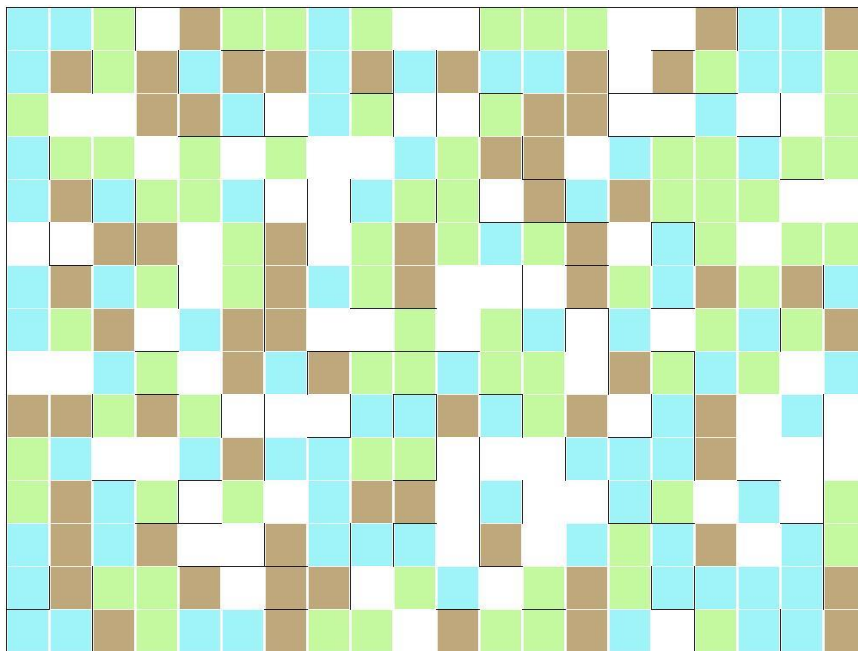


SISTEMAS INTELIGENTES – LABORATORIO

Documentación

‘Laberintos informados y Algoritmo de búsqueda’



Grupo B1-6:

- González Bermúdez, David – David.Gonzalez28@alu.uclm.es
- Gutiérrez Durán, Lucas – Lucas.Gutierrez@alu.uclm.es
- Gutiérrez Mariblanca, David – David.Gutierrez5@alu.uclm.es

Contenido

1. Tarea 1 – Generación del Laberinto	3
1.1 Introducción	3
1.1.1 Objetivo de la tarea	3
1.1.2 Lenguaje utilizado	3
1.1.3 Herramientas utilizadas	4
1.2 Generación del laberinto	4
1.2.1. Algoritmo de Wilson	4
1.3 Desarrollo de la tarea	5
1.3.1. Clases y métodos desarrollados	5
1.3.2. Problemas durante el desarrollo	7
1.4 Horas de trabajo	8
2. Tarea 2 – Definición del problema y las estructuras para el árbol de búsqueda	9
2.1 Introducción	9
2.2 Desarrollo de la tarea	9
2.2.1 Clases Node y Sucesor	10
2.2.2 Clases Border y Problem	10
2.2.3 Métodos creados en otras clases	11
2.3 Horas de trabajo	11
3. Tarea 3 – Laberintos informados y algoritmo de búsqueda	12
3.1 Introducción	12
3.2 Algoritmo de búsqueda	12
3.3 Desarrollo de la tarea	13
3.3.1 Cambios y mejoras en el código	13
3.3.2 Desarrollo de la tarea	14
3.3.3 Problemas en el desarrollo	16
3.4 Horas de trabajo	17
Anexos	18
Anexo 1: Conclusiones y opiniones de la práctica	18
Anexo 2: Manual de Usuario	21
Anexo 3: Ejemplos de laberintos creados	24

1. Tarea 1 – Generación del Laberinto

1.1 Introducción

En este primer punto se explicará cual era el objetivo de la tarea a realizar y cuales han sido algunas decisiones previas importantes que hemos tenido que tomar para afrontar el proyecto. Todo el código de la práctica está ubicado en el repositorio de GitHub: https://github.com/LuGuDu/B1_6.

1.1.1 Objetivo de la tarea

El enunciado de la tarea puede encontrarse en el Campus Virtual de la UCLM en el curso de Sistemas Inteligentes. Podríamos resumir ese enunciado en los siguientes puntos proporcionados por el profesor de la asignatura Luis Jiménez Linares:

1. Soy capaz de crear un laberinto correcto de un tamaño concreto (número de filas x número de columnas)
2. Soy capaz de salvar ese laberinto como un fichero JSON correcto y una imagen de este.
3. Soy capaz de leer un fichero JSON, validar su semántica y crear el laberinto correspondiente.
4. Soy capaz de crear la imagen del laberinto correcto.

Para la ayuda de la comprobación de la tarea el profesor puso a disposición del alumnado una serie de ficheros JSON con la información de laberintos junto con sus respectivas imágenes para poder realizar una correcta comprobación del programa.

Como detalles a destacar es importante saber que se pidió explícitamente que la imagen se guardara en formato JPG y no en otro como PNG. Además, se exigió la utilización del recurso de Git para una correcta y organizada gestión del proyecto, en nuestro caso lo empleamos por medio de GitHub.

1.1.2 Lenguaje utilizado

En cuanto al lenguaje utilizado para desarrollar el programa nos hemos decantado por Java en su versión 8 para evitar posibles problemas a la hora de la ejecución y compilación de código. Además, Java es el lenguaje en el que todos los componentes del grupo nos sentíamos más seguros ya que este es en el que se nos han enseñado las bases de la programación desde el comienzo de la carrera universitaria y lo conocemos mejor que otros como C o Python.

1.1.3 Herramientas utilizadas

La librería estándar de Java 8 nos ha proporcionado muchas herramientas para poder llevar a cabo el proyecto, sin embargo, hemos tenido que emplear una librería externa para poder realizar la manipulación de los archivos JSON de forma rápida y cómoda.

En concreto esa librería es gson en su versión 2.8.2. Gson es una biblioteca de Java de código abierto que permite la serialización y deserialización entre objetos Java y su representación en notación JSON. Esta biblioteca ha sido desarrollada por Google y puede ser conseguida desde internet con facilidad.

El IDE (Entorno de desarrollo integrado) que hemos utilizado todos los componentes del grupo ha sido Eclipse. Este IDE nos ha proporcionado comodidad a la hora de desarrollar el programa, facilidad a la hora de la compilación del código, y rapidez para poder visualizar los fallos cometidos.

Por último, como ya hemos comentado antes, hemos empleado la herramienta de GitHub para gestionar los repositorios de nuestro proyecto. GitHub nos ha proporcionado una claridad a la hora de gestionar las versiones de nuestro programa, además de una buena gestión del trabajo realizado por cada uno de los componentes del grupo, de las tareas a realizar por medio de una planificación Kanban, entre otras cosas.

Por último, hemos utilizado otros recursos como WhatsApp como principal medio de comunicación entre nosotros, además de llamadas vía Microsoft Teams para poder comentar y desarrollar en grupo.

1.2 Generación del laberinto

En nuestro programa, para generar laberintos aleatorios, partimos del número de filas y el número de columnas proporcionados por el usuario mediante el teclado. Una vez hecho esto se crea un laberinto de esas dimensiones, pero con la particularidad de que ninguna de sus celdas tiene ningún vecino.

1.2.1. Algoritmo de Wilson

Una vez generado el laberinto comentado en el párrafo anterior hacemos uso del 'Algoritmo de Wilson'. Este algoritmo tiene el siguiente funcionamiento: El algoritmo se inicia escogiendo una celda aleatoria del laberinto y después selecciona otra también de forma aleatoria. La primera celda se pondrá como visitada, y luego de esto desde la segunda celda se traza un camino aleatorio hasta llegar a una celda visitada (en esta primera iteración solo hay una celda visitada). En caso de aparecer un bucle se elimina y se sigue haciendo el camino aleatorio a partir de la celda donde se generó el bucle.

Ejemplo de secuencia con bucle: (0, 1), (1, 1), (1, 2), (2, 2), (2,1), (1,1).

En este caso la celda que se repite es la (1, 1) así que se desecha el camino entero hasta llegar al primer (1, 1) que generó el bucle.

Ejemplo de secuencia después de detectar y arreglar el bucle: (0, 1), (1, 1).

Una vez alcanzada la casilla este camino es registrado y todas las casillas de este camino serán visitadas, por lo tanto, en la siguiente iteración tendrá muchas más posibilidades de encontrar una casilla visitada en un menor tiempo que en la primera iteración.

Una vez hecho esto se escoge otra celda aleatoria no visitada y se genera un camino aleatorio hasta una celda ya visitada. De modo que este proceso se tiene que repetir hasta que no quede ninguna celda sin visitar en todo el laberinto.

1.3 Desarrollo de la tarea

En este tercer punto hablaremos acerca de cómo ha transcurrido el desarrollo del proyecto y explicaremos con detalles algunos puntos más concretos del programa. También hablaremos sobre problemas que hemos encontrado durante del desarrollo, dudas que hemos tenido que plantear al profesor de la asignatura y otras cuestiones de interés con respecto al desarrollo.

Es importante destacar que como grupo tomamos la decisión de escribir todo el código en inglés ya que en años anteriores los profesores nos lo recomendaban así y es lo ideal. A pesar de esto los comentarios añadidos por los creadores y la impresión por consola sí que es en español para una mejor comprensión.

1.3.1. Clases y métodos desarrollados

En nuestro proyecto tenemos ocho clases, que son las siguientes:

- *Cell.java*
- *DrawLab.java*
- *Funcionts.java*
- *Labyrinth.java*
- *Principal.java*
- *ReadJson.java*
- *WilsonAlgorithm.java*
- *WriteJson.java*

A continuación, procedemos a explicar de forma breve cada una de estas clases y los métodos que contienen en su interior, pero es muy importante tener en cuenta que dentro de cada clase se puede encontrar documentación interna que detalla con más precisión cual es la funcionalidad de la misma clase y de cada método.

1.3.1.1. Clase Principal

La clase Principal consiste únicamente de un método *main* que controla el menú del programa. Por medio de un *switch* llamamos a las diferentes funcionalidades que ofrece el programa. También se han creado dos excepciones para controlar la entrada por teclado del usuario, una para controlar la entrada de números negativos y otra para caracteres que no sean numéricos.

1.3.1.2. Clases Cell y Labyrinth

Las clases *Cell* y *Labyrinth* se encargan de definir los dos objetos más importantes que vamos a manipular durante todo el programa que son el laberinto y sus correspondientes celdas. Tanto el objeto *Cell* como el objeto *Labyrinth* tiene sus atributos específicos con sus correspondientes *getters* y *setters* además de un método *ToString*. Cabe destacar que el objeto *Labyrinth* tiene un atributo especial llamado *cells*. Este atributo es de tipo *Map<String, Cell>* y en su interior estarán contenidas todas las celdas del laberinto en cuestión.

1.3.1.3. Clases ReadJson y WriteJson

La clase *ReadJson* tiene el método *readJson* que es el encargado de pedir por teclado la ruta de un archivo JSON, leerlo y *parsear* toda la información de éste a un objeto laberinto por medio del método *fromJson* de la librería *gson*. Se han controlado diferentes aspectos por medio de excepciones como que la ruta del archivo no exista o que el archivo elegido no sea un JSON. También se controlan posibles errores sintácticos dentro del archivo JSON.

La clase *WriteJson* tiene por objetivo hacer lo contrario a la clase anterior. Esta contiene el método *writeJson* que se va a encargar de *parsear* un objeto laberinto en un archivo JSON empleando el método *toJson* de la librería *gson* y lo va a guardar en el escritorio del usuario con un nombre establecido por el usuario.

1.3.1.4. Clase Functions

La clase *Functions* contiene métodos de carácter general que utilizamos en diferentes partes del proyecto:

- El método *getCellsFromMap* es utilizado para extraer el objeto celda que se encuentra dentro de un tipo *Map<String, Cell>*.
- El método *genLab* es el encargado de llamar a los diferentes métodos necesarios para la generación del laberinto aleatorio.
- El método *saveLab* tiene como función guardar el laberinto en dos archivos diferentes, un JSON y un JPG. Ambos serán guardados en el escritorio con el nombre que quiera el usuario siempre teniendo en cuenta las posibles excepciones.

- El método *checkSemantic* tiene como parámetro de entrada un laberinto y tendrá como función el determinar si ese laberinto tiene una semántica correcta o no. En caso de que la semántica sea incorrecta el programa lanzara un mensaje y no se podrá leer el laberinto.

1.3.1.5. Clase *DrawLab*

La clase *DrawLab* se encarga del dibujado del laberinto y de su guardado en formato JPG. Los métodos más importantes de esta clase son *drawLab*, cuyos parámetros de entrada son el laberinto a dibujar y un nombre con el que se guardará el archivo, el método *drawNeighbours* que se encarga de dibujar en la imagen las paredes que tiene una celda en concreto, y el método *writelnImage* cuyos parámetros son la imagen a guardar y el nombre con el que se guardará dicha imagen en formato JPG.

1.3.1.6. Clase *WilsonAlgorithm*

La clase *WilsonAlgorithm* tiene dos métodos principales que tendrán por objetivo el transformar un laberinto vacío a otro con caminos. El laberinto que recibe es similar a una cuadrícula, ya que ninguna de sus celdas tiene ningún vecino, luego todo son paredes. Para realizar la transformación en el laberinto aplicamos el algoritmo de Wilson explicado anteriormente.

El método Wilson, al que pasamos el laberinto, tiene en su interior un bucle *while* que a su vez tiene otro anidado. En el cuerpo de este primer bucle se generará un camino y justo después empieza el bucle anidado. En este segundo *while* se transformará todo ese camino en celdas visitadas, y para esto se vaciarán las dos pilas que contienen el camino. También nos ayudamos de un *LinkedHashMap* donde guardamos todas las celdas (que habremos puesto en visitado anteriormente). El primer bucle iterará hasta que el tamaño de este *LinkedHashMap* sea igual que el *Map cells* del objeto *Labyrinth* para garantizar que todas las celdas estén visitadas.

En caso de que se produzca un bucle se guarda la posición de la celda donde comenzó el mismo y se retrocederá hasta volver a llegar al punto de inicio del bucle. Entonces se elegirá aleatoriamente otro vecino para seguir con el procedimiento.

Hemos creado también dos métodos llamados *fillNoVisitedKeys* y *pushCell*. El primero rellena la lista *noVisitedKeys* con todas las *keys* de un laberinto, y el segundo añade un *String* a la pila de *String* y un *Cell* a la pila de *cell*. Estos métodos surgen de la modularización de tareas que hace esta clase para una correcta y cómoda manipulación del código.

1.3.2. Problemas durante el desarrollo

A lo largo del desarrollo nos hemos enfrentado a diferentes problemas que nos han complicado el flujo de trabajo. Por ejemplo, añadir más funcionalidades de las pedidas, o el

tiempo requerido a la comprensión del algoritmo Wilson y su paso al lenguaje Java aplicándolo a nuestro problema concreto.

En específico hubo problemas en el hecho de plantear una u otra estrategia para hacer el algoritmo (iterativo, recursivo), y en la resolución de bucles ya que después de sacar las casillas que conformaban un bucle podía haber una situación inconsistente con vecinos que no son vecinos. Además de que no todas las situaciones en las que se produce un bucle son iguales y eso complica su tratamiento. Ejemplo de dos situaciones diferentes: bucle que abarca el camino completo y bucle que abarca solo un trozo del camino.

También surgieron diferentes problemas a la hora de manipular los archivos JSON.

Otro de los problemas fue comenzar la tarea con la visión de crear una interfaz gráfica para el mismo. Esto ha provocado una pérdida de horas considerable teniendo en cuenta que toda esa funcionalidad tuvo que ser removida por decisión del grupo de cara a futuras modificaciones del programa.

En general, a pesar de los problemas que han ido surgiendo, gracias a las horas dedicadas, a las búsquedas de información por medio de Internet y a la ayuda proporcionada por el profesor en caso de dudas, hemos podido seguir adelante y terminar la primera tarea.

1.4 Horas de trabajo

A continuación, vamos a desglosar el tiempo dedicado en esta tarea. Para ello vamos a escribir la actividad y el tiempo que ha supuesto. Tiempo dedicado a:

- Lectura y manipulación de archivos JSON
 - o David González Bermúdez – 10 horas
- Dibujado de laberintos
 - o Lucas Gutiérrez Durán – 10 horas
- Algoritmo de Wilson
 - o David González Bermúdez – 30 horas
 - o Lucas Gutiérrez Durán – 4 horas
 - o David Gutiérrez Mariblanca – 16 horas
- Comprobación de semántica
 - o David Gutiérrez Mariblanca – 6 horas
- Control de pruebas
 - o Lucas Gutiérrez Durán – 6 horas
- Documentación (PDF y código)
 - o David González Bermúdez – 1 horas
 - o Lucas Gutiérrez Durán – 5 horas
 - o David Gutiérrez Mariblanca – 1 horas
- Otras cuestiones (Aprendizaje GitHub, otras funcionalidades, horas de clase, ...)
 - o Todos los miembros – 12 horas

Tiempo total *aproximado* dedicado a la tarea 1: 125 horas

2. Tarea 2 – Definición del problema y las estructuras para el árbol de búsqueda

2.1 Introducción

En esta segunda tarea se pedía como objetivo el definir e implementar un artefacto software problema-salir-del-laberinto y todos los necesarios para implementar la creación y el manejo del árbol de búsqueda.

El enunciado de la tarea puede encontrarse en la página de Campus Virtual de la asignatura de Sistemas Inteligentes.

Con respecto a la tarea anterior ha habido algunos cambios relevantes en el código, como la redimensión gráfica de las celdas para poder generar laberintos de un mayor tamaño o una mejor organización del código por medio de paquetes. Esos paquetes contienen código relacionado con ciertas tareas como la creación del laberinto o la búsqueda de una solución. También se ha cambiado de sitio el método checkSemantics() a la clase ReadJson.java ya que pensamos que era más apropiado que estuviese ahí. Y por último hemos creado también dos clases nuevas que contienen excepciones que antes estaban dentro de otras clases.

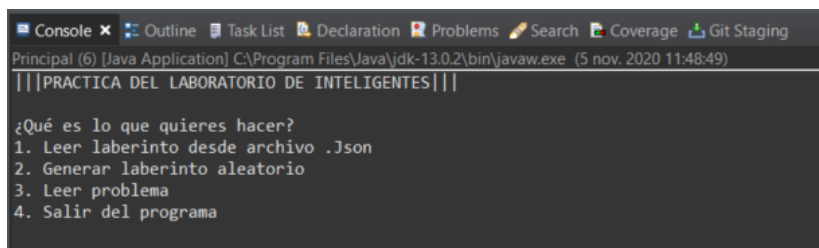
Como comprobaciones de la tarea se exige que nuestro programa pueda:

1. Construir manualmente los JSON para generar y leer problemas (JSON) con los laberintos suministrados en la tarea anterior.
2. Generar de forma aleatoria artefactos nodos.
3. Insertar dichos nodos en la frontera para comprobar su inserción correcta.

2.2 Desarrollo de la tarea

Para la elaboración de esta segunda tarea hemos creado clases nuevas y añadido métodos en clases que ya habíamos creado en la tarea anterior.

Hemos añadido una nueva funcionalidad en la interfaz por consola que permite leer un problema. Dicha opción pedirá la ruta de un archivo JSON que *PREFERIBLEMENTE SE ENCONTRÁRA EN EL ESCRITORIO DEL USUARIO*.



```
Principal (6) [Java Application] C:\Program Files\Java\jdk-13.0.2\bin\javaw.exe (5 nov. 2020 11:48:49)
|||PRACTICA DEL LABORATORIO DE INTELIGENTES|||

¿Qué es lo que quieres hacer?
1. Leer laberinto desde archivo .Json
2. Generar laberinto aleatorio
3. Leer problema
4. Salir del programa
```

Este fichero JSON tendrá la siguiente estructura:

```
1 [{"INITIAL": "(0, 0)", "OBJECTIVE": "(9, 9)", "MAZE": "sucesores_10X10_maze.json"}]
```

Donde INITIAL indica la posición de la casilla inicial, OBJETIVE la posición de la casilla objetivo, la que queremos alcanzar, y MAZE el laberinto en el que queremos realizar dicha búsqueda. (ES IMPORTANTE QUE ESTE ARCHIVO TAMBIEN SE ENCUENTRE EN EL ESCRITORIO).

Una vez leído el problema generamos unos nodos de prueba que posteriormente son añadidos a una frontera. Después la frontera es vaciada por orden. Después hacemos otra prueba y es imprimir por pantalla todos los vecinos del nodo "(2,3)". Una vez hecho esto generamos un problema JSON llamado hola que se guardará en el escritorio y contendrá el mismo problema leído anteriormente.

Realmente esta segunda tarea está orientada a crear clases y métodos que nos servirán de manera directa para la implementación de la siguiente tarea. Luego, como se ha podido ver, en esta tarea hemos hecho casos de prueba de las cosas creadas.

A continuación, vamos a detallar de forma concreta las clases y métodos diseñados para esta tarea:

2.2.1 Clases Node y Sucesor

La clase Node.java representa a una celda que se va recorriendo, con sus respectivos atributos en un árbol. Esta clase tendrá como atributos id, costo, su posición, el id del nodo padre, una acción, la profundidad del nodo, la heurística y el valor. Esta clase no tiene métodos característicos más allá de los getters y los setters, aunque hemos creado dos getters para obtener la fila y la columna del nodo.

La clase Sucesor.java únicamente almacena un sucesor de la celda. Tendrá como atributos un movimiento que será N, S, E u O. También tendrá un id y un costo. Esta clase será útil cuando necesitemos almacenar los posibles vecinos que tiene una celda.

2.2.2 Clases Border y Problem

La clase Border.java representa la frontera. Esta ha sido implementada con una PriorityQueue que almacena objetos de la clase Node.java. Los nodos serán ordenados según su valor por medio de un Comparator, de modo que conforme vayan entrando se ordenarán dejando más afuera a la celda con un menor valor. Si no, se ordenarán por fila, y si no, por columna.

La clase Problem.java define un problema a resolver. Este problema tendrá como parámetros la posición inicial de la que partir, la posición objetivo a la que queremos llegar, la ruta del laberinto donde lo queremos hacer y ese mismo laberinto como objeto. Hemos creado además un método que permite dar a conocer los vecinos de una celda y devolverlos en un ArrayList de objetos Sucesor (visto en el punto anterior de la documentación).

2.2.3 Métodos creados en otras clases

Las clases que más se han visto afectadas a cambios son WriteJson.java y ReadJson.java ya que hemos tenido que implementar los métodos necesarios para la lectura y la escritura de problemas en ficheros JSON.

En WriteJson.java hemos añadido el método writeJsonProblem al que se le pasan un problema y un nombre con el que guardar el archivo. Este método se encargará de guardar el problema en un fichero JSON con el formato adecuado con el nombre indicado en el escritorio.

Por otro lado, en ReadJson.java hemos añadido el método readProblem(). En su interior se pedirá al usuario una ruta para localizar el archivo JSON donde se encontrará nuestro programa. Todo con sus correspondientes excepciones.

2.3 Horas de trabajo

A continuación, vamos a desglosar el tiempo dedicado en esta tarea. Para ello vamos a escribir la actividad y el tiempo que ha supuesto. Tiempo dedicado a:

- Implementación de clases nuevas
 - o David González Bermúdez – 5 horas
- Mejoras en el programa
 - o Lucas Gutiérrez Durán – 30 minutos
- Documentación (PDF y código)
 - o Lucas Gutiérrez Durán – 2 horas
 - o David Gutiérrez Mariblanca – 30 minutos
- Otras cuestiones (Resolución de dudas, horas de clase, ...)
 - o Todos los miembros – 4 horas

Tiempo total *aproximado* dedicado a la tarea 2: 20 horas

3. Tarea 3 – Laberintos informados y algoritmo de búsqueda

3.1 Introducción

En esta tercera y ultima tarea los objetivos que se pedían eran los siguientes:

1. Introducir diferentes costes en las acciones aplicadas.
2. Definir una función heurística admisible.
3. Implementar el algoritmo de búsqueda en el Espacio de Estados
4. Resolver el problema con las estrategias siguientes: Anchura, Costo Uniforme, Profundidad Acotada, Voraz y A*.
5. Obtener resultados idénticos a los de los ficheros ejemplo proporcionados por el profesor.

Más información sobre el enunciado de esta tarea puede ser encontrado en la página de Campus Virtual de la asignatura de Sistemas Inteligentes. Datos a destacar en el enunciado son una indicación de los costos según el valor de la casilla:

- '0' – Asfalto
- '1' – Tierra
- '2' – Hierba
- '3' – Agua

Definimos la función heurística del estado (fila, columna) como la distancia manhattan:

$$\text{Heurística}((\text{fila}, \text{columna})) = |\text{fila} - \text{fila_objetivo}| + |\text{columna} - \text{columna_objetivo}|$$

3.2 Algoritmo de búsqueda

Para el algoritmo de búsqueda nos hemos ayudado del pseudocódigo proporcionado por los profesores de la asignatura:

```
BUSQUEDA (Problema, profundidad, estrategia): solución
```

```
visitado = vacío  
frontera = frontera vacía
```

```
#Nodo Inicial  
nodo = crea nodo  
nodo.padre = nadie  
nodo.estado = Problema.EstadoInicial  
nodo.costo = 0  
nodo.profundidad = 0  
nodo.acción = Ninguna  
nodo.heurística = Heurística(Problema, nodo.estado)  
nodo.valor = calcula(estrategia, nodo)
```

```
insertar nodo en frontera
```

```
solución = Falso
```

Pseudocódigo

```
Mientras (frontera no es vacía) y (no hay solución) hacer

    nodo = frontera.primer_elemento()

    Si Problema.objetivo(nodo.estado) entonces
        solución = Verdad
    Sino Si (nodo.estado no está en visitado) y (nodo.profundidad <
profundidad) entonces
        insertar nodo.estado en visitados
        lista_de_nodos_hijos = EXPANDIR_NODO(Problema, nodo,
estrategia)
        Para cada nodo_hijo en lista_de_nodos hacer
            insertar nodo_hijo en frontera
    Si solución entonces
        devolver camino(nodo)
    si no
        devolver no hay solución
```

Pseudocódigo

```
EXPANDIR_NODO(Problema, nodo, estrategia): Lista de nodos

crear lista de nodos

Para cada sucesor (acción,estado,costo) en
Problema.sucesores(nodo.estado) hacer
    crear nodoHijo
    nodoHijo.estado = estado
    nodoHijo.padre = nodo
    nodoHijo.acción = acción
    nodoHijo.profundidad = nodo.profundidad + 1
    nodoHijo.costo = nodo.costo + costo
    nodoHijo.heuristica = Heurística(Problema,estado)
    nodoHijo.valor = calcula(estrategia,nodoHijo)
    insertar nodoHijo en lista de nodos

devolver lista de nodos
```

Pseudocódigo

A modo de resumen de lo que se encarga este método es de ir recorriendo todos los sucesores de los nodos hasta encontrar una solución. Se empezará por un nodo inicial, se introducirá en una frontera y se comprobará si es la solución, si no lo es se buscarán los sucesores de dicho nodo y se introducirán a la frontera, acto seguido, depende de la estrategia, se comprobará un nodo u otro.

El proceso buscará si se encuentra una solución o si se vacía la frontera. Si la frontera se vacía antes de encontrar solución significa que no hay solución.

3.3 Desarrollo de la tarea

3.3.1 Cambios y mejoras en el código

Una mejora con respecto a nuestro código anterior ha sido la selección de las rutas de archivos por medio de un JFileChooser para hacer más ameno y sencillo la apertura y el guardado

de los archivos que va a necesitar nuestro programa. A la hora de abrir los archivos hemos incluido un filtro para que solo aparezcan los archivos con la extensión Json para que al usuario le sea más fácil seleccionar lo que busca.

Otra mejora implementada ha sido el borrado de paredes en los laberintos generados por nuestro programa. Lo que hacemos es generar un laberinto normal y corriente y, después, el usuario tiene la decisión de eliminar un porcentaje de las paredes que contiene. Para ello se ha empleado el uso de una formula deducida por nosotros y el uso de la técnica de la ruleta para la selección del lado a eliminar.

También hemos mejorado la lectura de archivos Json ya que antes leíamos línea por línea, convirtiendo eso en String. Ahora transformamos directamente todo el contenido de un documento en un String para sacarle la información. Con esto agilizamos mucho la lectura de los laberintos y los problemas.

Otra mejora ha sido personalizar el tamaño en pixeles de las celdas dibujadas en las imágenes. Para ello hemos creado una *interface Constants* donde se declara la constante TCELL. Para cambiar la dimensión de una celda el programador solo tendrá que hacer un solo cambio en esa clase, actualmente ese valor está en 50.

3.3.2 Desarrollo de la tarea

En este punto vamos a explicar las clases y métodos desarrollados para el correcto desempeño de esta tercera y última tarea. Vamos a descomponer la explicación diferentes puntos que se corresponderán a diferentes objetivos y, dentro de cada punto, se explicarán las clases y los métodos creados.

3.3.2.1 Dibujado del laberinto con pesos aleatorios

Para esta tarea hemos creado un método llamado *makeValue()* al que se le pasa por parámetro un laberinto. Este método asignará a cada celda del laberinto un peso aleatorio de 0 a 3 por medio de la función *Math.random()*. La operación de este método la realizamos después de aplicar el algoritmo de Wilson en nuestro laberinto.

Una vez tenemos nuestro laberinto listo el siguiente paso es guardarlo, lo cual lo hacemos como siempre. Sin embargo, hay un cambio en la clase *DrawLab*. Hemos creado un método llamado *drawCell()* con el propósito de dibujar en el JPG el laberinto con las celdas según su color correspondiente dependiendo de su peso. A las celdas con el peso 0 le corresponde un color blanco y simulan el asfalto, al peso 1 el color marrón simulando la tierra, al peso 2 el color verde simulando la hierba y al peso 3 el color azul simulando el agua.

Por lo demás el proceso de guardado de archivos es exactamente igual a las tareas anteriores.

3.3.2.2 Desarrollo de solución en base a un problema

El peso de esta nueva tarea está en la posibilidad de leer problemas y encontrar un camino solución en un laberinto dado. De modo que en este punto vamos a desarrollar como hemos llevado a cabo esa funcionalidad en nuestro programa. Cabe decir que la tarea anterior fue una preparación para esta, de modo que muchos elementos usados durante esta tercera y última tarea fueron creados en la tarea anterior.

En la clase *Principal* tras leer un problema declaramos un bucle de 1 a 5, donde cada número se corresponde a una estrategia distinta:

1. Anchura
2. Profundidad
3. Coste Uniforme
4. Voraz
5. A*

También hemos implementado la medición del tiempo en la que cada estrategia encuentra un camino solución, luego podrán compararse los tiempos para saber cual de las estrategias es la más eficiente.

Entramos de lleno en la clase *SearchAlgorithm* donde se encuentran ubicados los métodos necesarios para poder encontrar un camino solución, que será guardado en un *ArrayList<Node>*.

Como hemos dicho, desde la clase *Principal* se llama al método *search()* donde se le pasa por parámetro el problema, la profundidad de la búsqueda, la estrategia, un *ArrayList<String>* y un objeto de la clase *Border*. De lo que se encargará este método es de llevar a cabo el algoritmo de búsqueda que hemos mostrado anteriormente en esta memoria en pseudocódigo. Dicho algoritmo se ira encargando de meter nodos en una frontera, expandirlos, e ir visitando los nodos expandidos hasta encontrar el nodo solución o que la frontera se vacíe, lo cual significaría que no hay solución.

Para la expansión de un nodo hemos creado un método llamado *expandNode()* que se encargará de ir introduciendo en un *ArrayList<Sucesor>* los sucesores de un nodo que se pase por parámetro. Se devolverá un número id que será una unidad mayor al ultimo nodo introducido en el *ArrayList<Sucesor>*

Para la función heurística nos hemos ceñido al enunciado, donde el profesor proporcionaba la función heurística siguiente:

Heurística ((fila,columna)) = |fila-fila_objetivo| + |columna-columna_objetivo|

Para esto hemos creado el método *heuristic()* al que se le pasa por parámetros el problema y el nodo en cuestión, de modo que se calcula la heurística en un *double* y se devuelve.

Para calcular valor según la estrategia hemos creado el método *calculate()* al que se le pasa por parámetros el entero correspondiente a la estrategia y el nodo en cuestión, de modo que por medio de un switch y dependiendo del número de la estrategia se configura una estrategia u otra. Se devuelve el valor del nodo.

Como hemos dicho, el método *search()* devuelve un camino solución o un null en función de si existe solución o no. De modo que, una vez que tenemos este camino solución guardado en un *ArrayList<Node>* solo queda guardar las soluciones para que el usuario pueda verlas de alguna forma y lo hacemos de dos formas.

La primera es imprimir todo el camino solución en un fichero de texto txt. Para esto hemos creado una clase llamada *PrintSolution* donde se encuentra un único método llamado *printSolution()*. Este método se encargará de ir escribiendo toda la información de un nodo en cada línea. Un detalle es que dependiendo de la estrategia el valor lo casteamos a un entero. Si la estrategia no es profundidad los valores del nodo los casteamos a enteros ya que son valores que no tendrán decimales nunca, sin embargo, si la estrategia es en profundidad el valor lo dejamos como un *double* para que puedan verse los decimales del dato.

Una vez hemos generado el fichero de texto el siguiente paso es generar su JPG correspondiente para poder visualizar la solución dentro del mismo laberinto. Para esto hemos creado la clase *DrawSolution*. Desde la clase *Principal* llamamos al método *saveImageSolution()* que se encargará del proceso de la generación de la imagen final. Para ello lo primero es obtener la imagen del laberinto en cuestión, y para ello usamos el método *getPath()* que se encargará de crear el JPG del laberinto y devuelve la ruta de dicha imagen. El siguiente paso es obtener la imagen para poder manipularla dentro del programa y esto lo hacemos mediante el método *getImage()* que devuelve un *BufferedImage*. En este mismo método eliminamos el JPG generado antes ya que no es necesario y no queremos tener archivos basura. Lo siguiente que tenemos que hacer es, dado el *BufferedImage* del laberinto, dibujar la solución, y para esto usamos el método *drawCells()*. Además del camino solución también dibujamos todos los nodos visitados y todos los nodos que contenía la frontera en el momento en el que se encontró una solución. Por último, llamamos al método *generateFile()* para guardar físicamente ese *BufferedImage* nuevo en un archivo JPG para que el usuario pueda visualizar los resultados de la búsqueda.

Como dijimos al principio de este apartado, todo esto lo hará una vez por cada estrategia, y para ello necesitamos limpiar el *ArrayList<String>* de los nodos visitados para que no haya problemas a la hora de reutilizar el elemento más adelante. Con la frontera *Border* no será necesario.

3.3.3 Problemas en el desarrollo

En la realización de la tarea 3 nos han surgido una serie de problemas que relataremos a continuación.

Después de hacer el algoritmo de búsqueda con sus respectivas estrategias nos encontramos con que tanto la estrategia de búsqueda en profundidad como la estrategia de búsqueda en anchura eran inconsistentes comparando los resultados con los proporcionados por el profesor. Nuestra primera idea fue que estas dos estrategias son las únicas que usan la profundidad para calcular el valor por el que serán ordenadas, pero la profundidad se calculaba bien. Después de mucho probar nos dimos cuenta de que la culpa era de la estructura escogida en la tarea 2 para ordenar los nodos (*PriorityQueue*). Cambiamos dicha estructura por una *LinkedList<Node>* que se ordena accediendo a una Interface dentro de los paquetes de *java.util*.

Se ordena mediante valor, fila, columna e id con la siguiente línea de código:


```
Collections.sort(frontier, Comparator.comparing(Node::getValue).thenComparing(Node::getRow).thenComparing(Node::getCol).thenComparing(Node::getID));
```

Después de solucionar esto la estrategia en profundidad acotada seguía sin tener un funcionamiento correcto. Esto fue porque en la clase *SearchAlgorithm* dentro del método *calculate()* donde calculamos el valor de la variable *value* en vez de poner 1.0 poníamos 1 con lo cual asumía divisiones enteras y siempre nos daba el valor 0.

El siguiente problema no es crítico, pero es molesto. Hacíamos una lectura de Json en la que leíamos línea por línea y estas líneas las íbamos guardando en un String que luego parseábamos. Era lento, pero no sabíamos otra forma así que las lecturas de laberintos de tamaño medio nos empezaban a consumir cantidades de tiempo considerable. Esto fue solucionado mediante una función llamada *readFileAsString()*.

Por otra parte, hemos tenido otros problemas de diferente índole. Por otras asignaturas hemos tenido que cambiar la configuración de caracteres en eclipse y al actualizar el proyecto todos los caracteres especiales (á, ú, etc.) escritos en comentarios se cambiaron por caracteres mezclas de caracteres extraños. Para la siguiente práctica no pondremos tildes ni ñ en ningún sitio. Otro problema que hemos tenido es que aún no controlamos de todo GitHub desde eclipse así que hemos tenido algunos contratiempos relacionados con las versiones y mezcla de ellas.

3.4 Horas de trabajo

A continuación, vamos a desglosar el tiempo dedicado en esta tarea. Para ello vamos a escribir la actividad y el tiempo que ha supuesto. Tiempo dedicado a:

- Implementación del algoritmo de búsqueda
 - o David González Bermúdez – 6 horas
- Dibujado del laberinto (soluciones, frontera, visitados y colores según costo)
 - o Lucas Gutiérrez Durán – 5 horas
- Implementación de JFileChooser
 - o Lucas Gutiérrez Durán – 2 horas
- Generado de .txt solución
 - o Lucas Gutiérrez Durán – 1 hora
- Borrado de paredes
 - o David González Bermúdez – 3 horas
 - o David Gutiérrez Mariblanca – 3 horas
- Documentación (PDF y código)
 - o David González Bermúdez – 30 minutos
 - o Lucas Gutiérrez Durán – 4 horas
 - o David Gutiérrez Mariblanca – 30 minutos
- Arreglo de fallos
 - o David González Bermúdez – 3 horas
- Otras cuestiones (Resolución de dudas, horas de clase, mejoras, ...)
 - o Todos los miembros – 8 horas

Tiempo total *aproximado* dedicado a la tarea 3: 52 horas

Anexos

Anexo 1: Conclusiones y opiniones de la práctica

A modo de conclusión tenemos un programa que es capaz de:

1. Leer laberintos de un fichero Json
2. Generar laberintos con pesos aleatorios
3. Guardar nuestro laberinto generado en un fichero Json y una imagen JPG
4. Leer problemas de un fichero Json
5. Crear nuestros propios problemas y guardarlos en un fichero Json
6. Buscar una solución partiendo de un problema
7. Hacer esa búsqueda de solución con 5 estrategias distintas
8. Guardar los resultados en imágenes JPG y en ficheros txt

A continuación, para seguir con las conclusiones vamos a mostrar las *horas totales* de cada componente del grupo y las tareas a las que ha dedicado dichas horas:

- **Horas totales dedicadas al proyecto:** 197 horas
 - o Tarea 1: 125 horas
 - o Tarea 2: 20 horas
 - o Tarea 3: 52 horas
- **David González Bermúdez** – Horas totales: 82,5
 - o Tarea 1: 53 horas
 - o Tarea 2: 9 horas
 - o Tarea 3: 20,5 horas
- **Lucas Gutiérrez Durán** – Horas totales: 63,5
 - o Tarea 1: 37 horas
 - o Tarea 2: 6,5 horas
 - o Tarea 3: 20 horas
- **David Gutiérrez Mariblanca** – Horas totales: 51
 - o Tarea 1: 35 horas
 - o Tarea 2: 4,5 horas
 - o Tarea 3: 11,5 horas

Por último, para terminar con este anexo vamos a dar nuestras opiniones personales tanto de la practica como de los compañeros, y también una opinión grupal acerca de la práctica y la forma de realizarla.

David González Bermúdez

Me ha parecido una práctica bastante interesante que también ha generado una competitividad sana respecto a ver quién puede generar el laberinto más grande.

Según mi opinión el algoritmo de Wilson es muy interesante, pero es poco eficiente y el método en el que tratamos la frontera también me parece un poco ineficiente (metiendo nodos visitados en la frontera y luego sacándolos si son visitados). Se podría comprobar si son o no visitados antes de ser introducidos, pero no coincidiría con el algoritmo que seguimos en los ejercicios que hacemos a mano en la parte de teoría de la asignatura.

Respecto a un ámbito más personal a mí me ha gustado especialmente esta práctica más que la de otras asignaturas porque me gusta todo lo relacionado con algoritmos (estudiarlos, analizarlos, mejorarlos, etc.). Debido a esto en la práctica he sido el que al final ha acabado haciendo el algoritmo de creación y búsqueda del laberinto. También me he encargado otras tareas como lectura y escritura, etc.

Mi compañero Lucas se ha encargado de toda la parte de controlar excepciones, además ha hecho toda la parte gráfica, dibujar laberinto, etc. Le gusta bastante y se le da bien la parte de diseño. Mi compañero David Gutiérrez intentó abordar el algoritmo de generación de laberintos, ha comentado el código y ha hecho métodos auxiliares.

No creo que haya que penalizar a ningún integrante ya que cada uno ha realizado distintas tareas dependiendo de gustos y habilidades entre otros factores.

Lucas Gutiérrez Durán

Sinceramente cuando empezamos la asignatura yo estaba bastante asustado debido a que el profesor dijo que si no se superaba una tarea no se podría realizar la siguiente y eso desembocaría en que iríamos muy mal. Sin embargo, cuando las cosas fueron avanzando vi que la cosa no era para tanto.

La practica en general me ha gustado y me ha parecido incluso divertida, en el sentido de que trabajamos con laberintos y tenemos que resolverlos. Al principio pensamos que iba a ser sencillo, pero conforme íbamos avanzando vimos que tendríamos que dedicarle mucho más trabajo del esperado. Es destacable el apoyo del profesor a la hora del planteamiento de dudas y también es importante el hecho de que ha dejado a disposición de los alumnos diferentes archivos para poder comparar los resultados del programa con los suyos, lo cual ha facilitado en algunas ocasiones la localización de errores y fallos.

Hay que destacar en mi opinión el arduo trabajo de David González Bermúdez, su constancia y dedicación a esta practica ha sido muy notable ya que sé que le encanta la elaboración de algoritmos y resolver problemas lógicos. Yo he enfocado más mi trabajo a la parte visual del trabajo (la documentación, los dibujados de los laberintos, etc.) que es mi punto fuerte. También las aportaciones de David Gutiérrez Mariblanca han sido útiles al trabajo, y se ha notado su esfuerzo. Pero si pienso que alguien ha estado guiando el proyecto desde el principio y ha estado presente en todas y cada una de las tareas y objetivos, ese es David González.

David Gutiérrez Mariblanca

Esta práctica me ha parecido bastante interesante como una continuación de generación de otros tipos de algoritmos, después de los que ya hemos ido conociendo tiempo atrás. Considero esta práctica bastante completa, debido a que encontramos demasiadas

funcionalidades para llevar a cabo, en este caso, con unos laberintos. Tanto generar laberintos y leerlos, como generar problemas y leerlos para llevar a cabo algoritmos de búsqueda entre dos celdas.

Respecto a mis compañeros, no tengo queja ninguna. Son personas muy responsables que saben desenvolverse bastante bien a la hora de programar, cosa que a mí me cuesta horas y horas, respecto a la hora de generar líneas de código que sean eficientes para que el programa funcione correctamente.

He aprendido muchas cosas nuevas respecto esta práctica, sobre todo el uso de Git, el cual yo desconocía por completo. Pero sobre todo he aprendido de mis compañeros nuevas formas de desenvolverse a la hora de enfrentarnos a este tipo de problemas.

Destaco la aportación de David González en este trabajo debido al gusto que tiene a la hora de llevar acabo algoritmos, y sobre todo de Lucas Gutiérrez, dos compañeros que no han dudado en ayudar en todo lo posible.

Mi aportación en esta práctica ha sido a la hora de generar métodos auxiliares para llevar a cabo ciertas funcionalidades en la práctica, intentar llevar a cabo el algoritmo de Wilson (el cual daba muchos problemas y que mi compañero David supo solucionarlo mejorando el algoritmo), y controlar las inconsistencias de los laberintos.

Opinión grupal

En general la práctica ha sido muy interesante y ha sido divertida elaborarla, aunque no negamos que ha costado mucho trabajo y esfuerzo. Hacer este programa nos ha proporcionado nuevos conocimientos como la manipulación de archivos Json y el uso de la librería gson, la manipulación de *bufferedImage* y el guardado de archivos JPG, soltura a la hora de manejar diferentes estructuras de datos como *ArrayList* o *Hashmaps*, etc.

La estructura de la elaboración por tareas la hemos visto cómoda y apropiada, ya que se ha proporcionado un tiempo apropiado para cada tarea (2 semanas en el caso de las tareas 1 y 2 y 3 semanas para la tarea 3). El profesor ha estado dispuesto para la resolución de dudas y la comprobación del avance de la práctica y también es de agradecer.

También vemos apropiado que el número máximo de colaboradores en un grupo sea de 3 ya que facilita el reparto de tareas y disminuye el índice de disparidad en las opiniones de cada integrante a la hora del diseño del programa, aunque en este caso el profesor iba llevando la voz cantante.

En general estamos contentos con el resultado final del programa que hemos elaborado y será un bonito repositorio de cara al publico en nuestros perfiles de GitHub.

Anexo 2: Manual de Usuario

En este anexo vamos a mostrar cómo utilizar nuestro programa por medio de imágenes, mostrando de una en una todas las funcionalidades que tiene de cara a su interfaz para que el usuario que quiera usarla no tenga dudas de como utilizarla. Hemos optado por una interfaz por consola ya que es más sencilla de implementar y esta práctica no estaba enfocada al aspecto visual. Pero hemos empleado en ciertos momentos los *JFileChooser* para que al usuario no le sea tedioso escribir las rutas de los archivos cada vez que se quiera leer un archivo o que se quiera guardar.

Menú principal

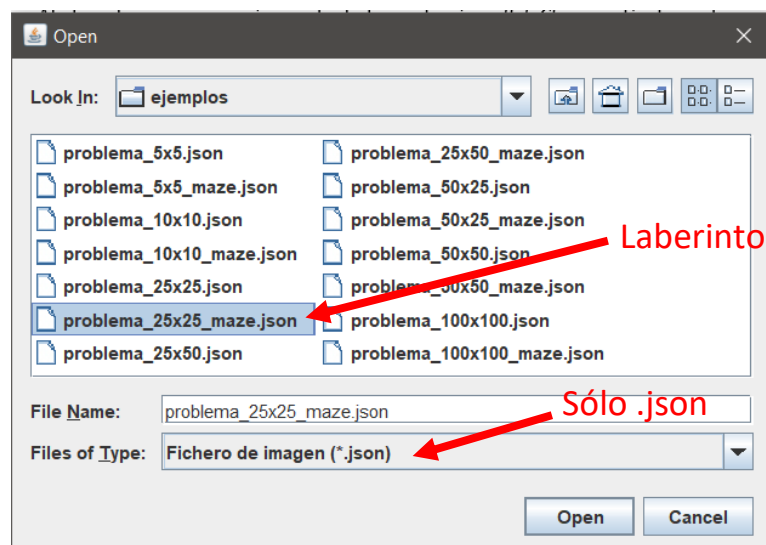
Lo primero que nos encontraremos al ejecutar nuestro programa es el menú principal dándonos la posibilidad de hacer 4 opciones más una quinta para salir del programa.

```
|||PRACTICA DEL LABORATORIO DE INTELIGENTES|||

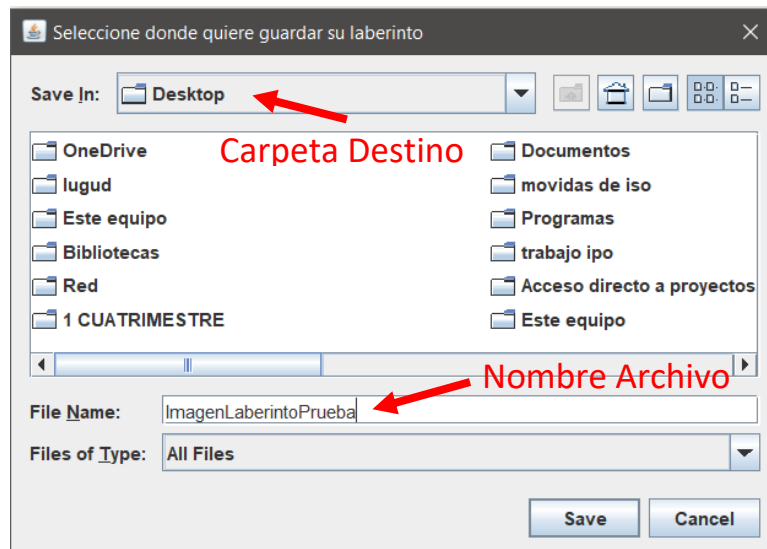
¿Qué es lo que quieres hacer?
1. Leer laberinto desde archivo .Json
2. Generar laberinto aleatorio
3. Leer problema
4. Generar problema
5. Salir del programa
```

1. Leer laberinto desde archivo Json

Al elegir la primera opción por teclado se abrirá un *JFileChooser* dándonos la posibilidad de buscar y elegir un archivo Json de nuestro equipo.

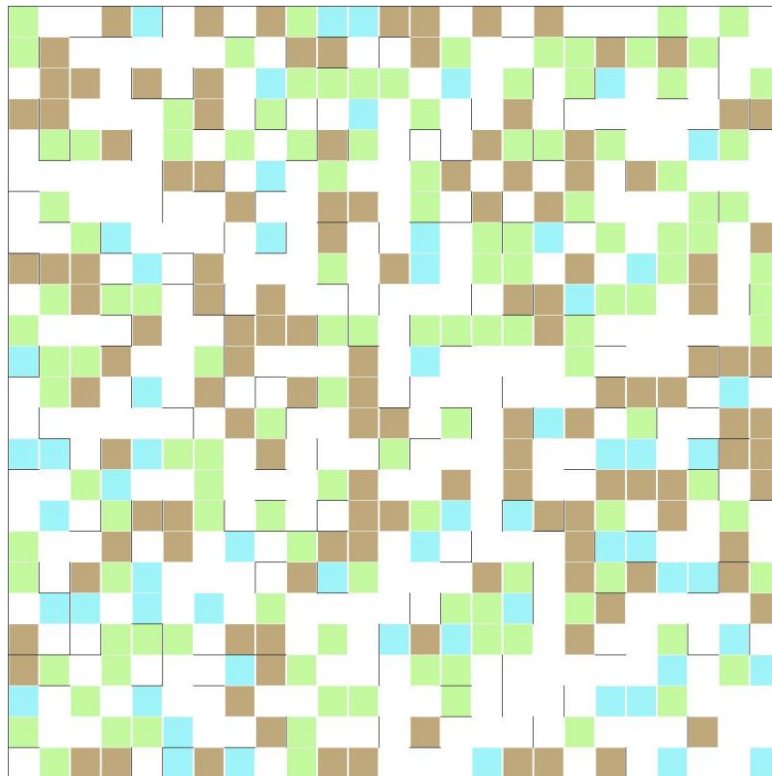


A continuación se abrirá otro *JFileChooser* en el que tendremos que indicar en que lugar del equipo y con que nombre queremos guardar la imagen JPG que se va a generar tras leer el laberinto Json anterior.



Por consola se indicará un mensaje de que el proceso ha sido satisfactorio y podremos comprobar que en la carpeta Desktop se nos ha creado un archivo ImagenLaberintoPrueba JPG.

```
|||PRACTICA DEL LABORATORIO DE INTELIGENTES|||
¿Qué es lo que quieres hacer?
1. Leer laberinto desde archivo .json
2. Generar laberinto aleatorio
3. Leer problema
4. Generar problema
5. Salir del programa
1
Seleccione la ruta de su archivo .json:
C:\Users\lugud\Desktop\1 CUATRIMESTRE\SISTEMAS INTELIGENTES\Laboratorio\problemas\ejemplos\problema_25x25_maze.json
Los archivos se han guardado correctamente!
```



2. Generar laberinto aleatorio

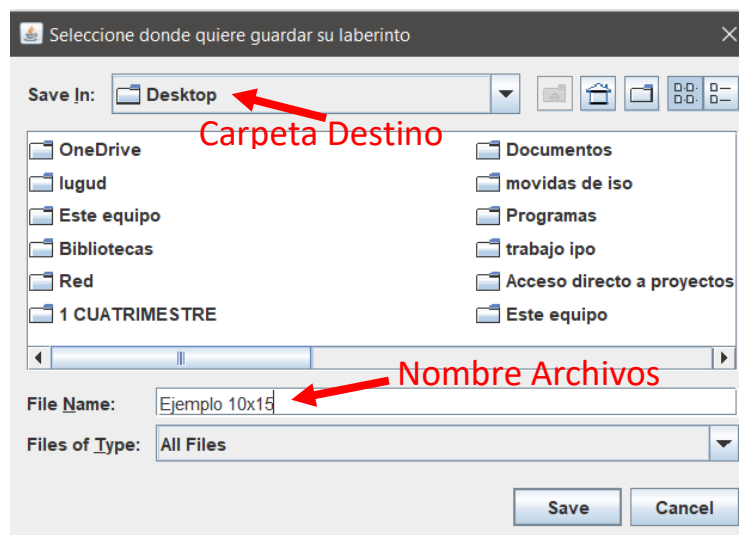
Probamos ahora la segunda opción que proporciona el programa.

```
¿Qué es lo que quieres hacer?  
1. Leer laberinto desde archivo .json  
2. Generar laberinto aleatorio  
3. Leer problema  
4. Generar problema  
5. Salir del programa
```

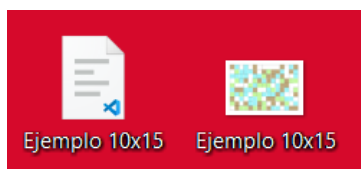
Lo primero que nos pedirá será introducir por teclado las dimensiones del laberinto, de modo que habrá que introducir primero las filas y después las columnas:

```
Indique las filas:  
10  
Indique las columnas:  
15
```

A continuación, indicamos en otro *JFileChooser* la ruta destino de los archivos que se van a generar:



Ahora podremos comprobar como en el escritorio se nos han generado dos archivos, un Json y un JPG correspondientes al laberinto que acabamos de generar:



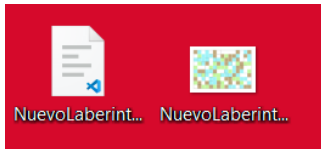
Por consola se mostrará la opción de indicar el porcentaje de paredes que queremos eliminar, en mi caso voy a indicar 50%:

```
Los archivos se han guardado correctamente!  
Indique el porcentaje de paredes que quiere quitar:  
50
```

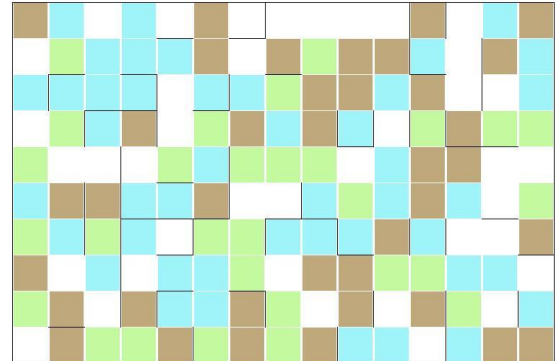
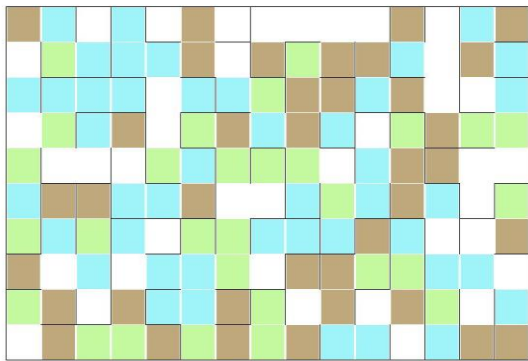
Ahora por consola indicará el numero de paredes que hay en el laberinto y el número de paredes que se van a eliminar en base al porcentaje anterior:

```
Numero de paredes en el laberinto: 126  
Se van a quitar un 50.0% de paredes. Total a remover: 63 paredes
```

Posterior a esto se volverá a abrir otro *JFileChooser* donde volveremos a elegir una ruta para guardar nuestros archivos nuevos, yo los guardaré en el escritorio con el nombre de NuevoLaberinto10x15:

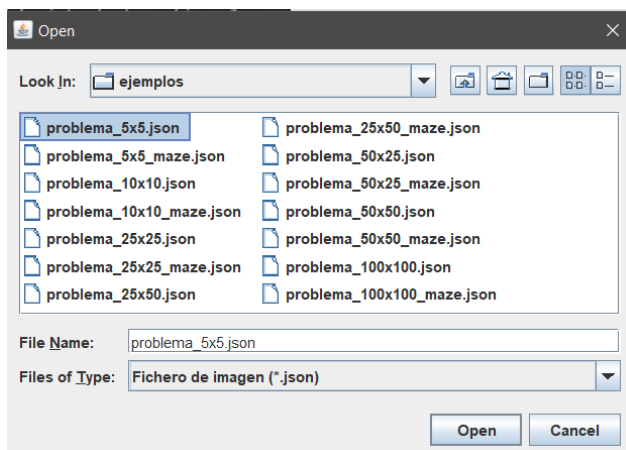
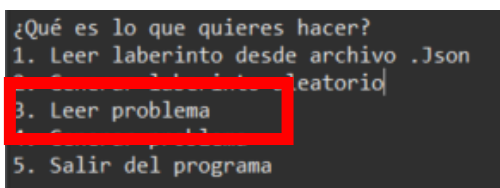


Podemos comparar la diferencia entre el anterior laberinto generado y el nuevo:



3. Leer problema

Esta opción nos permitirá leer un problema desde un archivo Json. Al elegir esta opción se abrirá un *JFileChooser* en el que tendremos que indicar la ruta de nuestro problema, igual que cuando leemos laberintos.

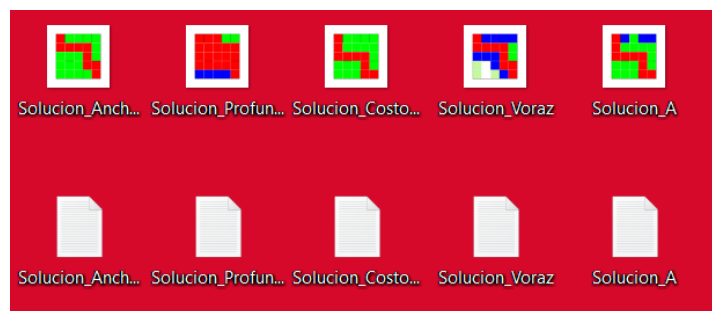


Cuando le demos a *Open* podremos ver que por consola se imprimen los siguientes datos:

```
Problema [initial=(0, 0), objective=(4, 4), maze=problema_5x5_maze.json]
El algoritmo de busqueda 1 tarda 7891400.0 nanosegundos.
El algoritmo de busqueda 2 tarda 2321900.0 nanosegundos.
El algoritmo de busqueda 3 tarda 2263200.0 nanosegundos.
El algoritmo de busqueda 4 tarda 502900.0 nanosegundos.
El algoritmo de busqueda 5 tarda 1070400.0 nanosegundos.
```

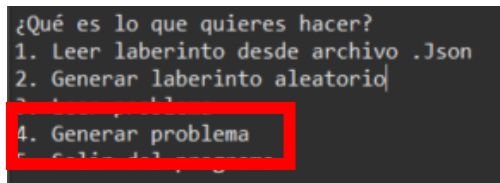
La primera línea corresponde con los datos del problema leído del archivo Json. Las siguientes 5 líneas indican cuanto tiempo ha tardado cada algoritmo de búsqueda dependiendo de la estrategia. (1 – Anchura; 2 – Profundidad; 3 – Coste Uniforme; 4 – voraz; 5 – A*).

Podemos comprobar como en el escritorio, por defecto, se nos generado los siguientes archivos: Un JPG y un TXT por cada estrategia

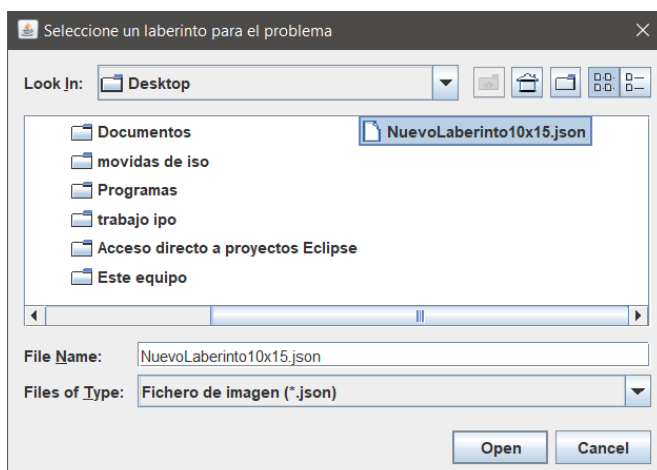


4. Generar Problema

La última opción nos permite generar un problema en base a un laberinto proporcionado.



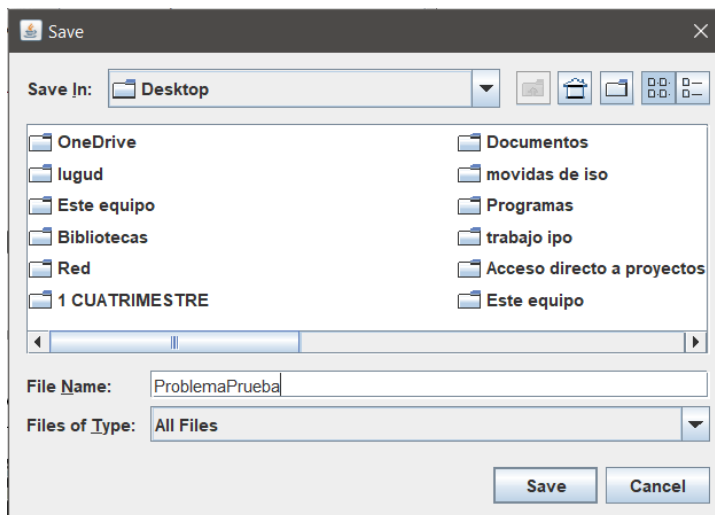
Para hacer las pruebas utilizaremos el laberinto generado en el paso 2 de este manual de usuario. Cuando seleccionemos la opción se va a abrir otro *JFileChooser* donde tendremos que indicarle cual será el laberinto sobre el que haremos el problema.



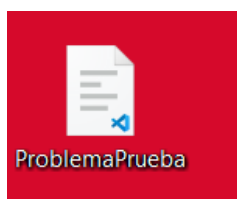
Una vez indicado el laberinto tendremos que introducir por teclado en la consola la fila y la columna tanto de la casilla inicial como de la casilla objetivo:

```
Casilla inicial:  
-> Introduzca la fila:  
0  
-> Introduzca la columna:  
0  
Casilla objetivo:  
-> Introduzca la fila:  
9  
-> Introduzca la columna:  
14  
(0, 0) (9, 14) NuevoLaberinto10x15.json
```

Ahora tendremos que indicar donde queremos guardar el archivo JSON del problema que acabamos de generar, en mi caso lo haré en el escritorio:



Podemos comprobar como en el escritorio se ha creado un archivo Json con nuestro programa:



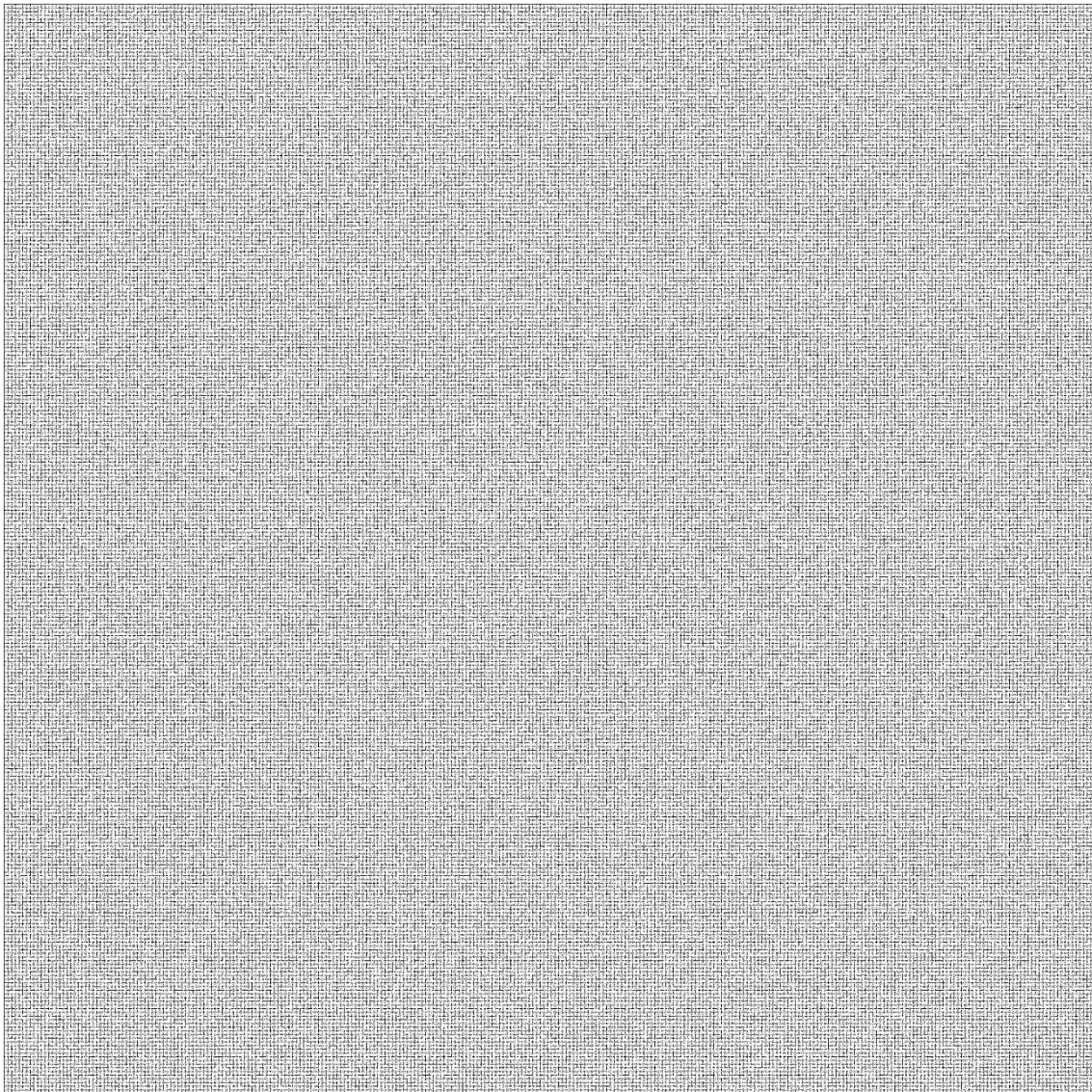
Y con esto daríamos concluido este manual de usuario donde hemos visto todas las funcionalidades que permite nuestro programa, desde la generación de laberintos y problemas hasta la lectura de los mismos.

Anexo 3: Ejemplos de laberintos creados

En este anexo vamos a mostrar algunos laberintos que creamos. Se destacan por ser más grande de lo normal. Hay que destacar que algunos los generamos durante la Tarea 1, luego hay cosas que no se mostrarán como el dibujo de los pesos o los caminos solución.

Laberinto 1000x1000:

- Tardó 50 minutos en generarse.
- 1 millones de celdas.
- 146 MB pesa su archivo Json.



Laberinto 2000x2000:

- Tardó 17 horas en generarse.
- 4 millones de celdas.
- 524 MB pesa su archivo Json.

