PROYECTO

INTELIGENCIA ARTIFICIAL

INTEGRANTES:

MARIA ALEJANDRA AGUIAR VASQUEZ – 1455775

LUIS FELIPE CASTAÑO LEDESMA – 1455721

PRESENTADO A:

ING. JOSHUA DAVID TRIANA

UNIVERSIDAD DEL VALLE

FACULTAD DE INGENIERIA

INGENIERIA DE SISTEMAS

TULUÁ

2017

INFORME

Especificaciones de la implementación:

Sobre manejo de archivos

Con el objetivo de permitir ambientes dinámicos, es decir, que se puedan usar múltiples laberintos para hacer varias pruebas, se ha creado una clase llamada **CargarDatos.java** y un paquete llamado **archivos**.

CargarDatos.java es la clase responsable de cargar los archivos necesarios para el correcto funcionamiento de los algoritmos Wall Tracing y Waypoint Navigation. En específico, **cargarDatos.java** posee tres funciones:

- La primera función, llamada **cargarTableroWallTracing**, es la responsable de cargar el mapa (matriz de enteros) necesario para la posterior ejecución del algoritmo Wall Tracing.
- La segunda función, llamada **cargarTableroWaypoint**, es la responsable de cargar el mapa (matriz de Strings) necesario para la ejecución del algoritmo Waypoint Navegation.
- Por último, la tercera función, llamada **cargar Caminos**, es la responsable de cargar la tabla con los caminos precalculados para apoyar la ejecución del algoritmo Waypoint Navigation.

Cabe destacar que, aunque las funciones que fueron implementadas para cargar el laberinto son distintas, el archivo plano que se cargará es el mismo, por ejemplo: el archivo **ejemplo1** contiene un laberinto que es apto tanto para Wall Tracing como para Waypoint Navegation. En conclusión, la diferencia radica en el tratamiento que se le da al archivo dentro de cada función.

Funcionamiento del algoritmo Wall Tracing

Este algoritmo posee 4 funciones principales que serán detalladas a continuación:

- **NextStep:** Es la función utilizada desde la vista para refrescar el mapa cada vez que ocurre un evento, la cual retorna el siguiente paso que debe dar el agente para llegar a la meta, además de eso, esta función esta contantemente verificando si hay game over, es decir, si el agente llego a su meta.

Otra tarea que recae sobre esta función es decidir si se aplicara Bresenham (si no hay obstáculos) o si el siguiente paso será dado por el comportamiento normal del agente.

- **Bresenham:** Este algoritmo retorna la ruta que será tomada por el agente hasta la meta si no existen obstáculos entre ellos, de otro modo retorna **null**, indicando que no es posible llegar a su destino.
- **ValidarPasos**: Esta función decide el siguiente paso si no es posible aplicar Bresenham, ésto aplicando las reglas descritas en el proyecto y validando que existan paredes circundantes en la siguiente posición.
- **ComprobacionParedes**: Esta función es la encargada de validar que existan paredes circundantes en una posición determinada del laberinto, es decir, esta es una función diseñada para apoyar en la toma de decisiones en la función validarPasos.

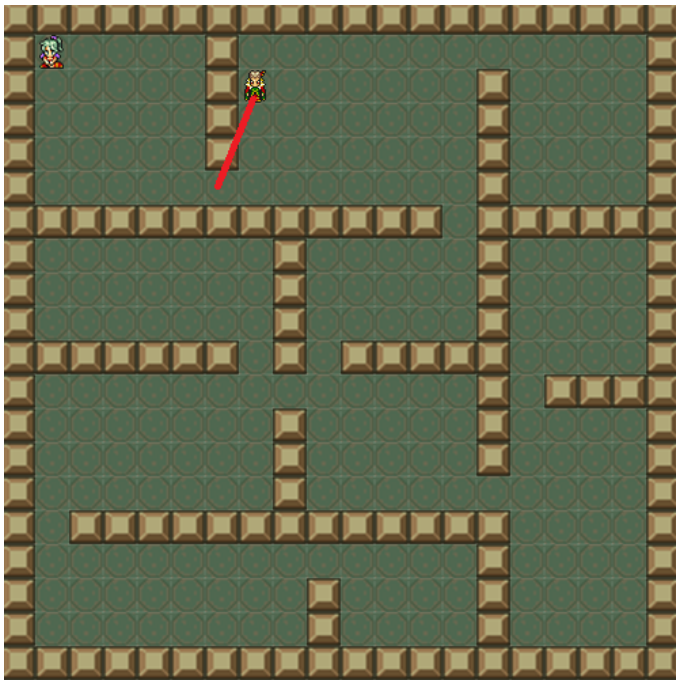
Funcionamiento del algoritmo Waypoint Navigation

Debido a la cantidad de funciones auxiliares implementadas para el correcto funcionamiento del Waypoint Navigation no se entrará en detalles sobre cada una de ellas, sino que se explicará el comportamiento general del algoritmo.

Igual que en el algoritmo anterior, la función que se comunica directamente con la vista es **nextStep**, el cual es el encargado de retornar el siguiente paso que debe realizar el agente para llegar a la meta. Para hacer esto, se programa el siguiente comportamiento dentro del algoritmo:

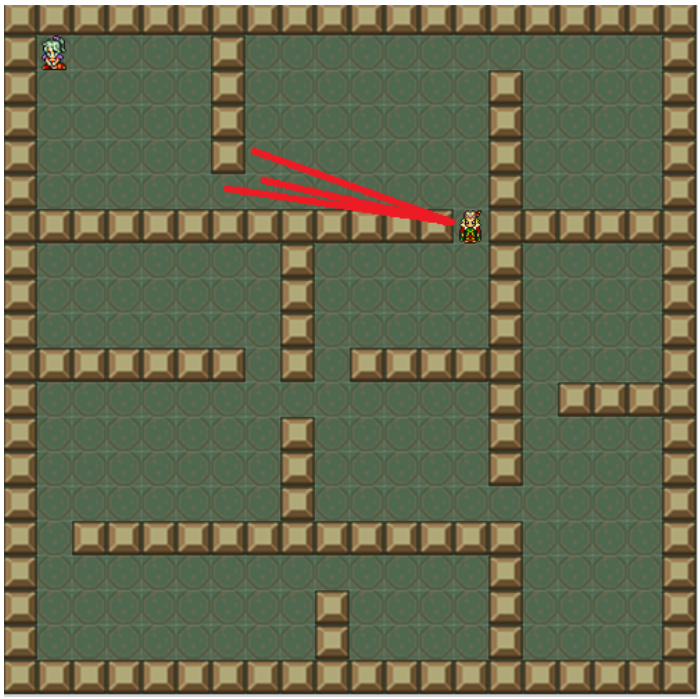
1. Se ejecuta Bresenham para verificar si es posible llegar directamente a la meta, si lo es, la función retorna el siguiente paso según Bresenham.
2. Si no es posible lo anterior, se busca el punto de corte entre la recamara que ocupa el agente y la siguiente recamara a la cual se debe ir según la tabla de caminos cargada previamente, luego se aplica Bresenham hacia este punto, si es posible llegar sin obstrucciones este es el valor que retorna nextStep.

Ejemplo de punto de corte obstruido entre recamaras:



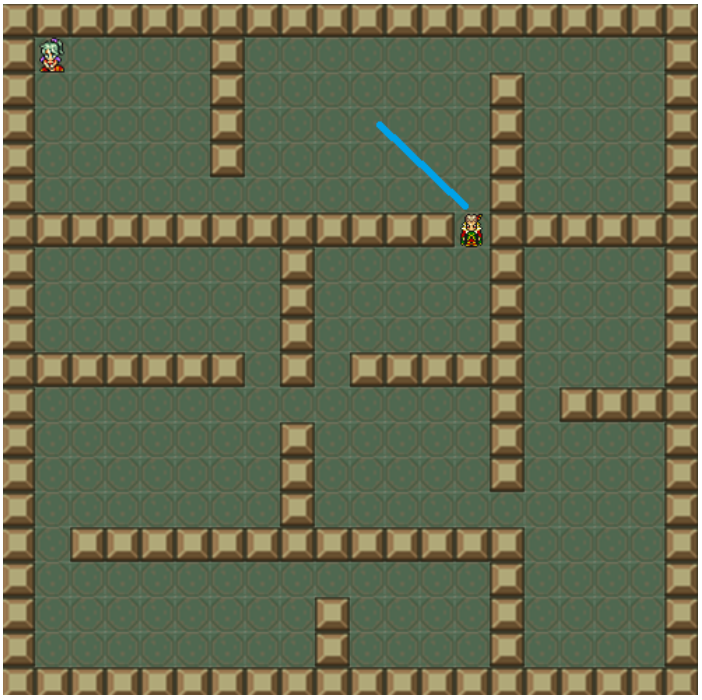
3. De lo contrario, se busca en las posiciones circundantes al punto de corte que pertenecen a la recamara donde se encuentra el agente y se aplica Bresenham hacia ellas, esto se debe a que es posible que el camino Bresenham hacia el punto de corte entre recamaras se vea obstruido, si se encuentra un camino sin obstrucción este es el valor que retorna nextStep.

   Ejemplo puntos circundantes obstruidos:

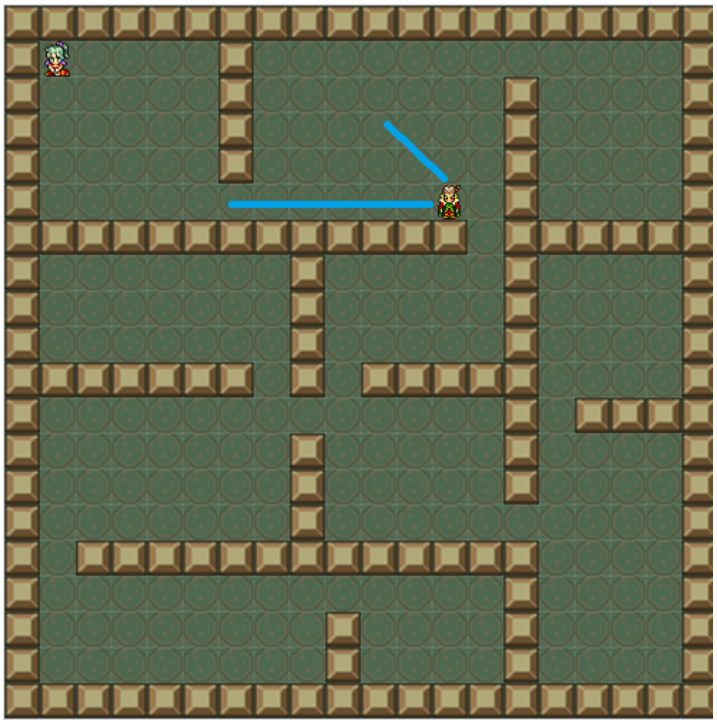


4. Por último, si no se encuentra ningún camino sin obstáculos de los anteriormente mencionados, el objetivo del agente es llegar al centro de su recamara con el fin de despejar de obstáculos su camino. Fijado ya el objetivo, se aplica Bresenham sobre este y se retorna a la vista.

   Ejemplo secuencia que no encuentra caminos disponibles:



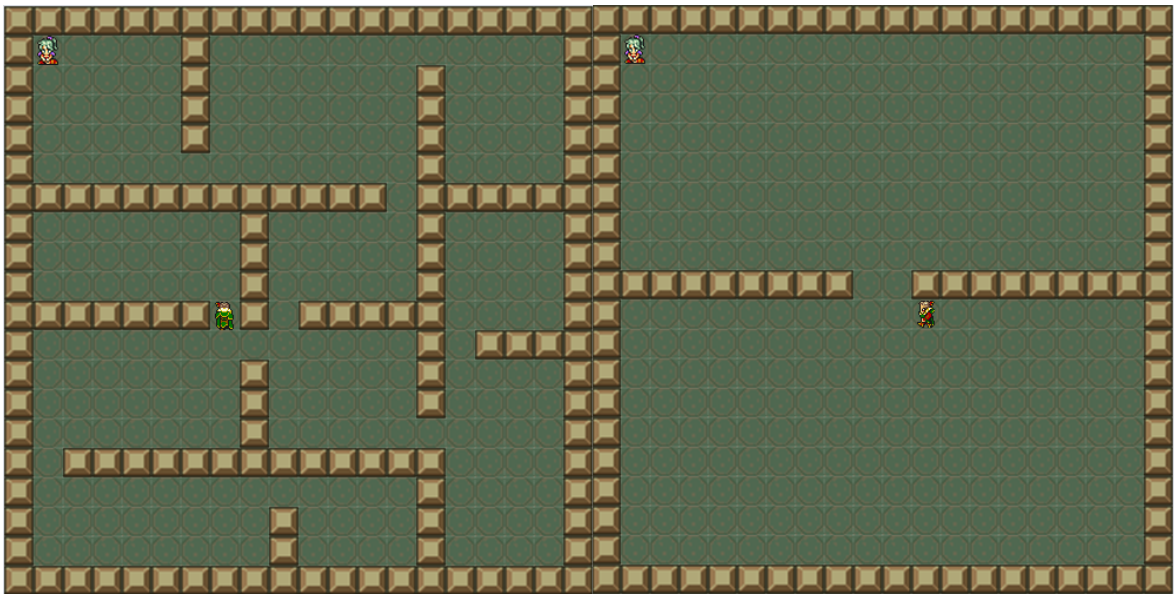
   Después de una iteración el resultado es:

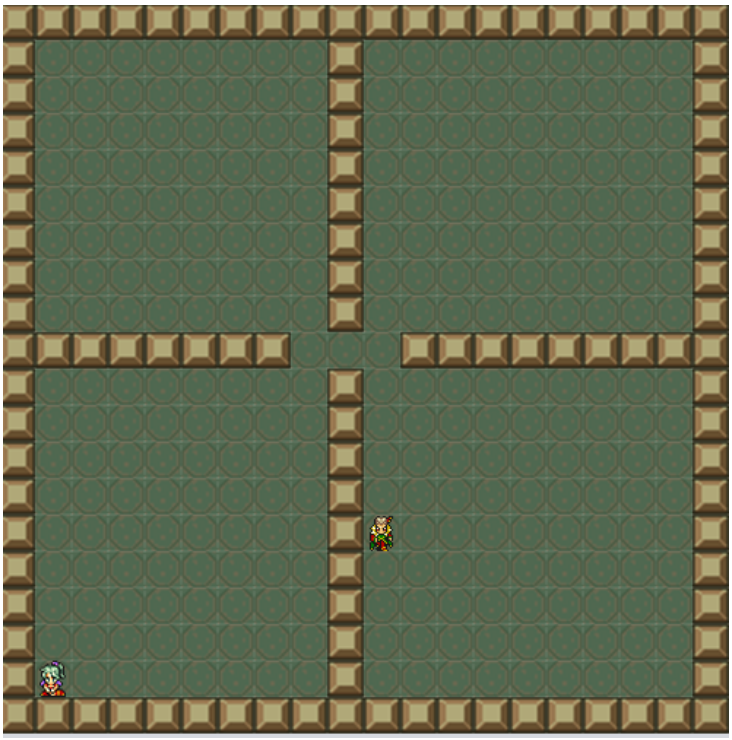


Como se puede observar, en este punto hay dos caminos posibles, uno es el camino hacia el punto de corte con la siguiente recamara según la matriz de caminos y el otro es el camino hacia el centro de la recamara actual. Debido a que el camino que tiene más prioridad es el que acerca a la meta del agente, el camino elegido en este caso es el camino hacia la siguiente recamara. Este proceso se repite para cada iteración o paso que da el agente.

Resultados encontrados

Wall Tracing

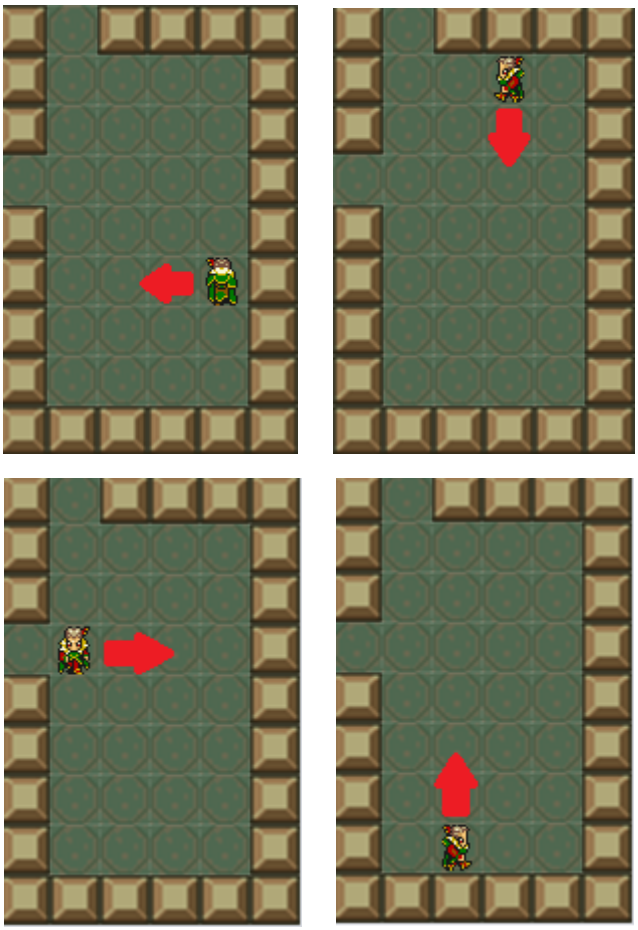




Después de una cantidad considerable de intentos, cambiando el mapa y además en cada uno modificando las posiciones iniciales, se puede concluir que el algoritmo Wall Tracing es muy propenso a caer en ciclos. Alrededor del 50% de las configuraciones iniciales caían en un ciclo infinito.

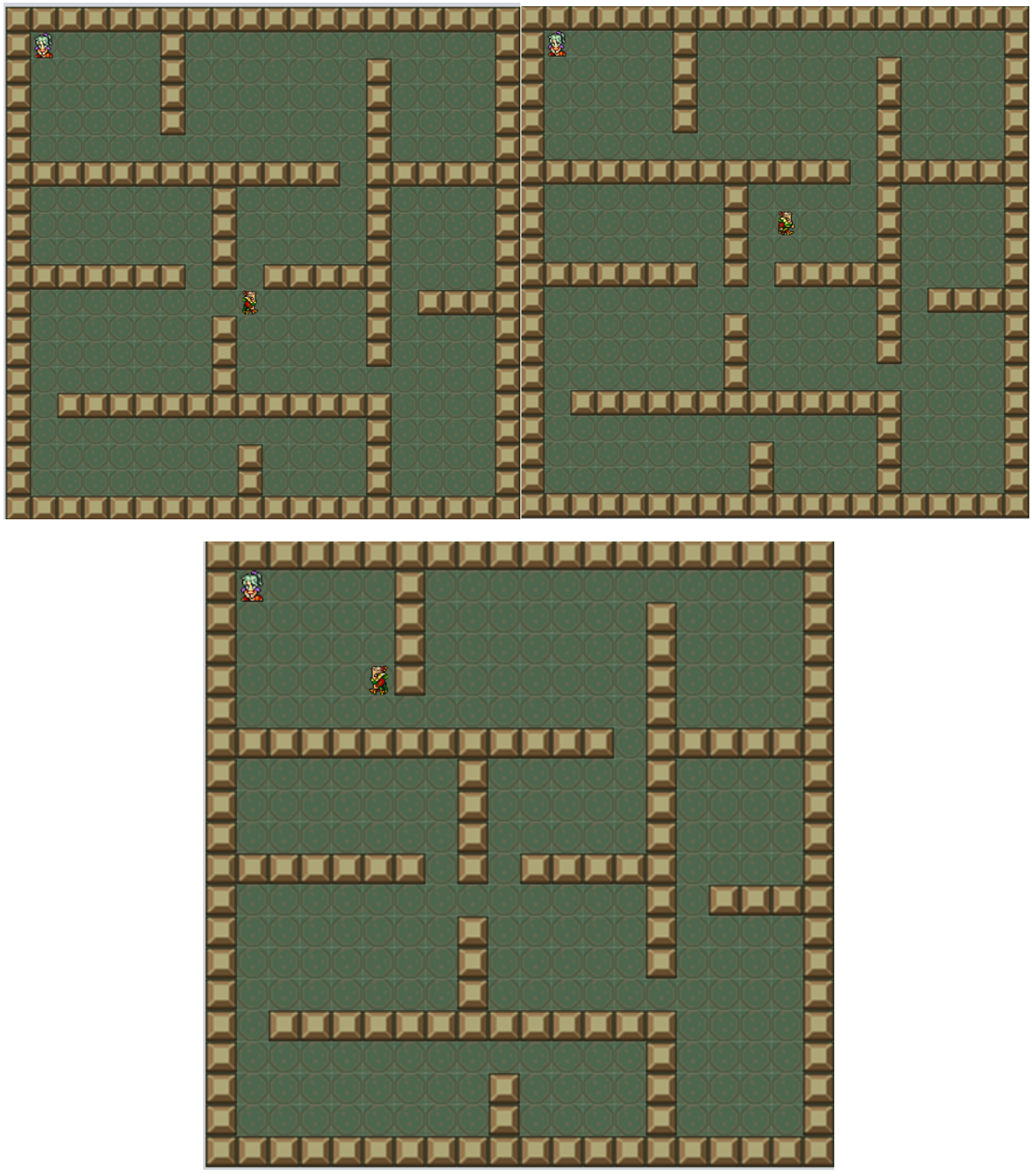
Al hacer un análisis más detallado se puedo observar la razón del fallo del algoritmo, para explicar esta razón se debe entender lo siguiente: El algoritmo Wall Tracing tiene prioridades sobre las operaciones, las prioridades son: **izquierda**, **frente**, **derecha** y **retroceder** respectivamente.

Se observó que el algoritmo caía en ciclo infinito siempre que la posición inicial del agente se encontraba en la parte derecha o superior de una recamara, esto se debe a que el lado izquierdo con respecto al agente apuntaba hacia el centro del laberinto, y ya que el lado izquierdo es el de mayor prioridad el agente siempre intentará ir primero hacia este lado.

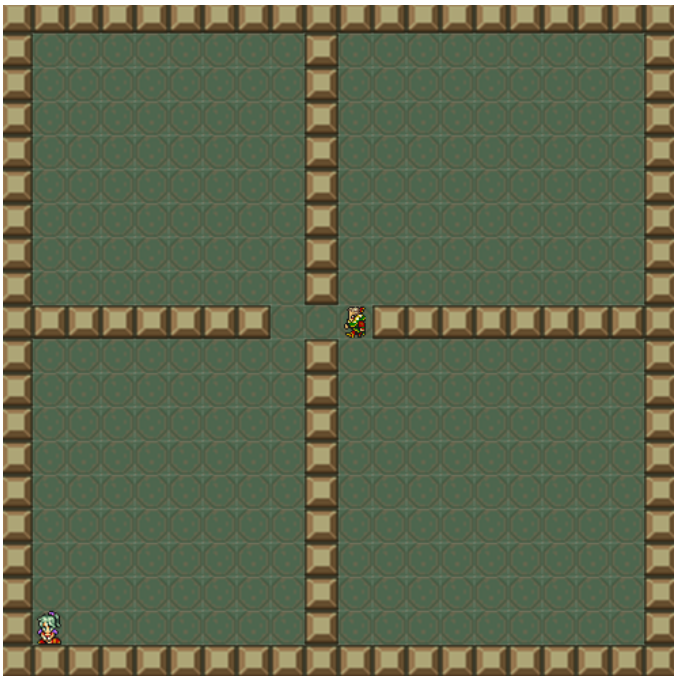


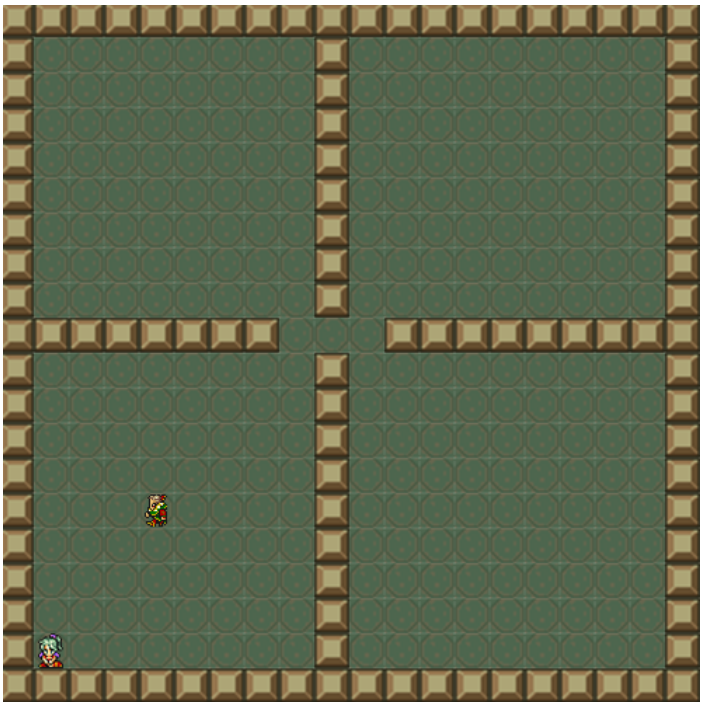
Waypoint Navigation

Ejemplo 1:



Ejemplo 2:





Este algoritmo tiene muy buenos resultados, llegando a su meta de una manera más óptima y sin estancarse en ciclos infinitos. Al igual que en Wall Tracing, este algoritmo fue probado en varios mapas y en cada uno con varias configuraciones iniciales diferentes, y al contrario del anterior sus resultados fueron de 100% éxito.

Tabla comparativa

Aspecto	Wall Tracing	Waypoint Navigation
Completitud	No es completo, porque a pesar de que exista una respuesta en algunos casos podría no encontrarla. Puede caer en ciclos.	Es completo, si existe una respuesta la encontrará.
Eficiencia	En algunos casos, aunque la meta este cerca del agente, éste puede recorrer todo el mapa antes de encontrarla. Por lo tanto, generalmente no es eficiente.	Gracias a la matriz pre-calculada de caminos, el agente va directamente a la meta de la manera más eficiente posible.
Complejidad espacial	Solo hace uso de una matriz y no es necesario guardar ninguna operación ni expandir un árbol de búsqueda.	Se hace uso de dos matrices, aún así, comparado a Wall Tracing su complejidad espacial no es mucho más alta.
Complejidad temporal	Las operaciones realizadas son muy simples y no requieren mucho procesamiento, es decir, su complejidad temporal es baja (poca).	Las operaciones son muy complejas y en algunos casos requieren recorrer varias veces el mapa en búsqueda de información. Por lo tanto, en comparación a Wall Tracing tiene una complejidad temporal alta.

En conclusión, los dos algoritmos tienen puntos fuertes, sin embargo, es preferible sacrificar algo de tiempo y espacio en memoria para asegurar la completitud de la solución.