llamarán: *palabras-solución*.

Las palabras-solución pueden estar ubicadas en cualquier posición y orientadas en cualquier dirección.





4.10 DENSIDAD DE UN LABERINTO

Podemos definir, de forma independiente dos tipos de densidad en un laberinto:

1. Densidad de ocupación: Se define como densidad de ocupación a *la cantidad de obstáculos por unidad de posición*. Es la relación entre la cantidad de posiciones de un laberinto y el número de obstáculos.

$$DO = n/p$$

siendo n el número de obstáculos y p el número de posiciones.

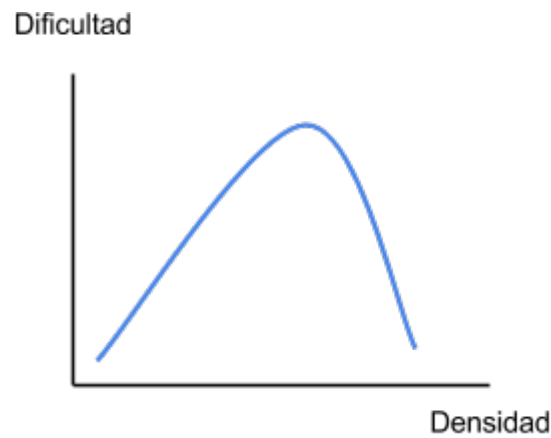
2. Densidad de movilidad: Se define como densidad de movilidad a *la libertad de moverse por un laberinto*. Es la relación entre los movimientos permitidos y los movimientos existentes.

$$DM = mP/mT$$

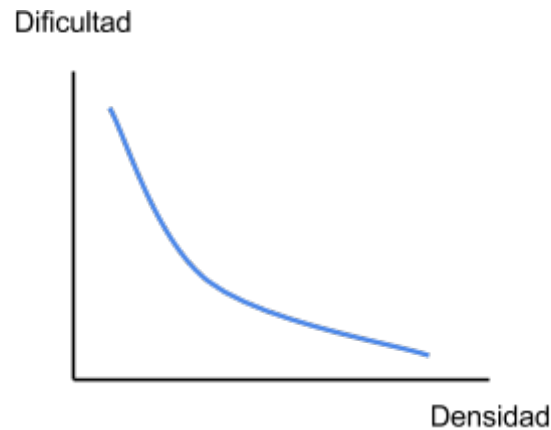
siendo mP el nº de movimientos permitidos y mT el nº movimientos totales.

La densidad de ocupación mide cuán lleno está un laberinto, y cuanto mayor sea dicho valor será más costoso moverse por el mismo ya que habrá más obstáculos. De forma análoga, la densidad de movilidad nos dice con qué facilidad podemos movernos dentro de un laberinto, y cuanto más limitada sea, al no poder realizar determinados movimientos se creará una “sensación” de obstáculos incluso donde no los hay.

En cuanto a la densidad de ocupación, su aumento puede implicar que la dificultad también aumente, aunque no lo hará siempre, de hecho llegará un momento en que la dificultad disminuya ya que una alta densidad de ocupación puede implicar un camino sin alternativas ni ramificaciones.



Sin embargo, la densidad de movilidad se comporta de forma diferente a la hora de afectar a la dificultad. Cuanto más baja sea la densidad de movilidad, mayor dificultad implica en cualquier caso.



5. DISEÑO E IMPLEMENTACIÓN

5.1 DISEÑO DE ALGORITMOS

La aplicación está prevista para poder generar los laberintos manual y automáticamente. Resultaría tedioso crear un laberinto completo cada vez que se necesitase o quisiese obtener un nuevo.

5.1.1 GENERACIÓN DE LABERINTOS

La generación de laberintos está diseñada con una serie de métodos y técnicas que permiten crear laberintos en el plano de forma que pueda ajustarse qué parámetros serán automatizados y en qué medida.

El proceso está constituido por multitud de parámetros aleatorios, lo que teóricamente hace que puedan generarse *todos los laberintos rectangulares posibles de dos dimensiones*, ya que el sistema puede no ser permisivo, es decir, pueden configurarse aquellas características que impidan la aleatoriedad total.

Para ello, se definirán varios elementos que se encargarán de parametrizar aquellos valores que hagan que un laberinto pueda generarse de forma aleatoria en el grado que se determine.

NÚMEROS ALEATORIOS EN RANGOS DEFINIDOS

Para la generación de laberintos será necesaria la capacidad de poder generar números aleatorios en un determinado rango de números enteros positivos. Esto será utilizado para establecer los valores que definirán la estructura del laberinto de una forma indeterminada.

*Sean n y m dos números enteros positivos que cumplen $n \leq m$,
entonces:*

$$a(n,m) = k$$

*siendo k un número entero positivo aleatorio entre n y m .
Por convenio definimos:*

$$a(n) = a(0,n)$$

PASOS PARA GENERAR UN LABERINTO

La secuencia de pasos que se sigue para la generación de un laberinto es la siguiente:

1. Generación de las dimensiones del laberinto
2. Generación de las posiciones de entrada y de salida
3. Forzar la generación de soluciones (opcional)
4. Generación de todos los obstáculos del laberinto
5. Generación de los movimientos disponibles en el laberinto

A continuación se muestra a detalle cada uno de ellos y cómo se retroalimentan.

1. GENERACIÓN DE LAS DIMENSIONES DE UN LABERINTO

Para generar las dimensiones de un laberinto, se necesita de un mecanismo que obtenga dos números enteros positivos que definan un rectángulo válido.

Por lo tanto, para generar las dimensiones de un laberinto bastará con generar dos números aleatorios en un rango que se determine, por defecto la dimensión mínima para considerar un laberinto será de tres unidades.

$$longitud = dimX = a(3,n)$$

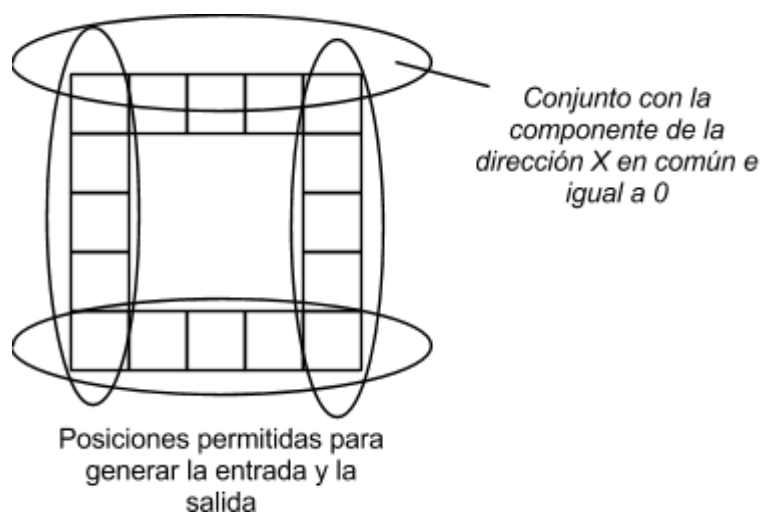
$$altura = dimY = a(3,n)$$

2. GENERACIÓN DE LAS POSICIONES DE ENTRADA Y SALIDA

Una vez con las dimensiones que definen el rectángulo que contendrá las posiciones de un laberinto, antes de comenzar a generar el contenido en bruto de posiciones u obstáculos, se hace preciso ubicar donde estará la entrada y la salida del laberinto.

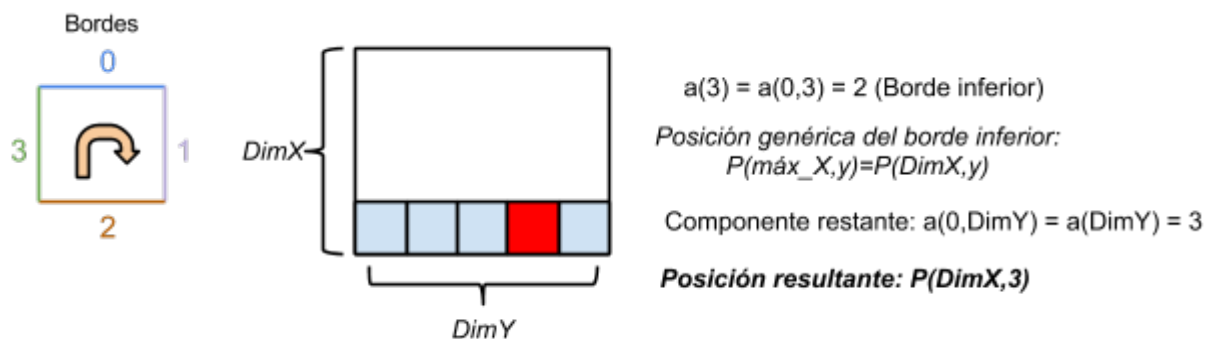
Tanto la entrada como la salida se ubicarán en los extremos del laberinto, es decir, que las posiciones que contengan la entrada y la salida estarán en algún borde. Dichos bordes de un laberinto son conjuntos de posiciones con una componente de la dirección en común y además dicha componente será 0 o el máximo de la dimensión X o Y.

Estas condiciones las cumplen cuatro conjuntos de posiciones, con lo cual definimos cuatro bordes, y allí será donde estén ubicadas tanto la entrada como la salida.



El algoritmo encargado de ubicar la entrada y la salida generará dos números aleatorios entre 0 y 3, este número se usará como índice para identificar a los bordes asignándoles dicho índice en el sentido de las agujas del reloj.

Con el borde ubicado, se tendrá definida una componente de la posición que quiere obtenerse, para obtener la otra se generará otro número aleatorio entre 0 y el valor máximo para la dimensión de ese borde.



3. FORZAR LA GENERACIÓN DE SOLUCIONES

Cuando se genera un laberinto de forma aleatoria, hay una probabilidad alta de que dicho laberinto no tenga solución debido a que dicha aleatoriedad puede provocar que se cierren los posibles caminos que lleven a la salida, al no controlar que casillas serán huecos u obstáculos.

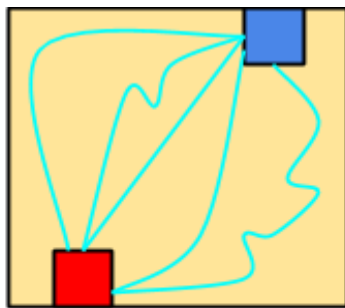
Por lo tanto, opcionalmente, se puede contar con el hecho de siempre que se desee generar un laberinto, éste tenga solución. Esto se traduce al hecho de generar un camino pseudo-aleatorio que atraviese el laberinto y a su vez pueda propiciar la ruptura de obstáculos y generar nuevos caminos.

Para hallar un método que genera dichos caminos analizamos la estructura del laberinto:

En primer lugar tenemos el laberinto con la entrada y la salida definida.

Esto vislumbra que se podrían conectar ambas posiciones con todos aquellos caminos que se necesitase, sean cuales sean, y ya se tendría una solución.

La cuestión es dar con la forma de obtener tan solo uno y emular un comportamiento casi-aleatorio.



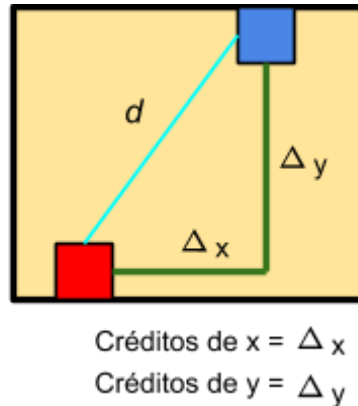
Cualquier secuencia de posiciones resolvería el problema. La verdadera cuestión es como obtener dicho camino de forma aleatoria en cualquier laberinto.

En segundo lugar, hay que considerar el hecho de limitar la longitud de dicho camino, si este aspecto no se considerase, se correría el riesgo de generar continuamente laberintos casi vacíos o con un número muy bajo de obstáculos. Hay que aclarar que el hecho de forzar la existencias de soluciones no debería afectar a otras características del laberinto.

Teniendo en cuenta los aspectos mencionados, se define el siguiente algoritmo para forzar la existencia de solución en un laberinto.

ALGORITMO DE LOS CRÉDITOS

1. Ubicación de las posiciones de entrada y salida. **P_e** y **P_s**
2. Asignar un valor llamado crédito de X a la variación entre la componente X de la posición de entrada y la componente X de la posición de salida.
3. Asignar un valor llamado crédito de Y a la variación entre la componente Y de la posición de entrada y la componente Y de la posición de salida.



4. Realizar un movimiento hasta llegar a la posición de salida. Dicho movimiento podrá tomar tres direcciones a lo sumo:

- Dirección con solo desplazamiento en la componente X y que reduzca la distancia en X entre las posiciones.
- Dirección con solo desplazamiento en la componente Y y que reduzca la distancia en Y entre las posiciones.
- Dirección con desplazamiento en X e Y y que reduzca la distancia entre las posiciones.

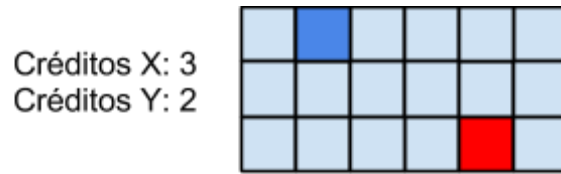
Cada movimiento restará un crédito a las direcciones involucradas en el movimiento, cuando los créditos de una componente hayan llegado a 0, la distancia en esa componente entre las posiciones será 0, por lo tanto si ambos créditos están a 0 se habrá alcanzado la posición de salida.

Para determinar qué movimiento hacer en cada momento se generará un número aleatorio entre 0 y 2, respectivamente cada número hará referencia a las direcciones mencionadas.

- Si el número generado es 0, se restará 1 a los créditos X y se desplazará una posición en la componente X hacia la componente X de la salida.
- Si el número generado es 1, se restará 1 a los créditos Y y se desplazará una posición en la componente Y hacia la componente Y de la salida.
- Si el número generado es 2, se restará 1 a los créditos X y a los créditos Y y se desplazará una posición en diagonal hacia la salida.

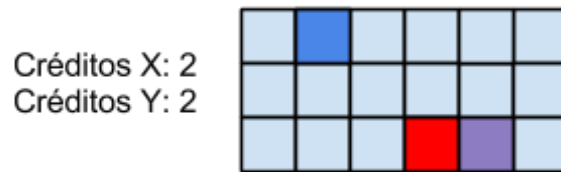
- Si los créditos de una componente llegan a 0, sólo se realizarán movimientos en la componente restante.

A continuación se muestra un ejemplo de un camino autogenerado:



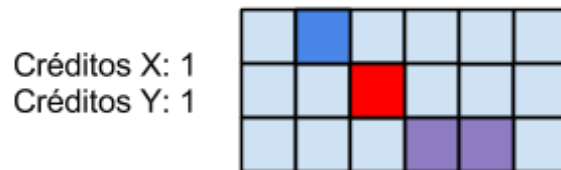
$$a(0,2) = 0$$

Se realiza movimiento en la componente X y se restó 1 a los créditos de X.



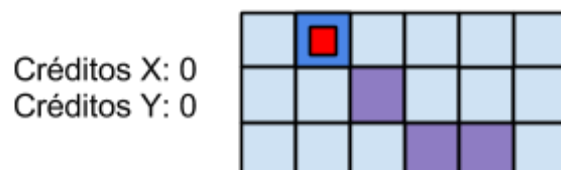
$$a(0,2) = 2$$

Se realiza movimiento en ambas componentes y se restó 1 a los créditos de las dos componentes.



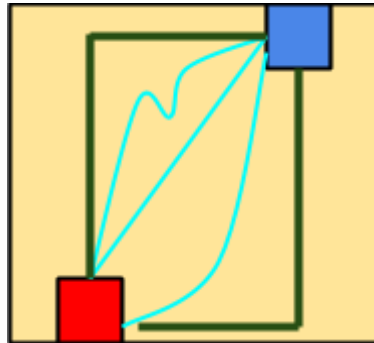
$$a(0,2) = 2$$

Se realiza movimiento en ambas componentes y se restó 1 a los créditos de las dos componentes.



El problema de este método radica en el hecho de que el conjunto de todos los caminos generados para forzar la existencia de solución no son todos los que pueden existir, debido al hecho de que éste método controla la longitud para evitar que un laberinto pueda contener un elevado número de obstáculos aun cuando se desee forzar la solución.

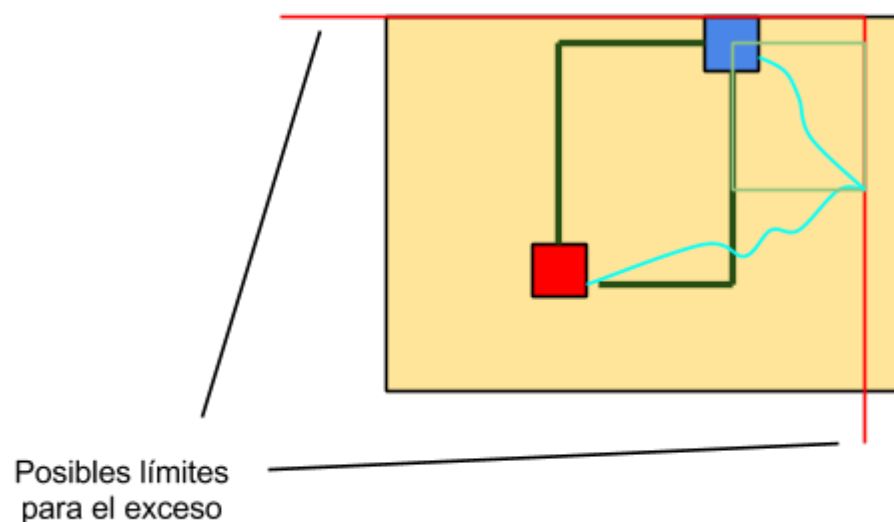
El algoritmo de los créditos genera todos lo caminos de la forma:



No se generarán caminos fuera del rectángulo delimitado por los caminos verdes, por lo cual parece innecesario disponer de un algoritmo especializado que se encargue de ello, ya que sería fácil generar dicho tipo de caminos mediante un método sencillo.

No obstante, un cambio en el algoritmo anterior, puede generar caminos que salgan del rectángulo, para ello, se puede considerar un exceso controlado en las componentes horizontal y vertical.

Permitiendo que en cada componente se pueda alejar una determinada cantidad de posiciones de los límites del rectángulo, una vez alcanzado el tope de dicho exceso se podrá volver a aplicar de nuevo el algoritmo a partir de dicha posición volviendo a establecer un exceso o no, ya que se podría volver al inconveniente de reservar demasiadas posiciones sin obstáculos y dejar “desnudo” el laberinto.



4. GENERACIÓN DE LOS OBSTÁCULOS DE UN LABERINTO

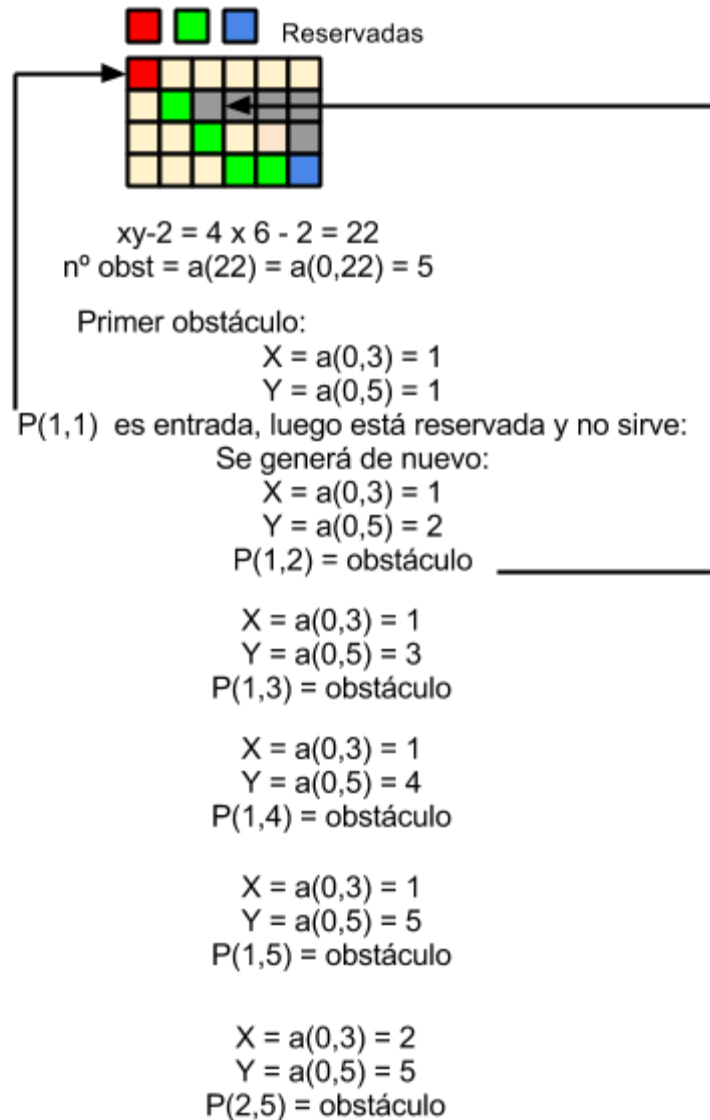
Para generar los obstáculos de un laberinto, han de generarse uno a uno mediante la generación aleatoria de sus componentes, establecidas en el rango permitido por las dimensiones del laberinto.

Tenemos que el número de obstáculos de un laberinto es como máximo $xy-2$, siendo x e y las dimensiones del laberinto. El número de obstáculos que contendrá un laberinto auto-generado puede obtenerse generando un número aleatorio entre 0 y $xy-2$ con $a(0,xy-2)$.

Una vez con dicho número, se generarán dos componentes aleatorias por cada obstáculo, si dichas componentes apuntasen a una posición que contenga la entrada, la salida o bien una posición reservada para un camino que fuerce solución, se desechará y se volverá a generar una posición hasta que se obtenga una libre.

El proceso se puede descomponer en los siguientes pasos:

1. Generar un número aleatorio de obstáculos.
2. Por cada uno de ellos:
 - a. Generar la componente X.
 - b. Generar la componente Y.
 - c. Si la posición resultante a la que apuntan X e Y está reservada, repetir este paso hasta que la posición resultante no esté reservada.



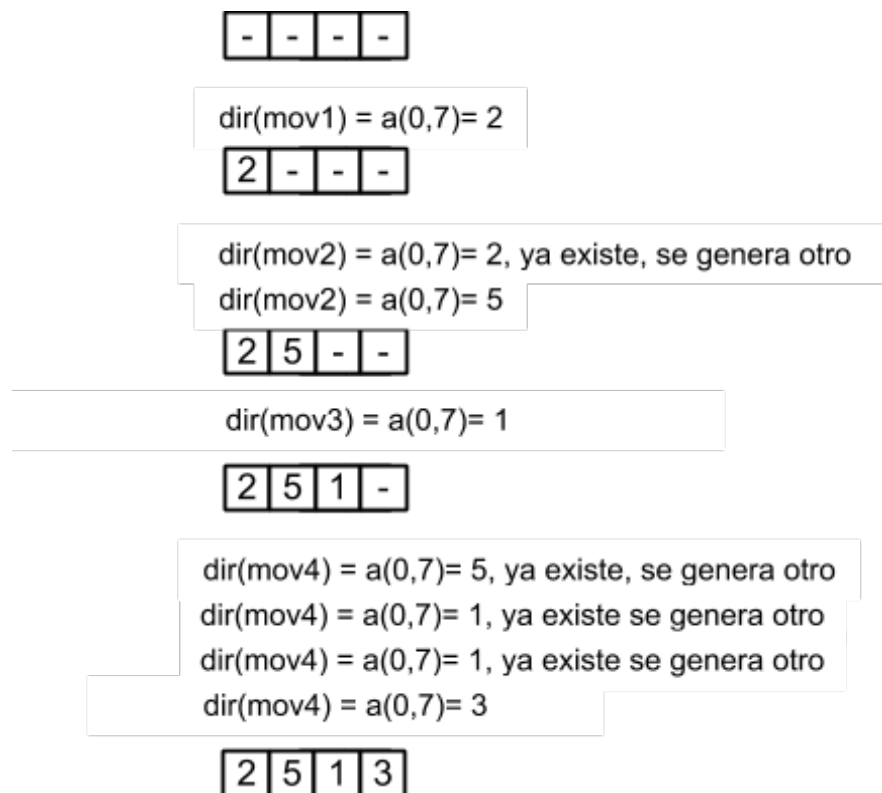
Por lo tanto, si se genera una dirección que ya se ha generado antes, se vuelve a generar otra, y así consecutivamente con cada uno de ellos hasta haber generado todos ellos. Se puede ver a continuación un ejemplo de su aplicación:

1. Se genera el número total de movimientos.

$$a(0,7)=4$$

El laberinto tendrá definidos 4 movimientos en total.

2. Se generan las direcciones de los movimientos que contendrá el laberinto.



Mediante estos procedimientos para generar laberintos, se deberían poder obtener todos los laberintos de dos dimensiones rectangulares.

Es importante indicar que las componentes no-conexas de un laberinto realmente pueden definir laberintos no rectangulares ya que no forman parte funcional del mismo.



Ejemplos de laberintos generados

5.1.2 RESOLUCIÓN DE LABERINTOS

La resolución de los laberintos se basa en el algoritmo de vuelta-atrás o backtracking. Cualquier camino, por muy enrevesado que sea o muy oculto que esté, puede contener una solución, por este motivo se recurre a la técnica de backtracking y no se plantea el diseño de una función heurística (que estaría muy relacionada con la *dificultad de un laberinto* explicada anteriormente).

Para llevar a cabo la resolución utilizando backtracking, hay que recurrir a las técnicas explicadas para ir explorando el laberinto hasta dar con una solución (si es que existe). Por lo tanto, para encontrar las soluciones de un laberinto habrá que recorrer de forma ordenada los caminos del laberinto utilizando los movimientos que se han definido para poder dar con la posible salida.

Un aspecto importante a destacar, es la diferencia entre la forma de actuar mientras se genera el laberinto a la forma de actuar mientras se resuelve. Durante la generación hay que tener un conocimiento total sobre todo aquello que se tiene en un momento dado. Sin embargo, durante la resolución se establece un conocimiento inicial nulo acerca del laberinto, es decir, será la exploración la que nos dé el conocimiento sobre él mismo.

Por ejemplo, durante la generación de determinados componentes de un laberinto, se tiene acceso a dónde está la posición de salida, cosa que no sucede en la resolución hasta que la exploración haya llevado a ella.

ALGORITMOS PARA ENCONTRAR LAS SOLUCIONES.

El algoritmo de backtracking o algoritmo de vuelta atrás es un algoritmo recursivo que permite explorar todos los caminos disponibles de forma ordenada (según algún criterio) para encontrar la solución a un problema.

El esquema genérico es:

Procedimiento Backtracking:

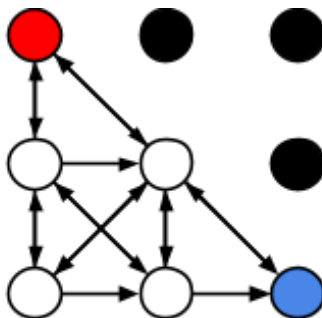
Mientras la estructura de datos sea explorable:

- Por cada operador aplicable del listado operadores:
- Aplicar operador
 - Si es solución entonces
 - Se añade la solución (*Puede comprobarse si se desea añadir, verificando si es mejor que las ya obtenidas*)
 - en otro caso
 - Se profundiza un nivel
 - Procedimiento Backtracking

Se irá explorando el laberinto, a medida que se vayan encontrando soluciones, se irán almacenando siempre y cuando las soluciones encontradas sean mejores que las que se han encontrado anteriormente. Progresivamente se irán encontrando mejores soluciones hasta dar con la mejor, a partir de la cual no se añadirá ninguna más (aunque la exploración continúe).

Los laberintos estudiados, con estructura matricial, conceptualmente también son grafos, y resulta conveniente analizarlos desde esa perspectiva para determinar qué método de búsqueda o exploración es más conveniente en promedio.

Un laberinto puede representarse con un grafo dirigido, en el cual los nodos representan las posiciones del tablero, las aristas dirigidas hacia un nodo representan si el jugador tiene definido un movimiento en esa dirección y además el mismo es aplicable, y los nodos obstáculos quedan inconexos con el resto del laberinto.



Laberinto representado por un grafo.
No tiene definido el movimiento hacia la izquierda en horizontal.

A la hora de explorar un grafo con el algoritmo de backtracking se puede formar un árbol de exploración de las posiciones.

Dicho árbol permite recorridos en profundidad y en amplitud o anchura. Se estudian ambos para comprobar cual se comporta de forma más eficiente en promedio para los laberintos.

Búsqueda en profundidad:

La búsqueda en profundidad es un algoritmo de resolución y exploración de grafos que permite recorrer todos los nodos de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

En concreto, para un laberinto, por cada posición que se explora, se obtienen todos los posibles movimientos que pueden realizarse en ella (nodos hermanos), y por cada uno de ellos se vuelve a aplicar el mismo procedimiento hasta profundizar al máximo (en el caso de hallar solución obviamente no seguiría por ese camino y no sería la máxima profundidad).

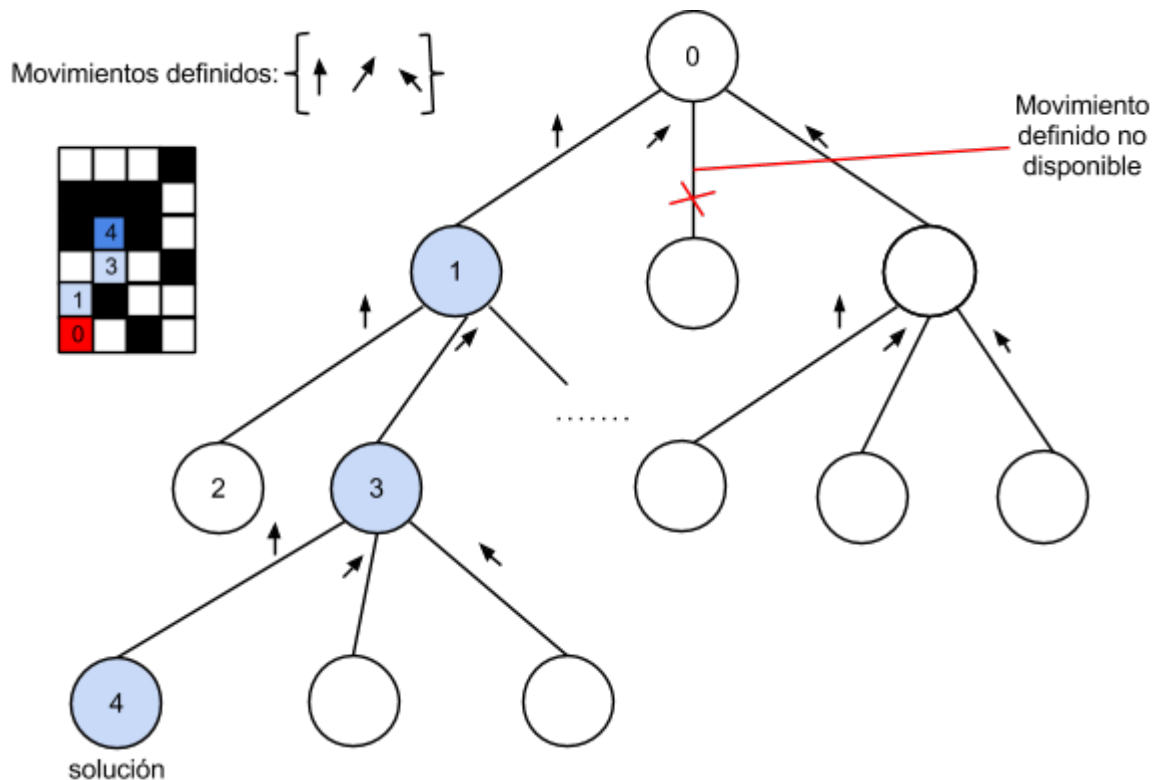
Como el laberinto impone condiciones de límite, es decir, restringe la profundidad de las soluciones, se trata de una búsqueda en profundidad iterativa. Dichas restricciones pueden venir dadas porque el número de posiciones del camino de exploración sea mayor que el número de posiciones transitables o vacías (en cuanto a complejidad, podría estudiarse el número de posiciones de áreas conexas del laberinto, aunque esto implicaría otras búsquedas), porque se haya encontrado un ciclo, o porque ya exista una solución con un número menor de elementos que el camino de exploración actual.

Además, con este algoritmo sólo es necesario almacenar las posiciones pertenecientes al camino que se está explorando en ese momento.

Complejidad del algoritmo de búsqueda en profundidad:

$$O(b.d)$$

siendo d la profundidad del árbol asociado al laberinto.



Se explora en profundidad hasta el camino {0,1,2} , como 2 es hoja y no es solución luego se vuelve hasta 1, se explora en profundidad hasta 4 y se añade como solución {0,1,3}. Se vuelve hasta 3 y así sigue explorándose el árbol.

Ejemplo de algoritmo de búsqueda en profundidad de un laberinto.

Búsqueda en amplitud:

La búsqueda en amplitud o anchura es un algoritmo para recorrer o buscar elementos en un grafo de manera uniforme. Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo, que se corresponde a la posición de entrada del laberinto) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol (limitando por supuesto aquellos caminos que acaben en solución antes de llegar a las hojas).

Expandir los nodos de forma uniforme garantiza encontrar la mejor solución de un problema de costo uniforme antes que ninguna, de manera que apenas se encuentre una solución, puede devolverse, porque será la mejor.

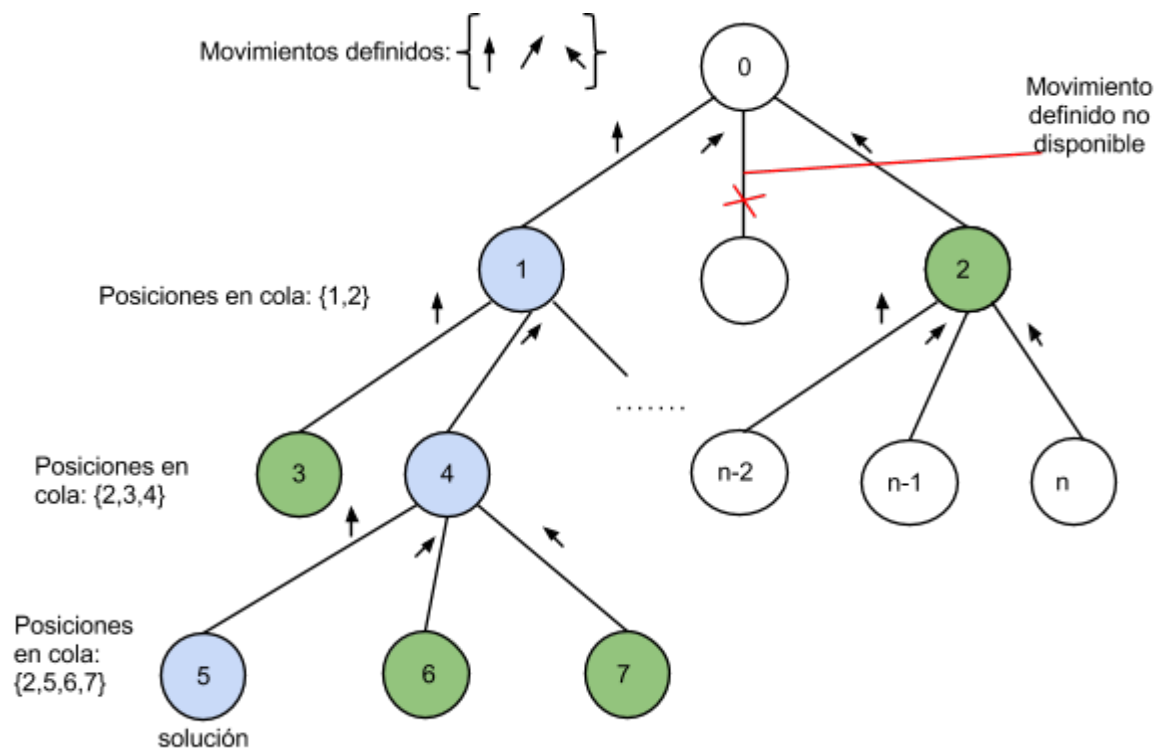
El problema principal de la búsqueda en amplitud es el alto orden de complejidad computacional, que hace que, de no mantenerse muy limitada la altura del árbol asociado a

un laberinto, crezcan rápidamente los requerimientos de memoria y se vuelva completamente inviable.

Complejidad del algoritmo de búsqueda en amplitud:

$$C = \frac{i^{n+1} - 1}{i - 1}$$

siendo i la altura del árbol, que puede crecer de forma indefinida. n hace referencia a los nodos hijos resultantes en cada nivel, es decir, al orden del árbol.



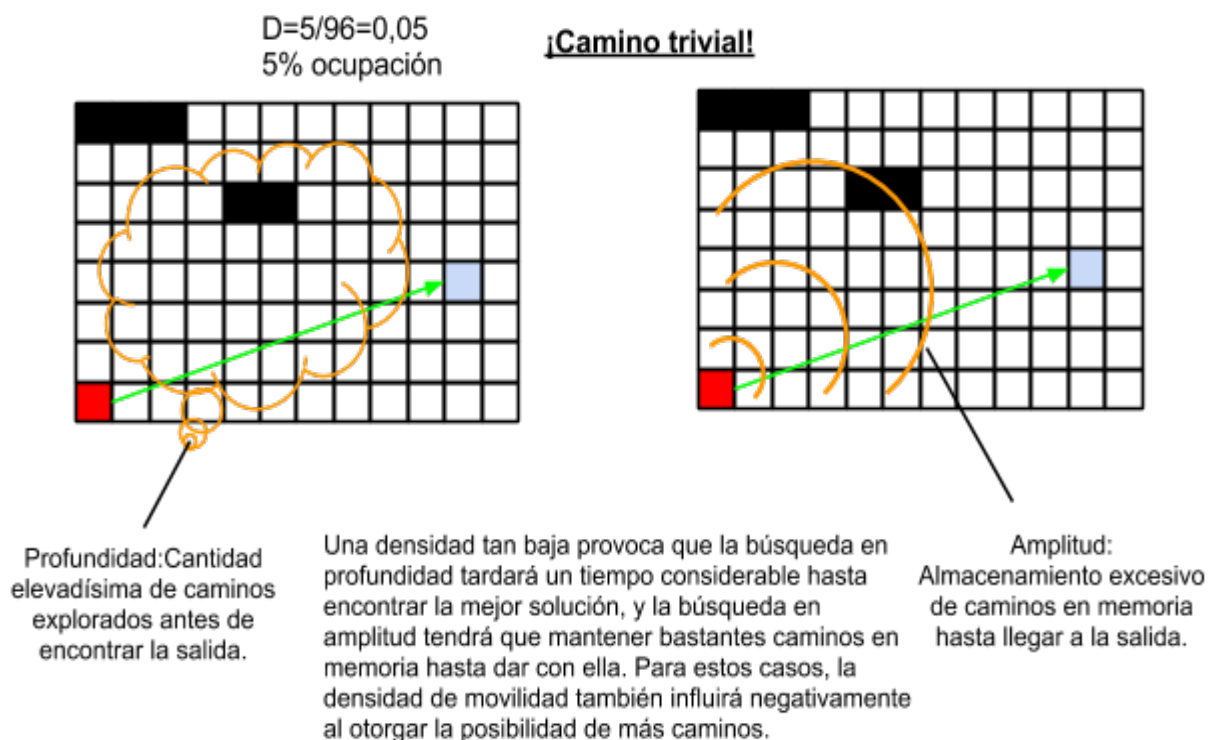
Se obtienen las posiciones adyacentes con los movimientos correspondientes, obteniéndose la posición 1 y 2, se extrae 1 y se encolan o añaden por el final sus adyacentes 3 y 4. Se extrae 2 y se añaden por el final n-2, n-1 y n. Se sigue explorando el árbol. Una vez se haya llegado a los hijos o adyacentes de 1, se da el caso de la figura.

Ejemplo de algoritmo de búsqueda en amplitud de un laberinto.

A priori, parece más adecuado utilizar el algoritmo de búsqueda en profundidad para laberintos con dimensiones elevadas y el algoritmo de búsqueda en anchura para

laberintos “pequeños”, no obstante, para estos últimos, cualquier algoritmo encontraría una solución rápidamente, con lo que basta determinar cual trabajará mejor en promedio con todos los laberintos que puedan generarse.

Atendiendo también a la densidad de ocupación, cabría esperar cambios en el comportamiento esperado, ya que por ejemplo, un laberinto con una densidad de ocupación casi nula, cuya solución para un humano fuese trivial (una línea recta incluso), podría provocar problemas de rendimiento para la búsqueda en profundidad encontrarse una solución óptima, así como la búsqueda en amplitud provocaría demasiados recursos de memoria.



5.1.3 GENERACIÓN DE SOPAS DE LETRAS

RELLENADO Y GENERACIÓN DE CONTENIDOS

Cuando se genera la estructura de una nueva sopa de letras, cabe la necesidad de rellenar de forma automática con contenido no repetitivo (el mismo elemento en toda la sopa), para evitar que sea el propio usuario el que tenga que introducir manualmente el contenido de toda la sopa (aparte de las palabras que contenga que no formen parte del relleno).

Para ello, se necesita un algoritmo que rellena de forma aleatoria todas las posiciones del tablero que conforma la sopa de letras.

De forma análoga, el usuario puede requerir un funcionamiento completamente autónomo, exigiendo también la generación de palabras que estén incluidas en la sopa de letras y también en el diccionario.

GENERACIÓN DE PALABRAS

El algoritmo se basa en generar de forma aleatoria la longitud de las palabras-solución comprendidas entre 1 y la longitud máxima de la sopa de letras dependiendo de la dirección en la que se ubique.

1. Generar un número aleatorio N entre 1 y la longitud máxima de la sopa.
2. Generar por cada posición entre 1 y N una letra aleatoria, generando para ello un número aleatorio entre 1 y 27 que indica la letra correspondiente del alfabeto.

5.1.4 RESOLUCIÓN DE SOPAS DE LETRAS

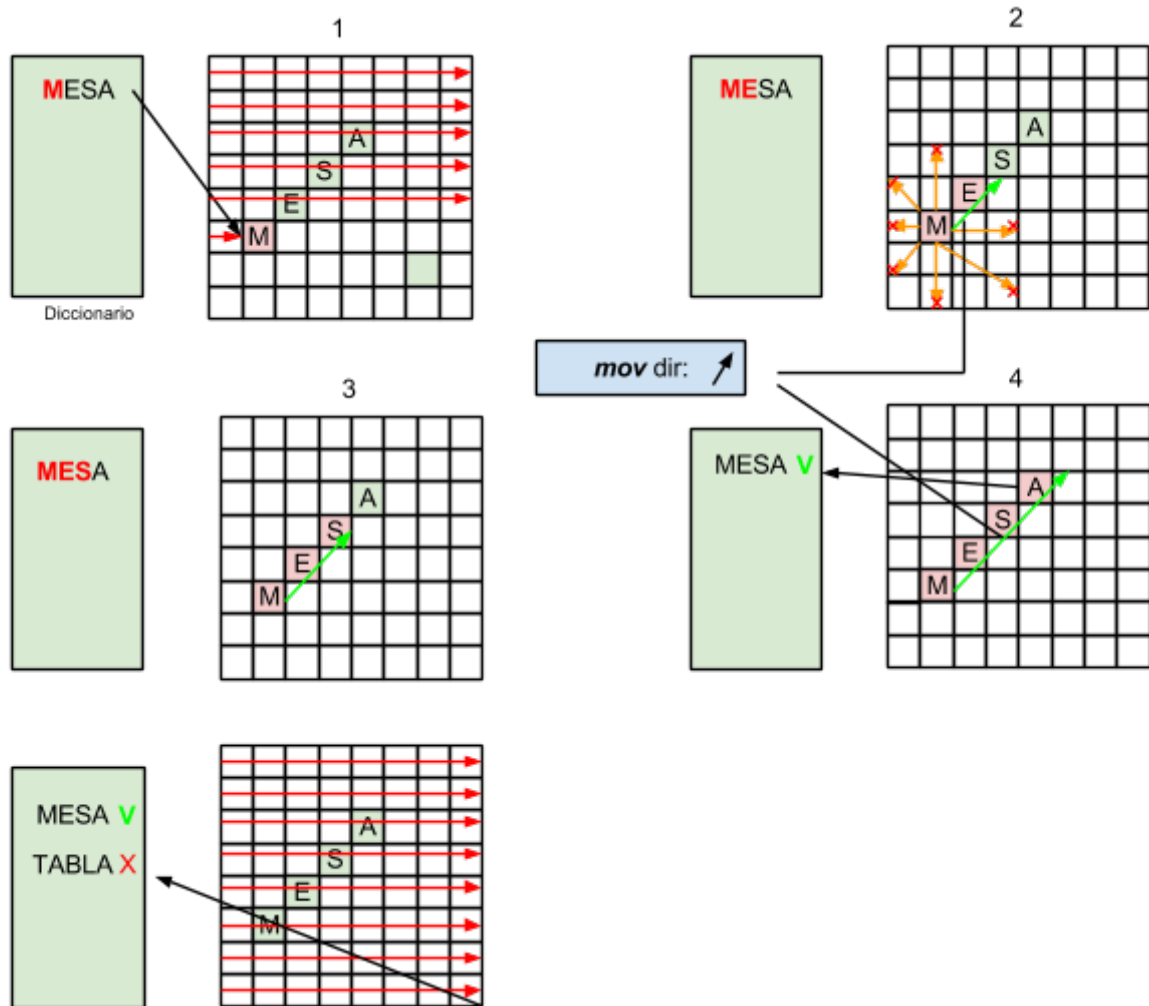
Cuando el usuario desee que el sistema resuelva una sopa de letras con las palabras-solución del diccionario marcadas, será en ese momento en el cual se ejecute un algoritmo de búsqueda que marque donde están ubicadas.

Algoritmo de resolución o búsqueda de palabras

Para encontrar las palabras que están en el diccionario y también están en la sopa de letras se usa el siguiente procedimiento:

1. Por cada una de las palabras del diccionario, se extrae su letra inicial.
2. Se comienza a recorrer de izquierda a derecha, y de arriba a abajo la sopa de letras hasta que una de las casillas contenga la misma letra que la inicial de la palabra del diccionario.
3. Se recorre el entorno de la casilla (posiciones adyacentes), hasta que la letra contenida coincida con la segunda letra inicial de la palabra del diccionario.
4. Si se encuentra, se fija la dirección de la primera letra a la segunda, y en esa dirección se comprueba si una a una las letras coinciden.
5. Si la segunda letra no coincide, o en la dirección fijada de búsqueda no coinciden todas las letras, seguirá recorriéndose la sopa de letras según el paso 2, hasta que se encuentre o termine la sopa.
6. Si la palabra se ha encontrado, se marcará en el diccionario para poder mostrar al usuario que dicha palabra se ha encontrado.

Para llevar a cabo el recorrido de la sopa de letras, basta con leer de forma secuencial una a una las posiciones de la misma. No obstante, una vez localizada una letra buscada que pertenece a una palabra buscada, se hará uso de los movimientos definidos para los laberintos para moverse en la dirección adecuada e ir comparando el contenido de la sopa de letras con la posible palabra-solución.



Debido a que resolver una sopa de letras, consiste en encontrar las palabras del diccionario que estén contenidas en la misma, resolver una sopa de letras se entiende en este caso como un proceso de marcado de palabras en el diccionario.

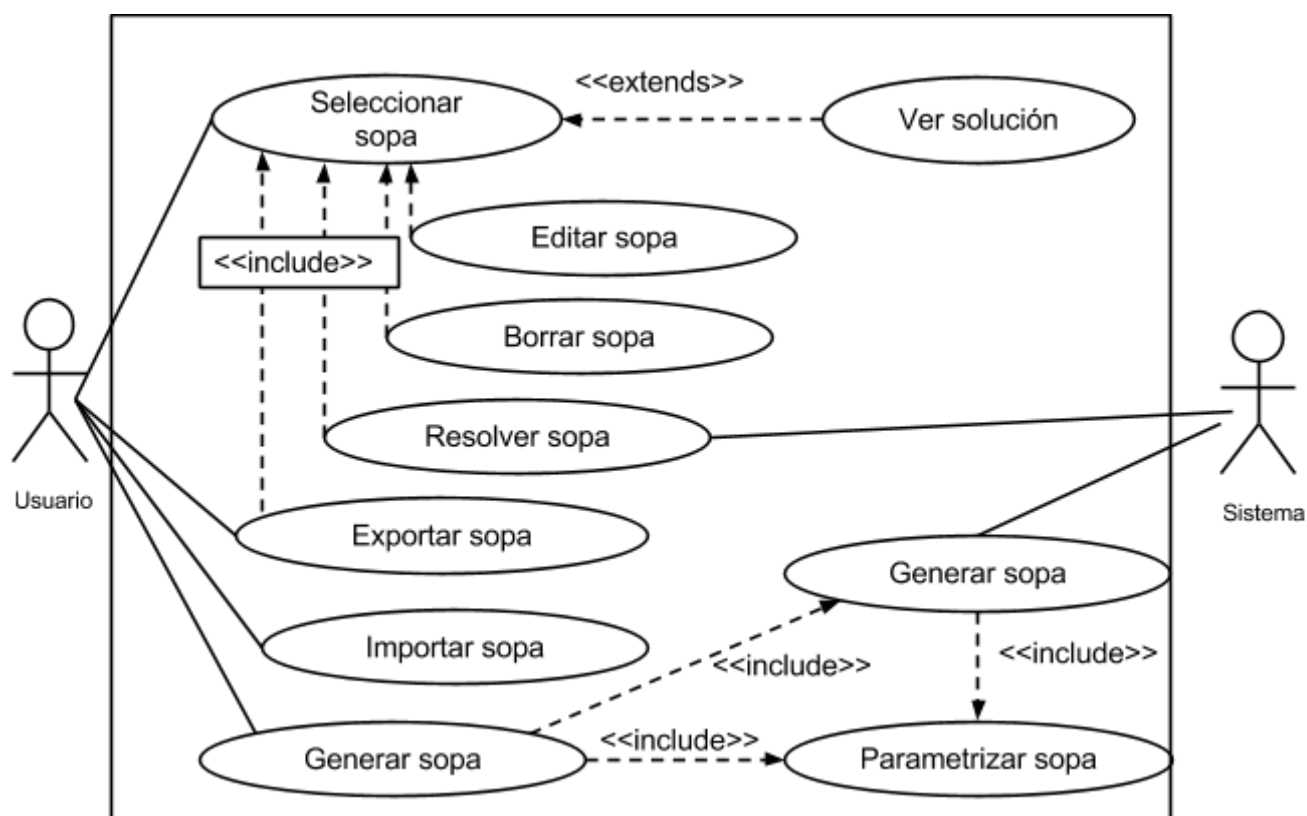
Si una palabra está, se marcará en el diccionario positivamente mientras que si no está en la sopa de letras se marcará en el diccionario negativamente.

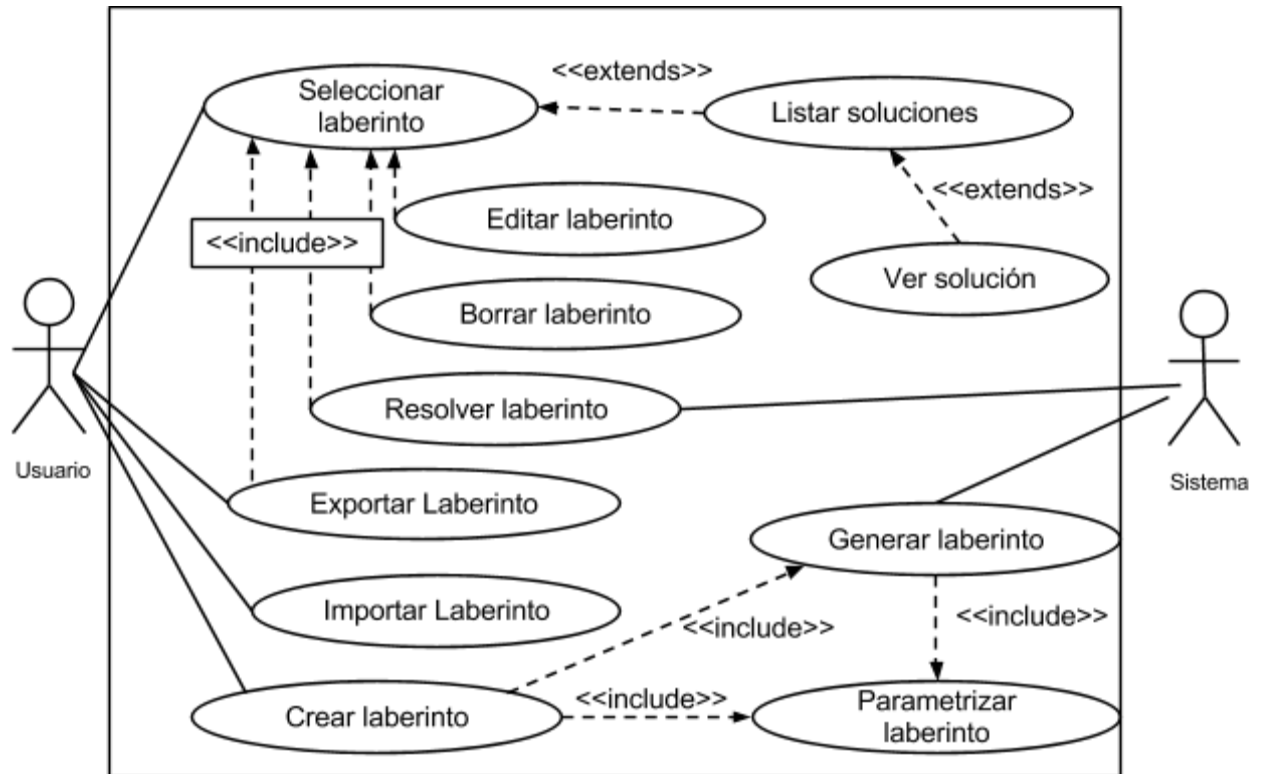
5.2 DISEÑO E IMPLEMENTACIÓN

5.2.1 DIAGRAMA DE CASOS DE USO

La aplicación divide las partes de laberintos y sopas de letras de tal forma y a propósito, que haga verse de forma clara la similitud entre ambos para en cierto modo expresar la forma de comportarse de dos tipos de juegos recreativos que son muy diferentes, pueden crearse de forma parecida.

Por esta razón, los casos de uso entre la parte de laberintos y de sopas de letras es muy parecido, a excepción de que las sopas de letras sólo almacenan una solución por estado actual.





Crear laberinto

Descripción: El usuario crea un nuevo laberinto con unas determinadas especificaciones.

Precondiciones: Ninguna.

Secuencia de actos:

1. El usuario elige crear un nuevo laberinto
2. El sistema ofrece al usuario opciones y ajuste de los parámetros de creación.
3. El usuario determina el nombre y las características. (Caso de uso parametrizar)
4. El sistema genera un nuevo laberinto. (Caso de uso generar)
5. El sistema muestra el nuevo laberinto en la lista de laberintos.

Alternativas: Cancelar la creación del laberinto.

4. El sistema detiene la creación.

Errores:

- Parámetros de creación no válidos, el sistema cancela la creación.

Postcondiciones: Nuevo laberinto disponible.

Editar laberinto

Descripción: El usuario edita los componentes de un laberinto.

Precondiciones: Laberinto existe y está seleccionado.

Secuencia de actos:

1. El usuario elige un laberinto.
2. El usuario selecciona la opción editar.
3. El sistema muestra al usuario la pantalla de edición.
4. El usuario elige editar el contenido interno del laberinto.
5. El sistema muestra al usuario la pantalla de edición con el laberinto.
6. El usuario puede dibujar caminos y obstáculos con los botones del ratón.

Alternativas:

4. El usuario elige editar las posiciones de entrada y de salida.
5. El sistema muestra al usuario la pantalla de edición con el laberinto.
6. El usuario puede cambiar las posiciones de entrada y salida con el ratón.

Errores: Ninguno

Postcondiciones: Laberinto editado.

Borrar laberinto

Descripción: El editar elimina de la lista actual el laberinto.

Precondiciones: Laberinto existe y está seleccionado.

Secuencia de actos:

- El usuario elige un laberinto.
- El usuario selecciona la opción borrar.
- El sistema borra el laberinto de la lista de laberintos disponibles.
- El sistema elimina al laberinto de la pantalla.

Alternativas: Ninguna

Errores: Ninguno

Postcondiciones: Laberinto deja de existir en memoria. (Puede existir en fichero)

Resolver laberinto

Descripción: El usuario solicita al sistema encontrar soluciones del laberinto seleccionado.

Precondiciones: Laberinto existe y está seleccionado

Secuencia de actos:

- El usuario elige un laberinto
- El usuario selecciona la opción resolver laberinto
- El sistema muestra una pantalla informativa de búsqueda mientras busca las posibles soluciones del laberinto.

- El sistema muestra una lista(caso de uso listar soluciones) con todas las soluciones que haya podido encontrar.

Alternativas: El sistema no encuentra ninguna solución y por lo tanto la lista es vacía.

Errores: Ninguno

Postcondiciones: El laberinto tiene asociada una lista de soluciones.

Ver solución

Descripción: Se muestra al usuario la solución o el camino de salida de forma gráfica en el laberinto.

Precondiciones: El laberinto existe, está seleccionado y tiene al menos una solución en su espacio de soluciones, por lo que previamente tiene que haber sido resuelto.

Secuencia de actos:

1. El usuario selecciona un laberinto.
2. El sistema muestra un listado de las soluciones del espacio de soluciones del laberinto.
3. El usuario selecciona una de ellas y pulsa sobre Ver solución.
4. El sistema muestra el laberinto con el camino solución destacado en otro color.

Alternativas:

2. El sistema no muestra soluciones.
 - 2.1. El usuario realiza el caso de uso Resolver laberinto.
 - 2.2 Se vuelve al paso 3 de la secuencia de actos.

Postcondiciones: No se modifica el estado del laberinto.

Generar laberinto

Descripción: Se genera la estructura interna de un laberinto en función de unos parámetros específicos.

Precondiciones: El usuario ha comenzado el proceso de creación de un nuevo laberinto, ha establecido los parámetros de generación.

Secuencia de actos:

1. Se realiza el caso de uso Crear laberinto hasta el punto 4.
2. El sistema crea las estructuras y objetos requeridos para el funcionamiento del laberinto.

Errores: Sucede un error interno de creación, se cancela la generación del laberinto.

Postcondiciones: Laberinto generado y añadido a la lista.

Exportar laberinto

Descripción: El laberinto se almacena en un nuevo fichero externo, o en uno existente (se sobrescribe el contenido que contenga ese fichero).

Precondiciones: El laberinto existe y está seleccionado.

Secuencia de actos:

1. El usuario selecciona un laberinto.
2. El sistema realiza la exportación del laberinto a un fichero de texto incluyendo las soluciones encontradas.

Errores: La extensión del fichero no es adecuada.

Alternativas:

3. El fichero seleccionado ya existe y se sobrescribe el contenido.

Postcondiciones: El laberinto se encuentra almacenado en un fichero del sistema operativo.

Importar laberinto

Descripción: El laberinto se carga dentro de la aplicación a través de un fichero externo.

Precondiciones: El fichero existe, está seleccionado y debe contener un laberinto.

Secuencia de actos:

1. El usuario selecciona un laberinto.
2. El sistema realiza la importación del laberinto de un fichero de texto a la lista de laberintos actuales, así como también se añaden sus soluciones.

Errores: El fichero no contiene un laberinto o contiene errores internos.

Alternativas: Ninguna

Postcondiciones: El laberinto se ha cargado dentro de la aplicación.

Los casos de uso entre la parte de laberintos y de sopas de letras es igual, a excepción de que las sopas de letras sólo almacenan una solución por estado actual.

Crear sopa de letras

Descripción: El usuario crea una nueva sopa de letras con unas determinadas dimensiones.

Precondiciones: Ninguna.

Secuencia de actos:

1. El usuario elige crear una nueva sopa de letras.
2. El sistema ofrece al usuario opciones y ajuste de los parámetros de creación.
3. El usuario determina el nombre y las dimensiones.
4. El sistema genera una nueva sopa de letras. (Caso de uso generar)

5. El sistema muestra la nueva sopa de letras en la lista.

Alternativas: Cancelar la creación de la sopa de letras.

4. El sistema detiene la creación.

Errores:

- Parámetros de creación no válidos, el sistema cancela la creación.

Postcondiciones: Nueva sopa de letras disponible.

Editar sopa de letras

Descripción: El usuario edita los componentes de una sopa de letras.

Precondiciones: Sopa de letras existe y está seleccionada.

Secuencia de actos:

1. El usuario elige una sopa de letras.
2. El usuario selecciona la opción editar.
3. El sistema muestra al usuario la pantalla de edición.
4. El usuario elige editar el contenido interno de la sopa de letras.
5. El sistema muestra al usuario la pantalla de edición de sopas de letras.
6. El usuario puede cambiar el contenido de las posiciones con otras letras.

Alternativas:

4. El usuario elige editar las palabras del diccionario.
5. El sistema muestra al usuario la pantalla de edición de las palabras.

Errores: Ninguno

Postcondiciones: Sopa de letras editada. Cambia a un estado de no-resuelta.

Borrar sopa de letras

Descripción: El editar elimina de la lista actual la sopa de letras.

Precondiciones: Sopa de letras existe y está seleccionada.

Secuencia de actos:

- El usuario elige una sopa de letras.
- El usuario selecciona la opción borrar.
- El sistema borra la sopa de letras de la lista de sopas de letras disponibles.
- El sistema elimina a la sopa de letras de la pantalla.

Alternativas: Ninguna

Errores: Ninguno

Postcondiciones: Sopa de letras deja de existir en memoria. (Puede existir en fichero)

Resolver sopa de letras

Descripción: El usuario solicita al sistema marcar las palabras solución.

Precondiciones: Sopa de letras existe y está seleccionada.

Secuencia de actos:

- El usuario elige una sopa de letras.
- El usuario selecciona la opción resolver sopa de letras.
- El sistema marca las posiciones de las palabras solución de forma interna en la sopa.

Alternativas: El sistema no encuentra palabras y no realiza cambios.

Errores: Ninguno.

Postcondiciones: La sopa de letras cambia a un estado de resuelta.

Ver solución

Descripción: Se muestra al usuario la sopa de letras con las palabras solución (si hay) marcadas.

Precondiciones: La sopa de letras existe, está seleccionada.

Secuencia de actos:

1. El usuario selecciona una sopa de letras.
2. El usuario y pulsa sobre Ver solución.
4. El sistema muestra la sopa iluminando las palabras solución.

Alternativas:

2. El sistema no muestra palabras solución.

Postcondiciones: No se modifica el estado de la sopa de letras.

Generar sopa de letras

Descripción: Se genera la estructura interna de una sopa de letras en función de unos parámetros específicos.

Precondiciones: El usuario ha comenzado el proceso de creación de una nueva sopa de letras, ha establecido los parámetros de generación.

Secuencia de actos:

1. Se realiza el caso de uso Crear sopa de letras.
2. El sistema crea las estructuras y objetos requeridos para el funcionamiento de la sopa de letras.

Errores: Sucede un error interno de creación, se cancela la generación de la sopa de letras.

Postcondiciones: Sopa de letras generada y añadida a la lista.

Exportar sopa de letras

Descripción: La sopa de letras se almacena en un nuevo fichero externo, o en uno existente (se sobrescribe el contenido que contenga ese fichero).

Precondiciones: La sopa de letras existe y está seleccionada.

Secuencia de actos:

1. El usuario selecciona una sopa de letras.
2. El sistema realiza la exportación de la sopa de letras a un fichero de texto.

Errores: La extensión del fichero no es adecuada.

Alternativas:

3. El fichero seleccionado ya existe y se sobrescribe el contenido.

Postcondiciones: La sopa de letras se encuentra almacenado en un fichero del sistema operativo.

Importar sopa de letras

Descripción: La sopa de letras se carga dentro de la aplicación a través de un fichero externo.

Precondiciones: El fichero existe, está seleccionado y debe contener una sopa de letras.

Secuencia de actos:

1. El usuario selecciona una sopa de letras.
2. El sistema realiza la importación de la sopa de letras de un fichero de texto a la lista de sopas de letras actuales.

Errores: El fichero no contiene una sopa de letras o contiene errores internos.

Alternativas: Ninguna

Postcondiciones: La sopa de letras se ha cargado dentro de la aplicación.

5.2.2 DIAGRAMA DE CLASES

Diagrama de clases de la aplicación. Se omiten aquellos atributos no relevantes en la jerarquía.

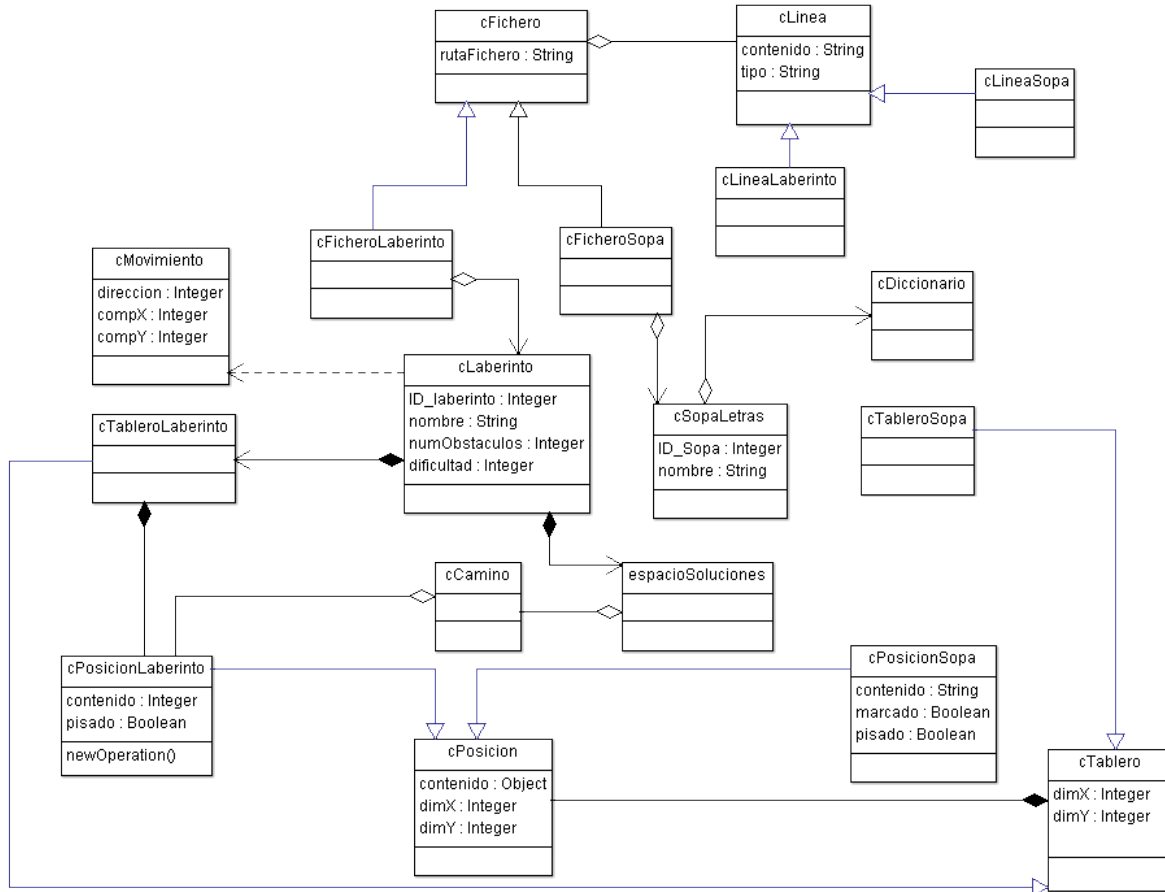
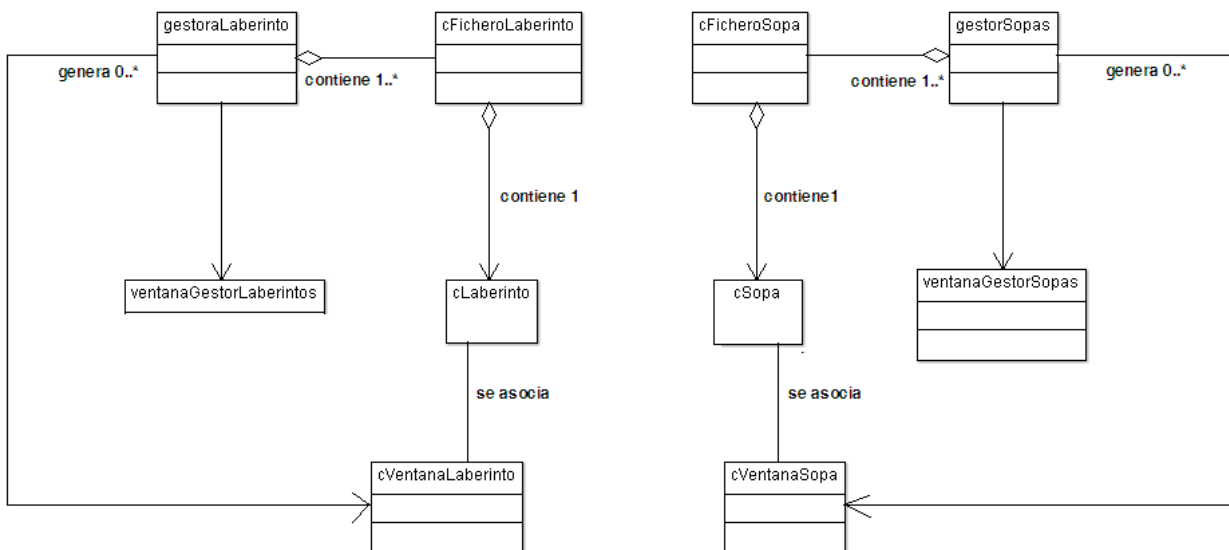


Diagrama de clases que definen la relación de clases de control gráfico y de los elementos.



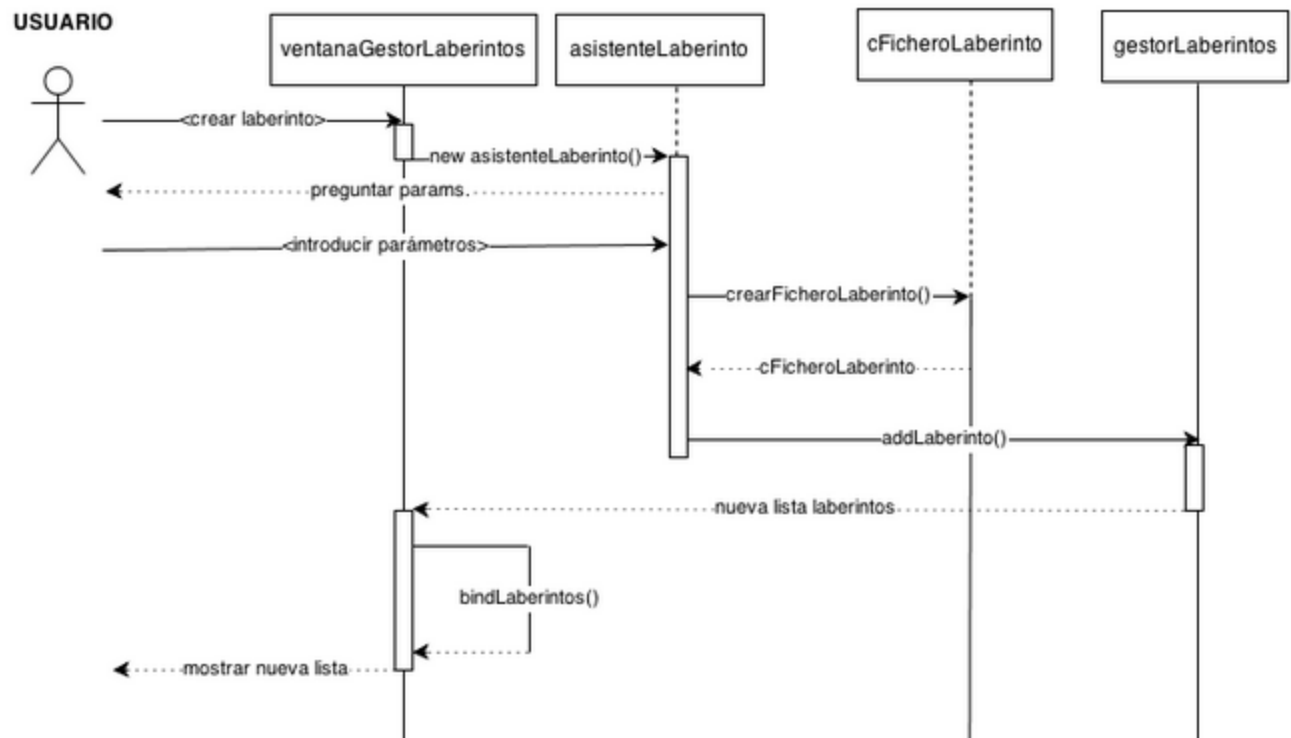
5.2.3 DIAGRAMAS DE SECUENCIA

Continuando con el modelado de objetos y aspectos dinámicos del sistema, se presentan los diagramas de secuencia para algunos de los casos de uso más representativos de la aplicación.

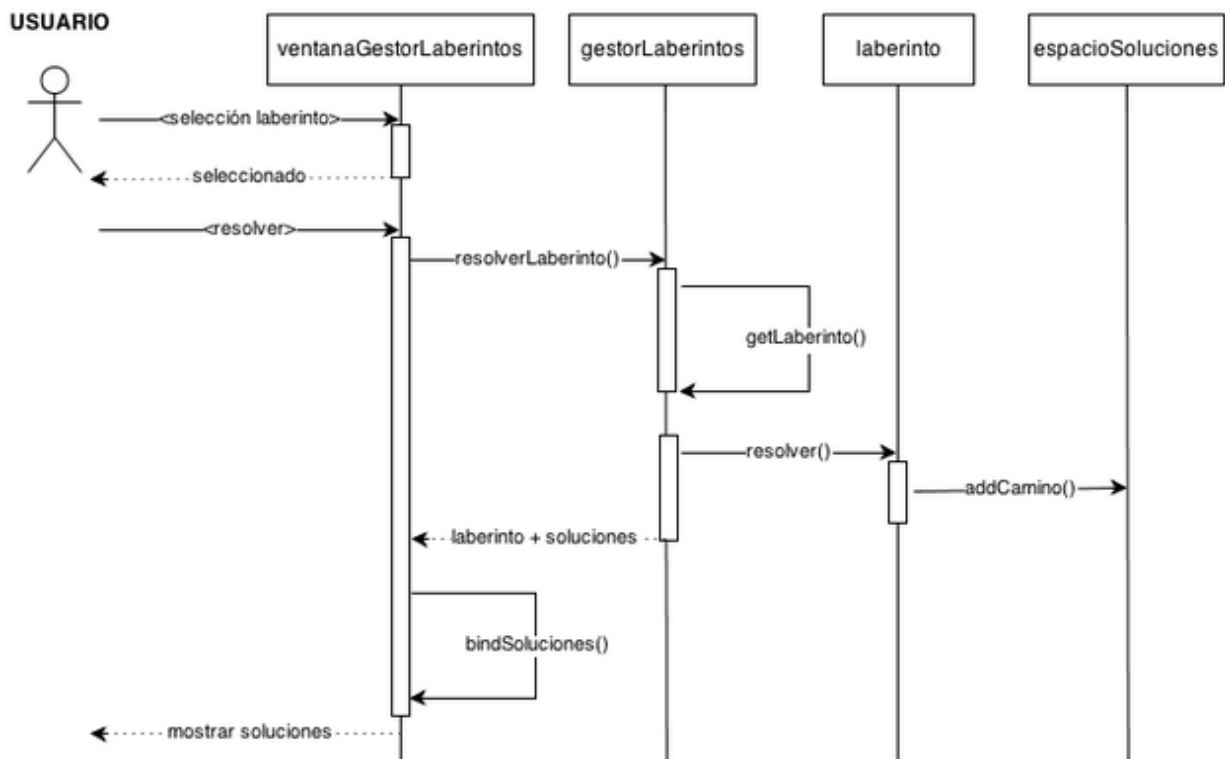
(Todos los casos de uso tienen un análogo similar entre laberintos y sopas de letras y la explicación de uno de ellos, explicará por consiguiente el de su análogo)

Hay que tener en cuenta que es posible que en la etapa de implementación aparezcan otras colaboraciones de clase no representadas por estos diagramas. Sin embargo, estos siguen siendo válidos en tanto presentan una primera visión de las interacciones entre clases que sucederán dentro de la aplicación.

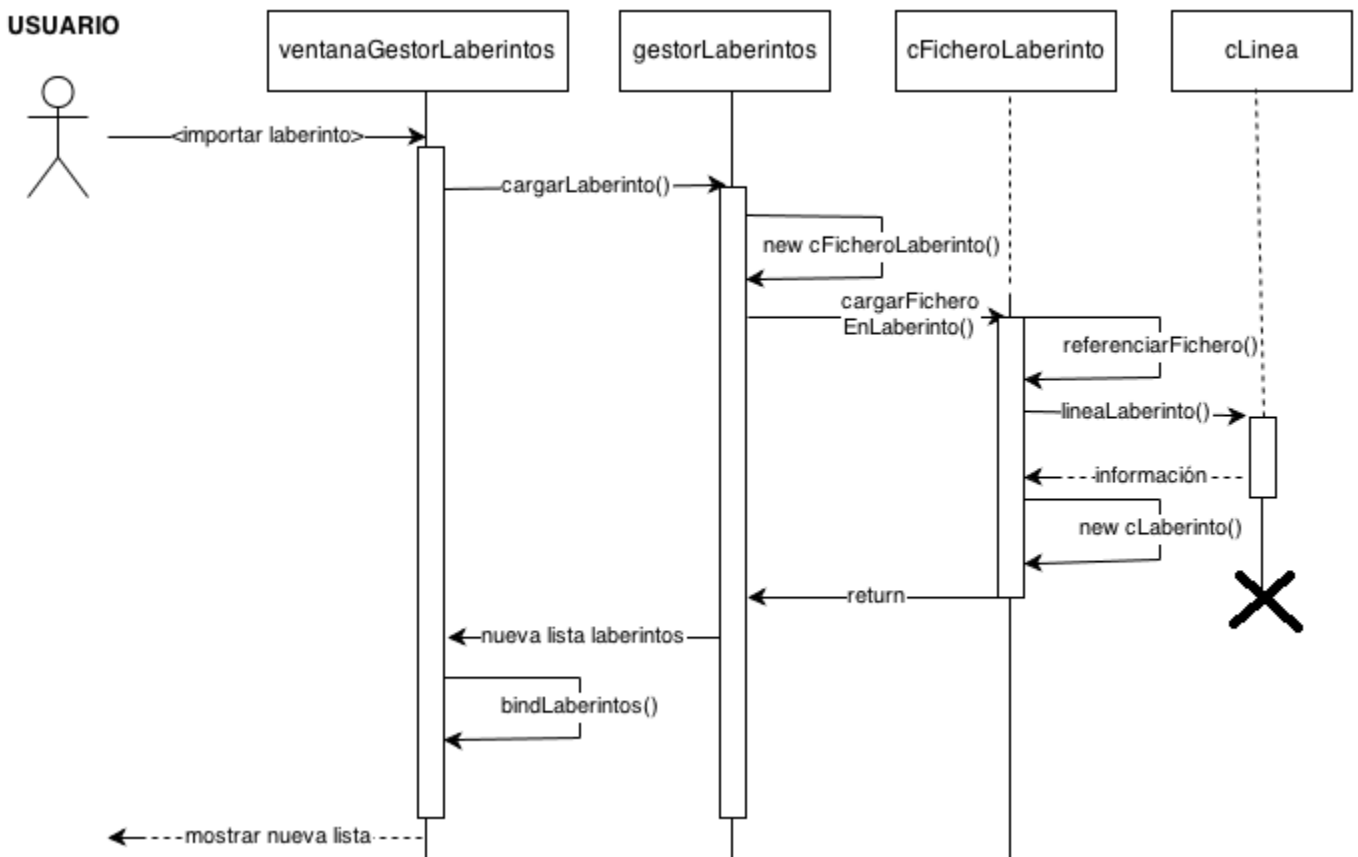
Crear un laberinto



Resolver un laberinto



Importar laberinto

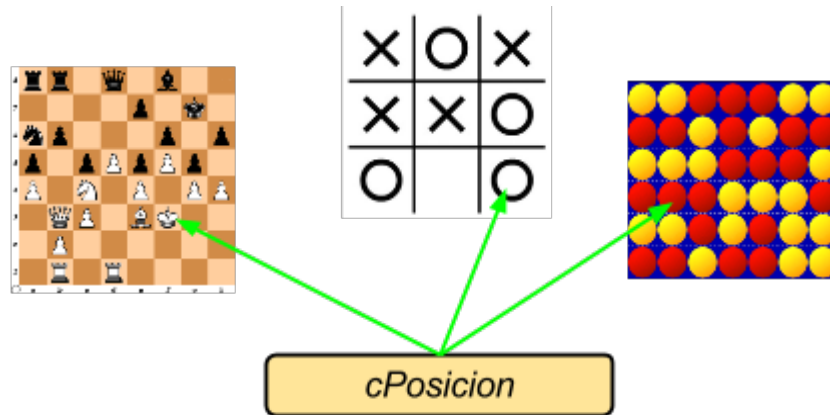


5.2.4 DEFINICIÓN E IMPLEMENTACIÓN DE CLASES

Para poder implementar los componentes de la aplicación se han diseñado una serie de clases que permiten estructurarlos de una forma lógica y coherente a como son.

Clase cPosicion

La clase *cPosicion* representa a los objetos que en conjunto conforman aquellas estructuras de forma matricial.



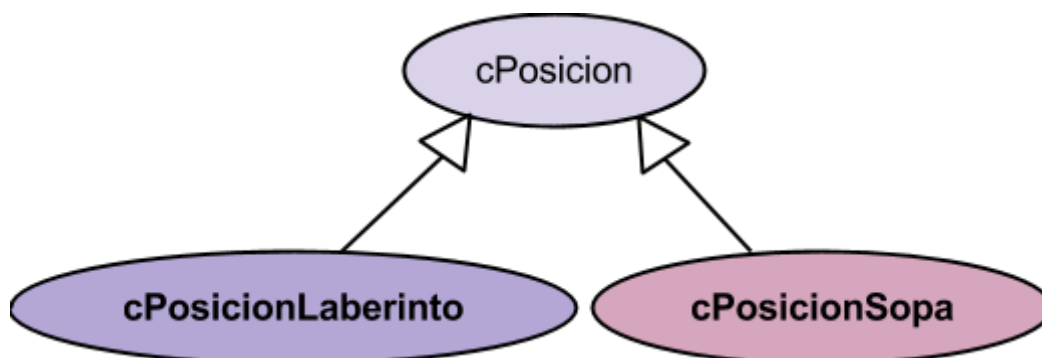
La clase posición define las características generales de posiciones de cualquier ámbito en el plano. Para cada caso en concreto habrá que redefinir una clase.

Ejemplos de casos con posiciones representables por cPosicion

cPosicion ilustra el hecho de ubicar un determinado **contenido** en un determinado lugar con un par de coordenadas. El contenido está definido como de tipo *Object*, ya que las clases que hereden el comportamiento de *cPosicion* tendrán como contenido el tipo o clase que necesiten.

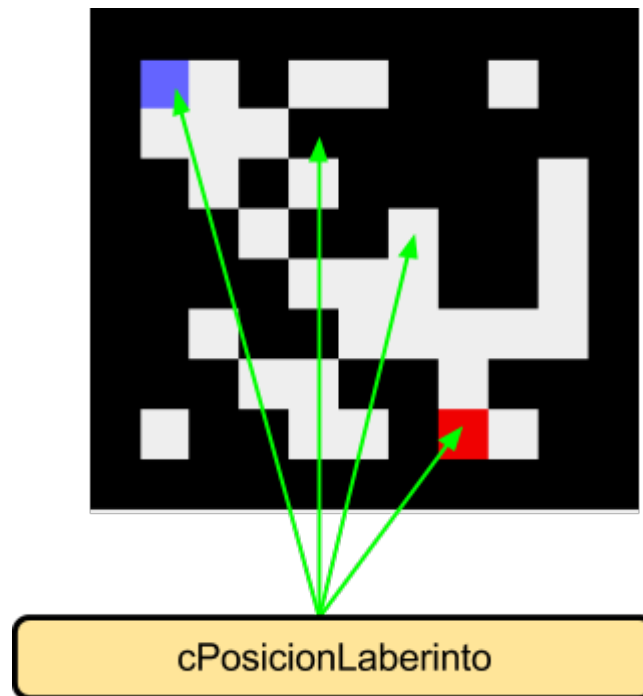
Desde luego *cPosicion* no entiende ni de contenidos, ni de estructuras que la contengan ya que solo define un comportamiento y no a ningún objeto.

Merece la pena recalcar que la clase *cPosicion* sirve para que se creen infinidad de clases hijas debido a que se trata de una estructura de datos muy recurrente y con una estructura muy común en forma de “registro”.



Clase cPosicionLaberinto

En los laberintos que se han definido, su estructura está representada mediante una matriz, y por lo tanto requieren de un objeto con un comportamiento que pueda representar sus posiciones hueco, obstáculo, etc. Por ello, se ha creado una clase *cPosicionLaberinto* que hereda dicho comportamiento y completa aquellas funciones que sean exclusivas de las posiciones de un laberinto.



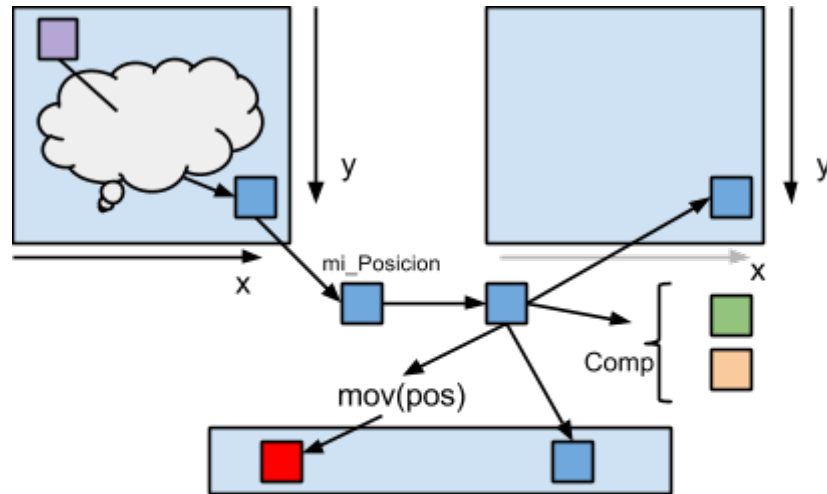
Los objetos de esta clase representan la unidad mínima que conforma un laberinto, son aquellos objetos de los que están formados los caminos, los separadores, las entradas y las salidas. Contienen un espacio vacío, un obstáculo o un jugador en su interior.

Un objeto de la clase *cPosicionLaberinto* guarda de forma interna la siguiente información:

- Coordenadas de ubicación en el laberinto: El objeto es “consciente” en todo momento en que posición está ubicado en la estructura matricial que lo contiene. Por supuesto, dicho objeto no sabe cual es esa estructura ni si está siendo manejado por otro tipos de objetos. El lector puede observar que la información sobre coordenadas va a estar duplicada por lo siguiente.
 - El objeto guarda él mismo sus coordenadas.
 - Debe existir una estructura maestra en forma de matriz que tenga almacenados todos los objetos *cPosicionLaberinto*, por lo tanto el hecho de que ya estén almacenados hace que de forma implícita esa ya exista esa

información, dadas por la fila y la columna que ocupe dicho objeto en la matriz.

La razón de que se almacene en los dos lugares es debido a que de esta forma, los objetos *cPosicionLaberinto* agilizan de una forma increíble la gestión de la información del laberinto, la comunicación entre los diferentes módulos, facilidad del uso de la información en algoritmos y comparación de datos.



Sea lo que sea que haya sucedido con la posición morada hasta llegar a azul
(habrá que casos en que puede que no podamos obtener nada de la azul)
basta con enviarla a otro apartado que lo requiera.

Ejemplo

Las coordenadas pueden obtenerse o cambiarse mediante los correspondientes métodos públicos get y set.

Merece la pena recalcar que la clase *cPosicion* sirve para que se creen infinidad de clases hijas debido a que se trata de una estructura de datos muy recurrente y con una estructura muy común en forma de “registro”.

Contenido

- Los objetos de clase *cPosicionLaberinto* alojan como contenido un entero cuyo valor hace referencia o a una hueco, un obstáculo, una entrada, una salida o al jugador. Dicho valor es provisto por métodos públicos que permiten modificarlo y obtenerlo desde otras partes.

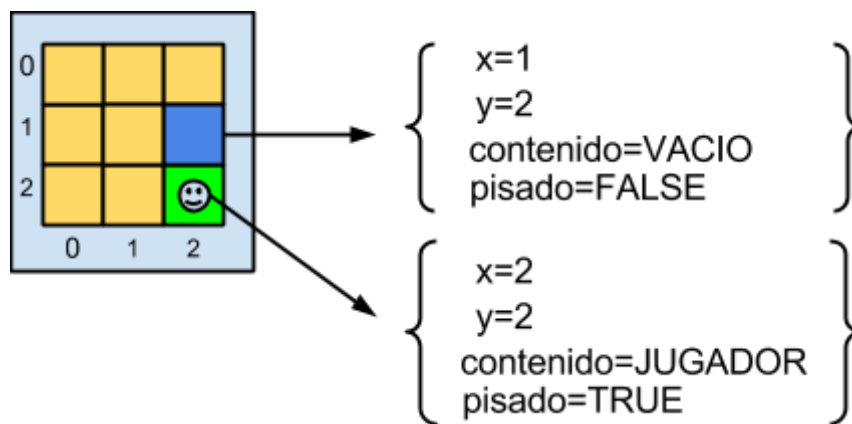
Estado

- El estado de una posición hace referencia al último hecho de lo que ha ocurrido.

En un laberinto, el estado de una posición puede tomar dos valores, pisada o no-pisada, esto es que el jugador haya pasado por la posición o no. Dicho estado se utiliza para controlar el hecho de que no se produzcan bucles.

Al igual que sus otros atributos, la clase provee de métodos públicos para establecer este valor o recuperarlo.

A continuación se muestra un ejemplo de los atributos de los objetos de esta clase.



Constructores

Los objetos de la clase *cPosicionLaberinto* pueden crearse de varias formas dependiendo de los valores que queramos forzar a que tenga el objeto en el momento de la creación, a continuación se enumeran sus constructores:

- Constructor indicando las coordenadas.
- Se creará una posición vacía, no-pisada en las coordenadas indicadas.
- Constructor indicando las coordenadas y el contenido.
- Se creará una posición con el contenido, no-pisada en las coordenadas indicadas.
- Constructor indicando las coordenadas, el contenido y el estado.
- Se creará una posición con todo lo indicado.
- Constructor indicando otra posición.
- Creará una posición idéntica a la que se le pasa.

Métodos

- Se disponen de los métodos públicos get y set que gestionen la posición.
- Método para crear una copia de sí mismo.

A modo de ejemplo, podemos usar los métodos provistos por `cPosicionLaberinto` para convertir la posición azul en una igual que la verde (de la imagen anterior). En un caso real, si se procediese a hacer esto, habría que reflejar también los cambios en otras partes, como se verá más adelante.

```
posAzul = new cPosicionLaberinto(1,2,cPosicion._VACIO,false);  
posVerde = new cPosicionLaberinto(2,2,cPosicion._JUGADOR,true);
```

- Forma 1

```
posAzul.setX(posVerde.getX());  
posAzul.setContenido(posVerde.getContenido());  
posAzul.setPisado(posVerde.getPisado());
```

- Forma 2

```
posAzul.setPosicion(posVerde); // Asigna posVerde a posAzul
```

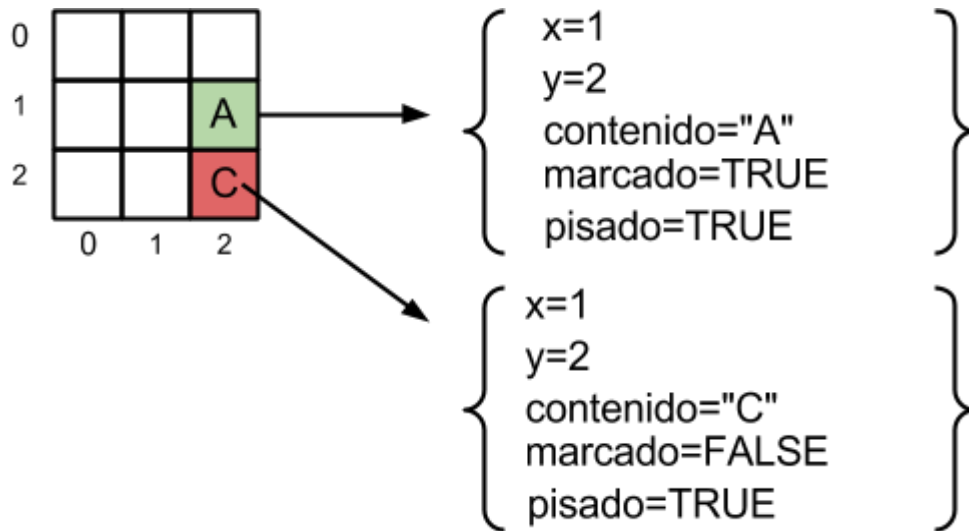
- Forma 3

```
posAzul = posVerde.replicar(); // Crea una copia de posVerde en posAzul
```

Clase `cPosicionSopa`

Los objetos de esta clase representan la unidad mínima que compone una sopa de letras, son aquellos objetos de los que están formadas las palabras. Contienen una letra de algún alfabeto definido.

La principal diferencia con la clase `cPosicionLaberinto` radica en el tipo interno de datos que maneja, y el estado de la posición dentro de la sopa de letras.



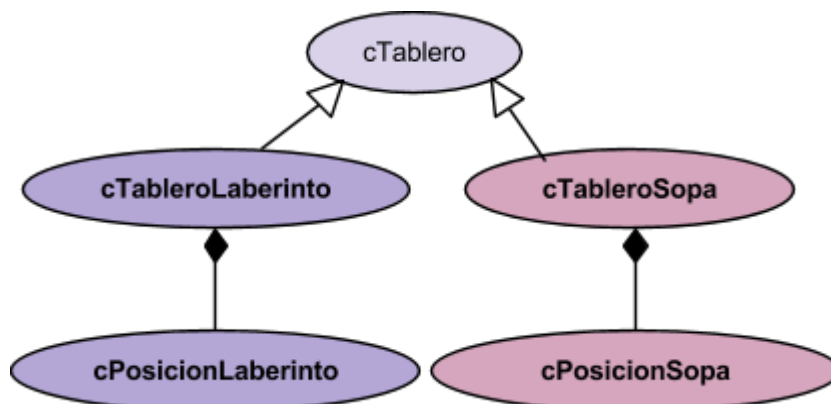
El funcionamiento en cuanto a constructores y métodos es muy similar, varía el tipo de contenido y que una posición de una sopa puede ser marcada como parte de una solución.

Clase c Tablero

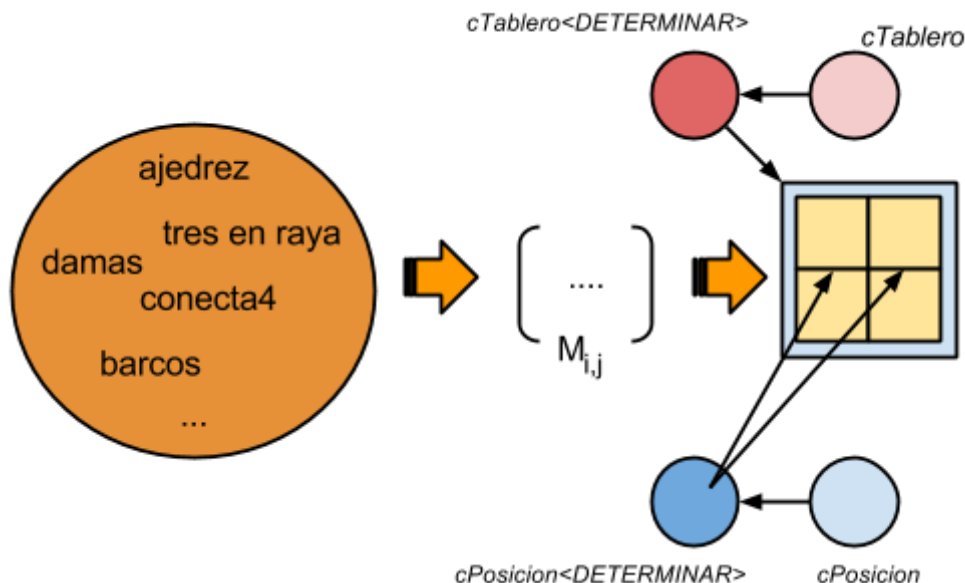
Un tablero es la estructura de datos que almacena la información relativa a las posiciones, es la matriz que guarda la información de problemas, juegos, gráficos, etc. que se traten.

cTablero es una clase que define el comportamiento de los objetos que representan a tableros o estructuras similares que tienen un comportamiento matricial.

Un objeto de la clase *cTablero* es aquella estructura de datos con forma matricial cuyos elementos siguen un comportamiento genérico de posición, es decir, dichos objetos tienen un comportamiento definido por la clase *cPosicion* (no son objetos de la clase *cPosicion* ya que ni siquiera existe un objeto de la clase *cTablero*).



Ejemplos pueden ser un tablero de ajedrez, un mapa de un centro comercial, un juego de hundir la flota, una tabla de una hoja de cálculo, etc y tendrán como contenedor de información a objetos de una clase que herede de la clase *cTablero*.



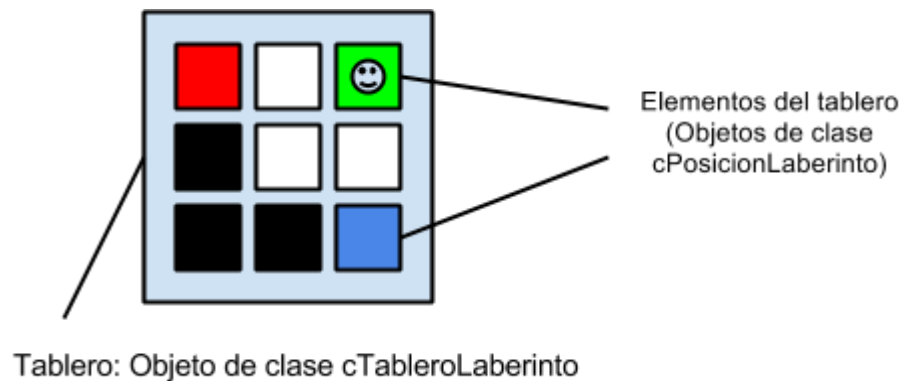
El tablero de un laberinto es la estructura de datos encargada de almacenar la información que define el área de recubrimiento de posiciones de un laberinto. Para ello se ha creado la clase *cTableroLaberinto*, que hereda la forma de comportarse genérica de los tableros y se especializa en el comportamiento de los laberintos.

Clase *cTableroLaberinto*

Un objeto de la clase *cTableroLaberinto* es una matriz de elementos en la que cada elemento es un objeto de la clase *cPosicionLaberinto*, de esta forma se almacena el contenido de los laberintos. Dicha matriz está implementada con array bidimensional de posiciones.

Por otro lado, en todo laberinto, su tablero tendrá que contener un objeto de la clase *cPosicionLaberinto* que almacene una entrada, otro que almacene una salida, y otro que almacene al jugador en caso de que éste no coincida con una entrada o una salida, en cuyo caso, como se dijo en la parte de representación, las entradas y salidas tienen un valor más alto que jugador y prevalecen sobre él. En dicho caso, puede pensarse que la información relevante al jugador se pierde. No se pierde, simplemente el tablero no se encarga de tenerlo ubicado. Esto es misión de otro módulo que se explicará más adelante (Objetos de clase *cLaberinto*).

En cuanto a que el tablero siempre tiene una posición que almacene la salida, es porque los laberintos siempre tienen definida una salida, si no tienen solución es debido a que no puede alcanzarse.



Un tablero se estructura de forma interna con los siguientes atributos:

- Dimensiones del tablero: Las dimensiones del tablero son las dimensiones del laberinto del cual almacena la información, es decir, su tamaño.

Las dimensiones son dos valores enteros mayores que 0. El caso concreto en el que ambas dimensiones sean igual a 1 no se contempla porque no cumple las condiciones establecidas de un laberinto, la existencia de una entrada y una salida, que requiere al menos de 2 posiciones.

Sean L_x y L_y las dimensiones del laberinto, entonces:
 $L_x, L_y \geq 0 - \{L_x = 1, L_y = 1\}$

Las dimensiones pueden obtenerse mediante los métodos públicos get asociados. El tablero guarda esta información mediante dos enteros y para no obtenerla accediendo al array bidimensional.

Matriz

Array de 2 dimensiones que representa las filas y columnas de posiciones del laberinto. Cada elemento del array es un objeto de la clase *cPosicionLaberinto*.

El tablero provee de los métodos get y set correspondientes para interactuar con el contenido de la matriz.

Constructores

Un tablero se puede crear de varias formas:

- Constructor especificando las dimensiones: Creará un nuevo tablero con las dimensiones indicadas con todas las posiciones hueco. En este caso, este tablero no estará listo aún para usarse, necesitará una asignación posterior de entrada y salida.
- Constructor especificando un objeto de tipo *cTableroLaberinto*: Creará un nuevo tablero igual a uno dado.
- Constructor especificando las dimensiones y las posiciones de entrada y salida. Creará un nuevo tablero con las dimensiones indicadas y asignando las posiciones de entrada y salida en su matriz, el resto de posiciones serán huecos.

Métodos

- Obtención y asignación de posiciones en la matriz.
- Procedimiento para el relleno del tablero: Cuando se instancia un objeto de la clase *cTableroLaberinto*, todos los elementos de su matriz que son objetos de la clase *cPosicionLaberinto* serán instanciados como posiciones hueco, indicando a su vez la posición interna de cada objeto posición según su ubicación en la matriz.
- Los objetos de esta clase contiene un método para generar una copia idéntica de sí mismos.

A continuación se muestra un ejemplo de creación e interacción con un objeto de clase *cTableroLaberinto*, en él se definen entrada y salida y luego se intercambian.

```
cPosicionLaberinto posicionEntrada = new PosicionLaberinto(0,0,cPosicion._ENTRADA);  
cPosicionLaberinto posicionSalida = new cPosicionLaberinto(9,9,cPosicion._SALIDA);
```

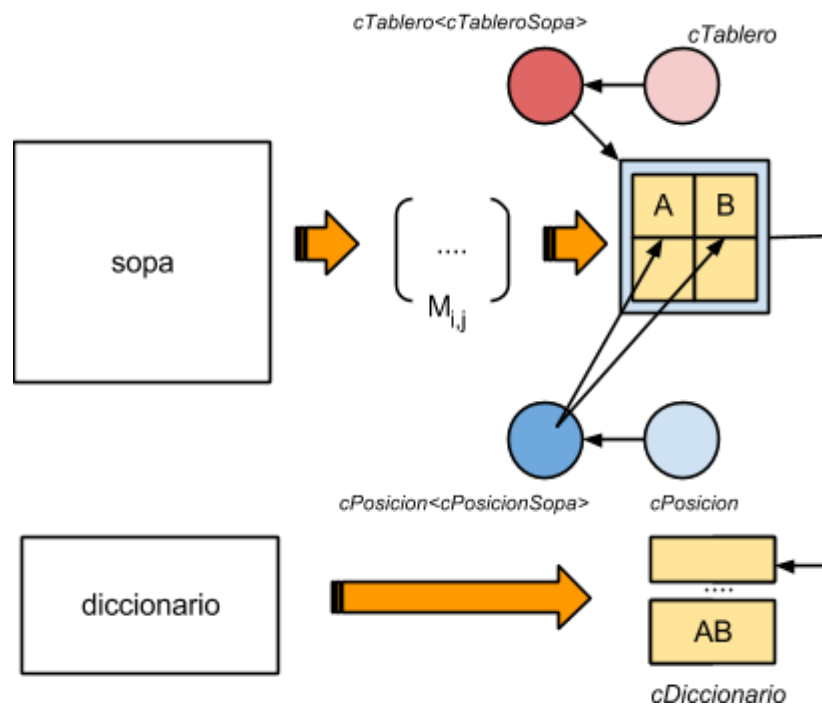
```
cTablerolaberinto tablero = new  
cTableroLaberinto(10,10,posicionEntrada,posicionSalida);
```

```
cPosicion posicionNuevaEntrada = new cPosicionLaberinto(posicionSalida);  
posicionSalida.setPosicion(posicionEntrada);
```

```
tablero.setPosicion(posicionNuevaEntrada);  
tablero.setPosicion(posicionSalida);
```

Clase cTableroSopa

Una vez más podemos hablar de que la estructura correspondiente a las sopas de letras encargada de almacenar las posiciones, es más simple que su análoga de los laberintos, el tablero. Ésta básicamente solo contiene referencias a los objetos posición correspondientes, así como la funcionalidad asociada al marcado y desmarcado de los mismos.



Clase cDiccionario

Los objetos de esta clase se componen de una tabla de dispersión para acceder de forma directa a los elementos (palabras) con los que tratar, que son la solución de la sopa de letras.

Constructores

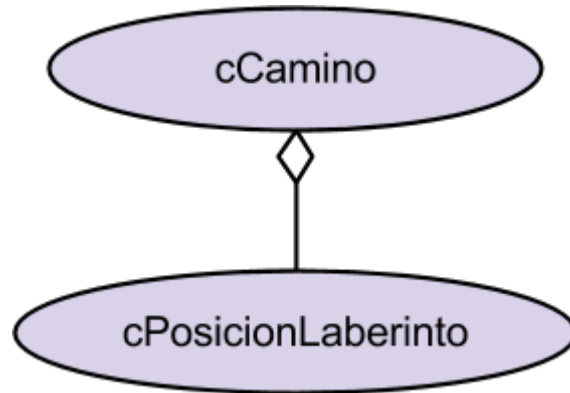
- Creación de un nuevo diccionario vacío.
- Creación de un diccionario especificando una lista de palabras.

Métodos

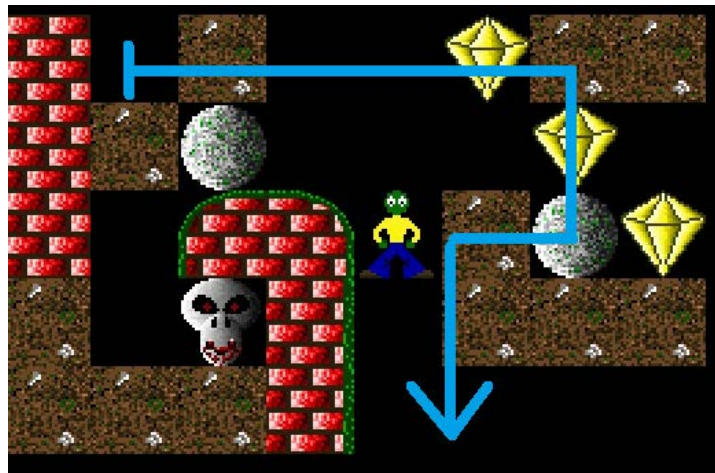
- Recuperación y asignación de palabras relacionadas con la sopa de letras.
- Comparación y verificación de la existencia de palabras buscadas.

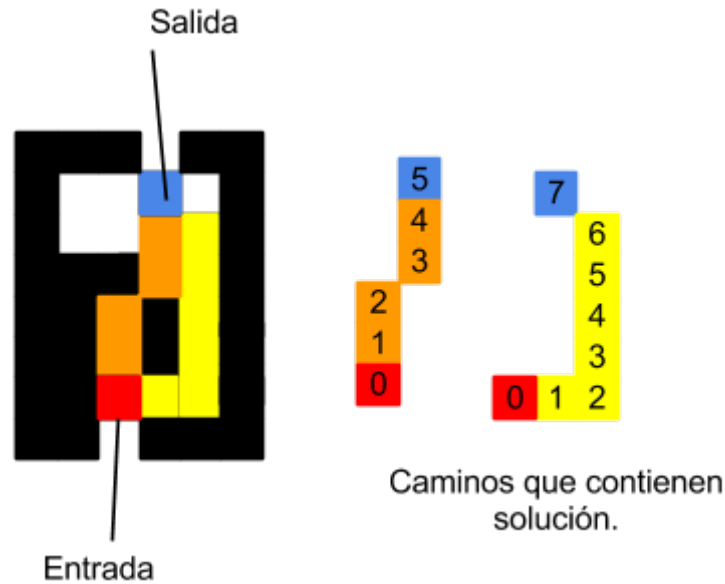
Clase cCamino

Como ya se ha tratado, los caminos son estructuras vectoriales dinámicas de una dimensión que almacenan una secuencia ordenada de posiciones.



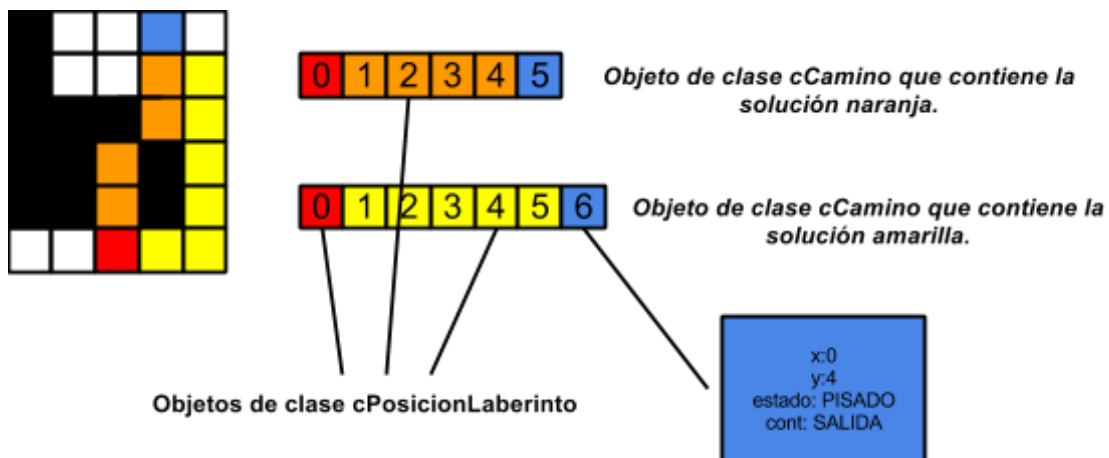
En un laberinto es la secuencia ordenada que hay que recorrer para llegar desde la entrada a la salida sin pasar dos veces por una misma posición. Para ello se dispone de una clase *cCamino*, los objetos de esta clase representarán los caminos que se tracen en un laberinto (aunque pueden ser usados para tratar cualquier secuencia de posiciones de cualquier tipo de tablero).



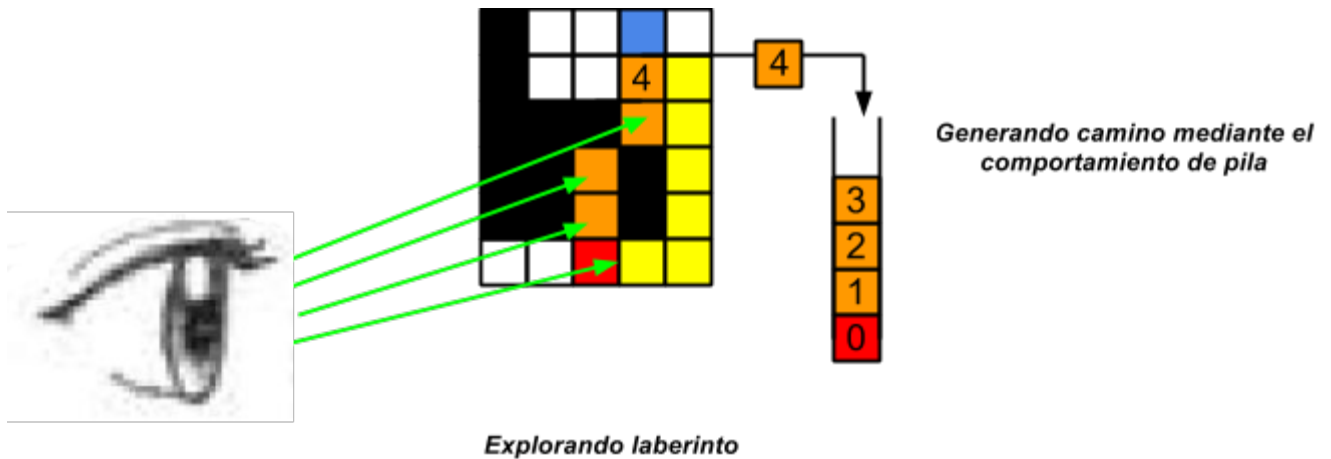


Como las soluciones de un laberinto se corresponden con caminos, se almacenarán en objetos de la clase *cCamino*.

Los caminos están formados por una lista (array dinámico) cuyos elementos son objetos de tipo *cPosicionLaberinto* y representan de forma ordenada como van siendo “pisados” (esto se refleja en el estado de las posiciones) a la hora de construir el camino.



Los objetos de la clase *cCamino*, están inicialmente vacíos, y disponen de métodos que les permiten comportarse como una pila, de este modo podrán ir apilando y desapilando posiciones según se vaya recorriendo el laberinto.



El número máximo de elementos que puede contener un camino estará limitado por las dimensiones del laberinto y por el número de obstáculos que haya en su interior.

Los objetos de la clase *cCaminos*, en el estudio y desarrollo del proyecto, solo se utilizan con el fin de obtener y almacenar soluciones, ya que con la matriz del tablero no se necesitan más medios de almacenamiento de las posiciones de un laberinto.

Un objeto de la clase *cCamino* está constituido por:

- Lista de posiciones: Se trata de una lista de elementos de la clase *cPosicionLaberinto* que almacena las posiciones recorridas. Se proveen de métodos para tratar a la lista como una pila, añadiendo un elemento al final, o eliminando el último elemento añadido. Así mismo pueden consultarse todos los elementos que pertenezcan a la lista.

Constructores

- Constructor único: Este constructor instancia un nuevo objeto de la clase *cCamino* sin ningún elemento.

Métodos

- Apilar posición: Esta clase contiene un método que permite añadir una nueva posición al final del camino.
- Eliminar última posición: Elimina la última posición que fue añadida al camino.

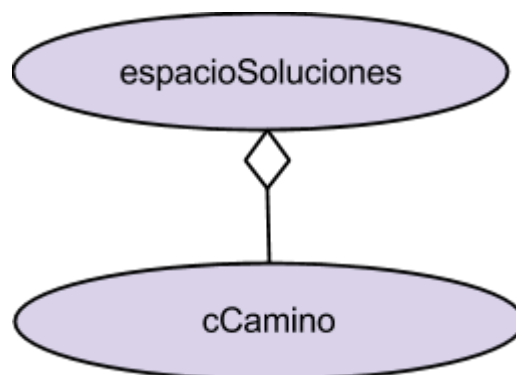
- Desapilar última posición: Devuelve el último elemento que fue añadido al camino y lo elimina del mismo.
- Contiene también los métodos get y set necesarios para obtener la longitud del camino y elementos en cualquier posición.
- Puede hacer una copia de sí mismo.

A continuación se muestra un ejemplo de recorrido de una parte de un objeto de la clase `tablero` y como se almacena en un camino.

// `tablero` es un objeto de la clase `cTableroLaberinto` bien formado

```
cCamino nuevoCamino = new cCamino();
for(int i=0; i<tablero.getDimX(); i++){
    cPosicionLaberinto p = tablero.getPosicion(i,0);
    nuevoCamino.apilarPosicion(p);
}
```

Clase `espacioSoluciones`



Un espacio de soluciones es aquel conjunto de elementos en el que cada uno de ellos supone una situación final, independientemente de lo favorable que resulte al jugador o jugadores implicados en un juego.

En un laberinto definimos espacio de soluciones E de un laberinto L , como un conjunto de elementos en el que todos ellos son soluciones del laberinto L . Se dispone por ello de una clase *espacioSoluciones*, que representa a aquellos objetos con una estructura que almacena objetos de la clase *cCamino* que contienen soluciones del laberinto.

Un objeto de la clase *espacioSoluciones* contiene una lista en la que se irán añadiendo las soluciones que se encuentren en un laberinto.

Un espacio de soluciones no podrá contener dos elementos iguales, es decir, dos caminos con las mismas posiciones y en el mismo orden.

Todo laberinto, independiente de cómo sea, tiene un espacio de soluciones. Los laberintos en los que no se pueda encontrar la salida, aunque no tengan soluciones, sí tendrán espacio de soluciones, pero estará vacío.

Los espacios de soluciones pueden resultar muy útiles para realizar estudios acerca de cuántas y cómo son las soluciones de un laberinto, relacionando el contenido del tablero con el espacio resultante.

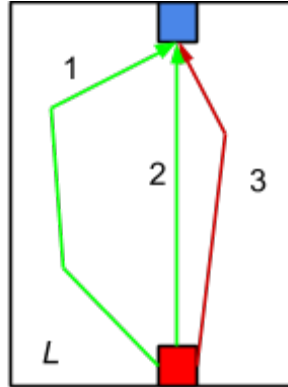
Debido a que dependiendo de como sea un laberinto éste puede llegar a contener una cantidad astronómica de soluciones, y esta situación es computacionalmente “fatal”, se puede afinar más con la definición del espacio de soluciones, o más bien el uso que puede darse al objeto que los representa:

"Un objeto de la clase espacioSoluciones es aquel objeto que almacenará soluciones cuyo número de elementos irá decreciendo según se vayan encontrando en el laberinto L al cual está asociado, no permitiendo la inclusión de soluciones con un número de elementos mayor o igual al de la última solución encontrada."

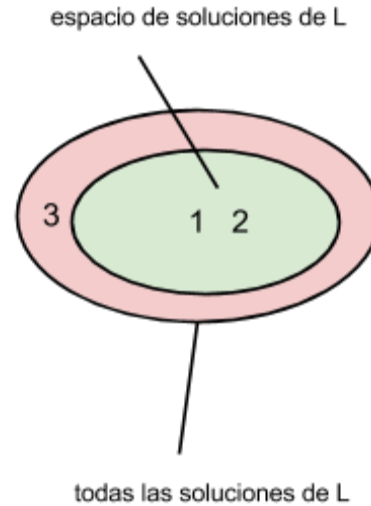
Esto provoca que un objeto de la clase *espacioSoluciones* sea un subconjunto de soluciones del conjunto que las contiene todas.

$$\text{espacioSoluciones} \subseteq \{ \text{todas las soluciones } L \}$$

longitud(1)=10
longitud(2)=6
longitud(3)=8



1. Se añade la solución 1 al espacio de soluciones.
2. Se añade la solución 2 al espacio de soluciones al contener menos elementos de las que están ya en el espacio de soluciones (la 1)
3. Se ignora el camino 3, el espacio de soluciones ya contiene una solución mejor.



Los objetos de la clase *espacioSoluciones* están formados únicamente por una lista de objetos de la clase *cCamino*.

Constructores

- Constructor único: Este constructor instancia un nuevo objeto de la clase *espacioSoluciones* sin ningún elemento.

Métodos

- Añadir camino: Añade los caminos solución que se van encontrando a la lista de soluciones.
- Métodos get para obtener las soluciones de la lista y el número de soluciones que contiene.
- Método para obtener la mejor solución (con el menor número de elementos).

A continuación se muestra un ejemplo del uso de la clase *espacioSoluciones*.

```
espacio espacioSoluciones = new espacioSoluciones();
// solucion1 y solucion2 son dos objetos de la clase cCamino
espacio.addCamino(solucion1);
espacio.addCamino(solucion2);
System.out.println("El espacio tiene " + espacio.getNumCamino() + " caminos"
    + " y el camino más corto tiene " + espacio.getCaminoMasCorto() + "
    posiciones");
```

Clase *cAleatorio*

Los números aleatorios se representan y se generan con los objetos de la clase *cAleatorio*. Los objetos de esta clase son los encargados de generar números al azar para utilizarse en la generación de los laberintos.

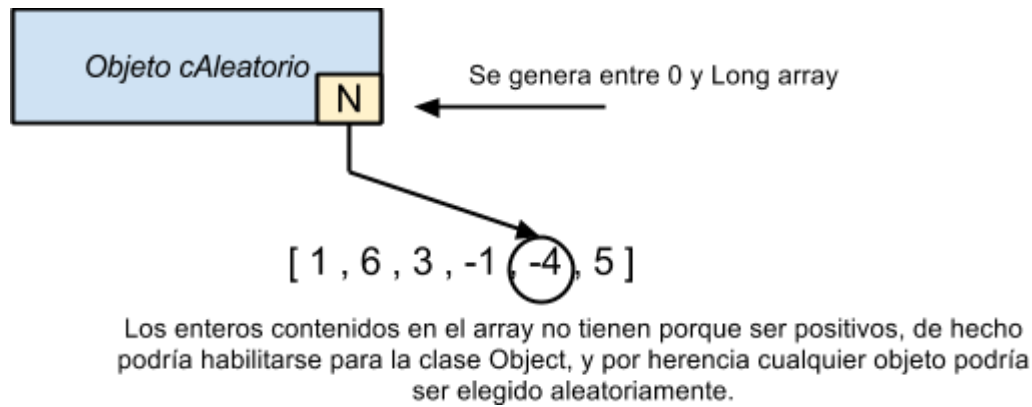
Están formados únicamente por un entero que almacenará el valor aleatorio que se genere y que se utilizará posteriormente.

Constructores:

- Puede instanciarse sin constructor, su contenido inicialmente estará vacío y necesitará generar algún valor aleatorio para poder utilizarse.
- Constructor mediante un número entero positivo: Creará un nuevo objeto que contendrá un valor aleatorio comprendido entre 0 y el número entero pasado como parámetro.

Métodos:

- Generar número aleatorio a través de un entero positivo: Genera un número aleatorio comprendido entre 0 y el número entero pasado por parámetro. El constructor utiliza este método para generar el valor aleatorio inicial al crear el objeto.
- Generar número aleatorio a través de un rango de enteros positivos: Genera un número aleatorio dentro del rango especificado.
- Generar valor booleano: Realmente es equivalente a generar números aleatorios comprendidos entre 0 y 1.
- Elegir entero de un array: Recibiendo un array de números enteros, elegirá de entre ellos uno al azar, dependiendo de lo que especifique su contenido interno, que no será más que un número aleatorio. Como dicho contenido puede ser mayor que el número de elementos del array, generará un número entero de máximo la longitud del array para poder obtener uno de ellos.



- Método get para obtener el contenido generado.

Clase cMovimiento

Los objetos de la clase *cMovimiento* representan a los movimientos que se realizan en el laberinto para moverse por él.

Los objetos de esta clase se aplican sobre objetos de la clase *cPosicion* y de aquellas que hereden de *cPosicion* para obtener otras con una ubicación distinta, "resultado" de haberse producido un desplazamiento.

Estos objetos guardan de forma interna dos componentes para aplicar el desplazamiento de las posiciones en un tablero. Básicamente se trata de una operación sobre las coordenadas de una posición. Sin embargo, solo necesitaremos especificar una dirección (un número entero) para que dichas componentes se aplique automáticamente.

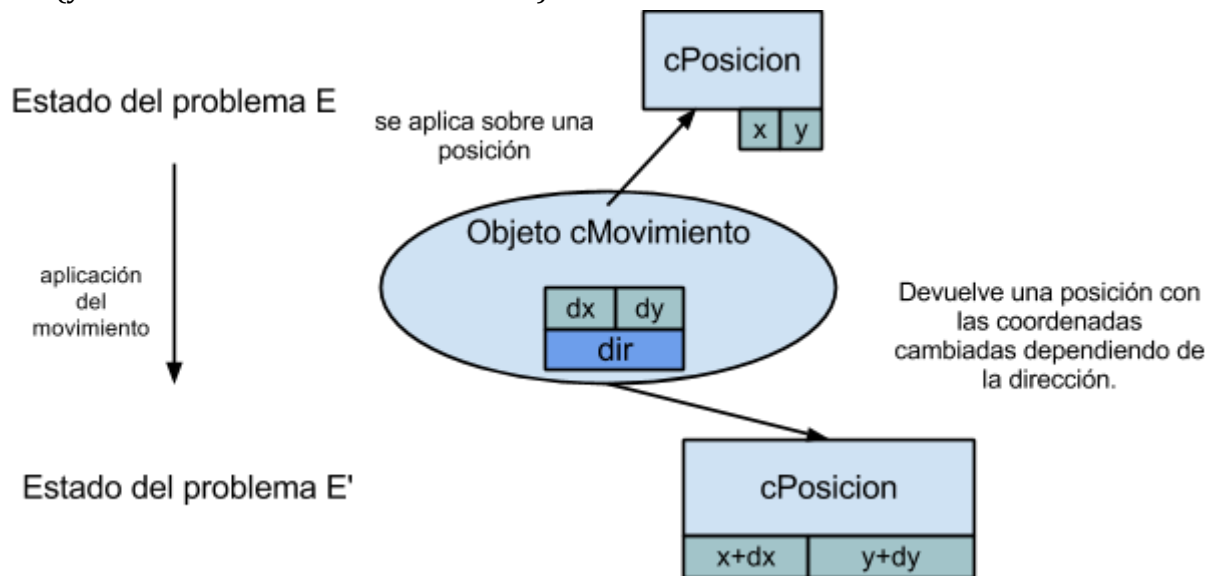
Esta clase resulta muy útil, no sólo para desplazarse normalmente por un tablero, sino porque permiten realizar fácilmente operaciones sobre posiciones adyacentes, obtención de posiciones consecutivas en cualquier dirección, etc.

Debido a que se han definido ocho direcciones diferentes, sólo pueden existir ocho objetos de la clase *cMovimiento* diferentes.

Los objetos de esta clase se comportan exactamente igual a los operadores de cambio de estado de cualquier problema de inteligencia artificial o de algorítmica en cuanto a la búsqueda de soluciones se refiere. Podemos considerar que son lo mismo.

Por lo tanto, un objeto de la clase *cMovimiento* al aplicar sobre una posición, si ésta está involucrada en el estado de un problema, el objeto también se aplicará sobre el estado del problema y lo cambiará a un nuevo estado.

De forma análoga, pueden realizar también el proceso inverso, “desaplicarse” sobre una posición (y en consecuencia sobre un estado).



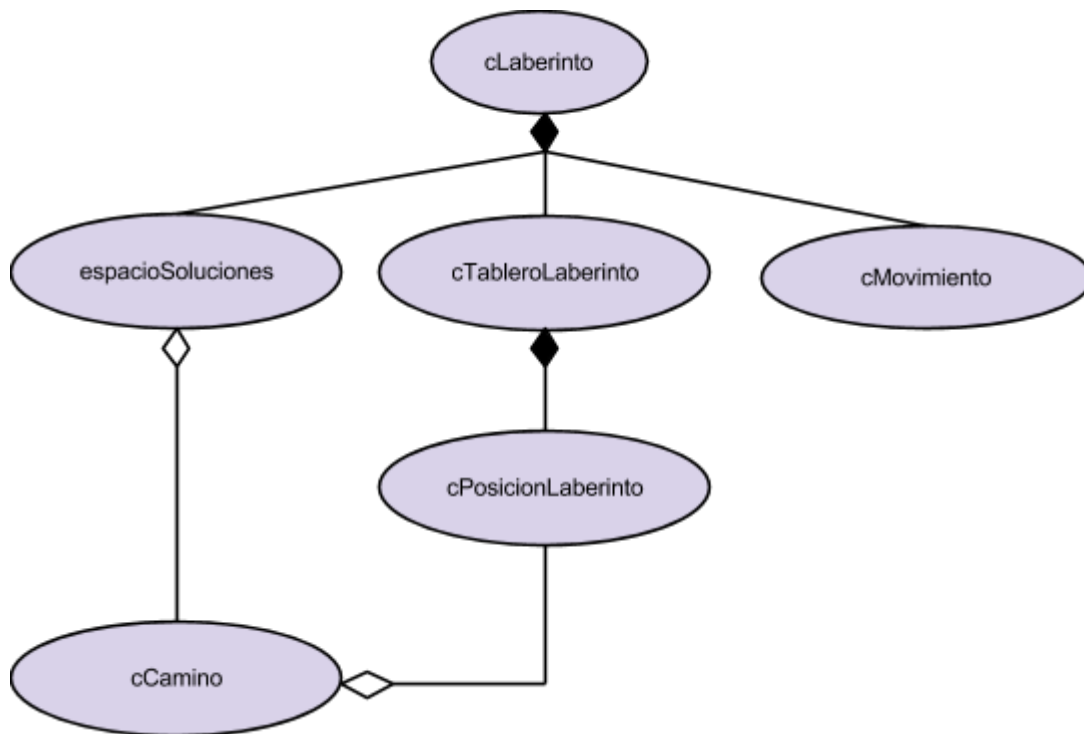
Constructores

- Constructor especificando la dirección: Crea un nuevo objeto de la clase *cMovimiento* que requiere como parámetro de entrada un número entero.

Métodos

- Aplicarse: Este método efectuará sobre una posición un cambio de ubicación de la misma.
- Desaplicarse: Actúa de la misma forma que al aplicarse pero en el sentido opuesto.
- Método set para la dirección que le permite así comportarse como movimientos con otra dirección sin necesidad de instanciar más.
- Método get para la dirección.
- Crear copias de sí mismo.

Clase cLaberinto



La clase *cLaberinto* es el representante máximo de un laberinto y contiene todos sus elementos. También hace uso de otros elementos independientes del laberinto, pero útiles para realizar determinadas funciones, como las funciones de generación de elementos para formar laberintos incluidas en la clase estática *cParametrizadorLaberinto*.

Un objeto de la clase *cLaberinto* contiene un tablero para almacenar el contenido de todas las posiciones que conforman el laberinto (huecos, obstáculos, entrada, salidas y posición del jugador).

Por otro lado, también contiene referencias a las posiciones más importantes, es decir, a la entrada, las salidas y a la ubicación del jugador para tener un control más eficiente a la hora de realizar operaciones sobre el tablero.

Un objeto de la clase *cLaberinto* está compuesto de :

- Objeto de la clase *cTablero* que contiene todas las posiciones del laberinto, es decir, un array o matriz bidimensional de objetos de la clase *cPosicionLaberinto*.

- Array que contiene todos los movimientos definidos para ese laberinto, es decir, un array de objetos de la clase *cMovimiento*.
- Un espacio de soluciones que contiene las soluciones encontradas la última vez que se exploró. Por defecto, cuando se inicializa un laberinto el espacio de soluciones está vacío.
- Una referencia a la posición de entrada del laberinto, es decir, un apuntador a un objeto de la clase *cPosicionLaberinto*.
- Una referencia a la actual posición del jugador en el laberinto, es decir, un apuntador a un objeto de la clase *cPosicionLaberinto*. El jugador podrá ocupar toda posición cuyo contenido no sea un obstáculo.
- Una referencia a la posición de salida del laberinto, es decir, un apuntador a un objeto de la clase *cPosicionLaberinto*. En realidad, la clase *cLaberinto* funciona en base a una salida del laberinto, pero el código permite la funcionalidad de crear más salidas, por lo tanto hay un array de referencias a objetos de la clase *cPosicionLaberinto*. El motivo por el cual solo se utiliza una posición de salida es la objetividad de buscar y comparar soluciones en cuanto a una misma salida. Como además se pretende buscar las mejores soluciones, las supuestas posiciones de salida más alejadas no cobrarían mucha importancia.
- Un entero que indica el número de obstáculos que existe en el laberinto, lo que equivale al número de objetos de la clase *cPosicionLaberinto* contenidos en el tablero cuyo contenido es un obstáculo. Este valor se utiliza tanto durante la generación del laberinto como para hacer comprobaciones en la exploración.
- Un nombre o identificador del laberinto que el propio usuario podrá decir.

Constructores

- Constructor sin especificar ningún parámetro: Si se instancia un laberinto con este constructor, absolutamente todos sus componentes se generarán de forma aleatoria.
- Constructor especificando características: Se generará un laberinto con las características especificadas, las que no se especifiquen se generarán de forma aleatorio teniendo en cuenta en los parámetros que se hayan indicado.

- Número de obstáculos que contendrá el laberinto.
- Movimientos disponibles en el laberinto para que realice el jugador.
- Forzar solución: Se creará al menos un camino seguro entre la entrada y la salida.

Métodos

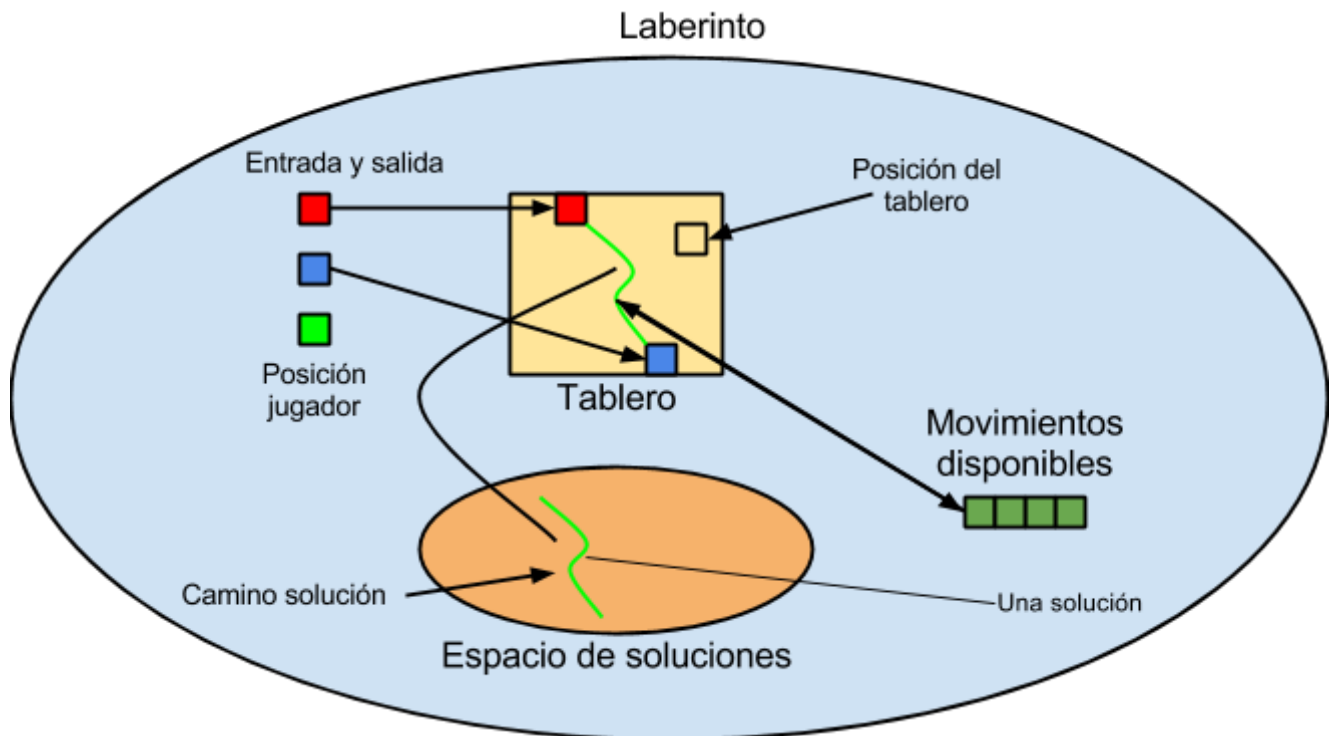
Los laberintos, generan de forma aleatoria, o semi-aleatoria, dependiendo de los parámetros que especifique el usuario a la hora de generarlo. No obstante, una vez generados, los laberintos son editables y configurables, lo que permite muchísima más versatilidad para trabajar con ellos.

Luego la clase *cLaberinto* brinda de varios métodos para poder editar su contenido.

- Obtener y establecer el número de obstáculos del laberinto.
- Obtener y establecer el tablero asociado al laberinto. Este método está restringido a usarse en la inicialización ya que el tablero influye en el resto de componentes del laberinto, y cambiar éste implicaría cambiar otros.
- Obtener y establecer las posiciones de entrada y salida.
- Método para interactuar con el espacio de soluciones.
- Aplicar movimientos sobre la posición actual del jugador en el laberinto.
- Método para la resolver el laberinto.
- Obtener los movimientos disponibles para que realice el jugador.
- Obtener los movimientos aplicables en un momento dado para que realice el jugador.
- Establecer un nombre identificador para el laberinto.
- Forzar solución: Este método consta de un algoritmo que intenta forzar una conexión entre la entrada y la salida si los movimientos definidos así lo permitiesen.

(Si las posiciones de entrada y salida estuvieran alejadas una distancia mayor que cero en la coordenada **Y**, y no hubiese definido ningún movimiento en la componente **Y**, sería imposible llegar incluso conectandolos.

Como el laberinto puede obtener las referencias de cualquier de sus componentes, puede enviarles mensajes para realizar cualquier operación de resolución o configuración que influya a cualquier de sus componentes.



A continuación se muestra un ejemplo del uso de un objeto de la clase *cLaberinto*.

```
cLaberinto L = new cLaberinto();
L.setNumObstaculos(L.getNumObstaculos()+1);
L.setMovimientosDisponibles(movs); // movs es un array de objetos de la clase cMovimiento
L.resolver();
cCamino c = L.getEspacioSoluciones().getCamino(0);
```

Clase cParametrizador

La clase *cParametrizador* es una clase estática que se encarga de realizar la generación de los componentes de un laberinto.

Esta clase es llamada por un objeto de la clase *cLaberinto* durante su inicialización para poder construirse.

Métodos:

- Generar número de obstáculos: Este método genera un número de obstáculos válido para un laberinto. Controla en mayor o menor medida la homogeneidad del laberinto para no desbordarlo con muchos obstáculos ni tampoco para dejarlo excesivamente vacío.
- Existe el riesgo de que aún así, las posiciones de entrada y salida queden encerradas,
- por lo cual el propio laberinto consta de un método para *“forzar caminos de solución”*.
- Generar posiciones de apertura: Este método genera posiciones en los bordes del laberinto. Se utiliza para ubicar la entrada y la salida a la hora de generar el laberinto.
- Generar posición aleatoria: Generalmente se usa para generar aquellas posiciones que van a ser obstáculos y van a rellenar el tablero.
- Generar tablero: Genera un tablero a partir de un rango de dimensiones, entre las cuales se eligen de forma aleatoria.
- Generar movimientos: Genera los movimientos que estarán definidos en un laberinto. Es poco recomendable dejar que este aspecto se genere de forma automática, es altamente probable que el laberinto no tenga solución si no se generan los movimientos adecuados, este aspecto es preferible realizarse de forma manual por parte del usuario.

Clase cMatematicas

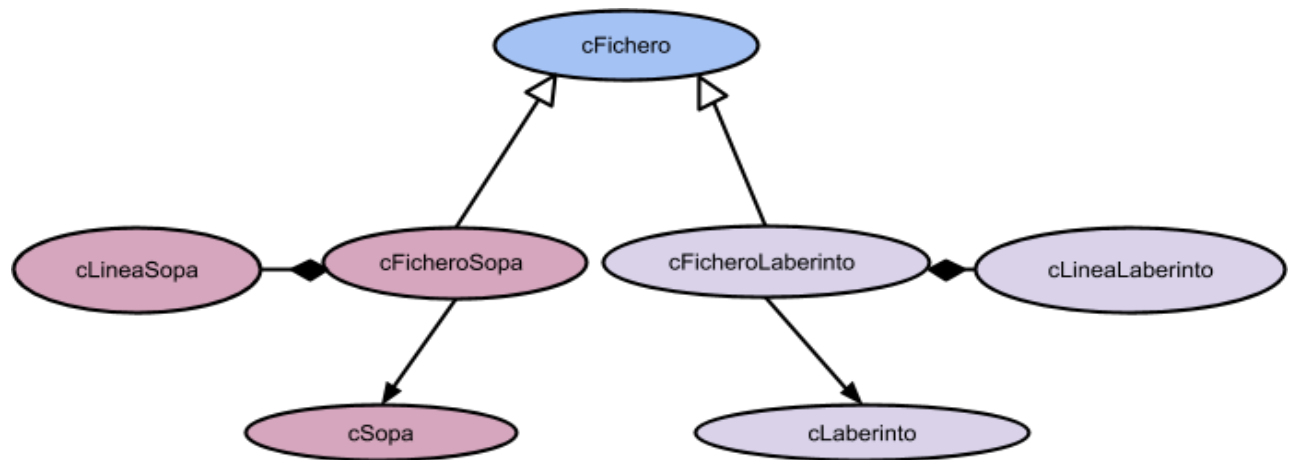
Es una clase estática que proporciona métodos que operan sobre números.

Concretamente se utiliza para obtener el menor o el mayor de dos números enteros, o obtener dichos valores de un vector de números enteros determinado.

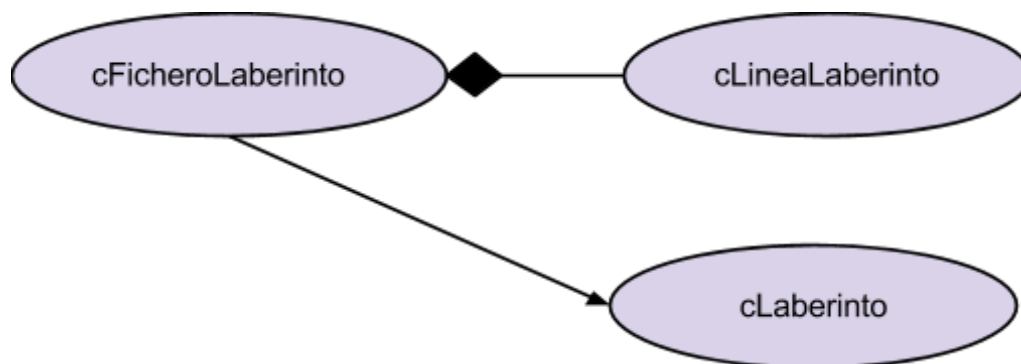
Clase cVector

Clase estática que permite construir arrays de posiciones a partir de posiciones por separado o incluidas en otros arrays.

5.2.5 ALMACENAMIENTO EXTERNO



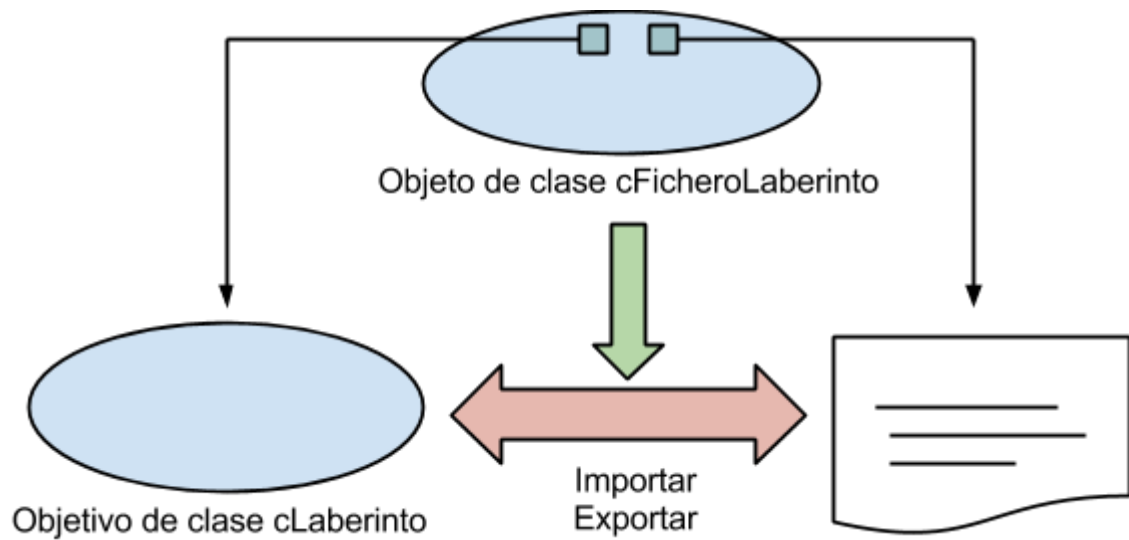
Clase cFicheroLaberinto



La clase *cFicheroLaberinto* implementa la funcionalidad de importar y exportar el contenido de laberintos a ficheros externos y viceversa.

Un objeto de la clase *cFicheroLaberinto* está compuesta por una referencia a un clase que controle el fichero externo que almacenará el contenido de un laberinto, y por la referencia a ese laberinto, es decir, en relación de asociación.

Tanto para importar como para exportar la información, se hace uso de objetos de la clase *cLineaLaberinto*, que se explica más adelante.



Constructores:

- Constructor especificando un laberinto: Si se inicializa especificando un laberinto, el fichero en el que se almacene se determinará posteriormente. Será el usuario el que decida cuál será el fichero.
- Este modo de inicializar un constructor está pensado para exportar un laberinto.
- Si elige un fichero que ya existe, eliminará el contenido que existiese previamente, en caso contrario creará un nuevo fichero con el contenido del laberinto.
- Constructor especificando la ruta de un fichero: Al inicializar el objeto especificando la ruta de un fichero, se podrá importar directamente el contenido. Un nuevo objeto de la clase *cLaberinto* se creará a partir del contenido del fichero especificado.

Los constructores no importarán ni exportarán ningún contenido, sólo dejarán preparado el objeto para hacer dichas operaciones posteriormente.

Métodos:

- Métodos para obtener el laberinto o ficheros asociados al objeto.
- Métodos para establecer una ruta de fichero o un laberinto distintos.

- Método para exportar el laberinto: Se exportará el contenido del laberinto referenciado al fichero cuya ruta se haya especificado.
- Método para importar el laberinto: Se importará el contenido del fichero especificado a un nuevo laberinto.

Formato del fichero externo que almacena un laberinto:

Nombre del laberinto	9 <Nombre>
Posiciones	0 <X>, <Y>, <Contenido>
Movimientos	2 <dir1>, <dir2>..., <dirN>
Soluciones	1 <pos1>, <pos2>, ... ,<posN>

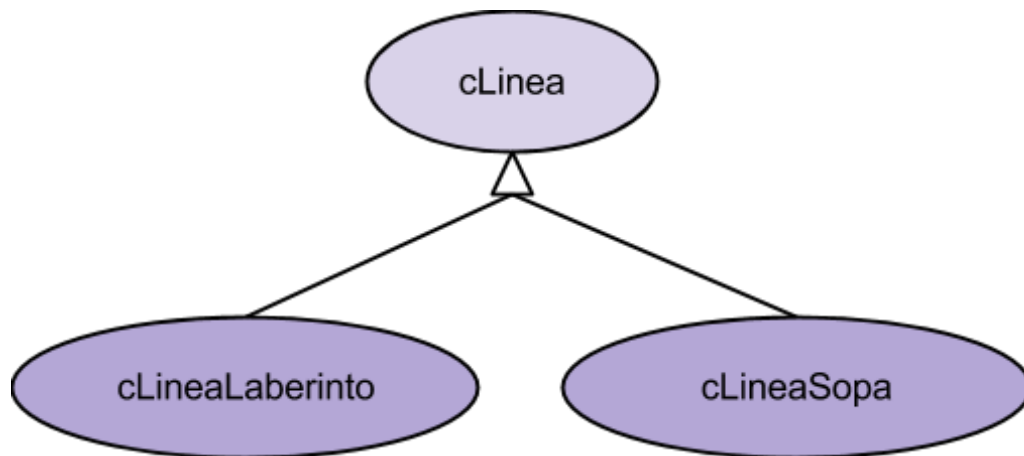
A continuación se muestra un ejemplo del uso de la clase estática *cFicheroLaberinto*:

```
cLaberinto lab_prueba = new cLaberinto();
cFicheroLaberinto f1 = new cFicheroLaberinto(lab_prueba);
cFicheroLaberinto f2;

// Se guarda el laberinto en el fichero "prueba.txt"
f1.setFichero("prueba.txt");
lab_prueba.cargarLaberintoEnFichero();

// Guardamos el contenido del fichero "prueba2.txt" en un nuevo laberinto
f2 = new cFicheroLaberinto("prueba2.txt");
f2.cargarFicheroEnLaberinto();
cLaberinto lab_prueba2 = f2.getLaberinto();
```

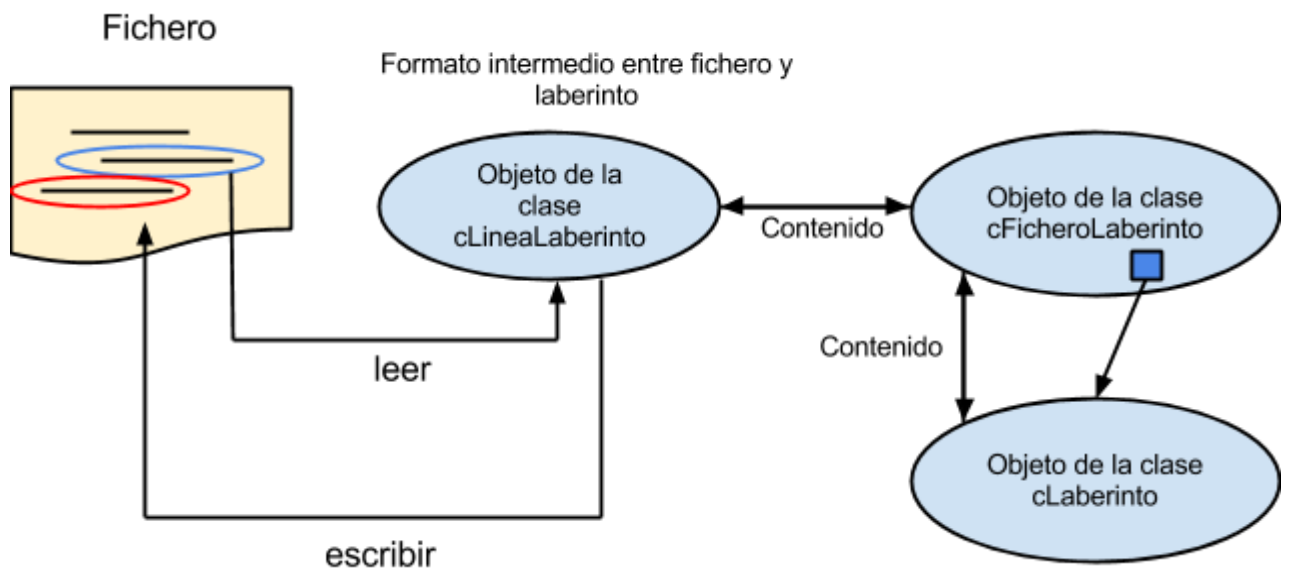
Clase cLinea



Como ya se ha mencionado, se utilizan objetos de la clase *cLinea* para ir traspasando la información de la aplicación a ficheros y/o viceversa.

Clase cLineaLaberinto

Los objetos de la clase *cLineaLaberinto* se instancian para almacenar información de una línea del propio fichero, ya sea para pasar el contenido de la línea al laberinto, o bien, escribir el contenido de la línea al propio fichero.



Clase cLineaSopa

Al igual que los objetos de la clase *cLineaLaberinto*, los objetos de la clase *cLineaSopa* se instancian para almacenar información de una línea del propio fichero de sopas, ya sea para pasar el contenido de la línea a la sopa, o bien, escribir el contenido de la línea al propio fichero.

Fichero de ejemplo que contiene un laberinto

```
9 Sin título1370551884732
3 60,60
0 0,1,0,0
0 0,2,1,0
0 0,3,0,0
0 0,4,0,0
0 0,5,0,0
0 0,6,0,0
0 0,7,1,0
0 0,8,1,0
0 0,9,1,0
0 0,10,1,0
0 0,11,1,0
0 0,12,0,0

0 59,56,0,0
0 59,57,0,0
0 59,58,1,0
0 59,59,1,0
2 0,1,2,3,4,5,6,7
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;3,10;2,10;3,11;4,11;5,11;6,11;7,12;6,13;5,13;5,14;6,15;5,16;4,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;3,10;2,10;3,11;4,11;5,11;6,11;7,12;6,13;5,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;3,10;2,10;3,11;4,11;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;3,10;3,11;4,11;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;3,10;4,11;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;4,11;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
1 9,6;8,7;7,7;6,7;6,8;5,9;4,10;5,11;6,11;7,12;6,13;5,14;6,15;5,16;5,17;5,18
```

Nombre

Dimensiones

Posiciones

Soluciones

Fichero de ejemplo que contiene una sopa de letras

```
8 gdfgd
9 7,14
0 0,0,E,false
0 0,1,M,false
0 0,2,J,false
0 0,3,H,false
0 0,4,M,false
0 0,5,Y,false
0 0,6,Ñ,false
0 0,7,G,false
.....
.....
.....
1 XTOUIEW,false
1 HAQXTVVIKÑA,false
1 YBAAFNQNXPT,false
1 EKCWNK,false
```

Nombre

Dimensiones

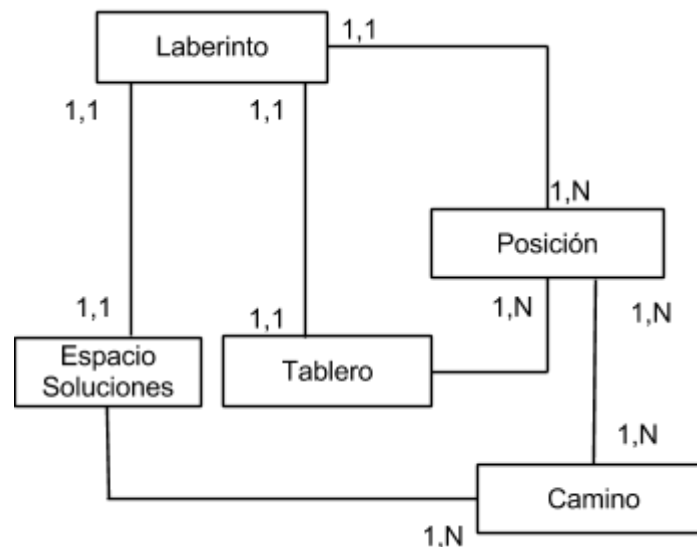
Tablero

Diccionario

5.2.6 ALMACENAMIENTO EN BASES DE DATOS

ESQUEMA RELACIONAL

Los ficheros que almacenan información de laberintos y sopas de letras pueden derivar en el almacenamiento de base de datos relacional que seguiría un esquema como el siguiente:



ALMACENAMIENTO Y PORTABILIDAD EN FICHEROS XML

El esquema relacional de la figura anterior, da una visión bastante precisa de cómo podrían almacenarse los laberintos en ficheros XML mediante un árbol de nodos asociados a dichos objetos con esa jerarquía.

```
<laberinto nombre="nombre del laberinto">
    <tablero dimension1="dimension1" dimension2="dimension2">
        <posicion tipo="tipo" x="x" y="y"/>
        ...
        <posicion tipo="tipo" x="x" y="y"/>
    </tablero>
    <espacioSoluciones>
        <camino>
            <posicion x="x" y="y"/>
            ...
            <posicion x="x" y="y"/>
        </camino>
        ...
        <camino>
            ...
        </camino>
    </espacioSoluciones >
</laberinto>
```

Y de la misma forma para las sopas de letras.

```
<sopa nombre="nombre">
...
</sopa>
```

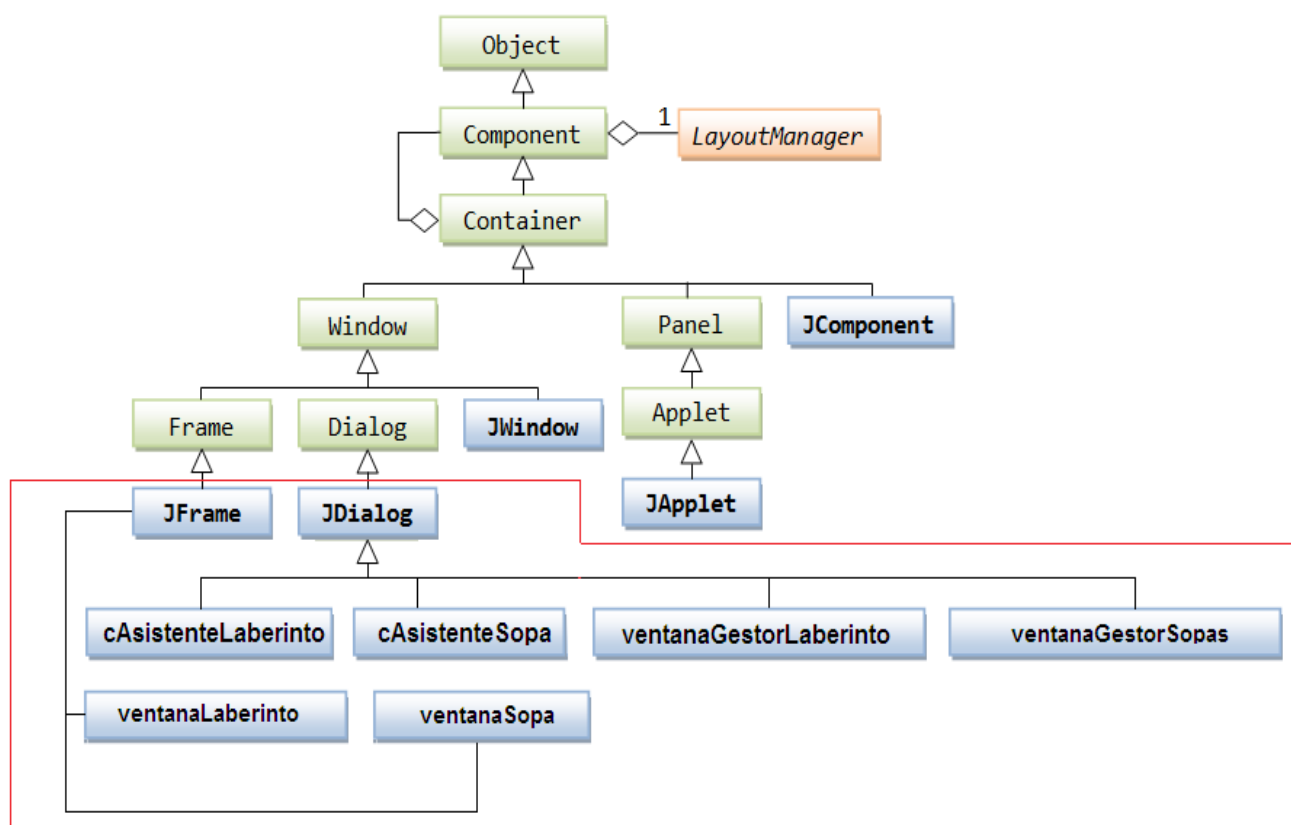
Este tipo de ficheros facilita el hecho de importar y exportar información de una base de datos relacional, mediante herramientas de uso extendido de SQL, o librerías de aplicación como el DOM Parser de Java.

5.3 INTERFAZ GRÁFICA DE USUARIO

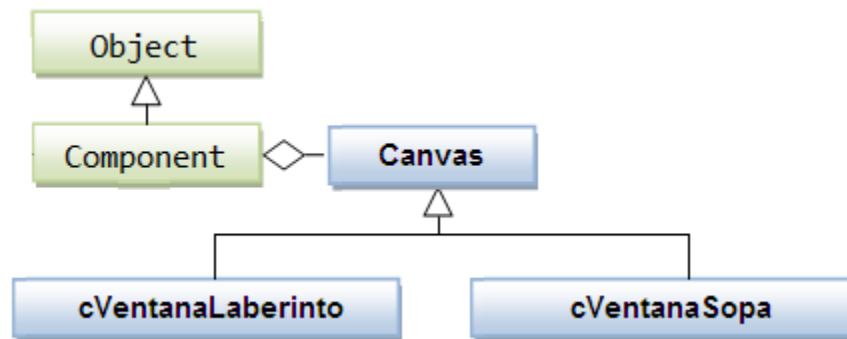
En este apartado se muestran los componentes del proyecto encargados de mostrar una interfaz al usuario, y se definen e identifican las clases y objetos que conforman esta parte de la aplicación.

- Clases que implementan las ventanas gráficas.
- Clases que controlan y vinculan la información de las estructuras de datos de los laberintos a la interfaz gráfica (en cierto modo sigue el modelo-vista-controlador).
- Las clases de control y de visualización gráfica funcionan del mismo modo tanto para laberintos como para sopas de letras, por lo que para no caer en la redundancia solo basta explicarlo para uno de ellos.

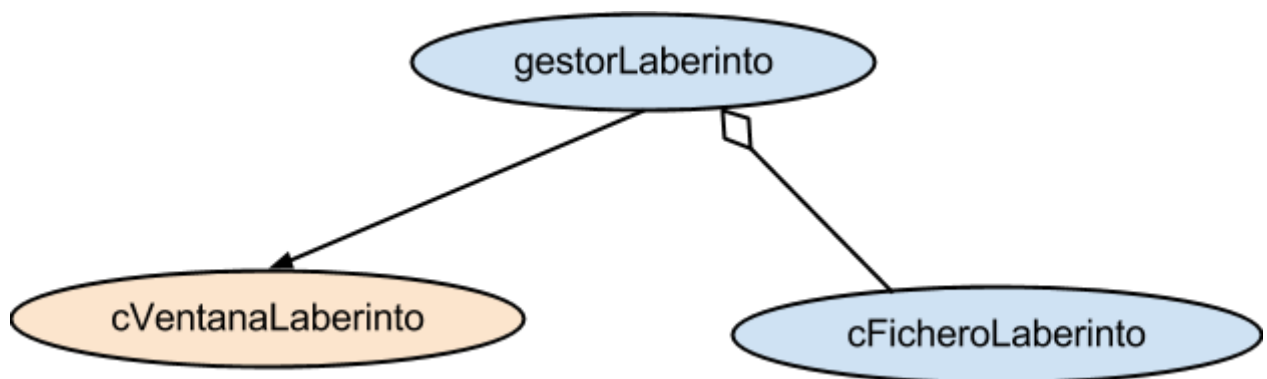
Para el desarrollo de las ventanas de diálogo e interacción con el usuario, se han utilizado la librería de Java JDialog y JFrame que permite crear ventanas con diversos controles para manejar información y dar lugar a la gestión de la aplicación (pudiendo utilizar imágenes). Con ella se extienden las clases que gestionan la interfaz, se muestra a continuación el esquema correspondiente de las clases:



Para el desarrollo de las ventanas que muestran laberintos y sopas de letras, y también permiten editarlos, se han utilizado clases que heredan la funcionalidad gráfica de la librería Canvas de Java.

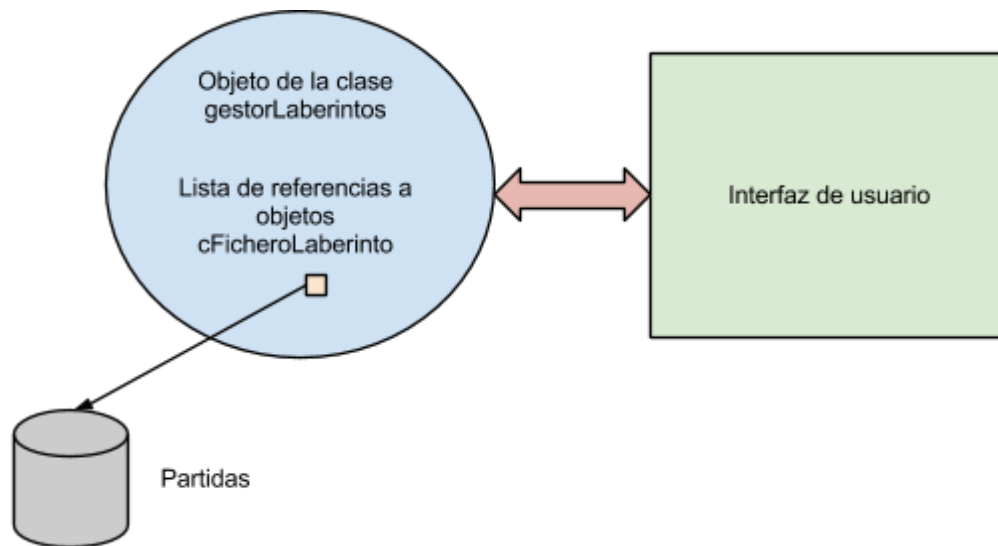


Clase gestorLaberinto



Esta clase es la encargada de gestionar la información referente a los laberintos que se estén utilizando en el sistema, es decir, almacena de forma interna una lista de objetos de la clase `cFicheroLaberinto`, en dicha lista se irán generando los laberintos que el usuario solicite, se podrán borrar, editar, importar o exportar a ficheros externos.

Los objetos de esta clase también se encargan de realizar las operaciones solicitadas por el usuario a través de la interfaz sobre los laberintos.



Las partidas almacenan toda la información de los laberintos, y dichas partidas son exportables a fichero externos, cuya información a su vez es perfectamente almacenable en una BBDD por ejemplo SQL.

Constructores

- Constructor sin especificar ningún parámetro: Inicializa un nuevo gestor de laberintos con su lista de objetos de la clase *cFicheroLaberinto* vacía y sin una interfaz gráfica asignada.
- Constructor especificando una interfaz: Inicializa un nuevo gestor de laberintos con su lista de objetos de la clase *cFicheroLaberinto* vacía con la interfaz gráfica indicada.

Métodos

- Métodos para agregar, cambiar o borrar laberintos, mediante la lista de objetos de la clase *cFicheroLaberinto*.
- Método para llamar a la exploración de un laberinto: Especificado un elemento de la lista que se gestiona, hace una llamada al objeto de la clase *cFicheroLaberinto* correspondiente y éste a su vez envía un mensaje a su objeto laberinto asociado para que comience con la exploración para encontrar soluciones.

- Métodos para importar y exportar laberintos entre la aplicación y ficheros externos.

La interfaz gráfica se corresponde a un objeto de la clase *ventanaGestorLaberintos* que implementa los elementos gráficos y la ventana que los contiene para manejar los laberintos.

Clase cVentanaLaberinto

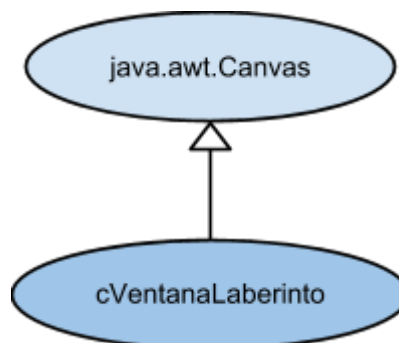
Implementa una ventana de diálogo para que el usuario gestione la lista de laberintos. Está asociada a un controlador (objeto de la clase *gestorLaberintos*) para manipular el modelo de datos a través de él.

Esta clase implementa los métodos necesarios para vincular el contenido de los laberintos al cual su controlador tiene acceso, a sus controles de interfaz gráficos (botones, listas, etc).

Desde esta ventana el usuario tendrá acceso a:

- Generar nuevos laberintos.
- Importar laberintos generados previamente a partir de ficheros externos.
- Exportar laberintos a ficheros externos.
- Configurar el contenido de los laberintos de forma gráfica.
- Explorar y solucionar los laberintos generados.

Siempre que un laberinto sea importado o exportado, lo hará incluyendo su espacio de soluciones (siempre y cuando no esté vacío).



Esta clase es la interfaz gráfica o capa de presentación de un laberinto. Hereda de la clase Canvas de Java la funcionalidad para poder trabajar con ventanas de forma gráfica.

Los objetos de esta clase constan de un frame que es la ventana que contiene el contenido gráfico, contiene una referencia al laberinto (objeto de la clase *cLaberinto*) que representan, y el frame tendrá unas dimensiones en función del tamaño del laberinto.

Los objetos de esta clase se utilizan para que el usuario pueda interactuar de forma directa con el contenido de un laberinto de forma gráfica sin necesidad de tener que utilizar coordenadas ni técnicas parecidas para manipularlo.

A través de ella podrá:

- Visualizar los laberintos generados.
 - Visualizar contenido
 - Visualizar soluciones
- Editar el contenido de los laberintos.
 - Posiciones obstáculo
 - Posiciones hueco
- Editar las posiciones de entrada y salida.
- Ajustar el zoom para el laberinto a la hora de editarlo.

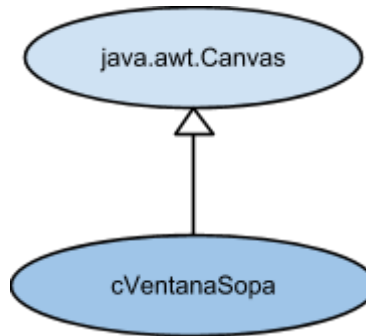
Clase gestorSopa

Al igual que gestorLaberintos se encarga (del mismo modo) de gestionar toda la información referente a las sopas de letras e interacción entre el usuario y las sopas.

Clase cVentanaSopa

Al igual que cVentanaLaberinto implementa una ventana de diálogo para que el usuario gestione la lista de sopas de letras.

Está asociada a un controlador (objeto de la clase *gestorSopas*) para manipular el modelo de datos a través de él.



Esta clase implementa los métodos necesarios para vincular el contenido de los sopas de letras al cual su controlador tiene acceso, a sus controles de interfaz gráficos (botones, listas, etc).

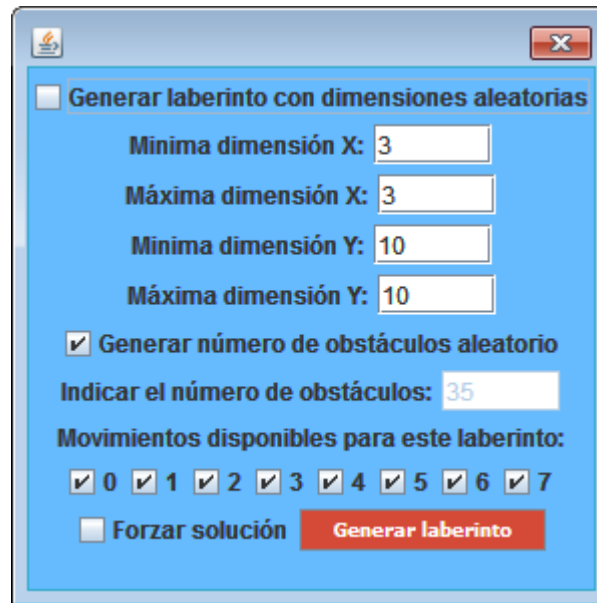
Desde esta ventana el usuario tendrá acceso a:

- Generar nuevas sopas de letras.
- Importar sopas de letras generados previamente a partir de ficheros externos.
- Exportar sopas de letras a ficheros externos.
- Configurar el contenido de las sopas de letras de forma gráfica.
- Mostrar las palabras-solución del diccionario en la sopa de letras.

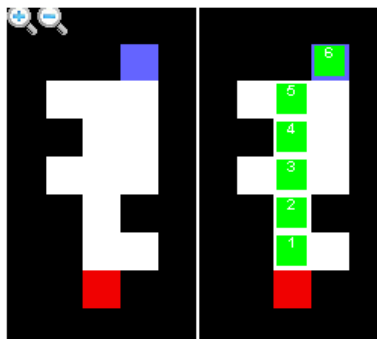
6. EJECUCIÓN Y RESULTADOS

A continuación se muestran una serie de resultados de la ejecución de la aplicación con distintos parámetros de configuración.

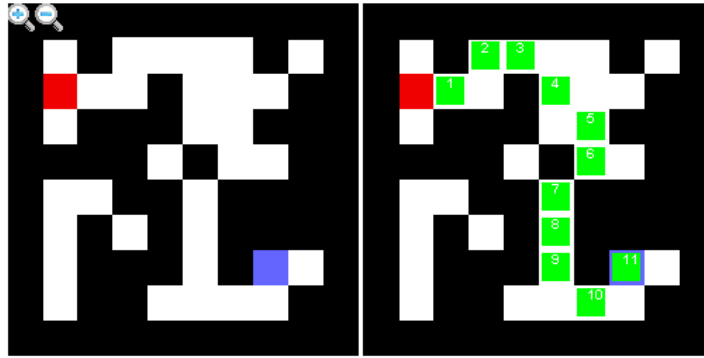
GENERAR LABERINTOS



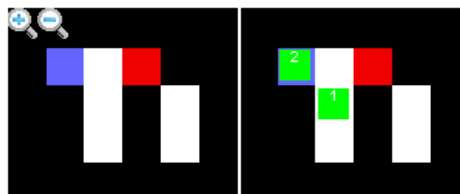
GENERAR LABERINTOS SIN ESPECIFICAR PARÁMETROS



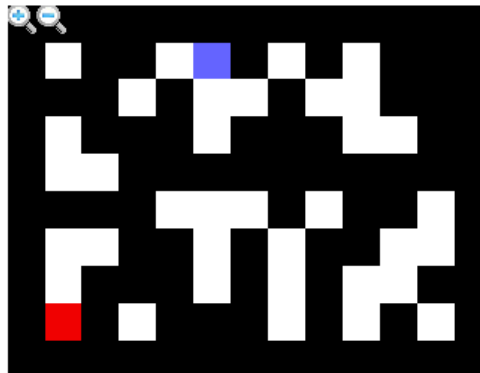
Laberinto auto-generado con solución.



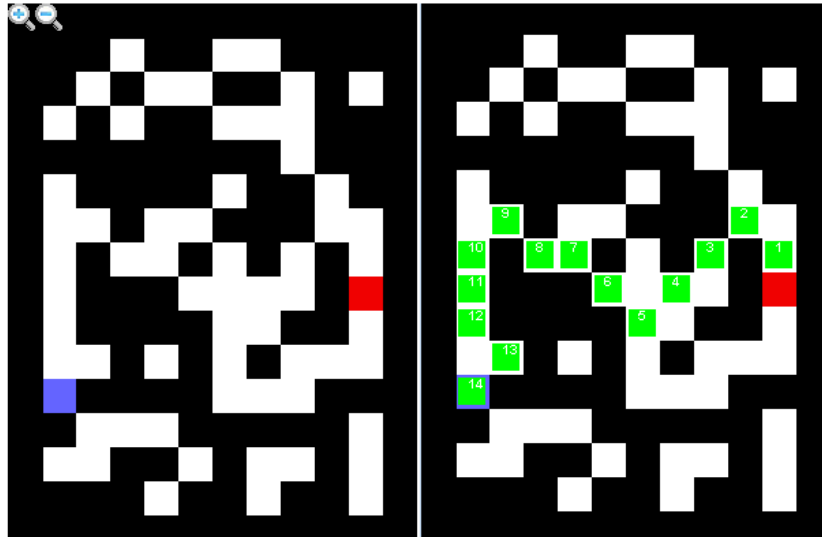
Laberinto auto-generado con solución.



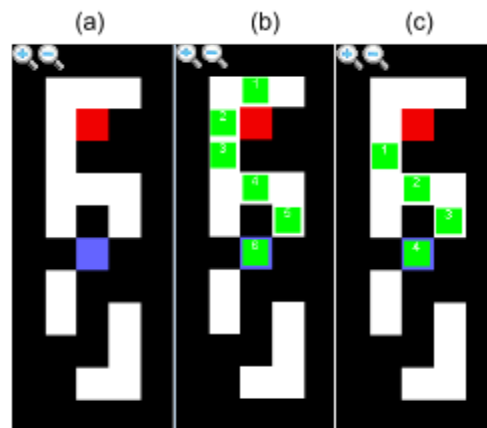
Laberinto auto-generado con solución.



Laberinto auto-generado sin solución.



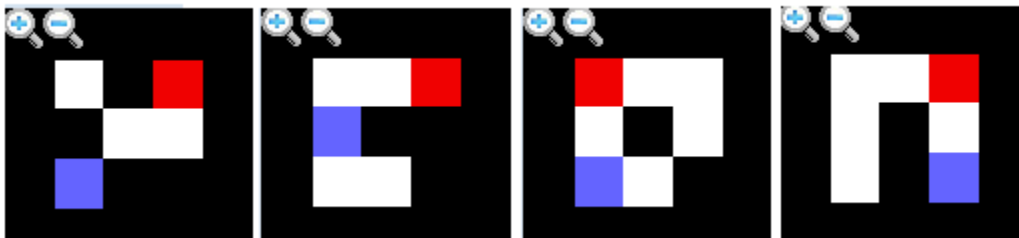
Laberinto auto-generado con solución.



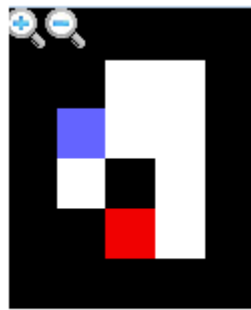
Laberinto con una solución cualquiera (b) y una solución óptima.(c)

GENERAR LABERINTOS ESPECIFICANDO PARÁMETROS

- Especificando dimensiones 3x3



- Especificando una dimensión constante, $X=4$

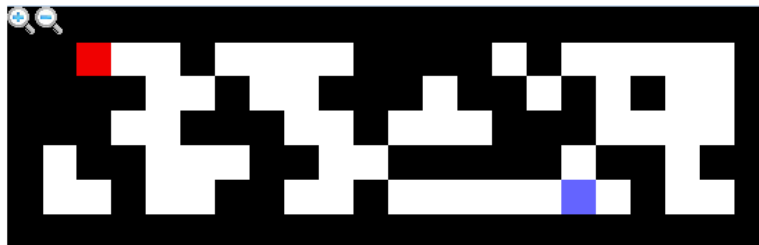


4x3

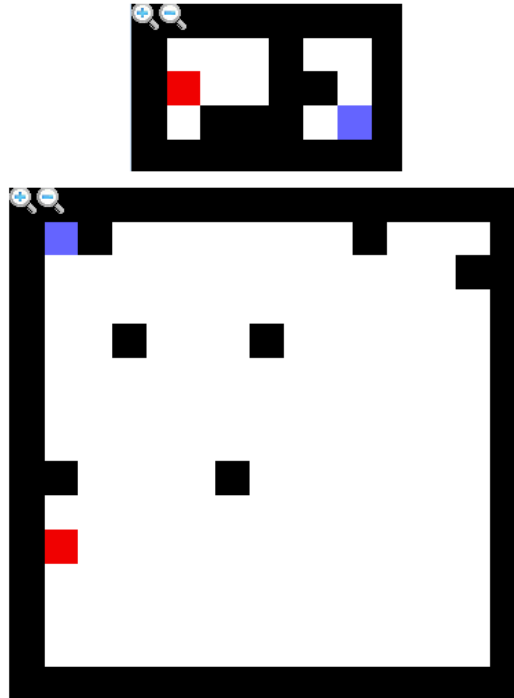


4x10

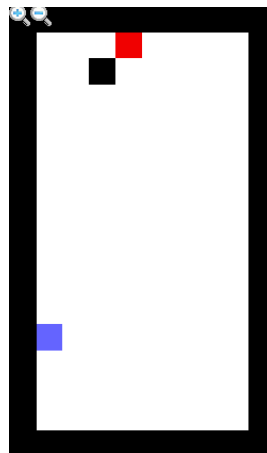
- Especificando $5 \leq X \leq 8$, $Y = 20$



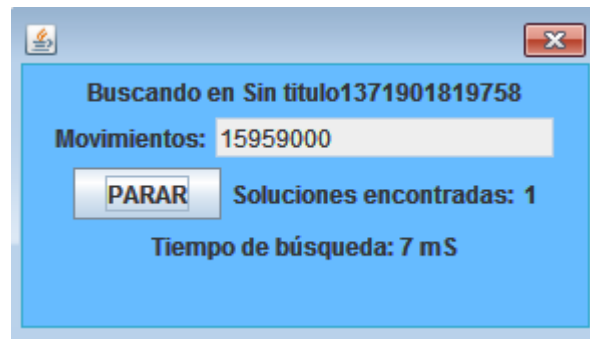
- Especificando el número de obstáculos = 7



- Especificando el número de obstáculos = 1



Monitor de resolución de laberintos con árboles de nodos inmensos

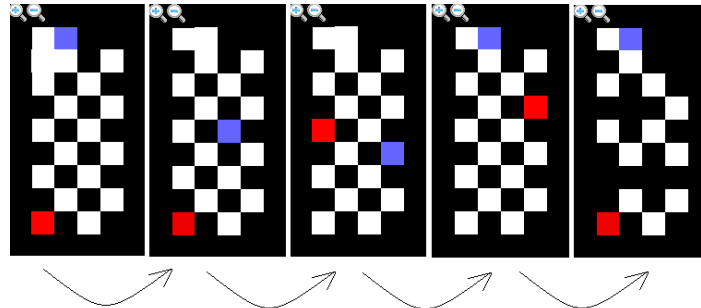


APLICACIÓN DE ZOOM EN EDICIÓN DE LABERINTO



EDICIÓN DE LABERINTOS

Cambio de las posiciones de entrada y salida y de algunos obstáculos



CREACIÓN DE SOPAS DE LETRAS

Especificando dimensiones 8x8

G	J	A	A	Ñ	C	O	I
K	W	H	P	X	I	S	Ñ
O	Y	G	F	G	N	L	O
L	B	Q	S	Q	K	J	T
D	A	G	A	E	V	S	C
S	U	E	R	Q	P	F	E
O	N	A	C	Q	C	Ñ	G
O	B	N	E	W	A	I	J

A	X	Y	Q	O	I	P	Y
F	G	A	V	L	A	P	Z
Y	P	T	I	J	S	F	G
C	Ñ	P	Z	Ñ	X	D	R
Ñ	M	Y	F	N	D	Z	T
U	B	M	H	D	F	N	Ñ
F	H	L	K	O	H	L	Q
J	E	J	Ñ	I	P	J	J

Edición del contenido de una sopa

A	X	Y	Q	O	I	P	Y
F	G	A	V	L	A	P	Z
Y	P	H	D	J	S	F	G
C	Ñ	P	O	L	X	D	R
Ñ	M	Y	F	L	A	Z	T
U	B	M	H	D	F	N	Ñ
F	H	L	K	O	H	L	Q
J	E	J	Ñ	I	P	J	J

A	X	Y	Q	O	I	P	Y
F	G	A	V	L	A	P	Z
Y	P	H	D	J	S	F	G
C	Ñ	P	O	L	X	D	R
Ñ	M	Y	F	L	A	Z	T
U	B	M	H	D	A	N	Ñ
F	H	L	K	O	H	L	Q
J	E	J	Ñ	I	P	J	J

Resolución de sopa de letras

X	Q	Q	M	Ñ	B	O	U	L
P	A	P	A	T	M	Y	E	P
A	X	S	I	V	A	Y	V	O
F	E	N	J	G	R	A	N	J
Q	T	N	Ñ	R	Y	Ñ	N	A
Z	A	W	Z	Q	J	I	K	V
Ñ	P	Q	Y	A	C	V	A	D
G	F	L	J	V	B	C	Y	A
G	F	B	T	S	W	V	R	T

Añadir palabra

Borrar seleccionada

GRAN

6. CONCLUSIONES

Los requisitos iniciales que se presentaban tras la elección del proyecto se han cumplido en su mayoría si son revisados uno a uno, lo que se puede considerar un éxito.

El volver a programar en Java después de haber trabajado en los últimos años en el entorno .NET, HTML y JavaScript, me ha servido de refresco para los conocimientos en ese lenguaje y en las herramientas de desarrollo empleadas como Eclipse, aparte de ser una dificultad añadida.

Además, la realización del proyecto me ha permitido adquirir conocimientos adicionales a los obtenidos durante la carrera, sobre todo de Java y de desarrollo de elementos gráficos. He aprendido a utilizar diferentes técnicas de forma conjunta que en la carrera suelen verse más aisladas, como combinar algoritmos de búsqueda, con el uso de clases y objetos, y a la vez con hilos y procesos concurrentes, teniendo que buscar estrategias y métodos para el correcto funcionamiento de todo a la vez, controlado a su vez por una interfaz gráfica.

El desarrollo me ha beneficiado como futuro ingeniero ya que he tenido que buscar soluciones a problemas como por ejemplo parar procedimientos recursivos que ya han lanzado millones de ejecuciones recursivas de sí mismos.

El resultado final en cuanto a la interfaz no ha sido perfecto, ya que he tenido difícil poder encontrar el tiempo suficiente seguido por cuestiones personales y de trabajo, y mi experiencia previa en este campo se basaba más bien en ficheros XML, XAML, HTML, etc. desde entornos que tratan mejor el aspecto gráfico. Aunque con el resultado me he dado cuenta que tendré que investigar y estudiar el uso de tecnologías como OpenGL para desarrollos gráficos más potentes.

7. BIBLIOGRAFÍA

[1] Ben-Ari, Ragonis, Ben-Bassat, 2002

C. Gregorio Rodríguez, L.F. Llana Díaz, R. Martínez Unanue (2002). Creativos y recreativos en C++.

Fco Javier Martínez Juan (2011). Construcción de Software Java con Patrones de Diseño.

Allen Weiss Mark (2008). Estructuras de datos en Java.

Menchén Peñuela, Antonio (2010). Diseño de programas.

7.1 BIBLIOGRAFÍA WEB

Estructuras de datos y algoritmos en java. Ediciones Paraninfo, www.paraninfo.com

Marcos Ortega Montenegro, Tesis sobre juegos informáticos.

<http://www.tesis.ufm.edu.gt/pdf/3400.pdf>

Oracle, librerías para ventanas Swing.

<http://docs.oracle.com/javase/tutorial/uiswing/>

Documento Oficial de Oracle, Canvas.

<http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/Canvas.html>

Pau Aguilar Fruto, IA - Algoritmos de juegos (2008).

http://www.lsi.upc.edu/~bejar/ia/material/trabajos/Algoritmos_Juegos.pdf

Serie de entradas y teoría sobre laberintos.

<http://eltopologico.blogspot.com.es/2007/02/laberintos-parte-2.html>

Ejemplo de solucionador de sopas de letras en Java.

<http://codesandtags.org/solucionador-sopa-de-letras-en-java/>

J.M. Bilbao, F.R. Fernández. Avance en teoría de juegos con aplicaciones económicas y sociales. <http://www.esi2.us.es/~mbilbao/pdf/files/libro.pdf>