Departamento de Sistemas y Automática

TRABAJO FIN DE GRADO

IMPLEMENTACIÓN Y ANÁLISIS DEL ALGORITMO FAST MARCHING Y SUS DISTINTAS VERSIONES

Autor: Pablo Muñoz Gely

Tutores: David Álvarez Sánchez
Javier Victorio Gómez

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Leganés, Septiembre de 2014

Título: Implementación y análisis del algoritmo Fast Marching y sus distintas versiones.
Autor: Pablo Muñoz Gely.

EL TRIBUNAL

Presidente: Dr. Miguel Ángel Monge Alcázar

Vocal: Dr. Marta Ruiz Llata

Secretario: Dr. Concepción A. Monje Micharet

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 9 de Octubre de 2014 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

Me gustaría agradecer a todas las personas que han permitido la realización de este Trabajo Fin de Grado.

Agradecer a David Álvarez, Javier V. Gómez y a la Universidad Carlos III de Madrid la oportunidad que me han brindado para la realización de este trabajo.

También agradezco a Javier V. Gómez y mi padre Julián Muñoz Jiménez toda la ayuda recibida en la elaboración de este trabajo.

Gracias a mis padres por permitir, no solo en la realización de este trabajo, sino también en el Grado en Ingeniería Electrónica Industrial y Automática. Y a mis amigos por aguantarme durante todo este tiempo.

Resumen

La modelización de la expansión de una onda en el espacio ha sido un campo muy estudiado, debido a su gran utilidad en muchos ámbitos, como por ejemplo, problemas de fluido-mecánica modelización de tumores, navegación robótica, análisis sísmico, videojuegos o procesamiento de imágenes. Al ser estudiado en tantos campos y tan diversos, es inevitable que existan una gran cantidad de soluciones al mismo problema. Por lo que se han creado diferentes algoritmos a lo largo de los años.

Este Trabajo Fin de Grado, recopila, analiza y compara varios de estos algoritmos. Consiguiendo, de esta manera, un documento en el que se comparen cada uno de ellos en diferentes situaciones, como por ejemplo, diferentes tamaños del espacio donde se propaga la onda, velocidades de propagación de onda distintas, grandes contrastes de esa velocidad, diferente número de dimensiones... .

De este modo, se consigue una buena comparativa del desempeño de cada uno de los algoritmos analizados y de sus características.

Palabras clave: Fast Marching Method, Group Marching Method, Fast Iterative Method, Untidy queue, Eikonal, Complejidad computacional, Comparación.

Contenido

Índice de figuras

Índice de tablas

Capítulo 1:

INTRODUCCIÓN Y MOTIVACIÓN

En este Trabajo Fin de Grado se tratarán los algoritmos de propagación de ondas. Esta área de investigación tiene gran utilidad en muchos ámbitos, como por ejemplo, problemas de fluido-mecánica[1][2], modelización de tumores[3][4], navegación robótica[5], análisis sísmico[6], videojuegos o procesamiento de imágenes[7].

Con esta amplia variedad de aplicaciones se han tenido que desarrollar diferentes métodos para encontrar la solución de una manera más rápida y eficiente.

En la Figura 1.1 se puede apreciar la utilidad de estos algoritmos en medicina para delimitar el contorno de una arteria mediante la propagación de una pequeña región hasta alcanzar el contorno deseado, usando uno de estos algoritmos[8].

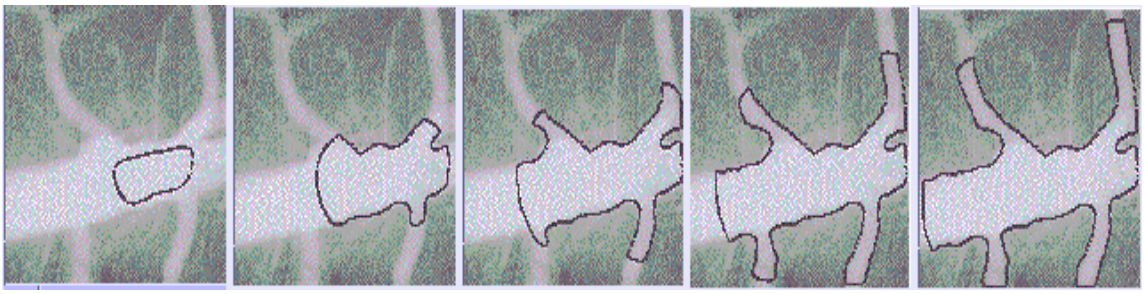


Figura 1.1: Contorno de una arteria, conseguido mediante Fast Marching Method[9].

El objetivo de este trabajo es implementar y comparar estos algoritmos en distintas circunstancias. De esta manera, saber cuál de ellos consigue un resultado correcto y en el mejor tiempo, determinando, con ello, cual es mejor en distintas situaciones.

Para la realización de este proyecto, se han implementado tres algoritmos, basados en el algoritmo *"Fast Marching Method"*[10] y se han utilizado otros tres ya existentes. El proceso ha consistido en experimentar diferentes configuraciones con cada uno de los seis algoritmos para luego obtener, analizar y comparar los resultados con cada uno de ellos.

La principal motivación para la realización de este trabajo, es conseguir una buena herramienta de comparación entre los diferentes algoritmos. Ya que, en la documentación usada, los métodos para comparar resultados no eran muy efectivos al no haber buenas comparaciones [11][12]. Solamente se usaba el método implementado y *"Fast Marching Method"* para comparar entre ambos exclusivamente. Además, un método de comparación de los resultados entre distintas documentaciones es complicado, ya que se realizan con diferentes ordenadores y esto afecta al resultado final.

1.1 Objetivos

Para cumplir con el objetivo principal, se han propuesto los siguientes objetivos parciales:

-Implementación del algoritmo *"Group Marching Method"*.

-Implementación del algoritmo *"Untidy Fast Marching Method"*.

-Implementación del algoritmo *"Fast Iterative Method"*.

-Experimentación y comparación de los tres algoritmos implementados además de otros tres algoritmos ya existentes basados en "*Fast Marching Method*".

1.2 Estado del arte

Un **mapa de distancias** es una representación en la que se sabe la distancia desde un punto, a cualquier otro punto dentro de ese mismo mapa.

Estos mapas comenzaron cuando Rosenfeld y PFaltz[13] desarrollaron una aplicación para la representación del esqueleto de una imagen (la mínima expresión de una estructura). A partir de ahí, muchos autores han realizado la implementación de distintos algoritmos, sobre todo, para el avance en tres campos: visión por computador, representación de ondas y generación de gráficos por ordenador.

Los algoritmos de **transformadas de distancia DT**(*distance transforms*), son un método para un cálculo rápido de las distancias, que incluyen un error respecto a la distancia euclídea pero permiten mayor rapidez. Según la estimación de la distancia y en función de cómo se propagan las ondas a lo largo del mapa, se establece una distinción de grupos[14]:

A. Según la estimación de distancia

- **Transformadas de distancia biseladas[14][15]**: el nombre de esta DT viene del término de carpintería, en el que también se realizan dos pasadas. El valor de distancia de una celda, se calcula con las distancias de sus vecinos usando una plantilla de distancias, tras hacer dos barridos.

  La plantilla es dividida en dos partes, una usada en el primer barrido (directo) y la otra en la segunda pasada (inverso).La pasada directa, calcula las distancias moviéndose desde la celda donde se ha iniciado, hasta el final del mapa, Figura 1.2 a). La segunda pasada calcula las distancias que quedan, después de la primera pasada, Figura 1.2 b).

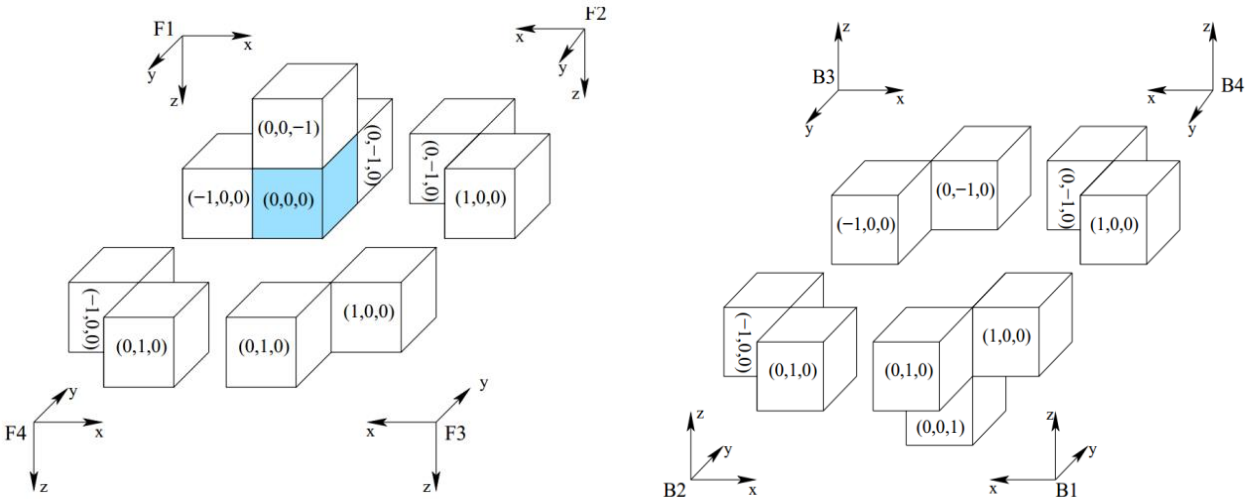
*	*	*	*	*	*	*		12	11	10	9	10	11	12
*	*	*	*	*	*	*		11	8	7	6	7	8	11
*	*	*	*	*	*	*		10	7	4	3	4	7	10
*	*	*	0	3	6	9		9	6	3	0	3	6	9
*	*	4	3	4	7	10		10	7	4	3	4	7	10
*	8	7	6	7	8	11		11	8	7	6	7	8	11
12	11	10	9	10	11	12		12	11	10	9	10	11	12

a) Mapa de distancia tras la primera pasada. *b) Mapa tras la segunda pasada.*

Figura 1.2: Proceso de cálculo de las distancias en las DT biseladas.

- **Transformadas de distancia vectoriales[16]**: dado que las transformadas biseladas tienen el problema de ser poco precisas a medida que el número de celdas aumenta, se tienen que desarrollar las transformadas de distancia vectoriales.

  Este método utiliza vectores para propagar la onda, en una sucesión de dos pasadas, al igual que en el caso de las transformadas biseladas. Cada celda guarda un vector a la celda más cercana y otro vector a una celda que no haya sido procesada aún, usando una plantilla de vectores, Figura 1.3. La segunda pasada se realiza también de manera inversa, como en el caso de las transformadas biseladas.



a) Plantilla de vectores en la pasada directa. *b) Plantilla en la pasada inversa.*

Figura 1.3: Plantilla de vectores usada en la transformada de distancia[14].

- **Resolvedores de la Eikonal**: este tipo de algoritmos se basan en la resolución de la llamada **ecuación de la Eikonal**. Los cálculos de las distancias se realizan mediante los datos de distancias de sus vecinos.

  Este trabajo se basa en uno de los algoritmos que son de este tipo, el *Fast Marching Method*, que se explicará en capítulos posteriores.

B. <u>Según la propagación</u>

Según la propagación se pueden utilizar dos métodos:

- **Sweeping Methods[17]**: se usan en casos donde la trayectoria óptima no cambia su dirección, por ejemplo, en mapas donde la velocidad sea homogénea. Así, en estos métodos, se puede solucionar la ecuación de la Eikonal mediante la actualización de la solución a lo largo de una dirección.

  Para la resolución del problema se necesitan 2^n barridos. Siendo *n*, el número de dimensiones que existan, debido a que hay dos posibles sentidos para cada dirección.

- **Frentes de onda**: se calculan las distancias desde donde se origina la onda, y se incrementan hasta que todas las celdas son procesadas. Este es el sistema de propagación de ondas que usa *Fast Marching Method* y sobre el que se centrará este trabajo.

Capítulo 2:

FAST MARCHING METHOD

El *Fast Marching Method, FMM*, se trata de un algoritmo desarrollado por Osher y Sethian[10], que modeliza el movimiento de una onda. Es un caso especial del *"Método del conjunto de nivel"*, una técnica que es usada para delinear interfaces y formas.

FMM es muy similar al *"Algoritmo de Dijkstra"* que se basa en el hecho de que la onda solamente se desplaza hacia el exterior del punto donde se inició dicha onda. La onda generada tiene la capacidad de ser el camino más corto, temporalmente hablando, desde donde se inició hasta el destino.

Para entender fácilmente FMM[18] basta con imaginar el efecto que produciría tirar una piedra sobre una superficie acuática en reposo. Al contacto de la piedra con la superficie del agua en un punto, se formaría una onda circular que empezaría a expandirse hacia el exterior de dicho punto.

Si en una parte de la superficie no hubiese agua, y hubiera otro líquido, la velocidad a la que se expande la onda cambiaría y la forma circular no se mantendría, como en el ejemplo que se puede apreciar en la Figura 2.1.

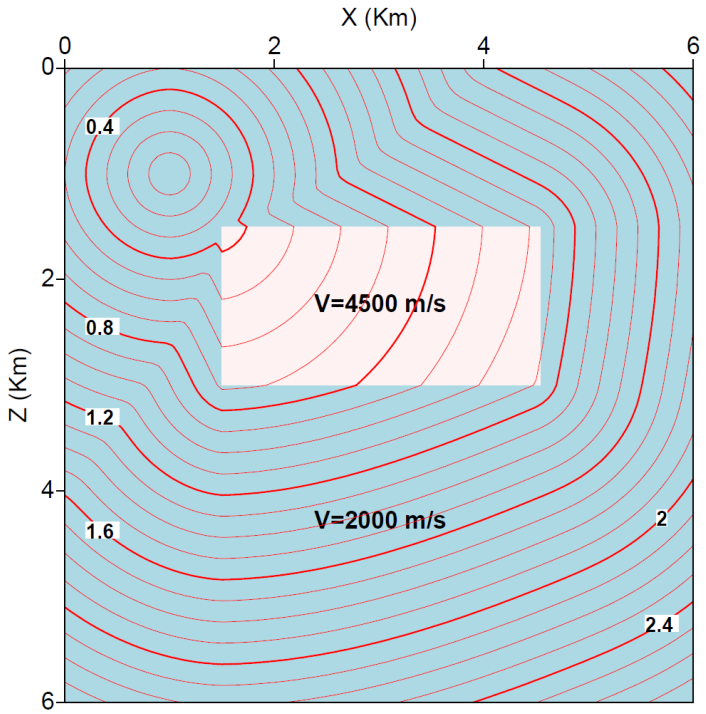


Figura 2.1: Comportamiento de una onda ante velocidades de propagación de onda distintas en un mismo medio[11].

FMM calcula cuanto tiempo tarda en llegar la onda generada desde el punto inicial hasta cualquier punto del espacio, siendo el punto o puntos iniciales los que tienen asociado un tiempo cero.

La velocidad de propagación en el medio de desplazamiento de la onda no tiene que ser siempre la misma, lo que supone, entonces, que el camino más corto no tiene por qué ser, necesariamente, el más rápido.

2.1 Ecuación de la Eikonal

Mediante la ecuación de la Eikonal es posible determinar el movimiento de la onda en el espacio, y de ella se servirá el algoritmo para hacer avanzar el frente de la onda generada.

$$\frac{1}{F(x)} = |\nabla \mathrm{T}(x)|$$

En la ecuación, x es la posición de un punto, F(x) la velocidad a la que una onda puede propagarse en ese punto y T(x) es el tiempo que le toma a la onda llegar a la posición x.

Para la resolución de la ecuación, Osher y Sethian [10] proponen, mediante el uso de una rejilla, discretizar el espacio donde se propaga la onda. Tras la discretización y posterior simplificación del gradiente ∇T, según Sethian, siendo las filas y las columnas denominadas como *i* y *j* de la rejilla respectivamente, lleva a:

$$max\left(\frac{T_{ij} - \min(T_{i-1,j}, T_{i+1,j})}{\Delta x}, 0\right)^2 + max\left(\frac{T_{ij} - \min(T_{i,j-1}, T_{i,j+1})}{\Delta y}, 0\right)^2 = \frac{1}{{F_{ij}}^2}$$

Siendo Δx y Δy la medida de la rejilla en la dirección *x* e *y*, asumiendo velocidad positiva de la onda F > 0, entonces T_{ij} siempre es mayor que $\min(T_{i-1,j}, T_{i+1,j})$ y $\min(T_{i,j-1}, T_{i,j+1})$. Así se puede resolver finalmente para dos dimensiones:

$$max\left(\frac{T_{ij} - \min(T_{i-1,j}, T_{i+1,j})}{\Delta x}\right)^2 + max\left(\frac{T_{ij} - \min(T_{i,j-1}, T_{i,j+1})}{\Delta y}\right)^2 = \frac{1}{{F_{ij}}^2}$$

Para la implementación de los algoritmos se han establecido rejillas cuadradas:

$$\Delta x = \Delta y$$

Para simplificar:

$$T_1 = \min(T_{i-1,j}, T_{i+1,j})$$
$$T_2 = \min(T_{i,j-1}, T_{i,j+1})$$

Entonces:

$$(T - T_1)^2 - (T - T_2)^2 = 1$$
$$T^2 - 2TT_1 + {T_1}^2 + T^2 - 2TT_2 + {T_2}^2 = 1$$
$$2T^2 - 2T(T_1 + T_2) + {T_1}^2 + {T_2}^2 - 1 = 0$$

Resolviendo se puede hallar el valor de T, que será el valor de tiempo de viaje de la onda para llegar a esa celda.

$$T = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

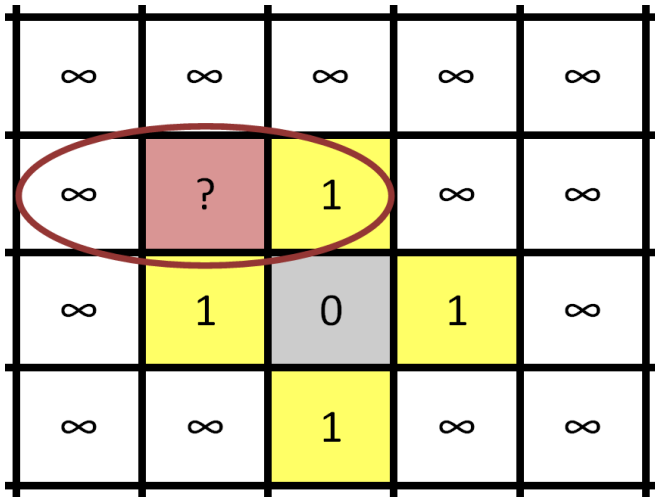
Siendo:

$a = 2$ $(número\ de\ dimensiones)$.

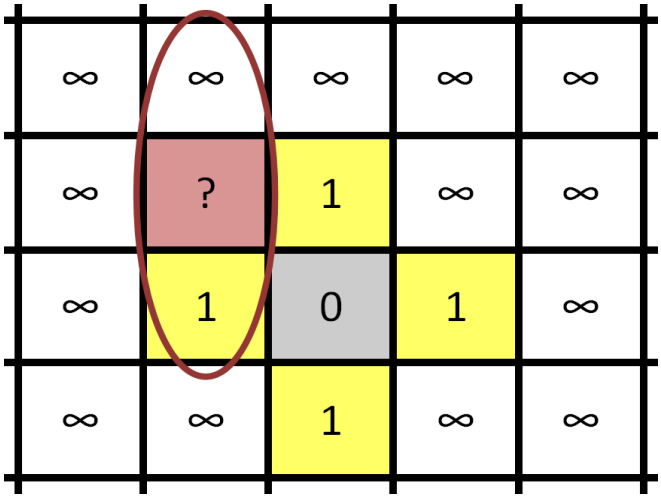
$b = -2(T_1 + T_2)$

$c = {T_1}^2 + {T_2}^2 - \frac{1}{{F_{ij}}^2}$

En la Figura 2.2 a) y Figura 2.2 b) se puede ver un ejemplo de qué valor debe tomar T_1 y T_2, para la resolución del tiempo de llegada de la onda para dos dimensiones.



a) T_1 es el mínimo entre las celdas adyacentes en el eje X



b) T_2 es el mínimo entre las celdas adyacentes en el eje Y.

Figura 2.2: Valores de T_1 y T_2 para la resolución de la ecuación de la Eikonal.

Existe la posibilidad de que la raíz cuadrada, cuando se halla el tiempo, salga negativa. En ese caso la ecuación de la Eikonal no se puede solucionar. Para hallar el valor del tiempo, se calcula mediante:

$$F(x) = \frac{e}{T(x)}$$

Dado que, para las implementaciones, se consideran celdas cuadradas con tamaño unidad:

$$T(x) = \frac{1}{F(x)}$$

Es decir, el tiempo de la celda a calcular, es el tiempo de una celda contigua (T_0), más el tiempo que se tarda de llegar desde la primera celda a la segunda, Figura 2.3.

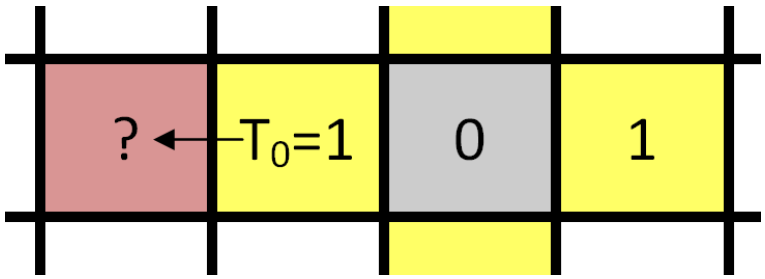


Figura 2.3: Cálculo de tiempo de llegada (TT) cuando no se puede calcular el valor de la ecuación de la Eikonal.

Finalmente, el tiempo de viaje se calculará como:

$$T(x) = \frac{1}{F(x)} + T_0$$

2.2 Fases del algoritmo

Como se ha descrito anteriormente, el espacio donde se propaga la onda ha sido dividido en forma de rejilla.

Para la implementación del algoritmo cada celda de la rejilla tienen que ser etiquetada según el estado en el que se encuentre. Los posibles estados diferentes en los que se puede encontrar una celda pueden ser tres [18]:

- **Frozen**: La onda ya ha pasado por esa celda y se ha calculado un *tiempo de llegada (TT)*. Su valor es definitivo, por lo que no puede ser cambiado.

- **Open**: Celdas sin un TT asignado, pues la onda aún no ha llegado y no se ha calculado todavía.

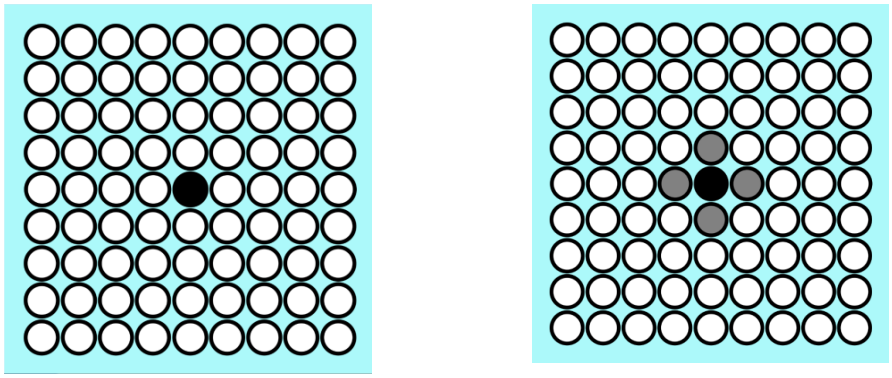
- **Narrow band**: celdas candidatas a ser el frente de onda en la siguiente iteración del algoritmo. Tienen un TT asignado, pero al no ser definitivo puede llegar a cambiar en las siguientes iteraciones.

En la Figura 2.4 y Figura 2.5 se puede observar, gráficamente, cómo funciona el algoritmo. Los puntos negros, grises y blancos corresponden a los estados *Frozen, Narrow y Open*, respectivamente.

A su vez, el algoritmo también consta de tres fases diferenciadas:

-Inicialización: el algoritmo empieza por establecer en todas las celdas que sean puntos iniciales un TT de T=0, además de etiquetarlas como *Frozen*, Figura 2.4 a).

A continuación todas las celdas, que no sean *Frozen*, que estén en la vecindad de von Neumann, son etiquetados como *narrow band*, calculando para cada celda su TT, Figura 2.4 b).



a) ***b)***

Figura 2.4: Etapa de inicialización del algoritmo FMM.

- **Iteración**: En cada repetición, de esta parte del algoritmo, se va a resolver la ecuación de la Eikonal para los vecinos de von Neumann de la celda de menor TT de la *narrow band*. Una vez calculado los TT de los vecinos se añaden a la *narrow band*, y la celda será etiquetada como *Frozen.* Solamente se añaden vecinos, a la *narrow band,* que tengan un estado *Open.*

Encontrar la celda de menor valor TT es posible ya que se mantiene una lista de celdas, ordenadas por sus TT, sabiendo siempre cual es la de menor valor para la próxima iteración, Figura 2.5.

Esta fase continuará mientras la *narrow band* tenga alguna celda almacenada.

- **Finalización**: a esta fase del algoritmo se llega cuando en la *narrow band* no queda ninguna celda, por lo que el algoritmo ha llegado a su fin y no se realizarán más iteraciones. Todas las celdas ya han sido etiquetadas como *Frozen* y todos los TT calculados.

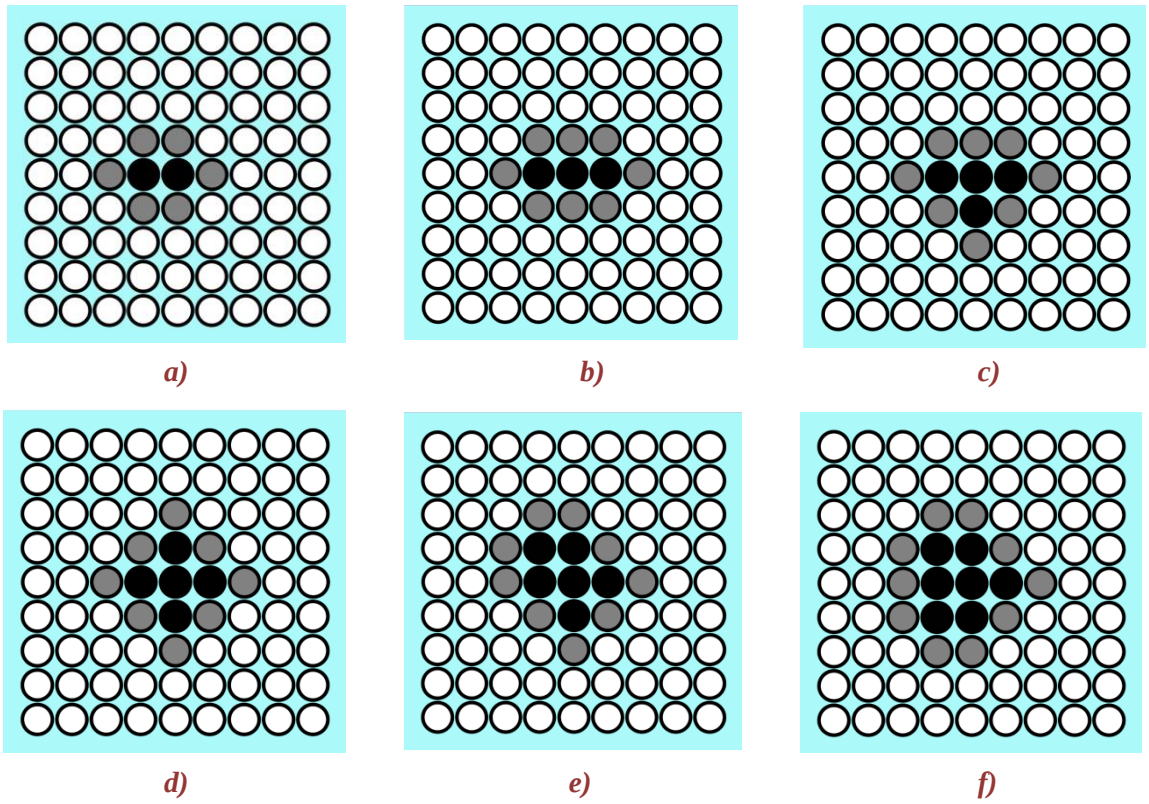


Figura 2.5: Proceso iterativo del algoritmo FMM.

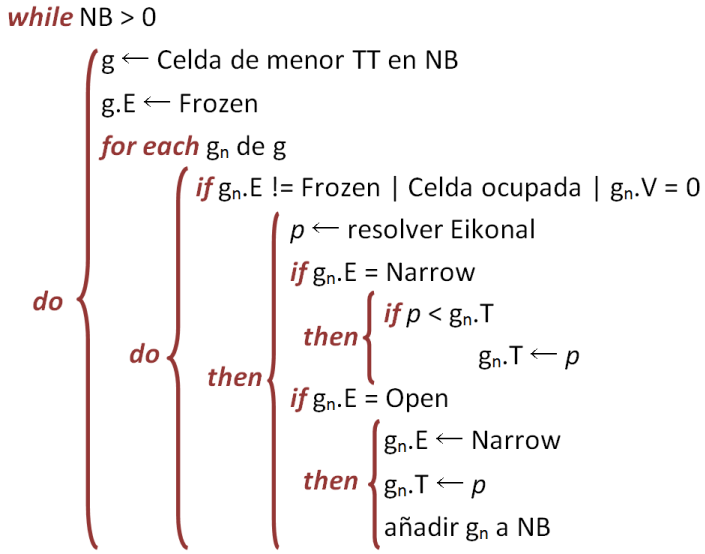
2.3 Pseudocódigo

- Inicialización:

G: rejilla de tamaño m x n.
g_s: celdas donde se origina la onda.
NB: narrow band.
V:velocidad de la celda.
g: celda.
g_n: vecino von Neumann.
T:tiempo de viaje.
E:etiqueta de la celda: Open | Narrow | Frozen.

```
for each g_s ∈ G
do {
    g_s.T ← 0
    g_s.E ← Frozen
    for each g_n de g_s
    do {
        if g_n.E != Frozen | Celda ocupada | g_n.V = 0
        then {
            p ← resolver Eikonal
            if g_n.E = Narrow
            then {
                if p < g_n.T
                    g_n.T ← p
            }
            if g_n.E = Open
            then {
                g_n.E ← Narrow
                g_n.T ← p
                añadir g_n a NB
            }
        }
    }
}
```

- Iteraciones



2.4 Complejidad computacional

Al estar realizando comparaciones de diferentes algoritmos, es necesario hacer hincapié en la complejidad computacional(O), ya que esta es independiente de la potencia de procesamiento donde se ejecute el algoritmo.

La notación O, se usa para expresar los órdenes de crecimiento de los algoritmos en el peor de los casos. Es decir, cuál es el desempeño de un algoritmo, en el peor de los casos, a medida que el tamaño de los datos a operar se incrementa[19].

Sabiendo el orden de complejidad que tiene el algoritmo se va a poder predecir el comportamiento del mismo. Los órdenes de complejidad son:

- **O(1)** - orden constante: no tiene una curva de crecimiento, su desempeño es independiente de la cantidad de datos que maneja.

- **O(n)** - orden lineal: el desempeño del algoritmo es proporcional al crecimiento de los datos que se van a utilizar.

- **O(log n) y O(n log n)** - orden logarítmico: este tipo de orden de complejidad suele significar que el algoritmo trabaja con datos que han sido divididos repetidas veces, como por ejemplo, un árbol binario.
  El término *log n* suele ser $log_2 n$ (aunque no siempre), significando el número de veces que se puede dividir un conjunto en dos partes sin tener como resultado partes subconjuntos vacíos.

$$log_2 1024 = 10$$
$$log_2 2048 = 11$$

Doblar *"n"* tiene muy poco efecto sobre un algoritmo *log n*, eso significa que la curva de crecimiento es muy suave y es más tolerante al aumento del volumen de datos.

- **O(n^2)** - orden cuadrático: el algoritmo tiene un desempeño proporcional al cuadrado de los datos introducidos. Estos algoritmos suelen existir cuando, teniendo una serie de datos, es necesario trabajar con cada uno de los datos dos veces.
  Un ejemplo perfecto, de este tipo de complejidad, es el algoritmo de ordenación de burbuja[20].

- **O(n^a)** - orden polinomial(a>2): este tipo de algoritmos tiene bucles internos que le hacen tener que usar cada dato en repetidas ocasiones.

- **O(a^n)** - orden exponencial(a>2): este tipo de algoritmos, en el caso de a=2, dobla el tiempo usado en el cálculo por cada nuevo elemento que se añade. No son muy útiles en la práctica, ya que con pocos datos es necesario una gran potencia computacional.

- **O(n!)** - orden factorial: son algoritmos con tan poca escalabilidad que son intratables.

Suponiendo que se tiene un problema de tamaño *"n"*, cada algoritmo de cada complejidad computacional resuelve el problema en un determinado tiempo. Cuando el problema tiene el doble de tamaño, siendo ahora *"2n"*, cabría esperar que tardasen el doble de tiempo en resolver el problema, pero no sucede así. Por ejemplo, los algoritmos de orden logarítmico, tardarán poco más en resolver el problema dos veces mayor y los exponenciales tardarán varias veces más que el doble, respecto al problema *"n"*.

Debido a que se tiene conocimiento de la complejidad computacional de cada algoritmo, se puede predecir cuánto tardará cada uno en resolver el problema *"2n"*. En la Figura 2.6 se puede observar gráficamente cada uno de estos órdenes computacionales según crece el número de datos.

Se tiene que tener en cuenta que **escalabilidad** no significa **eficiencia**. Así, un algoritmo de complejidad menos escalable puede arrojar mejores resultados que otro con mejor escalabilidad y peor eficiencia. Finalmente el volumen de datos se sobrepondría a la eficiencia, pero es importante saber que mejor escalabilidad no significará mejores tiempos de cálculo, sobre todo con poco volumen de datos.

En el caso de este trabajo, el tamaño de problema "*n*", viene dado por el número de celdas de la rejilla que componen el mapa.

FMM es un método que tiene una complejidad *O(n log n)*, siendo el factor *log n* introducido por la administración de la cola de prioridad, al tener que obtener la celda de menor TT de la *narrow band* en cada iteración.

En este trabajo, el algoritmo *Fast Marching Method* utilizará tres colas de prioridad diferente. Cada una de estas colas de prioridad tiene complejidad computacional *log n* y

serán las tres modificaciones del algoritmo FMM que se usarán para comparar en los experimentos posteriores.

Como se puede observar en la Figura 2.6, la complejidad óptima para este tipo de algoritmos es la de orden lineal *O(n)*. Los algoritmos que se han implementado y que se explicarán en los siguientes capítulos, tienen este tipo de complejidad.

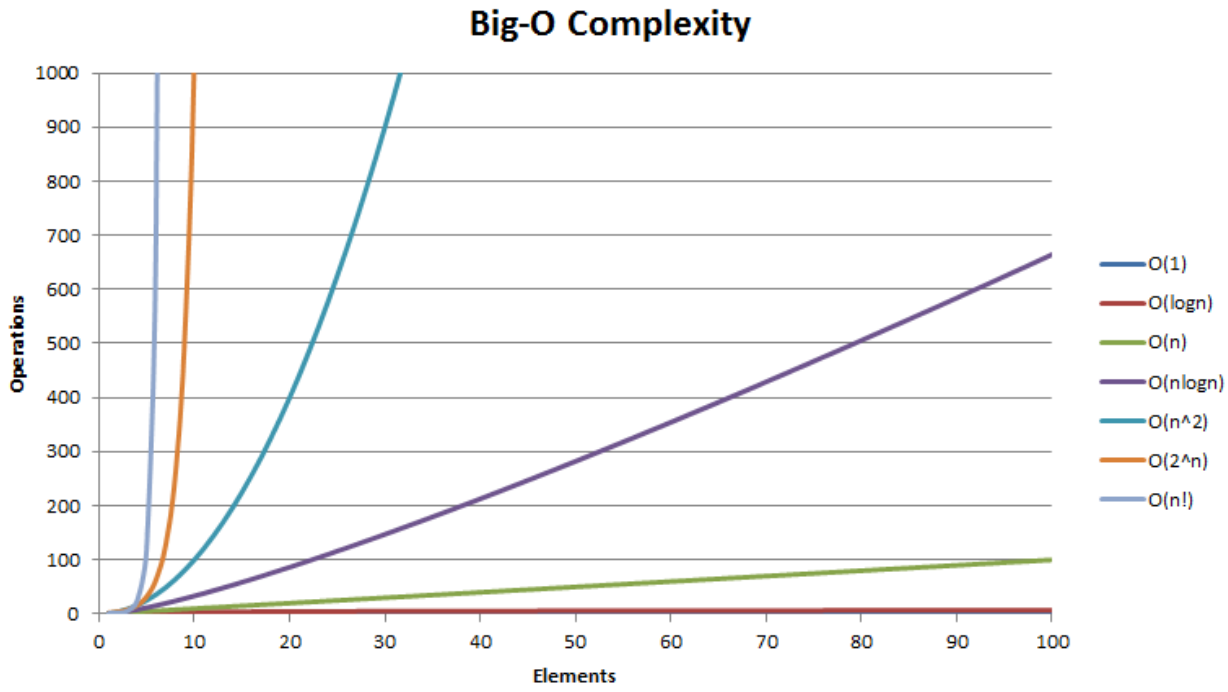


Figura 2.6: Gráficas de cada uno de los órdenes de complejidad computacionales.

2.5 Modificaciones de Fast Marching

Como se ha comentado en anteriores capítulos, se van a usar diferentes colas de prioridad para comparar con los algoritmos del Capítulo 3. Estas tres modificaciones, cada una con su cola de prioridad, no se han implementado pero en este capítulo se explicará el funcionamiento de cada modificación. Pudiendo entender mejor las comparaciones con los demás algoritmos, los cuales sí se han implementado.

Los algoritmos modificados tienen el mismo funcionamiento que FMM, explicado en los puntos 2.2 y 2.3, pero cambiando la cola de prioridad que controla *la narrow band*, por otra que introduce cada modificación.

Las colas de prioridad que se han usado, han sido: una pila binaria, una pila de Fibonacci y una cola de prioridad de la librería Boost. La complejidad computacional de cada una de las operaciones se puede observar en la Tabla 2.1.

Las operaciones básicas de las colas de prioridad son:

- Obtener mínimo: se extrae el menor dato de los datos guardados. Es una operación sencilla ya que la cola de prioridad ordena, por valor, los datos introducidos.
- Borrar mínimo: el menor dato es borrado.
- Insertar: añade un nuevo dato a la cola de prioridad.
- Incrementar/decrementar clave: aumenta o disminuye el valor de un dato de la cola de prioridad.
- Unir: se combinan dos colas de prioridad.

	Encontrar mínimo	**Extraer mínimo**	**Incrementar clave**	**Insertar**	**Borrar**	**Unir**
Pila binaria	O(1)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Pila de Fibonacci	O(1)	O(log(n))*	O(1)*	O(1)	O(log(n))	O(1)
Boost - Priority queue	O(1)	O(log(n))	-	-	-	O(m+n)
*: Significa tiempo amortizado						

Tabla 2.1: Complejidad computacional de cada operación de la cola de prioridad [21][22].

2.5.1 Con pila de Fibonacci

Se trata de una estructura de datos, muy parecida a la pila binaria. Su principal característica es la amortización de procesos, es decir, que hace que las operaciones rápidas tarden un poco más, pero a cambio las operaciones lentas se aceleran.

En la Tabla 2.1 se puede observar la complejidad computacional de la pila de Fibonacci. Como se decía en anteriores capítulos, la complejidad del uso de esta pila, para obtener el menor valor TT de la *narrow band,* es *O(log n).*

La pila de Fibonacci [23][24] se basa en el uso de varios árboles formados por nodos. Estos árboles tienen la propiedad de que un hijo de un nodo tiene siempre igual o mayor valor que el padre. Debido a esta propiedad, la raíz siempre será el nodo con menor valor.

La amortización de procesos, de la que se ha hablado anteriormente, se mide mediante el potencial de la pila:

$$Potencial = t + 2m$$

Siendo *t* el número de árboles y *m* el número de nodos marcados. Para considerar un nodo como marcado, tiene que haberse eliminado un hijo suyo y ser, a su vez, hijo de otro nodo cualquiera. Por ello, los nodos raíz no pueden ser marcados. Así, por ejemplo, si se tienen 3 árboles y 3 nodos marcados le potencial es de 9.

Para un rápido borrado y concatenado, las raíces de todos los árboles son una *double linked list*. Se puede observar una representación gráfica de la pila de Fibonacci en la Figura 2.7.

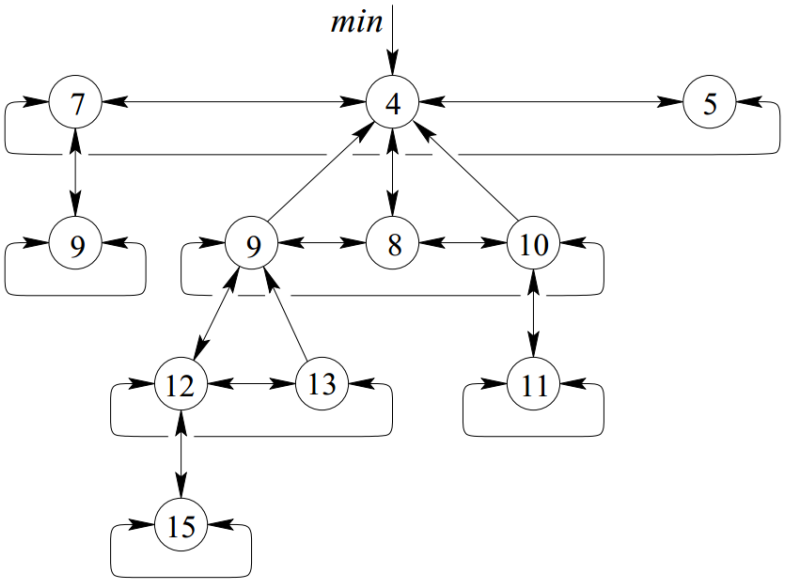


Figura 2.7: Representación gráfica de los árboles que forman la pila de Fibonacci[24]

Las operaciones básicas de una pila de Fibonacci [24] son:

- Obtener mínimo: es una operación muy sencilla, ya que el valor mínimo corresponde al dato situado en la raíz.

- Unir: se pueden unir dos pilas de Fibonacci. Para ello, se añade la pila con mayor valor en la raíz, como hijo de la otra pila. Como se puede observar en la Figura 2.8.

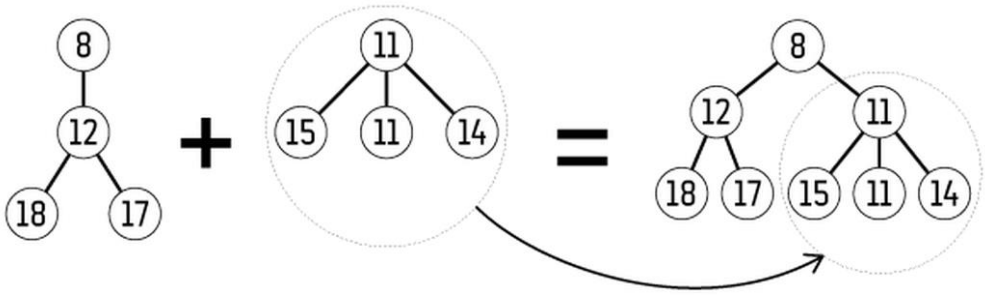


Figura 2.8: Proceso de unión de dos pilas de Fibonacci en una sola[25].

- Insertar: para añadir un nuevo dato, solamente es necesario realizar la operación de "unión" con una pila de Fibonacci nueva. Esta nueva pila solamente contendrá el nuevo dato a añadir.

- Borrar mínimo: el valor mínimo es borrado. Para ello:
  - Se borra el nodo con la menor clave (la raíz).
  - Unión de los hijos del nodo eliminado.
  - Mientras existan dos raíces con el mismo grado, se realiza la operación "unir". El grado de un nodo es equivalente a la distancia que se encuentre de la raíz. Así, el hijo del nodo raíz, tiene grado uno.

  Un ejemplo se puede observar en la Figura 2.9.

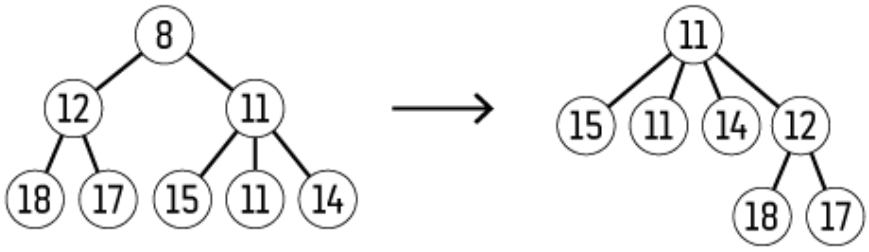


Figura 2.9: Proceso de borrado del dato con el menor valor de una pila de Fibonacci[25].

- Decrementar clave: Se disminuye el valor de un dato en la pila de Fibonacci. Se realiza de la siguiente manera:
  -Separar el nodo a decrementar del árbol al que esté conectado.
  -Reducir el valor del nodo.
  -Añadir el nodo decrementado al árbol.

  Un ejemplo gráfico se puede observar en la Figura 2.10, en el que se desea decrementar el nodo de valor 12 a valor 2.

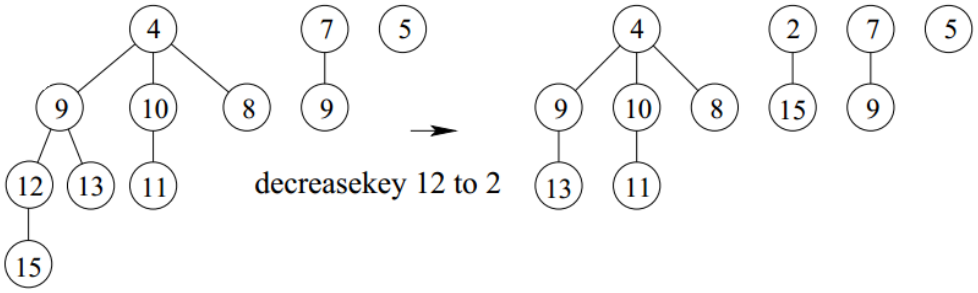


Figura 2.10: Proceso de decremento de clave de un dato en una pila de Fibonacci[24].

2.5.2 Con pila binaria

Al igual que la pila de Fibonacci, la pila binaria[26] es una estructura de datos que se basa en árboles y nodos de complejidad *O(log n)*.

Tiene la característica de que cada nodo tiene un valor superior al de sus hijos (pila binaria de máximos), o al revés, más pequeño que el de sus hijos (pila binaria de mínimos). Obviamente la versión que interesa para los algoritmos, es la de mínimos. Un ejemplo de una pila binaria de máximos se puede observar en la Figura 2.11.

Para poder usar un array como pila binaria, como en la Figura 2.11, es necesario tener en cuenta las posiciones de la raíz y de dónde se encuentran sus hijos. El nodo raíz siempre va en la posición 0, y la posición de los dos hijos de cada nodo, se puede calcular de manera sencilla, como:

$$Posición\ K\ hijo\ 1 = 2K + 1$$

$$Posición\ K\ hijo\ 2 = 2K + 2$$

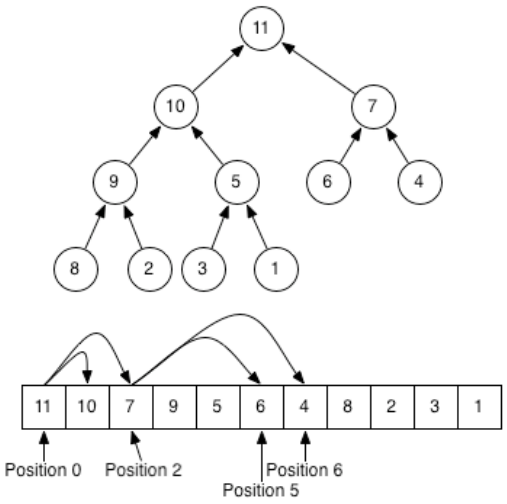


Figura 2.11: Representación de árbol binario y el array equivalente.

El algoritmo, usado en las comparaciones, tiene una modificación de la pila binaria llamado, pila *d-ary*[27].

Esta modificación tiene la capacidad de hacer más rápidas las operaciones de **decrementar prioridad** a costa de hacer más lenta la operación de **borrar mínimo**.

Esto permite obtener mejores tiempos en los algoritmos en los que decrementar prioridad se usa más veces. Estas operaciones son más comunes en los algoritmos como el de Dijkstra o como es el caso, FMM.

La estructura consiste en un array de *n* elementos que tiene las mismas propiedades que la pila binaria.

Las operaciones básicas de una pila binaria [26] son:

- Insertar elemento: se realiza, agregando un elemento en la posición que respeta la condición del árbol semicompleto, es decir, que las inserciones se realicen de izquierda a derecha por orden.
  Una vez colocado en su posición, debe cumplirse que los hijos sean mayores. Por lo que se intercambian posiciones de hijos a padres, mientras el hijo sea menor, hasta llegar a la raíz en caso de que fuese necesario. El proceso se puede observar, gráficamente, en la Figura 2.12.

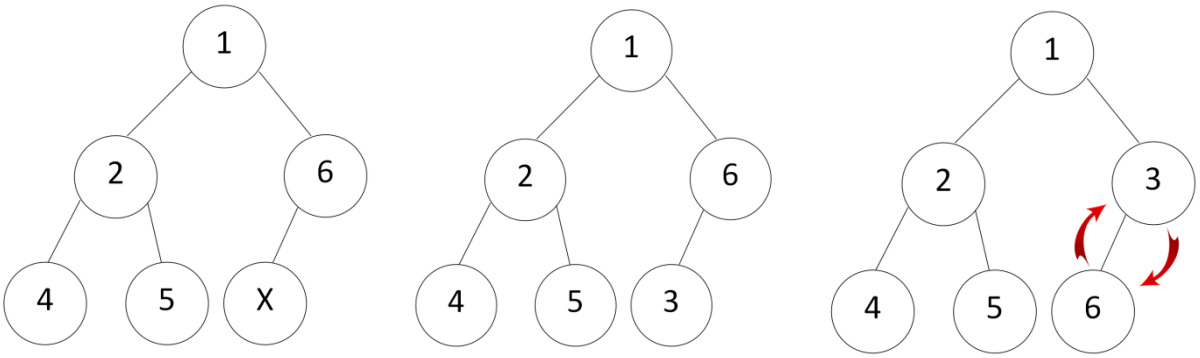


Figura 2.12: Proceso de inserción de un dato (X=3), en una pila binaria.

- Extraer mínimo: se extrae el elemento que esté en la raíz, pues es el dato de menor valor. Luego se comprueba cuál de sus hijos tiene menor valor y se convierte en la nueva raíz, si se da el caso, se intercambian con los hijos mientras la raíz sea mayor. El proceso se puede observar en la Figura 2.13.

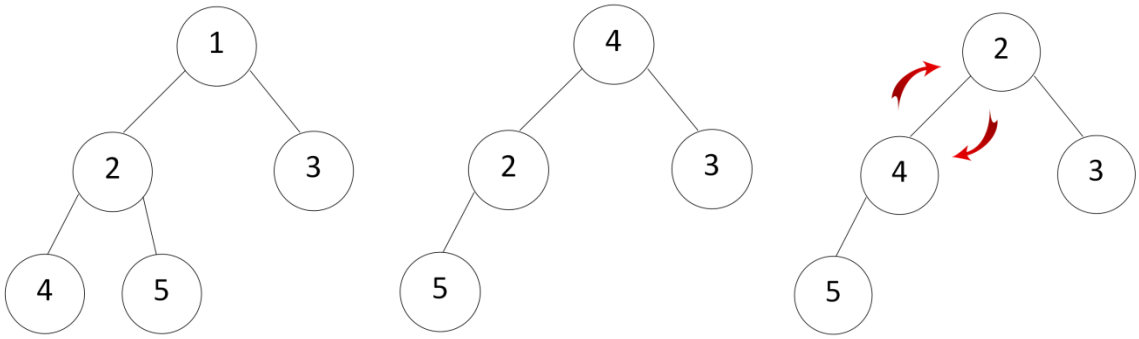


Figura 2.13: Proceso de extracción de la raíz en una pila binaria.

2.5.3 Fast Marching simplificado

Esta modificación aparece para resolver el problema de que en FMM, las celdas en la cola de prioridad pueden volver a ser calculadas. Si cualquier celda es recalculada, es necesario que se muevan dentro de la cola de prioridad, necesitando así, controladores para manejar la cola.

Para evitar el uso de estos controladores, se propone usar el algoritmo SFMM *(Simplified Fast Marching Method*), que se obtiene de los siguientes cambios[14]:

- Si se ha recalculado un valor para una celda, se acepta que tenga dos valores distintos. Cuando se necesita saber el valor de la celda, el valor más pequeño es el que mira primero y la celda pasa al estado *Frozen*. En caso de aparecer un segundo valor, este es descartado. Si no se usara este mecanismo, aparecerían errores debido al recálculo de las celdas.

- Los errores también aparecen si las celdas que no son *Frozen*, se usan para el cálculo de los nuevos valores de las celdas en la cola de prioridad. Entonces, no es necesario asignar un valor a las celdas antes de que sean etiquetadas como F*rozen*.

Estas simplificaciones implican que se puede usar una cola de prioridad estándar. En el caso de los experimentos se ha usado la cola de prioridad de la librería Boost.

Capítulo 3:

MODIFICACIONES O(N) DEL ALGORITMO FAST MARCHING METHOD

Como se ha descrito en anteriores capítulos, es muy importante el orden de complejidad de los algoritmos para que sean eficientes. Debido a que en el algoritmo FMM sus modificaciones no son del orden de complejidad óptimo, se han desarrollado otras variaciones que sí que lo cumplan. A continuación se explicarán tres de estos algoritmos que se han implementado. Cada uno de ellos tiene una complejidad computacional *O(n)*.

3.1 The Group Marching Method

Group Marching Method [11], en adelante GMM. Se trata de una modificación de FMM desarrollada por Seongjai Kim y Daniel Folie, de complejidad *O(n)* al evitar usar una cola de prioridad compleja para controlar la *narrow band*. Así, GMM es capaz de mejorar la eficiencia de FMM pero manteniendo la misma precisión.

Este algoritmo se basa en la técnica de la *narrow band*, como se ve en la Figura 3.1, en la cual los puntos dentro de ella forman el frente de avance de la onda generada. Al contrario que en FMM, el algoritmo no seleccionará la celda de la *narrow band* de menor valor. A cambio, hará avanzar varias celdas al mismo tiempo. Este avance se produce mediante una doble iteración en cada celda, aunque no dobla el tiempo de cálculo de los *tiempos de llegada*(TT), sino que el tiempo usado es ligeramente mayor a una sola iteración.

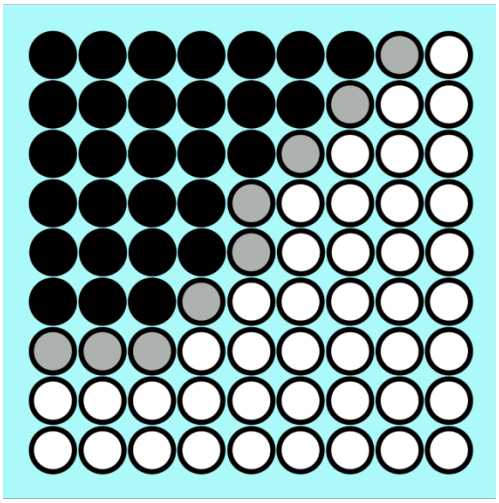


Figura 3.1: Expansión de la onda en GMM, puntos blancos, negros y grises corresponden a los estados: Open, Frozen y Narrow respectivamente.

Para la resolución de los *tiempos de llegada*, el algoritmo se sirve de la ecuación de la Eikonal desarrollada en el punto 2.1. Además del sistema de etiquetas *Open, Narrow y Frozen*, que permite saber el estado en el que se encuentra cada celda de la rejilla.

Como se ha mencionado, GMM evita usar una cola de prioridad, pero a cambio tiene que mantener una lista de celdas. De esa lista de celdas llamada *Gamma Γ*, varias de ellas podrán avanzar tal y como si se tratasen de la celda con menor TT de la *narrow band* en FMM.

Los autores han creado un método para decidir que celdas son las que pueden hacer avanzar el frente de onda. Dadas dos celdas cercanas de la *narrow band*, si la diferencia entre sus tiempos de llegada son menores que:

$$\text{Diferencia TT} \leq \frac{1}{\sqrt{D}} \cdot h \cdot s$$

Siendo *D*, el número de dimensiones, *h* la longitud de la celda en esa dimensión y *s* la lentitud, es decir, la inversa de la velocidad de expansión de la onda $\left(\frac{1}{F}\right)$.

Si se cumple esta condición, quiere decir que la línea que une las dos celdas y la dirección hacia la que se expande la *narrow band* es mayor que 45 grados, como se puede comprobar en la Figura 3.2.

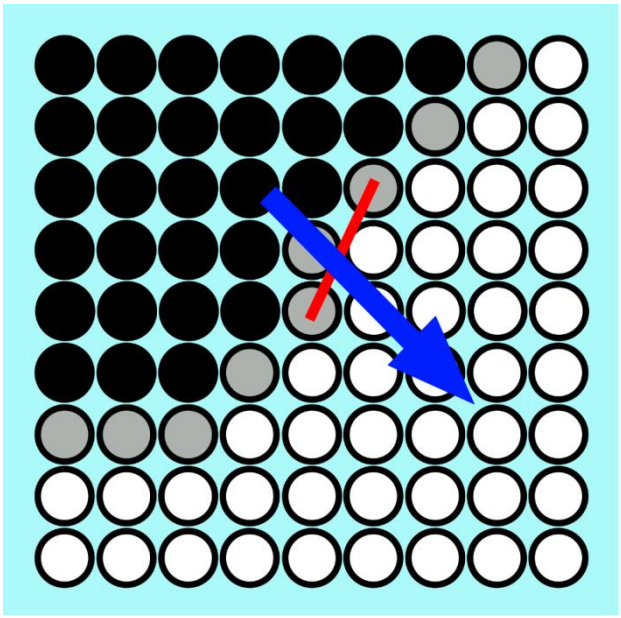


Figura 3.2: Ángulo entre la dirección de expansión de la onda (flecha azul) y la línea que une dos celdas (línea roja).

Si las dos celdas no están juntas, el procedimiento de actualización del *tiempo de llegada* (TT), resuelto por la ecuación de la Eikonal, no afectan una a la otra. En caso de ser adyacentes, el TT de una afecta a la otra en caso de que la dirección de expansión de la onda y la línea que une ambas celdas, sea menor de 45 grados.

En conclusión, dos celdas no se afectan entre ellas y pueden ser expandidas a la vez en una misma iteración, si:

$$G = \left\{ x \in \Gamma : \mathrm{T}(x) \leq \mathrm{T}_{\Gamma,min} + \frac{1}{\sqrt{D} F_{\Gamma,min}} \right\}$$

Siendo *G* el conjunto de celdas que serán avanzadas de una sola vez en la iteración. Sin embargo, para resolver la ecuación es necesario tener un conocimiento del tiempo de llegada mínimo, $\mathrm{T}_{\Gamma,min}$, de todas las celdas de *Gamma* y de la velocidad $F_{\Gamma,min}$ en cada iteración posterior. Para encontrar estos valores, es necesario un algoritmo *O(n)*. Por lo que los autores sugieren usar un límite global, e ir incrementándolo según la onda se propague.

Este valor será llamado *TM*, y es hallado al principio del algoritmo una sola vez. El valor de *TM* corresponde al valor del mínimo TT de la *narrow band* inicial, y es incrementado en cada iteración por:

$$delTAU = \frac{1}{F_{global,max}}$$

Esta simplificación tiene como consecuencia un empeoramiento de la eficiencia cuando los contrastes de la velocidad de expansión de la onda son grandes. La baja eficiencia se debe a que si *delTAU* tiene un valor bajo, en cada iteración pocas celdas son capaces de expandirse. Por lo tanto, el algoritmo necesita más iteraciones para completar todas las celdas, tardando más que con un *delTAU* de mayor valor.

Para evitar inconsistencias cuando dos frentes de onda chocan, debido a que la vecindad de celdas común puede ser afectada por ambos frentes, se realiza una doble actualización de las celdas de *Gamma* que se expandirán en esa iteración.

3.1.1 Fases del algoritmo

En la Figura 3.3, se puede observar cómo se expanden varios puntos a la vez en cada iteración, al contrario que en FMM. Para el cálculo de todos los *tiempos de llegada* (TT) se realizarán las siguientes fases:

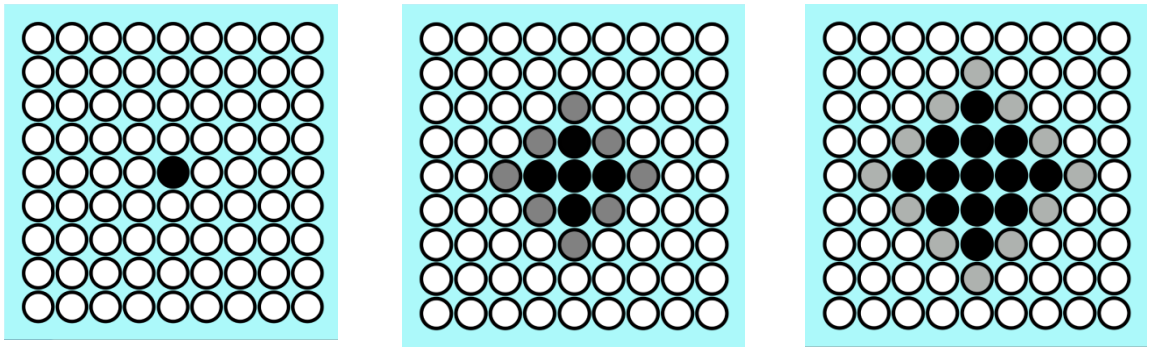
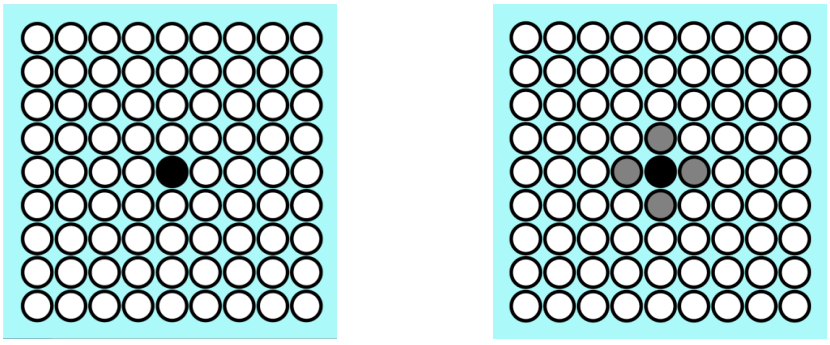


Figura 3.3: Proceso de expansión de varios puntos en GMM en una sola iteración.

-Inicialización: Se asignan a todas las celdas de la rejilla valor infinito, o un valor muy alto en su defecto. También se calcula el valor *delTAU* del que se ha hablado anteriormente.

Se hallan los *tiempos de llegada(TT)* de los vecinos von Neumann adyacentes a la celda o celdas donde se inició la onda y se etiquetan como *Narrow* al guardar esas celdas en *Gamma*. Ver Figura 3.4.

Guardar el mínimo valor de TT de las celdas halladas anteriormente, que fueron añadidas a *Gamma*, como el valor de *TM*. Finalmente, etiquetar como *Frozen* las celdas donde ser origina la onda.



a) *b)*

Figura 3.4: Etapa de inicialización del algoritmo GMM.

- **Iteración**: Mientras que *Gamma* contenga alguna celda, esta etapa se repetirá. Se recalcula el valor de TM en cada nueva iteración:

$$TM = TM + delTAU$$

Para cada celda de *Gamma*, en el orden inverso de cómo fueron añadidos, que cumpla:

$$\text{TT de la celda} \leq \text{TM}$$

Se calculan los TT de todos sus vecinos von Neumann que sean *Narrow* u *Open.* Si el tiempo calculado es menor que el que ya se tenía guardado, se sustituye, Figura 3.5 a), b), c) y d).

Para cada celda de *Gamma* en el orden directo, que cumplan la condición de ser menor que TM. Se calculan lo TT de los vecinos que sean *Narrow* u *Open*(sustituyendo si el nuevo valor TT es menor que el que ya tenía). Si las celdas eran *Open*, se añaden a *Gamma* y se cambia su etiqueta a *Narrow*.

Finalmente se eliminan de *Gamma* las celdas que han cumplido la condición de TM y cambiando sus etiquetas a *Frozen*, Figura 3.5 e), f), g) y h).

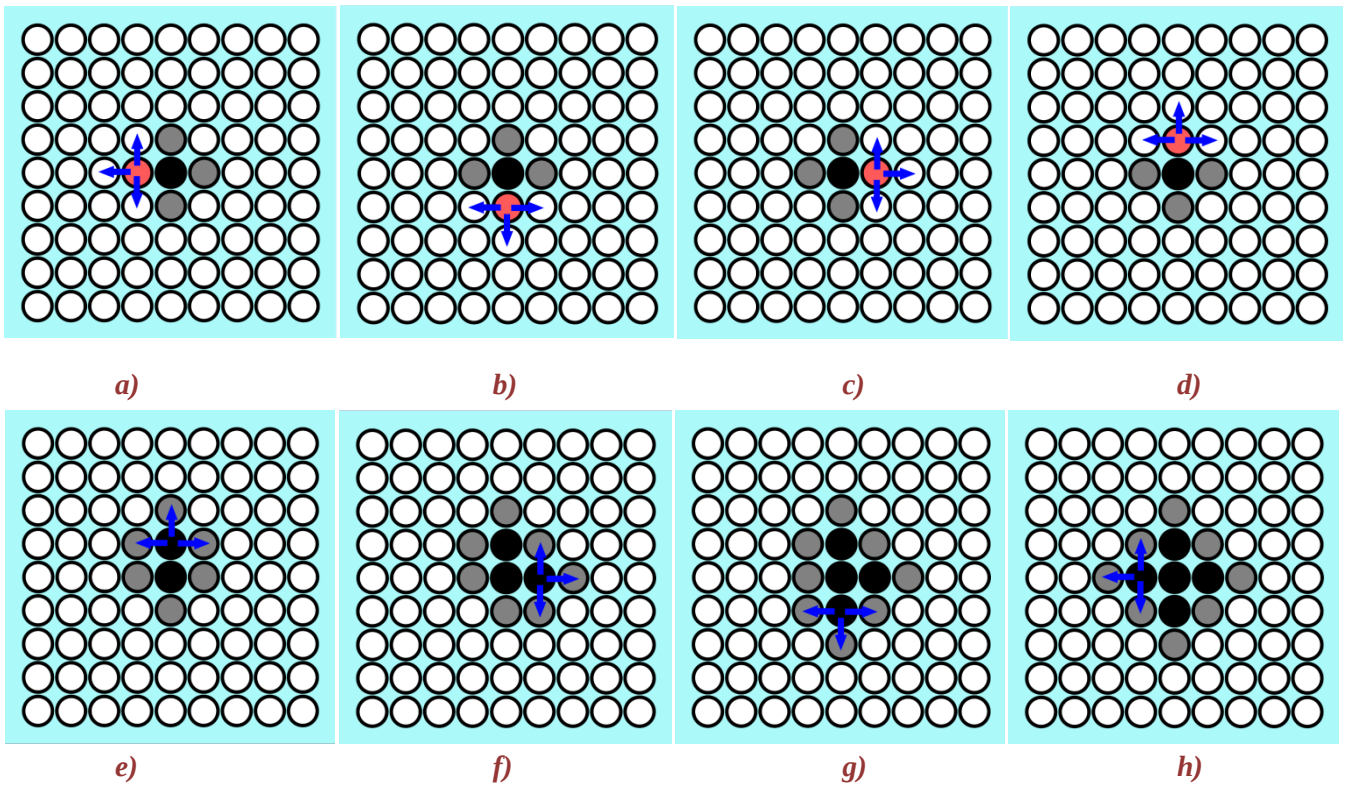


Figura 3.5: Proceso iterativo del algoritmo GMM en dos partes. Expansión de la narrow band(punto rojo) en orden inverso y después directo.

- **Finalización**: Todos las celdas han sido etiquetadas como F*rozen* y *Gamma* no contiene ninguna celda, por lo que se ha llegado a la última iteración y el algoritmo acaba al tener todas las celdas un TT asignado.

3.1.2 Pseudocódigo

- Inicialización:

G: rejilla de tamaño m x n.
g_s: celdas donde se origina la onda.
NB: narrow band.
V:velocidad de la celda.
g_r : celda perteneciente a gamma.
g: celda.
g_n: vecino von Neumann .
T:tiempo de viaje.
E:etiqueta de la celda: Open | Narrow | Frozen.

```
for each g ∈ G
    do  g.T ←∞
for each gs ∈ G
    do  gs.T ← 0
delTau ← 1 / F_global,max
TM←∞
for each gn de gs
    do  gn.T← resolver Eikonal
        if gn.T <  TM
                TM← gn.T
        gn.E← Narrow
        Γ← gn
gs.E ← Frozen
```

-Iteraciones

```
while Γ > 0
    do  TM←TM + delTAU
        for each  gr en el orden inverso
            do  if  gr .T ≤ TM
                then  for each gn de gr
                          do  if  gn .E  =  Narrow| Open
                              then  p ← resolver Eikonal
                                    if p < gn.T
                                          gn.T ← p
        for each  gr en el orden directo
            do  if  gr .T ≤ TM
                then  for each gn de gr
                          do  if  gn .E  =  Narrow| Open
                              then  p ← resolver Eikonal
                                    if p < gn.T
                                          gn.T ← p
                      if  gr .E  = Open
                          do  Γ← gn
                              gn.E← Narrow
                      Eliminar  gr de Γ
                      gr .E← Frozen
```

3.1.3 Comentarios sobre la implementación

Los autores del algoritmo no proponen la utilización de una determinada estructura de datos para la implementación de *Gamma*, por ello, en este trabajo, se ha probado con dos tipos de estructuras.

En un principio se probó con un vector. El problema ocurría en rejillas de grandes proporciones, el vector se hace muy grande al tener que almacenar una *narrow band* enorme, sobre todo en 3D. En caso de tener que borrar un elemento en mitad del vector, se tienen que mover todos los elementos una posición a la izquierda para cubrir el hueco, como se puede ver en la Figura 3.6. Esto hace que aumente, de manera considerable, el tiempo de cálculo. Por lo que se optó por usar una *double linked list*.

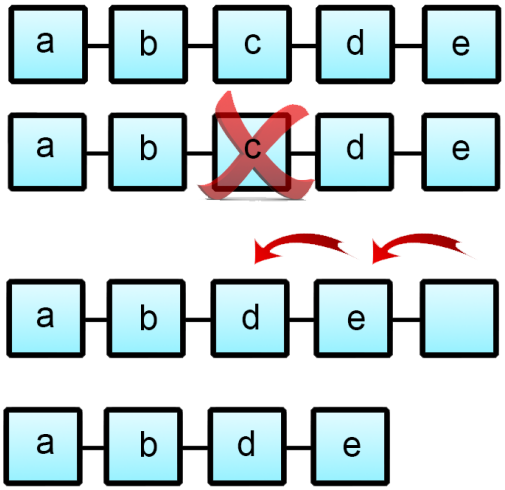


Figura 3.6: Proceso de eliminación de un elemento en un vector.

En una lista de este tipo, se tiene referencia de dónde está el siguiente dato y el anterior, pero no están seguidos en la memoria. Esto tiene como ventaja que es mucho más simple borrar un dato almacenado en cualquier posición, ya que solo hace falta escribir en el puntero de direcciones las nuevas direcciones de memoria para sustituir al eliminado, Figura 3.7.

Así, el tamaño de la *narrow band* no interfiere con el algoritmo.

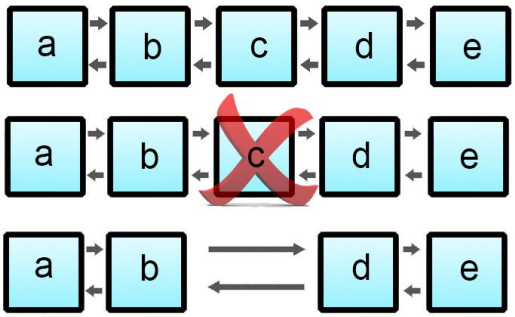


Figura 3.7: Proceso de eliminación de un elemento en una double linked list.

3.2 Untidy Fast Marching Method

Como se ha visto en el punto 2.4, la cola de prioridad evita la optimización del algoritmo según crece el número de celdas. Para obtener el deseado orden de complejidad O(n), L.Yatziv, A.Bartesaghi, y G.Sapiro [12], han optado por crear una cola de prioridad que no introduzca ordenes complejos. Obteniendo así, una mejor eficiencia respecto a FMM e introduciendo errores de la misma magnitud que cuando se realiza la discretización del espacio en celdas.

Este algoritmo, a partir de ahora UFMM, es una extensión del algoritmo de Dijkstra y se basa en encontrar en cada iteración la celda con menor tiempo de llegada de la *narrow band*, al igual que FMM.

Como se vio en FMM, para encontrar esta celda era necesario usar un cola de prioridad de orden *O(log n).* Para evitar su uso, los autores proponen usar una estrategia de los métodos de *"fast sweeping algorithms"*. Este tipo de estructura ha sido nombrada como *"Untidy Priority Queue"*, que cambia la complejidad de la cola de prioridad de *O(log n)* a *O(1).*

3.2.1 Funcionamiento de la "Untidy Prioriy Queue"

Esta cola de prioridad se basa en la propiedad de que el valor de los *tiempos de llegada* (TT) de todas las celdas añadidas a la cola, siempre son mayores o iguales al de la última celda extraída.

Este tipo de colas se llaman *colas monótonas de prioridad*, y se usarán en este algoritmo porque la *narrow band* tiene un comportamiento monótono.

Dado que el valor de la velocidad está limitado, se puede asumir que los TT de las celdas de la cola tienen un incremento máximo sobre el mínimo TT de la cola de prioridad [12]. Debido a que los valores TT de las celdas están dentro de un rango máximo, es posible realizar una estructura de datos basada en un array circular, Figura 3.8.

Cada entrada del array circular, contiene una lista de puntos de similar valor TT, en el que añadir y eliminar tiene una complejidad O(1). Estas colas de prioridad basadas en arrays circulares se denominan *calendar queue*[28].

En una *calendar queue*, *Pmin* es la celda de la cola con menor TT y *Pr* es la próxima celda con *tiempo de llegada Tr* que será eliminada, pudiéndose garantizar que *Pr=Pmin.*

Debido a que FMM introduce errores, los autores usan la *untidy priority queue*, en donde *Pr* ≈ *Pmin*. Es decir, la cola propuesta es una simplificación de la *calendar queue.*

Cada entrada del array representa una discretización de una valor de tiempo T uniformemente distribuida. En la Figura 3.8, estos valores discretizados de tiempo se corresponden con T0, T1, T2..., cada discretización es una lista FIFO(*First In, First Out*), es decir, el primer dato que entra en la pila, es el primero que sale de ella. En cada lista FIFO se guardan valores de tiempo de llegada parecidos.

En la Figura 3.8, los valores discretizados de tiempo T1, T2, T3... equivalen a T0+Δ, T0+2Δ, T0+3Δ...respectivamente, siendo Δ la diferencia, en tiempo, entre dos niveles consecutivos. Dado que en ese ejemplo, el máximo incremento es 7, los valores TT van a estar fijos entre [T0, T0 + 7Δ].

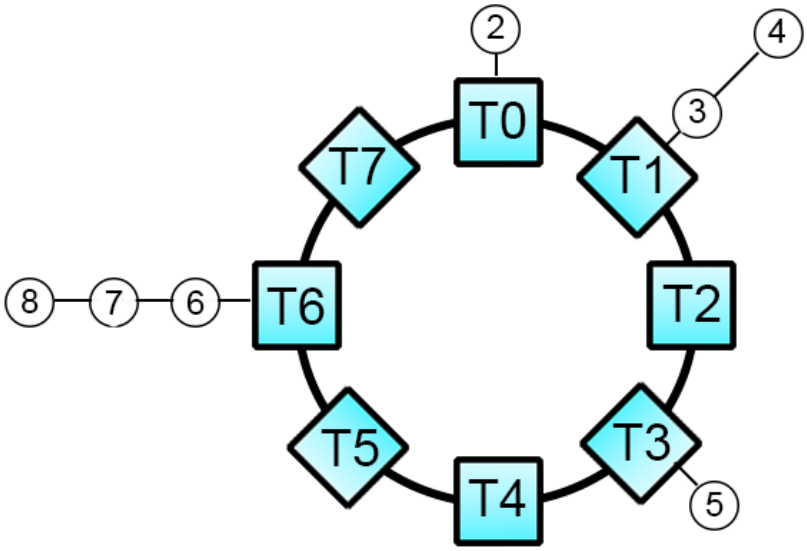


Figura 3.8: Representación del array circular y de las discretizaciones de tiempo T del algoritmo untidy priority queue.

La cola de prioridad tiene como referencia la dirección memoria $L_{0,}$ dónde está la localización de T0. Cuando se añade una nueva celda P_a, con valor de tiempo de llegada T_a al array circular, se colocará en la discretización de tiempo T_j que corresponda. Esta posición se calcula como:

$$j = \frac{T_a - T_0}{\Delta}$$

Para encontrar la entrada de memoria L_j que corresponde a la discretización j, basta con realizar la siguiente operación:

$$L_j = \left| \frac{j - L_0}{\Delta} \right|$$

La celda con menor TT de la cola de prioridad, estará en la discretización de tiempo T0. Debido a que se introducen errores en la discretización, *Pr* no tiene que ser necesariamente la celda de menor TT, *Pmin*. Como ocurre en el ejemplo de la Figura 3.9.

Cuando la discretización T0, ya no tiene más celdas porque han sido borradas, se usará la siguiente discretización que no esté vacía. La celda *Pr* será buscada en esa discretización ya que contendrá la celda con menor TT, es decir, será la nueva T0. Se puede observar el proceso gráficamente comparando la Figura 3.8 y Figura 3.9.

El pseudocódigo y las fases del algoritmo son iguales que en el algoritmo FMM visto en el punto 2.2, ya que solo cambia la utilización de la cola de prioridad *O(log n)* por la *untidy priority queue*.

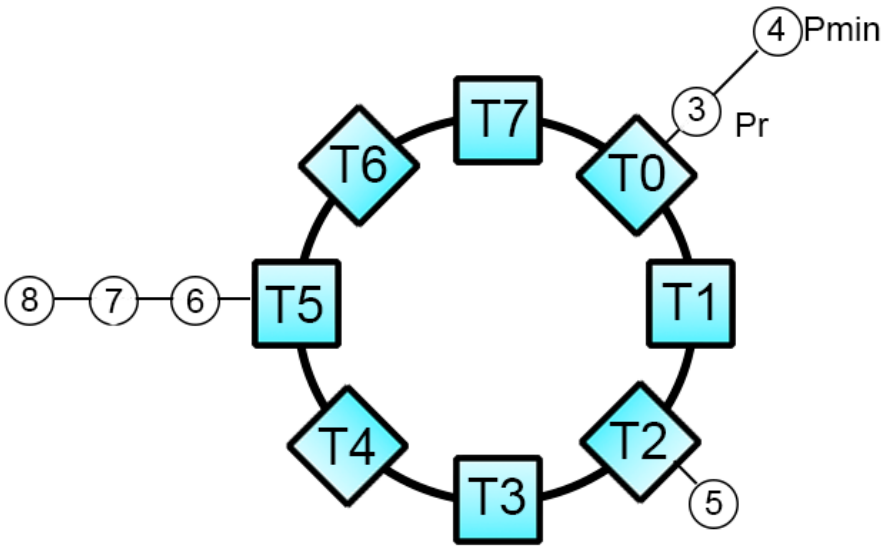


Figura 3.9: Representación del array circular, una vez se ha borrado todas las celdas de la discretización T0. Pasando a ser T0 la antigua discretización T1.

3.3 Fast Iterative Method

Fast Iterative Method, FIM es un algoritmo desarrollado por Won-Ki Jeong y Ross Whitaker [29]. Su principal idea es evitar el uso de una cola de prioridad compleja pero manteniendo el sistema de la *narrow band*. De esta manera, el algoritmo pasa de tener complejidad computacional *O(log n)* a *O(n).*

Para almacenar las celdas de la narrow band, FIM crea lo que los autores llaman, *active list*, equivalente a la *narrow band* en los métodos anteriores.

En cada iteración, la lista de celdas se expande para actualizar los valores de los *tiempos de llegada* (TT). Una celda solamente se elimina de la *active list*, una vez se llega a su solución. Para hallar los TT de las celdas según se expande la *narrow band*, se usará la ecuación de la Eikonal del punto 2.1.

Los objetivos del algoritmo incluyen, dar una buena prestación, coherencia de memoria caché y escalabilidad para múltiples procesadores.

Para superar estos objetivos, los requisitos a cumplir[29] son:

- El algoritmo no debe imponer un orden de actualización de la *narrow band* en particular.
  Este requisito es necesario para mantener una coherencia de la memoria caché, por ejemplo, el algoritmo FMM requiere un acceso a memoria aleatorio que impide una buena coherencia de caché.

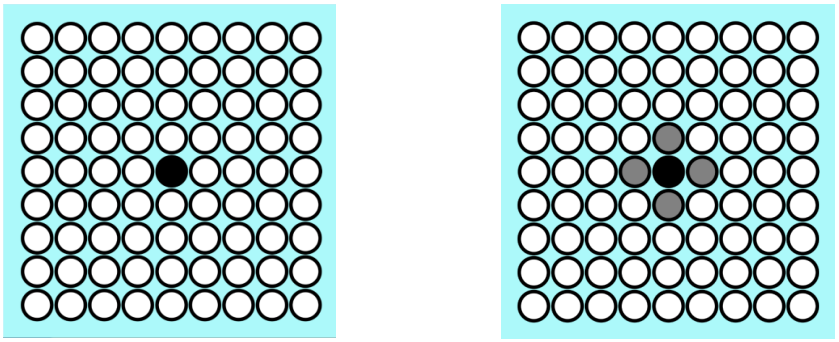
- No debe usar una estructura de datos complicada para ordenar la *narrow band*.
  Este criterio es necesario para SIMD *(Single Instruction, Multiple Data)[30]*, un método para mejorar la eficiencia en aplicaciones que usan la misma operación en varios datos simultáneamente.
  FMM mantiene una cola de prioridad con una lista de celdas, relativamente pequeña respecto al total de celdas. Esta circunstancia no puede ser implementada eficientemente en SIMD, debido a que es más eficiente procesar muchos datos en una operación común.

- En cada iteración se deben actualizar varios puntos, y no solo el de menor valor, como ocurre en FMM.

3.3.1 Fases del algoritmo

Este algoritmo consta de dos partes diferenciadas:

-**Inicialización**: en esta primera parte se ponen las condiciones de la rejilla, poniendo todos los valores TT de cada celda a infinito o un valor muy alto. A continuación, los vecinos von Neumann, de las celdas donde se origina la onda, serán añadidos a la *active list*, Figura 3.10.



a) *b)*

Figura 3.10: Etapa de inicialización del algoritmo FIM.

-**Actualización**: Una vez establecidas las condiciones iniciales, por cada punto en la *active list*, se calcula su TT mediante la ecuación de la Eikonal y se compara con el valor que tenía esa celda. Figura 3.11 a), b), c) y d).

Una celda se considera que ha convergido, cuando la diferencia entre el TT actual que tiene esa celda y el nuevo TT, calculado por la Eikonal, es menor o igual al error límite ε.

En caso de que hubiera convergido en un valor, se borra de la *active list* y se expanden los vecinos von Neumann que no estén en la *active list* o que ya convergieron(*Frozen*) y se etiquetan como *Narrow*. Esos vecinos se añaden a la *active list*, justo delante del punto borrado, para poder resolver los valores de *tiempo de llegada* en la siguiente iteración, Figura 3.11 e), f), g) y h).

El paso de actualización continua hasta que la *active list* esté vacía, pues ya todos los puntos han convergido, es decir, se han etiquetado como *Frozen*.

Aumentar el error ε hace aumentar la velocidad, pero disminuye la precisión del algoritmo.

Para la implementación del algoritmo FIM, se ha escogido que la *active list* sea una *double linked list* tal y como recomiendan los autores.

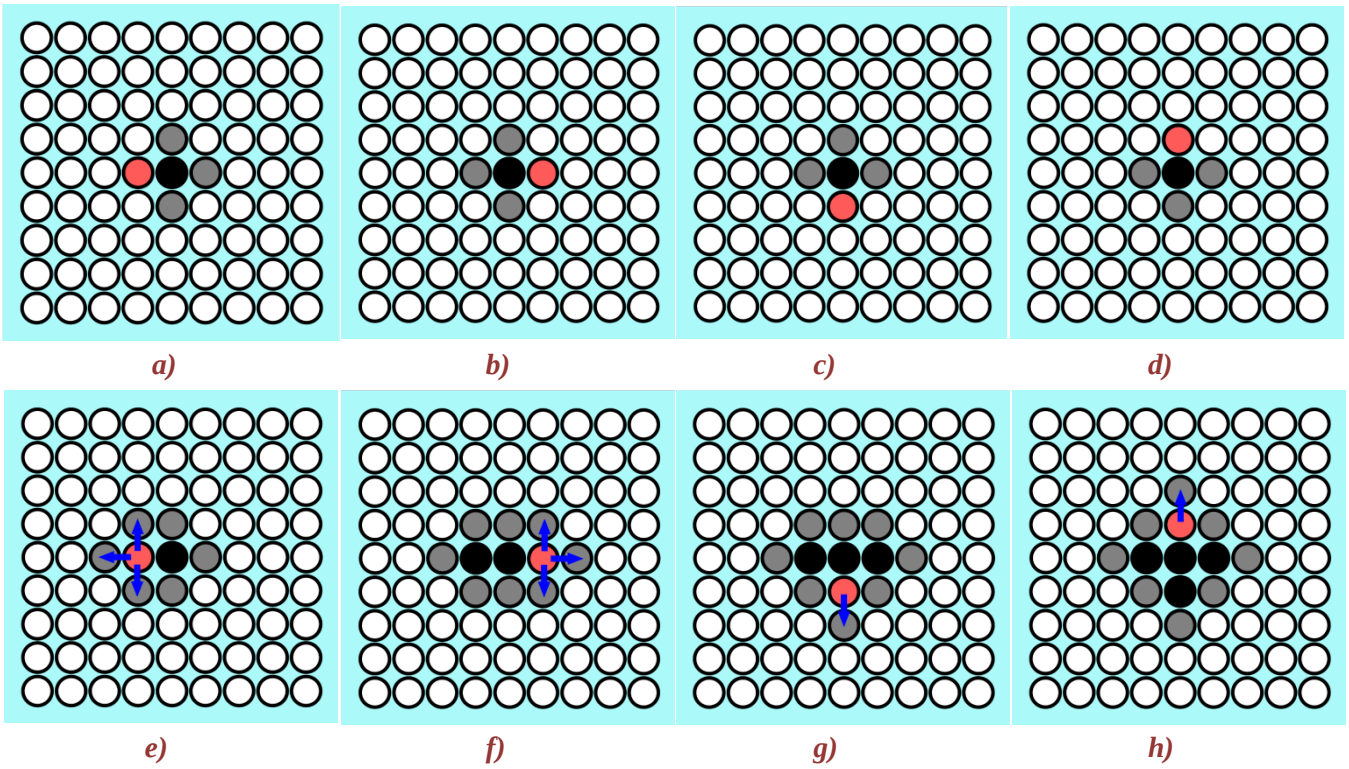


Figura 3.11: Proceso de convergencia de los puntos de la active list para la posterior expansión de la narrow band

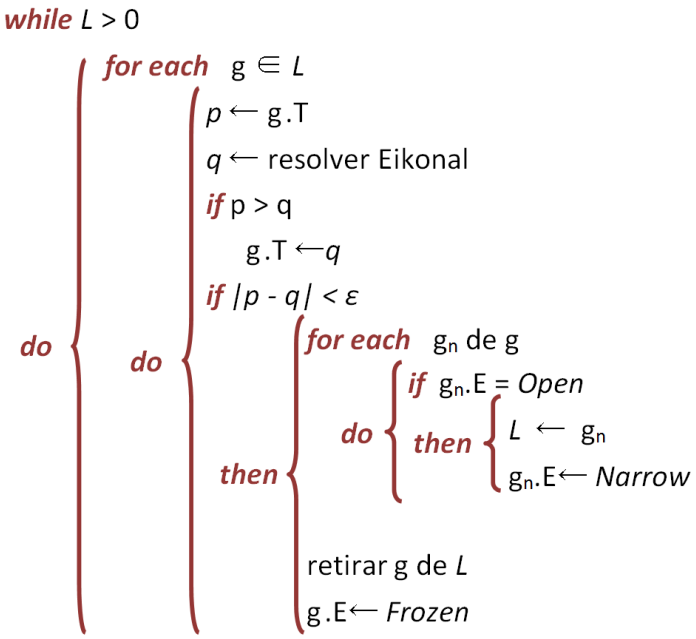
3.3.2 Pseudocódigo

G: rejilla de tamaño m x n.
$\mathbf{g_s}$: celdas donde se origina la onda.
ε: error límite.
V:velocidad de la celda.
L: *active list* que contiene la narrow band.
g: celda.
$\mathbf{g_n}$**:** vecino von Neumann.
T: tiempo de viaje.
E: etiqueta de la celda: Open | Narrow | Frozen.

-Inicialización

for each $g \in G$ *do*
- *if* $g = g_s$
  - $g.T \leftarrow 0$
- *else*
  - $g.T \leftarrow \infty$
- *if* $g = g_n$ de g_s
  - $L \leftarrow g_n$

-Actualización



Capítulo 4:

EXPERIMENTACIÓN Y RESULTADOS

En este capítulo se expondrán las condiciones bajo las que se han realizado todos los experimentos de los algoritmos. Además, se explicarán las conclusiones a las que se ha llegado viendo cada uno de los resultados.

4.1 Configuración de los experimentos

Los algoritmos anteriormente explicados, han sido implementados en un PC con el sistema operativo Ubuntu 12.04, equipado con un Intel Core I5 de 3.1 Ghz y 8 gigabytes de memoria RAM. Otro dato a destacar, es el uso de las librerías Boost 1.5.

Todos los resultados temporales que se proporcionan son resultado de la media realizada con los resultados obtenidos de 10 medidas. El número de medidas realizado es alto para que los resultados se vean menos comprometidos con otros parámetros y sean más fiables. Los resultados numéricos de las gráficas, de todos los experimentos, se proporcionan en el apéndice c. *"Resultados numéricos"*.

Para llegar a las conclusiones, se han realizado diferentes experimentos, según se iban observando los resultados y cuál era su tendencia. Los experimentos finalmente realizados han sido:

- **Experimento Nº 1**: mapa con velocidades de propagación constante (F=1). El mapa tiene forma cuadrada y con una sola celda de inicio de onda en el centro.
  Se cambia el tamaño del mapa usado para el experimento, y se repiten las medidas para comprobar cómo cambian los tiempos de cálculo para cada algoritmo.

- **Experimento Nº 2**: mapa con velocidades de propagación constante (F=1). El mapa tiene forma cuadrada y con una sola celda de inicio de la onda, en una de las esquinas del cuadrado.
  Se cambia el tamaño del mapa usado para el experimento, y se repiten las medidas para comprobar cómo cambian los tiempos de cálculo para cada algoritmo.

- **Experimento Nº 3**: mapa con velocidades de propagación constantes (F=1), con forma rectangular y con una sola celda fuente, situada en la mitad de un lado.

Se mantiene el número de celdas que componen el mapa, pero disminuye el tamaño de la *narrow band* cambiando el tamaño de los lados del rectángulo.

- **Experimento Nº 4:** mapa con velocidades de propagación constantes F=1 y F=0.1, con forma cuadrada y con una sola celda fuente en el centro.
  Se contrasta el tiempo de cálculo que necesita cada uno de los algoritmos en cada uno de los mapas, para comprobar así, si la velocidad de propagación afecta de alguna manera en el tiempo de cálculo.

- **Experimento Nº 5:** mapas con velocidades de propagación no constantes, con forma cuadrada y con una sola celda fuente en la mitad de un lado.
  Se usan dos mapas con franjas horizontales en la que la velocidad de propagación es, en esas franjas, un 10% y siendo, en el otro mapa, un 90%. Así se consiguen dos niveles de contraste de velocidad, fuerte o débil.

- **Experimento Nº 6**: mapas con velocidad de propagación constante (F=1), con forma cuadrada.
  El inicio de la onda se posiciona, según el mapa utilizado, de manera que se produzcan cambios de dirección en la propagación de la onda.
  Para estudiar mejor como afectan estos cambios de dirección, se crean dos mapas, uno con tres cambios de dirección y otro con doce.

- **Experimento Nº 7:** mapa con velocidad de propagación aleatoria, con una sola celda fuente en el centro. En este mapa se comprobarán los errores que cometen cada uno de los algoritmos al calcular los tiempos de llegada(TT).

- **Experimento Nº 8:** el mapa con velocidades de propagación aleatorias del experimento Nº7, se vuelve a usar para este experimento. Se comprueba los errores que comete el algoritmo UFMM según se varía el número de discretizaciones y cómo afecta al tiempo de cálculo.

- **Experimento Nº 9:** mapa con velocidad de propagación constante (F=1), con una sola celda fuente en el centro y forma cúbica.
  Se cambia el tamaño del cubo usado para el experimento, y se repiten las medidas para comprobar cómo cambian los tiempos de cálculo para cada algoritmo.

4.2 Resultados de los experimentos

Experimento Nº 1:

Para este experimento, se ha seleccionado un mapa con velocidad constante (F=1), con forma cuadrada y con una sola celda de inicio en el centro. Las dimensiones del mapa se cambian, repitiendo las medidas, en cada cambio, para comprobar cómo afecta a los tiempos de cálculo de los *tiempos de llegada*, para cada uno de los algoritmos.

En la Figura 4.1 se observa la forma circular de la onda que se ha generado situando la celda fuente en el centro del mapa, debido a que las velocidades de propagación de onda de cada celda son iguales.

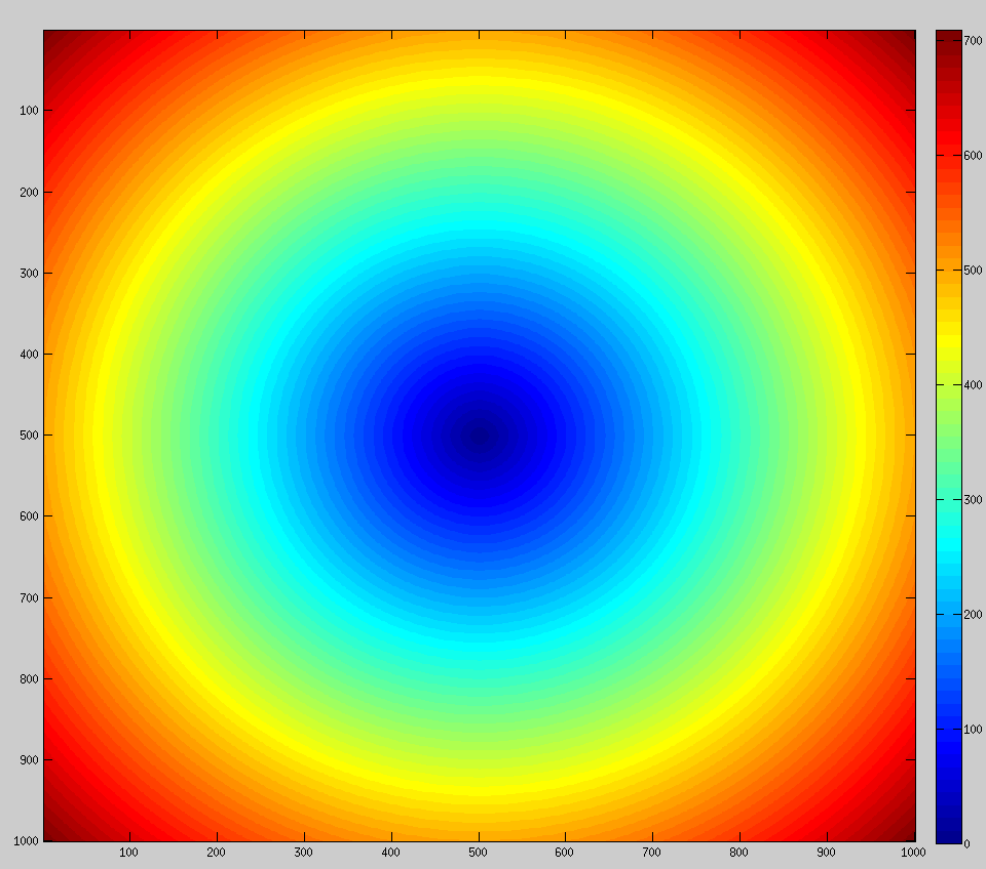


Figura 4.1: Representación gráfica de los tiempos de llegada (TT) en milisegundos del experimento Nº1.

Observando la Figura 4.2 y comparando los algoritmos, se puede apreciar las tendencias de cada uno de los órdenes de complejidad *O(n)* y *O(n log n)* vistos en el capítulo 2.4.

Los algoritmos lineales *O(n) y* logarítmicos *O(log n)*, actúan según lo esperado. Si tomamos el valor de un millón de celdas y lo comparamos con el resultado de cuatro millones, se obtiene la relación de la Tabla 4.1(Cuatro veces el tamaño del problema, cuatro veces más tiempo en resolver) y de la Tabla 4.2.

Tamaño	Nº Celdas	GMM	UFMM	FIM	Relación teórica (*n*)
1000x1000	**1.000.000**	337,6	241	203	**1.000.000**
2000x2000	**4.000.000**	1368	1004	814,6	**4.000.000**
Relación		**4.05**	**4.16**	**4.01**	**4**

Tabla 4.1: Comprobación de que, realmente, los algoritmos son de complejidad O(n).

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	Relación teórica (*n log n*)
1000x1000	1.000.000	462,2	450,7	346,7	**6.000.000**
2000x2000	4.000.000	1968,1	2205,7	1659	**26.408.240**
Relación		**4.25**	**4.89**	**4.78**	**4.4**

Tabla 4.2: Comprobación de que, realmente, los algoritmos son de complejidad O(n log n).

Atendiendo a la comparativa entre los distintos algoritmos, como era de esperar, los algoritmos de complejidad lineal se ven menos afectados por el aumento de celdas.

De los algoritmos lineales el que mejor resultado ofrece en este mapa, es decir, velocidad constante (F=1) y con la onda iniciada en el centro, ha sido FIM. Mejorando el tiempo en un 19% aproximadamente respecto el segundo mejor método UFMM y 40% respecto a GMM situado en tercera posición.

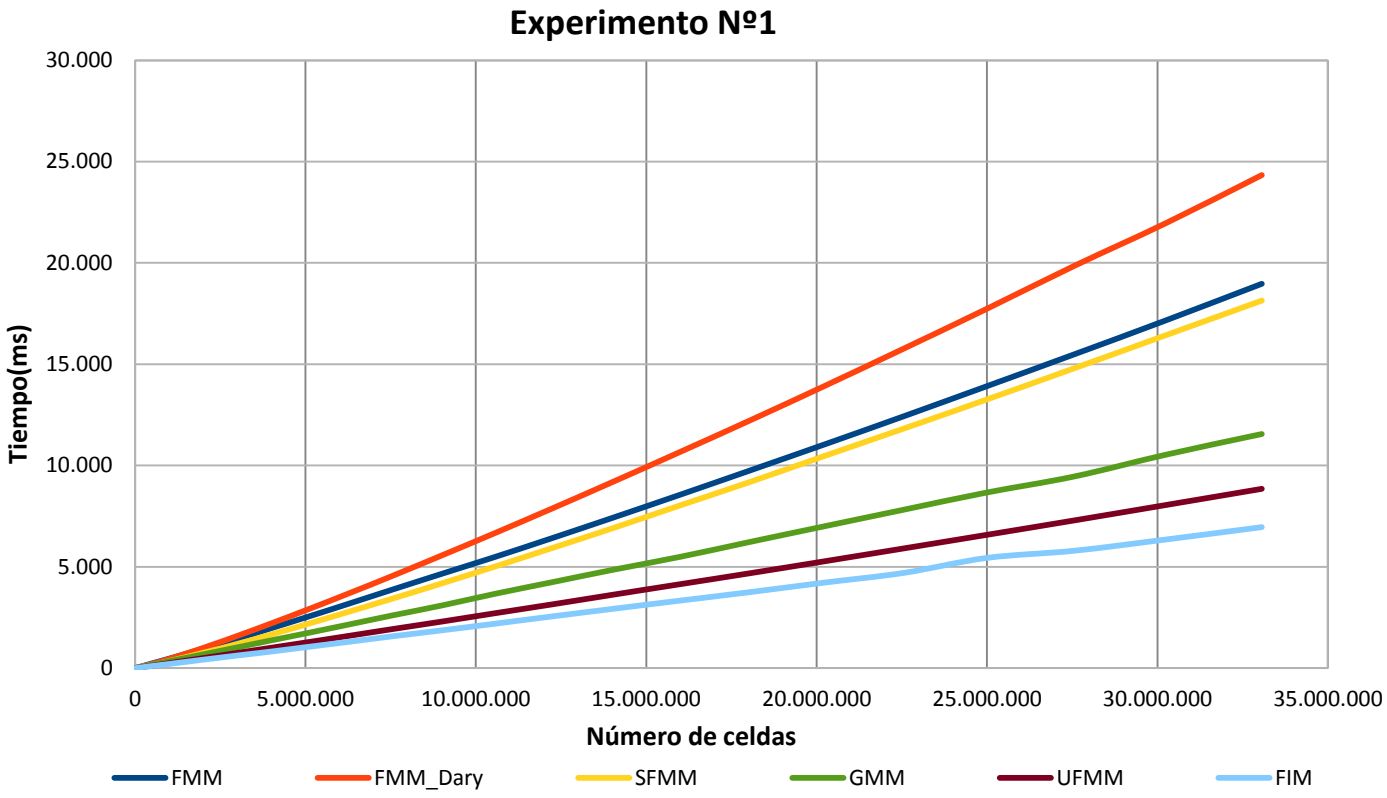


Figura 4.2: Resultado gráfico del experimento Nº1. Mapa con velocidades constantes y con una sola fuente en el centro.

Para observar mejor el experimento, se han realizado medidas adicionales del experimento Nº1 entre 0 y 1,5 millones de celdas. Como se puede apreciar en la Figura 4.3, GMM y SFMM tienen una eficiencia parecida hasta las novecientas mil celdas, finalmente GMM logra mejores resultados cuando las celdas aumentan más allá de ese punto.

Comparando el resultado de la pila binaria (FMM_Dary) y la pila de Fibonacci (FMM), para pocas celdas, la pila binaria registra mejores resultados. A partir del millón de celdas, la pila de Fibonacci se impone.

Dado que este es el experimento más básico, se usará como base para comparar los siguientes experimentos.

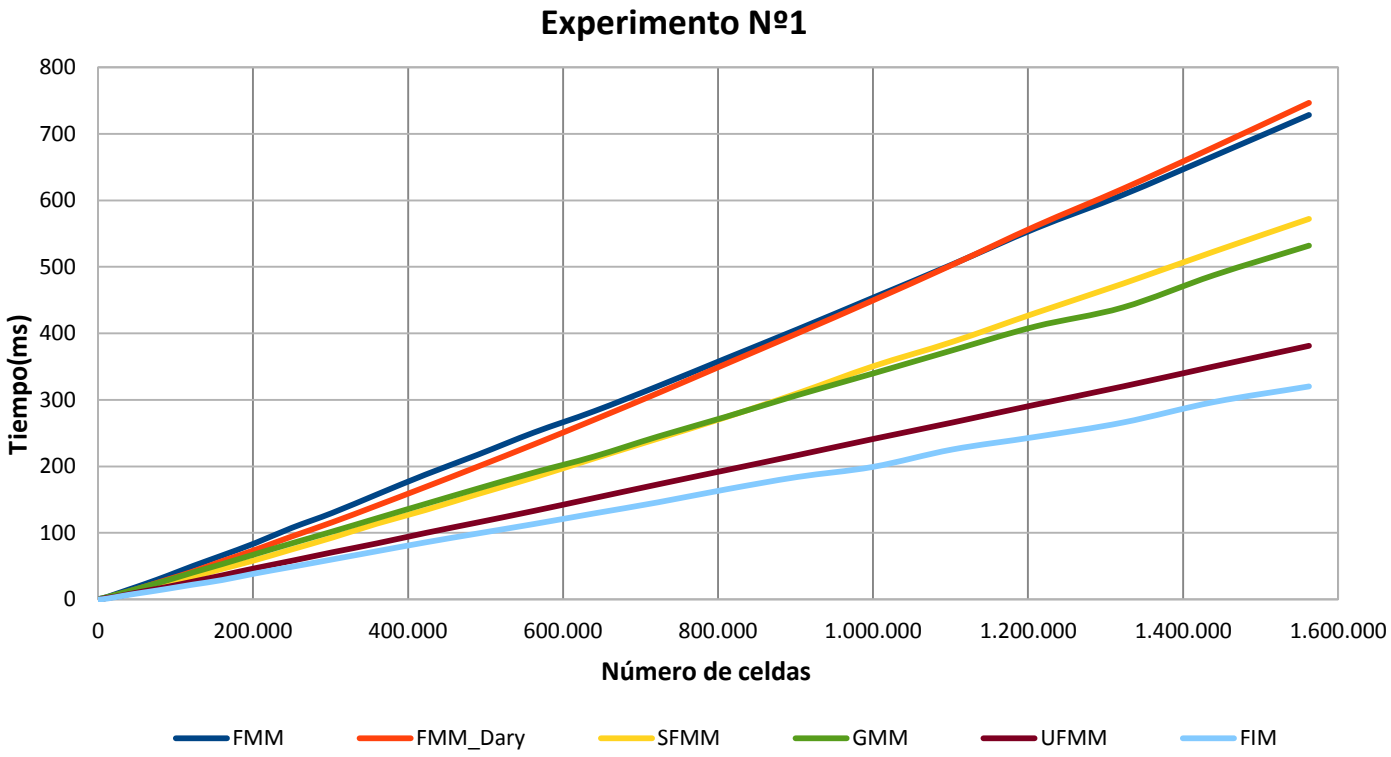


Figura 4.3: Resultado gráfico del experimento Nº1. Zoom entre 0 y 1.6 millones de celdas.

Experimento Nº 2:

Para este experimento, al igual que el experimento Nº1, se ha seleccionado un mapa con velocidad constante (F=1), con forma cuadrada y con una sola celda de inicio en una de las esquinas. Las dimensiones del mapa se cambian, repitiendo las medidas para comprobar cómo afecta los tiempos de cálculo de los *tiempos de llegada* para cada uno de los algoritmos.

En la Figura 4.4 se puede apreciar gráficamente la expansión de la onda, la forma es igual que en el experimento Nº1, pero en un solo cuadrante.

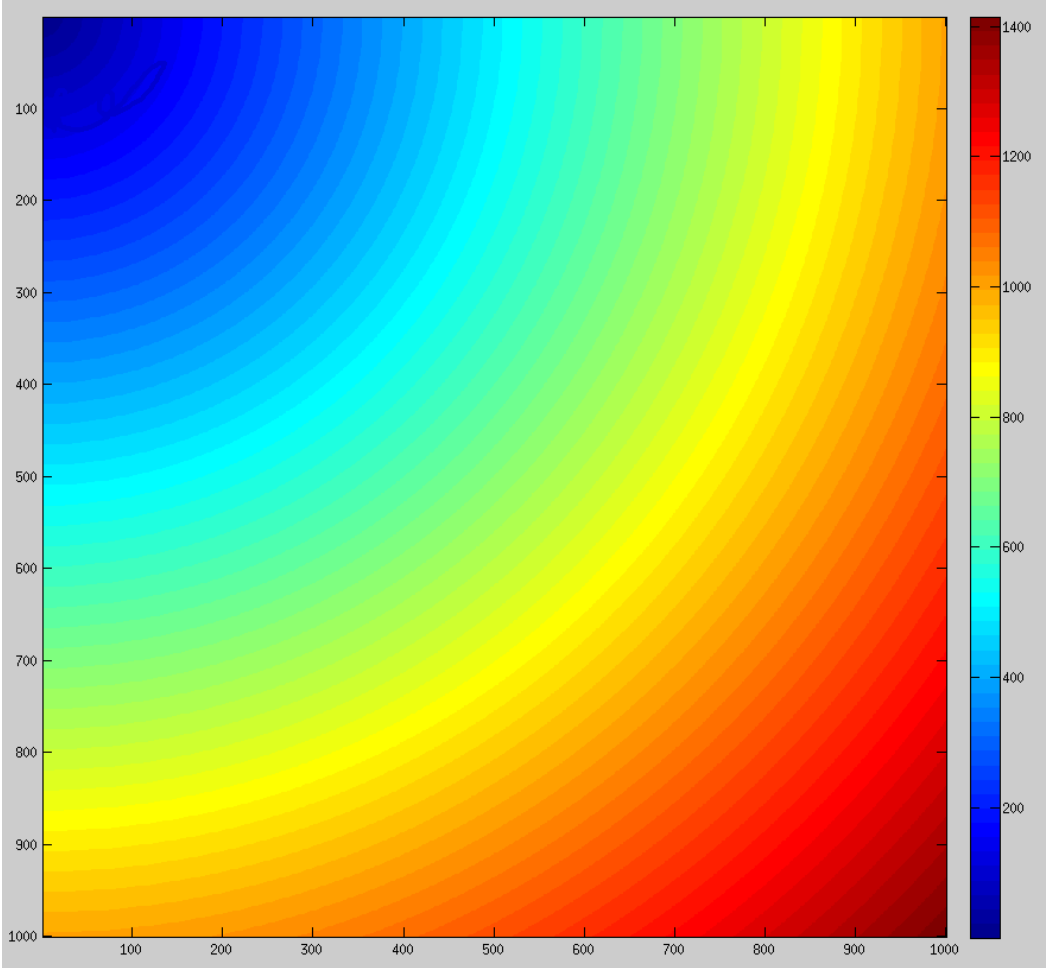


Figura 4.4 Representación gráfica de los tiempos de llegada (TT) en milisegundos del experimento Nº2.

Observando las Figura 4.5 y Figura 4.6, se observan las mismas tendencias que en el experimento Nº1, además de mantenerse el resultado final en la comparación de algoritmos.

Si se examinan las diferencias, se puede encontrar que el cruce de líneas entre los métodos FMM y FMM_Dary se produce a los 27 millones de celdas. En el experimento Nº1, este fenómeno se producía con muchas menos celdas (1 200 000).

GMM en el experimento Nº1, empezaba a dar mejor resultado temporal que SFMM en torno al millón de celdas. En cambio en el experimento Nº2, el cruce de líneas del algoritmo GMM y SFMM se produce con un número más elevado de celdas.

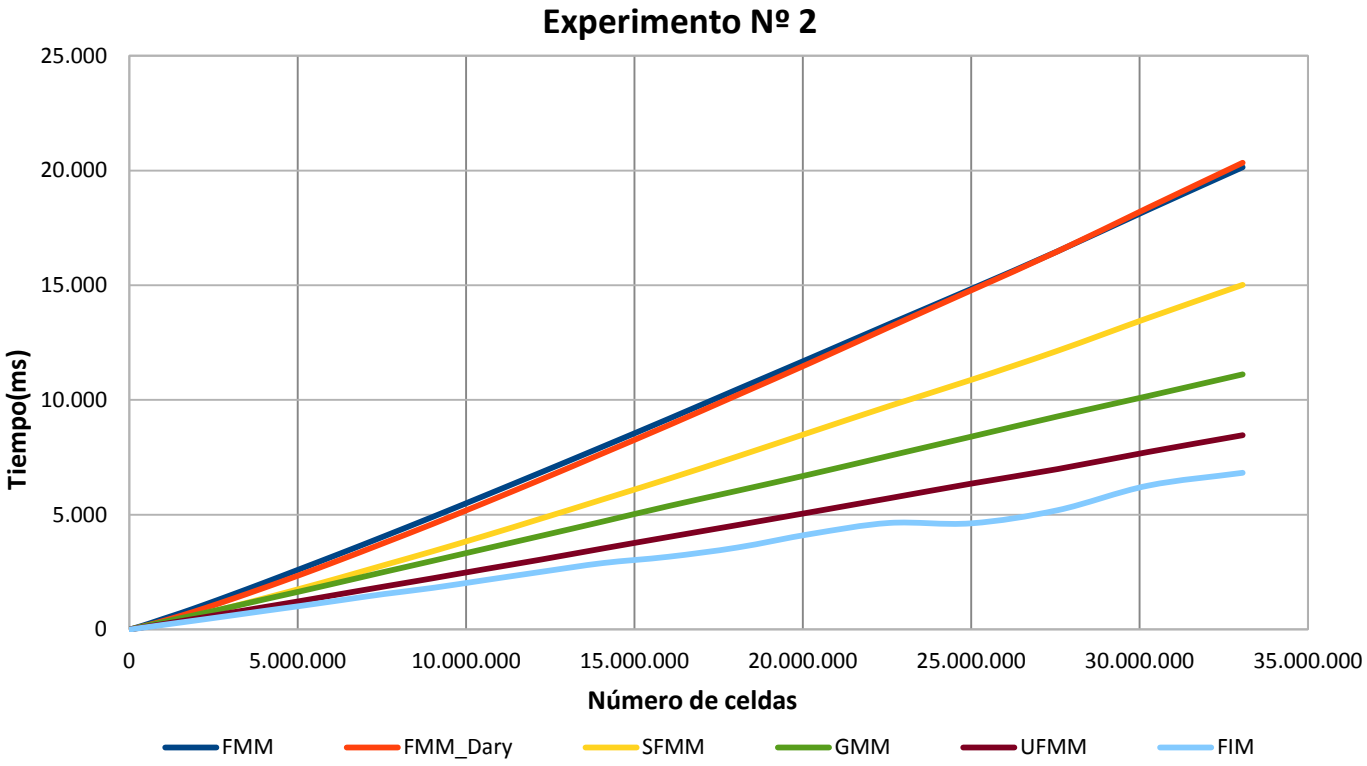


Figura 4.5: Resultado gráfico del experimento Nº1. Mapa con velocidades constantes y con una sola fuente en una esquina.

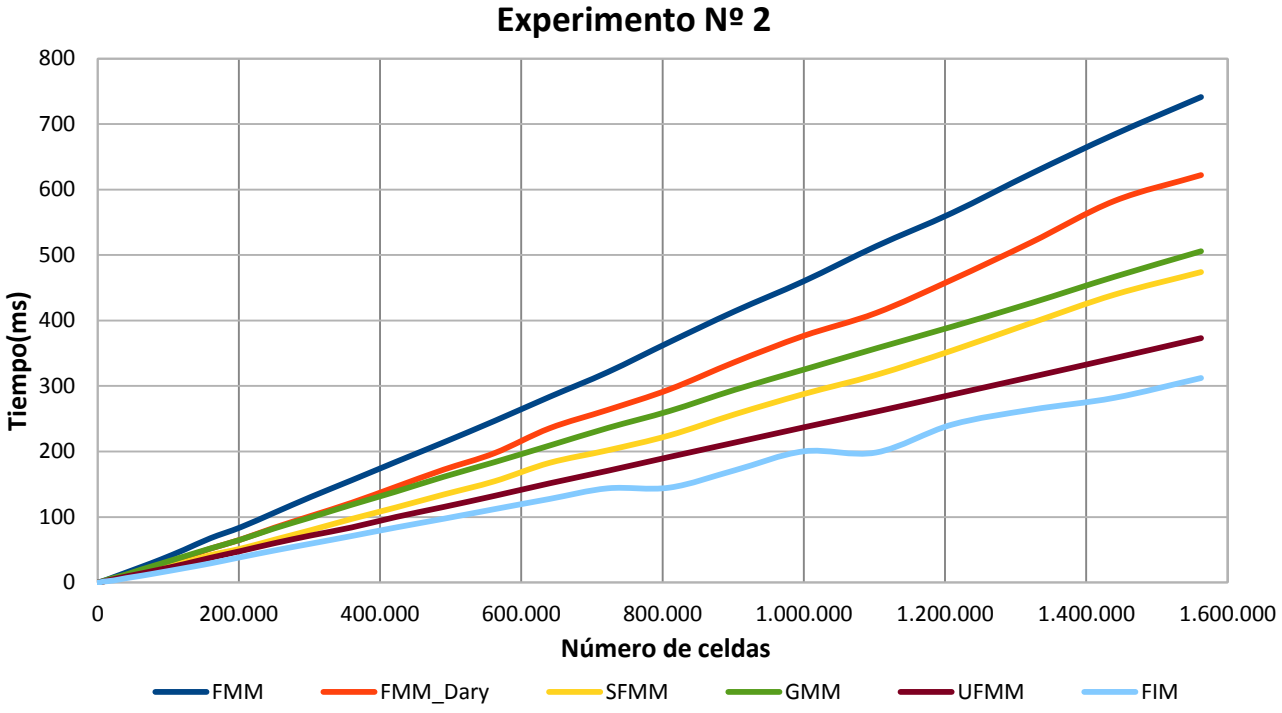


Figura 4.6: Resultado gráfico del experimento Nº2. Zoom entre 0 y 1.6 millones de celda.

Tomando en cuenta estos sucesos, se puede concluir que el tamaño de la *narrow band* podría afectar a los resultados. En el experimento Nº2, la *narrow band* es cuatro veces menor que en el experimento Nº1 ya que la onda solamente se expande en un cuadrante.

Experimento Nº 3:

La información que se ha obtenido en el experimento Nº2 ha sido que el tamaño de la *narrow band* afecta de alguna manera a los algoritmos. Para comprobar mejor el resultado, se ha diseñado el experimento Nº3.

Se realiza modificaciones en el tamaño del mapa, para conseguir una reducción de la *narrow band*, pero manteniendo constante el número de celdas. El experimento empieza con un cuadrado de 4000x4000 celdas, Figura 4.7, seguido por un rectángulo de 8000x2000 celdas, Figura 4.8, y acaba con un rectángulo de 100x160000, manteniendo, de esta manera, constante el número de celdas en 16 millones, como se puede comprobar en la Tabla 4.3.

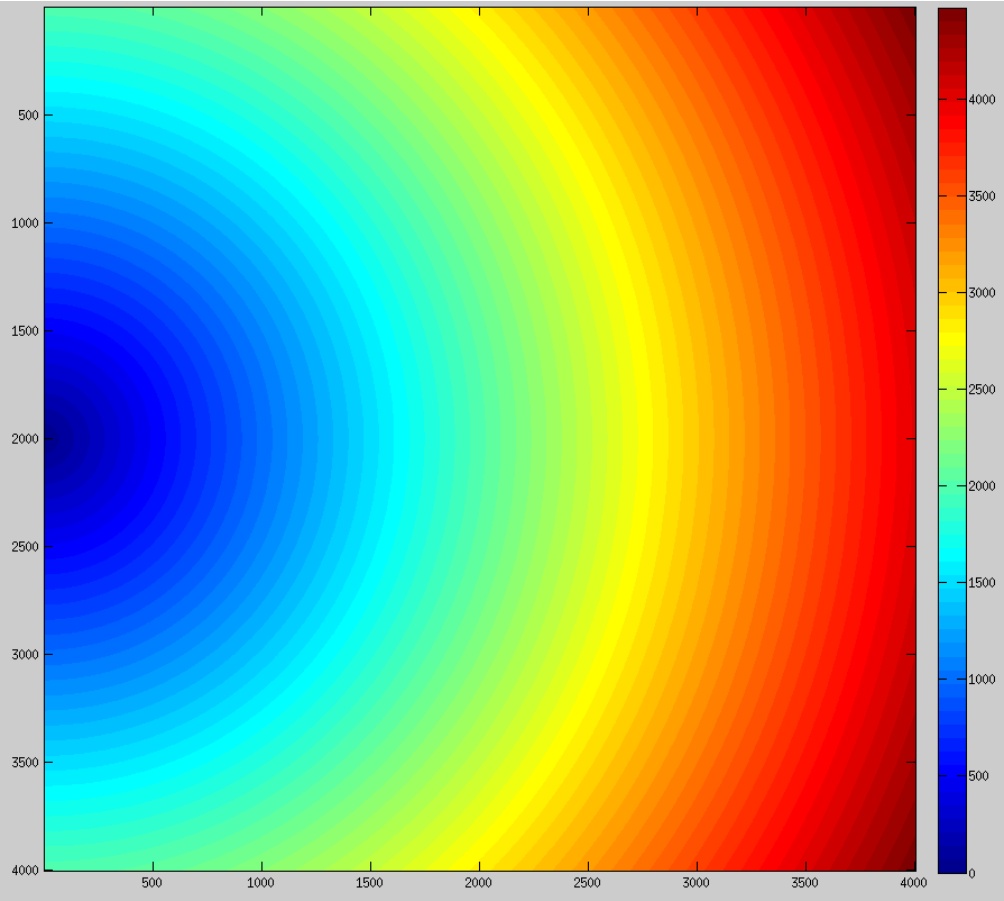


Figura 4.7: Representación gráfica de los tiempos de llegada (TT) del experimento Nº3.Mapa cuadrado 4000 x 4000 celdas.

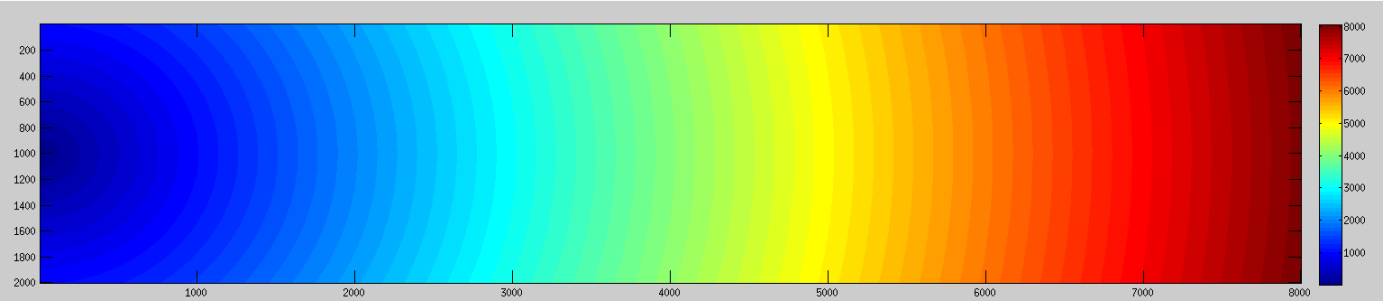


Figura 4.8: Representación gráfica de los tiempos de llegada (TT) del experimento Nº3. Mapa rectangular de 8000 x 2000 celdas.

Para tener una buena comparación de cómo afecta el tamaño de la *narrow band*, se van a relacionar los resultados obtenidos con el tamaño máximo que puede alcanzar la *narrow band* a lo largo de todas las iteraciones, Figura 4.9. El tamaño máximo de la *narrow band* se corresponde con el número de celdas que contenga el rectángulo en el lado donde se inicia la onda, en el caso de este experimento es la altura.

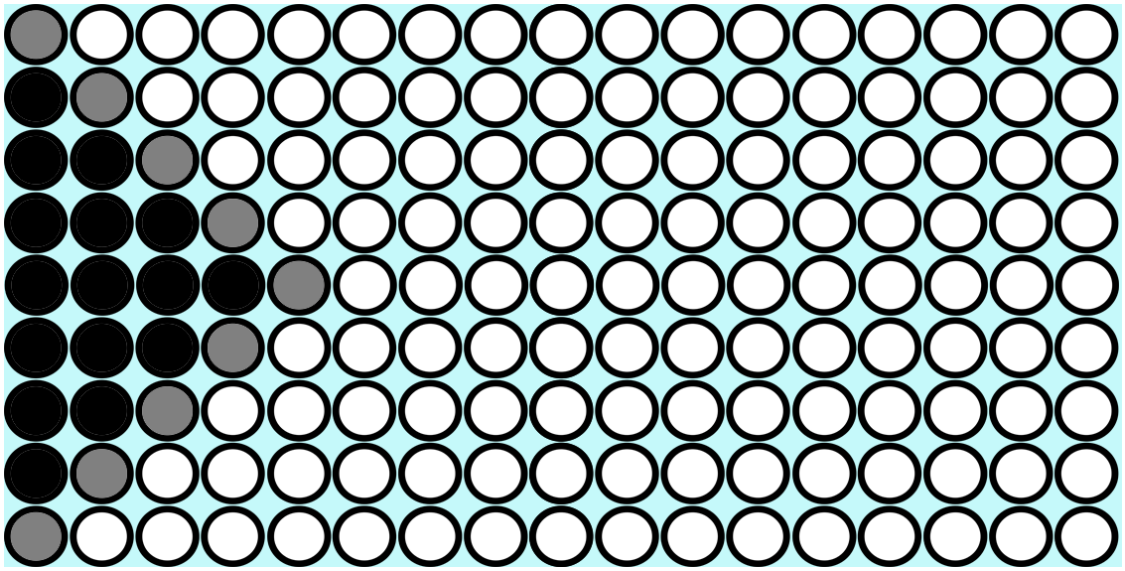


Figura 4.9: Medida máxima de la narrow band en un rectángulo discretizado. Coincide con el número de celdas de la altura de ese rectángulo.

Para las distintas dimensiones del mapa que se está usando, las *narrow band* quedarían de tamaño:

Tamaño(*x*·*y*)	**Nº Celdas**	**Longitud de la NB (celdas)**
4000x4000	16.000.000	4000
8000x2000	16.000.000	2000
16000x1000	16.000.000	1000
40000x400	16.000.000	400
80000x200	16.000.000	200
160000x100	16.000.000	100

Tabla 4.3: Tamaños máximos de cada una de las narrow band, según las dimensiones del mapa.

Observando la Figura 4.10, se demuestra que el tamaño de la *narrow band* sí tiene efecto directo sobre el resultado de los experimentos.

En el experimento los algoritmos lineales *O(n)*, no les afecta, prácticamente, el tamaño de la *narrow band*, las líneas de la Figura 4.10 salen constantes, como era de esperar.

En cambio, los algoritmos *O(n log n)* se ven notablemente más afectados por el aumento de celdas de la *narrow band*. Se observa que FMM_Dary, según va aumentando el número de celdas que componen el mapa, tiene peor tiempo de cálculo que FMM.

Comparándolo con los experimentos Nº1 y Nº2, se puede deducir por qué ambos métodos se mantienen parejos en el Nº2. Se debe a que la *narrow band* solo es una cuarta parte del experimento Nº1, afectando en menor medida al resultado.

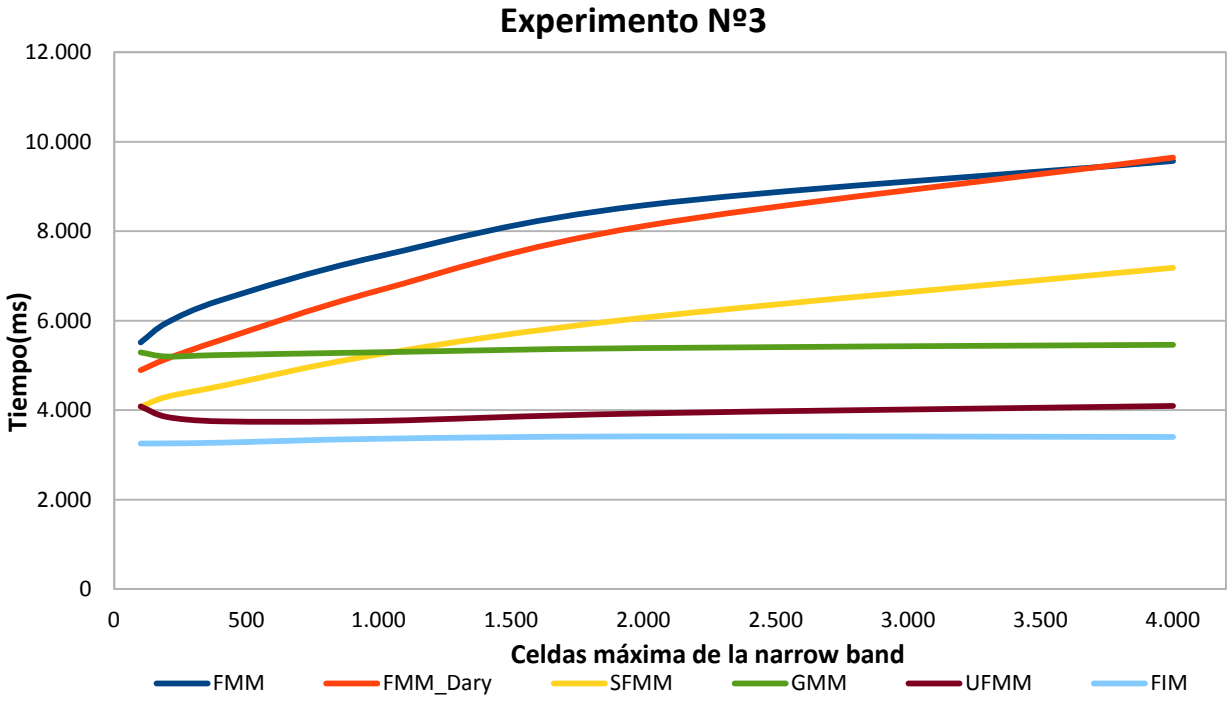


Figura 4.10: Comprobación del experimento Nº3, cómo afecta el tamaño de la narrow band a cada uno de los algoritmos.

Se pueden obtener dos conclusiones:

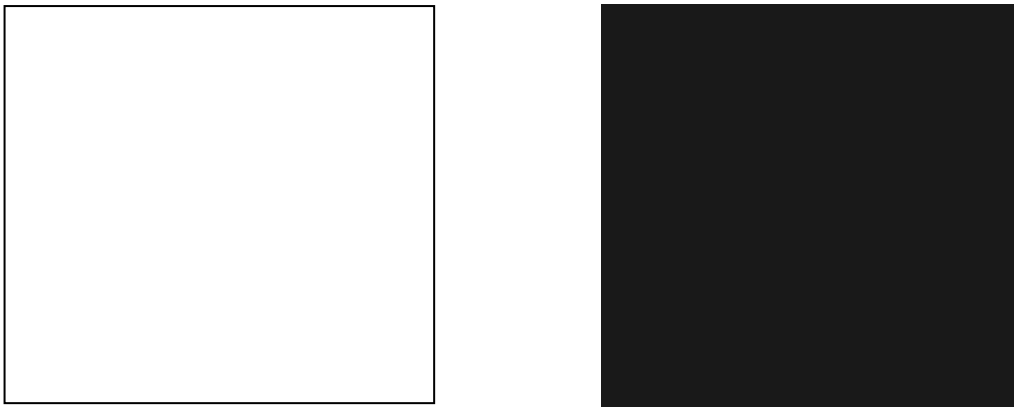
- El tamaño de la *narrow band* no afecta prácticamente a los métodos lineales *O(n)*.

- Un tamaño grande de la *narrow band* afecta en gran medida a la pila binaria FMM_Dary y afecta de manera moderada a FMM. SFMM se ve afectado, pero menos que los dos algoritmos anteriores.

  Dado que en mapas 3D, la *narrow band* se hace inmensa, no es recomendable usar ninguno de los métodos no lineales, sobre todo, la pila binaria FMM_Dary.

Experimento Nº 4:

En este experimento se va a estudiar la influencia de la velocidad de propagación de onda en los algoritmos. Para ello, se van a comparar dos mapas de 1000 x 1000 celdas, en las que van a tener la velocidad máxima de propagación de onda, Figura 4.11 a), y el segundo mapa con un 10% de la velocidad de propagación máxima, Figura 4.11 b).



a) Mapa con máxima velocidad de onda. *b) Mapa con un 10% de la velocidad máxima.*

Figura 4.11: Mapas de velocidad de propagación de onda constante, usados en el experimento Nº4.

Dando como resultado los tiempos de llegada de la Figura 4.12 y Figura 4.13. Si se comparan, se puede observar que en la Figura 4.13 los tiempos de llegada son diez veces mayores, ya que la velocidad de propagación es un 10% respecto a la velocidad máxima.

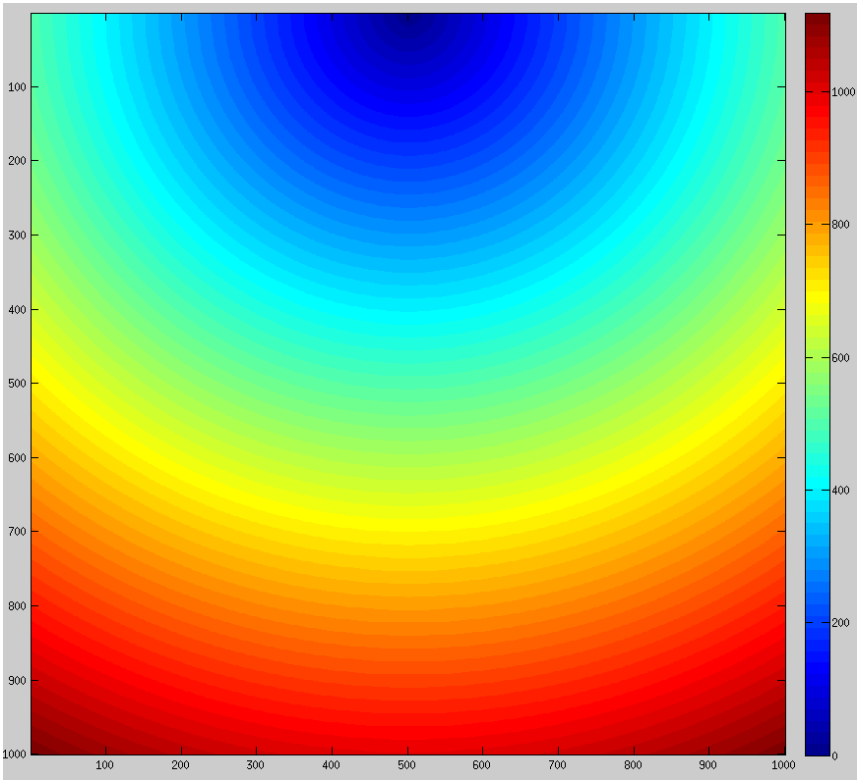


Figura 4.12: Representación gráfica de los tiempos de llegada (TT) del experimento Nº4. La velocidad de propagación de la onda es un 100%.

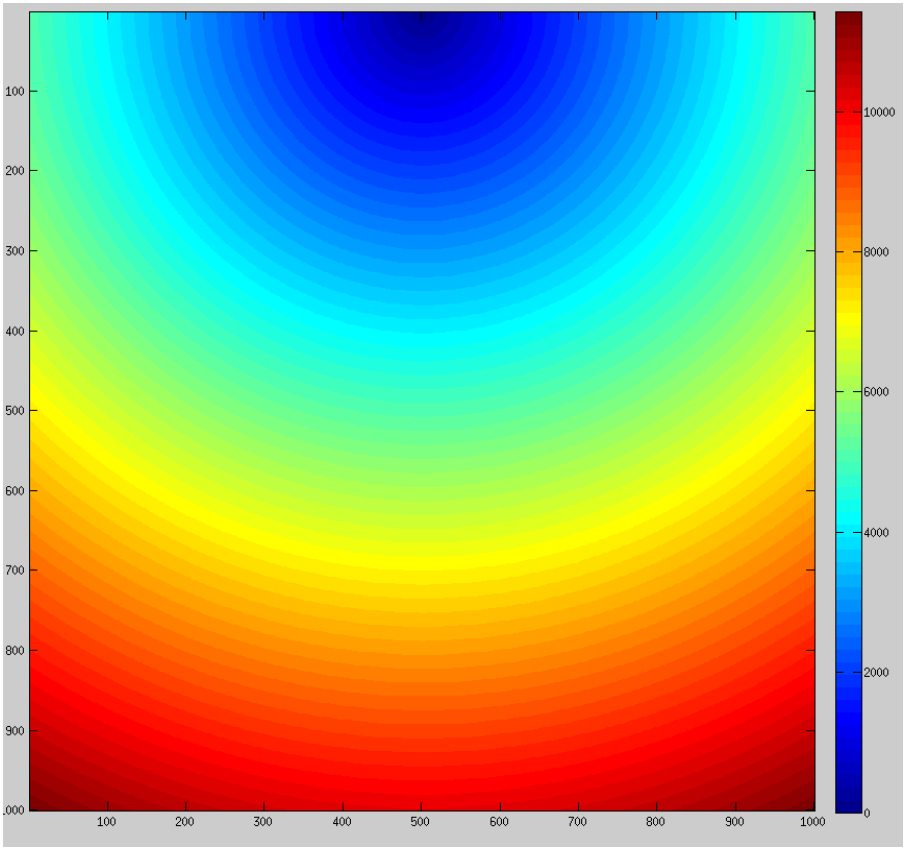


Figura 4.13 Representación gráfica de los tiempos de llegada (TT) del experimento Nº4. La velocidad de propagación de la onda es un 10% de la velocidad máxima.

La Tabla 4.4 representa el tiempo que le ha tomado a cada algoritmo calcular todos los tiempos de llegada de los mapas mencionados. Se puede comprobar que los algoritmos no cambian el tiempo de cálculo con una velocidad de propagación u otra, excepto el algoritmo GMM.

Velocidad	100%	10%
FMM	469,7	470
FMM_Dary	392,1	393,4
SFMM	303,3	304,6
GMM(del_TAU = 10)	-	339,6
GMM(del_TAU = 1)	336	463,4
UFMM	241,3	252,1
FIM	197,4	199,4

Tabla 4.4: Tiempo de cálculo, en milisegundos, que ha necesitado cada algoritmo para calcular todos los tiempo de llegada en el experimento Nº4.

Comparando los resultados obtenidos, el algoritmo GMM tiene un problema con la velocidad de propagación. Este problema se debe a que hay que tener en cuenta, el valor de la máxima velocidad de todas las celdas. Ya que interviene, como se puede recordar consultando el punto 3.1, en el cálculo del valor de *delTAU*.

$$delTAU = \frac{1}{F_{global,max}} \begin{cases} F_{global,max} = 1 \rightarrow delTAU = 1 \\ F_{global,max} = 0.1 \rightarrow delTAU = 10 \end{cases}$$

Si ese valor no se ajusta con el valor máximo del mapa se pierde efectividad y el tiempo de cálculo aumenta.

Como conclusión se puede decir que:

- La velocidad del medio no afecta al tiempo de cálculo, excepto a GMM en el caso de que el valor de *delTAU* no se ajuste con los parámetros del mapa usado.

- La necesidad de ajuste de este valor, hace que GMM no sea óptimo para mapas desconocidos del que no se sepa el valor máximo de la velocidad.

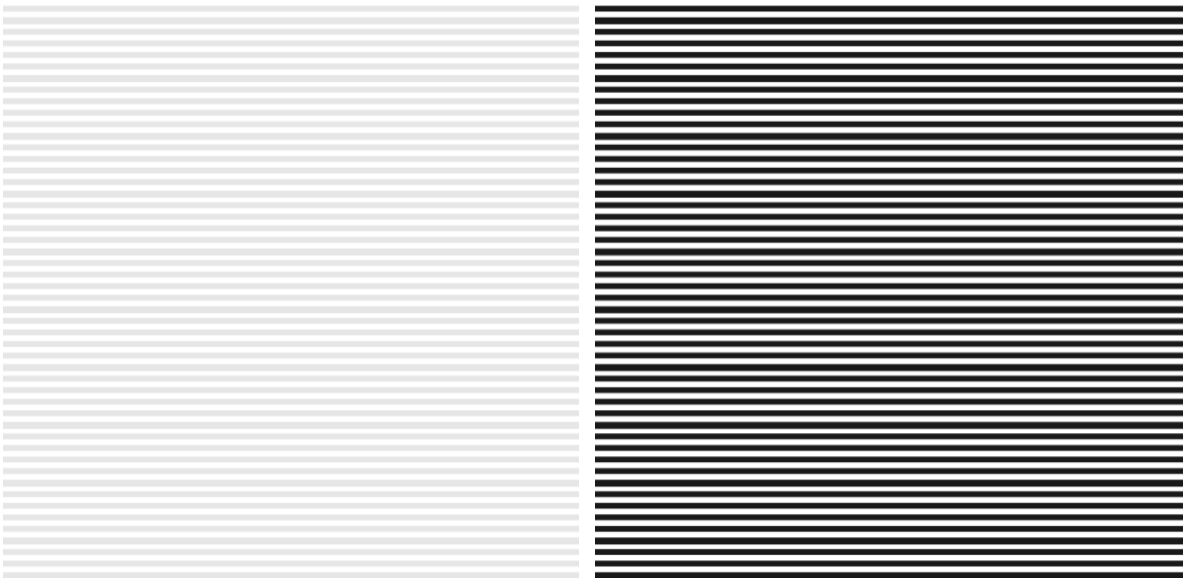
Experimento Nº 5:

En el experimento Nº4 se ha comprobado que la velocidad homogénea no afecta al tiempo de cálculo de los algoritmos. Con este experimento se podrá comprobar cómo el cambio constante de velocidades de propagación de la onda al resultado en el tiempo de cálculo. Para ello, se han creado los mapas de la Figura 4.14.

Estos mapas tienen unas dimensiones de 1000 x 1000 celdas, que tienen franjas horizontales en las que el valor de la velocidad de propagación de onda cambia. En el caso de la Figura 4.14 a), se alterna entre el 100% de la velocidad y un 90%. En caso de la Figura 4.14 b), se alterna entre franjas con 100% de la velocidad y un 10%.

Así, se ha conseguido dos mapas con dos grados de contraste en la velocidad de propagación, fuerte o débil.

La representación gráfica de los tiempos de llegada se puede observar, cuando el contraste es débil, en la Figura 4.15 y , cuando el contraste es fuerte, en la Figura 4.16 donde se aprecia una deformación de la onda circular.



a) Contraste débil *b) Contraste fuerte*

Figura 4.14: Mapas con contrastes en la velocidad de propagación de la onda, usados en el experimento Nº5. Con franjas horizontales que alteran la velocidad de propagación de la onda.

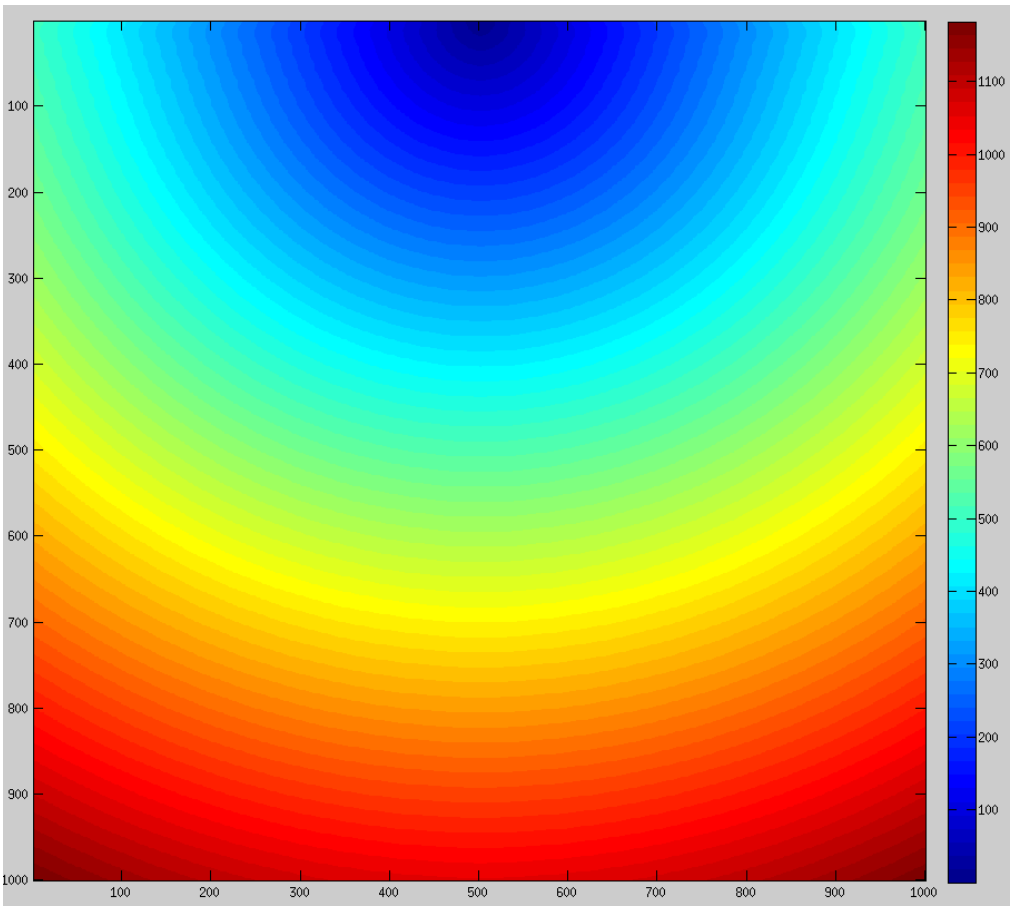


Figura 4.15: Representación gráfica de los tiempos de llegada (TT), con franjas alternadas de 100% y de 90% de velocidad.

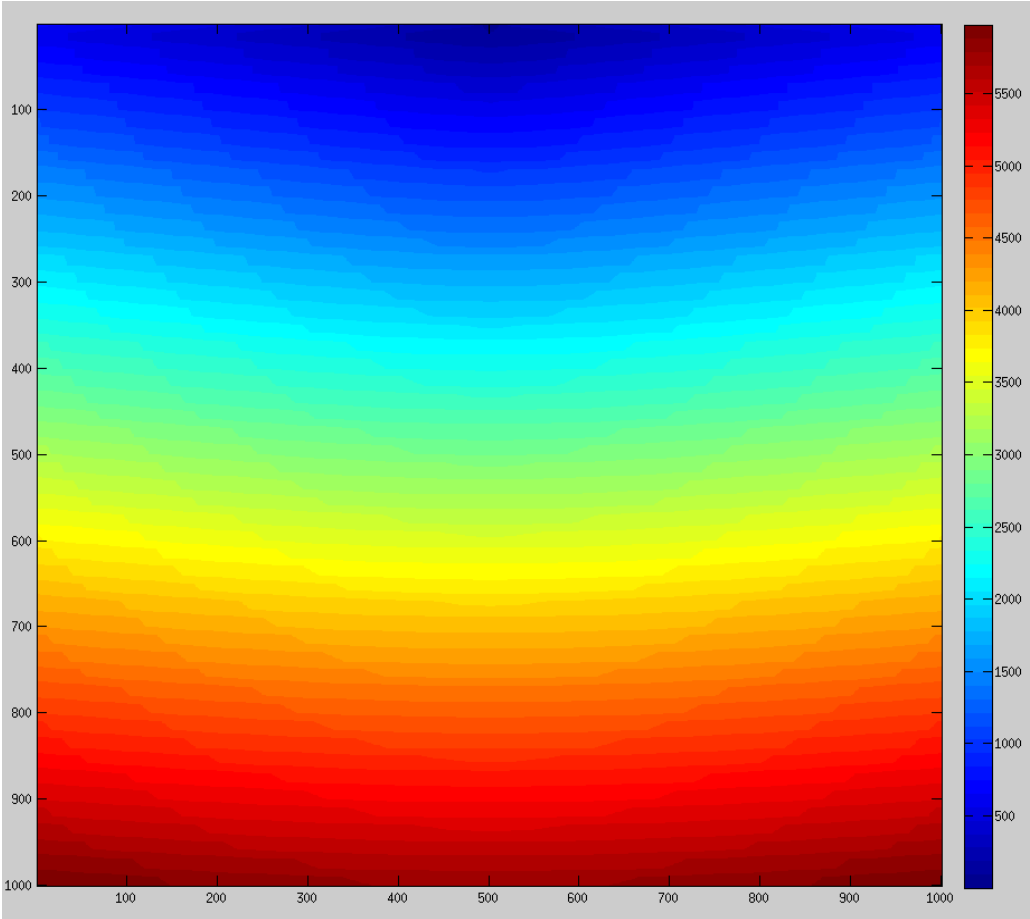


Figura 4.16: Representación gráfica de los tiempos de llegada (TT), con franjas alternadas de 100% y de 10% de velocidad.

El contraste de velocidades de propagación de onda, afecta muy levemente a los algoritmos, comprobando los resultados de la Figura 4.17. Cuando los contrastes no son fuertes, el tiempo que se tarda en calcular los tiempos de llegada es ligeramente menor. Sin embargo, la disminución de tiempo es tan leve, que no es determinante.

Como se esperaba, el más afectado es GMM, ya que al tener un fuerte contraste, no se puede ajustar el valor de *delTAU* y su eficiencia se ve afectada. Esa falta de eficiencia le hace ser peor que FMM_Dary cuando el contraste es fuerte.

Como conclusión:

- GMM no es eficaz para rejillas de celdas con grandes contrastes de velocidad de propagación de la onda. Esta desventaja hace que el algoritmo sea inviable para situaciones en las que no se conoce las propiedades del mapa.

- Los contrastes no afectan prácticamente en el tiempo de cálculo a los demás algoritmos.

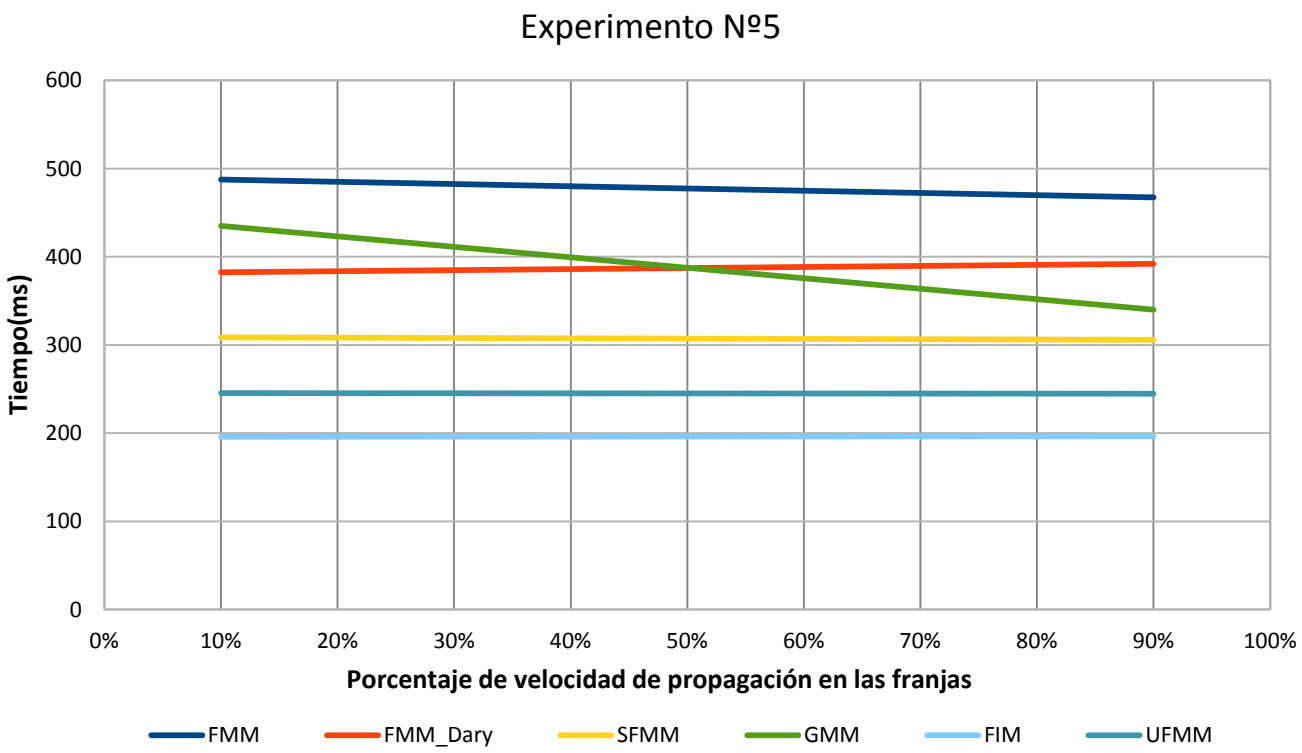


Figura 4.17: Resultado gráfico del experimento Nº5. Experimentación con contrastes de velocidad de propagación.

Experimento Nº 6:

En una de las referencias [29], se estudia el efecto que tienen los cambios de dirección en la propagación de la onda en algunos de los métodos. Este experimento pondrá a prueba esta condición en todos los algoritmos. Para ello, se usan dos mapas, ambos con la misma superficie. Las diferencias entre ellos es el número de cambios de dirección efectuados en la propagación de la onda. Uno de los mapas usados solamente tendrá tres cambios de dirección, Figura 4.18 mientras que el otro tendrá doce, Figura 4.19.

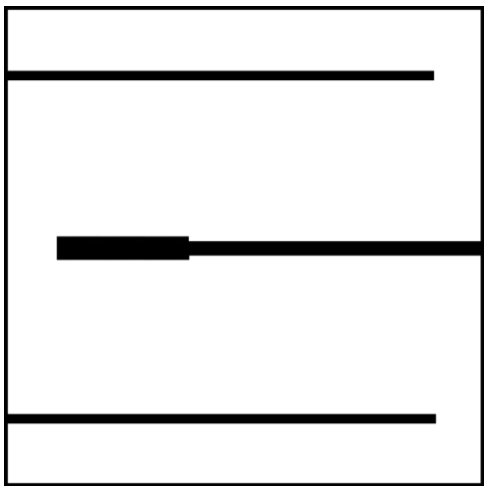


Figura 4.18: Mapa usado en el experimento Nº6, se establecen tres cambios de dirección en la propagación de la onda.

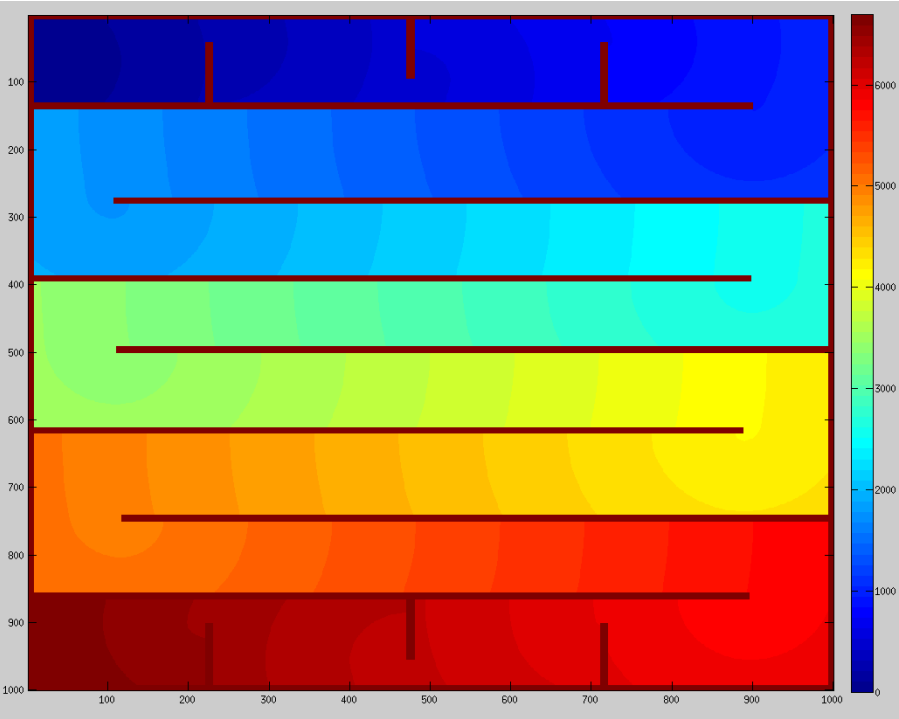


Figura 4.19: Representación gráfica de los tiempos de llegada (TT), en milisegundos, del experimento Nº6. Mapa en el que se establecen doce cambios de dirección en la propagación de la onda.

En los resultados de la Tabla 4.5, se puede apreciar una ligera disminución en el tiempo de cálculo en los algoritmos *O(n log n)* para 12 giros. Dado que en los otros algoritmos *O(n)* esta situación no se produce, la disminución del tiempo de cálculo se puede achacar a la disminución del tamaño de la *narrow band*.

En el experimento Nº 3, se demuestra que el tamaño de las *narrow band* grandes afectan a estos algoritmos lastrando su rendimiento. Cuando se producen 12 giros, la *narrow band* generada es ligeramente menor que cuando solamente existen 3 giros, tardando así menos en el tiempo de cálculo.

Este experimento nos lleva a la siguiente conclusión:

- Los cambios de dirección no afectan a los algoritmos basados en frentes de onda

Número de cambios de dirección	3	12
FMM	358,1	340,1
FMM_Dary	287,7	266,5
SFMM	229,8	216,4
GMM	300,3	311,6
UFMM	210,1	213,5
FIM	178,8	179

Tabla 4.5: Tiempo de cálculo, en milisegundos, que han necesitado cada algoritmo para calcular todos los tiempo de llegada en el experimento Nº6.

Experimento Nº 7:

En este experimento se ha creado un mapa de velocidades, mediante ruido generado aleatoriamente, unos trazos y posterior difuminación Gaussiana del conjunto. El resultado final ha sido la Figura 4.20. Este mapa se usará como dato de la velocidad de propagación de la onda. La representación de los *tiempos de llegada* cuando una onda ha sido generada en este mapa, se puede observar en la Figura 4.21. La onda circular se ha distorsionado debido a los cambios de la velocidad de propagación de la onda, causando finalmente que pierda la forma.

Este experimento servirá para averiguar la magnitud de los errores cometidos por cada uno de los algoritmos. Desde la Figura 4.22 a la Figura 4.25, representa gráficamente los errores cometidos por cada uno de los algoritmos en el cálculo de los *tiempos de llegada*(TT) en milisegundos. La cuantificación de los errores se puede ver en la barra coloreada a la derecha de cada imagen, asociando cada color a un valor de tiempo.

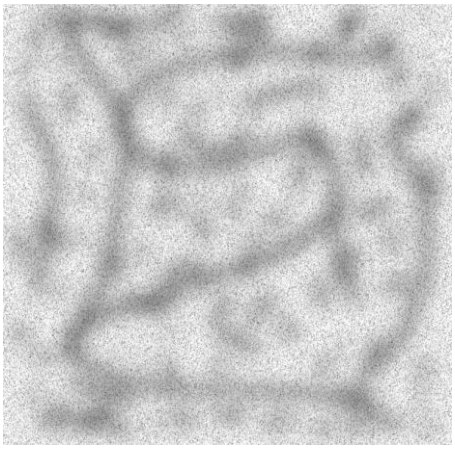


Figura 4.20: Mapa usado como cambios en la velocidad de propagación de la onda según escala de grises, utilizado en el experimento Nº7.

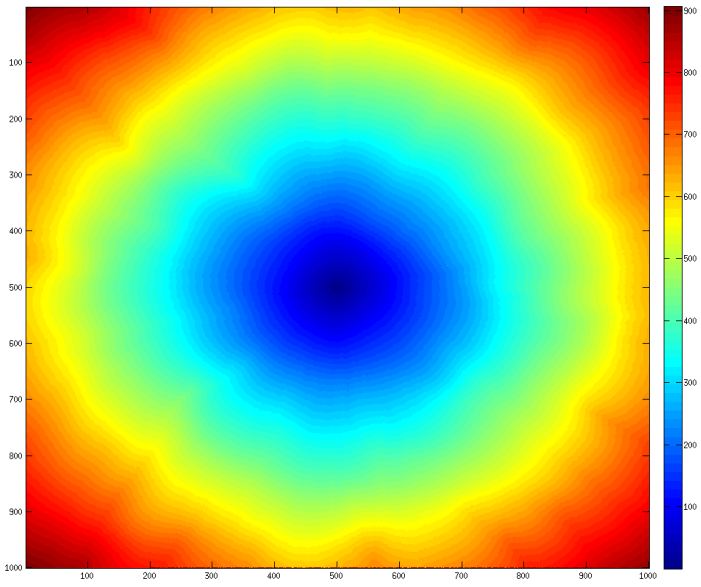


Figura 4.21: Onda resultante generada por el algoritmo FMM cuando se ha iniciado la onda en su centro.

Las conclusiones que se pueden extraer, son:

- **FMM_dary:** Se obtienen los mismos resultados de TT que en FMM. No tiene errores (respecto a FMM) pero en la mayoría de los casos, ha tardado menos que FMM para mapas con reducido número de celdas. Figura 4.22.

- **SFMM:** Se obtiene un error muy leve, del orden del milisegundo. Pero a cambio se obtiene la mejor velocidad de los algoritmos *O(n log n).* Figura 4.23.

- **GMM:** Los errores obtenidos son un poco mayores, en torno a los 15 milisegundos. Los errores se concentran en los ejes cardinales de la celda desde donde se inició la onda. En las diagonales, los errores son menores, en torno a 5 milisegundos. Figura 4.24.

- **UFMM**: Los resultados de los errores en este algoritmo se estudiarán en el experimento Nº8, debido a que su precisión depende de un parámetro.

- **FIM:** Ha sido el algoritmo con más errores de todos. También se concentran en los ejes cardinales, como pasa con GMM. Los errores llegan a ser de alrededor de 160 ms en los puntos críticos. Aunque en las diagonales se mantiene estable, sin errores. A cambio de la mayor rapidez, es el que más errores desarrolla. Figura 4.25.
  Como se comentaba en la explicación del algoritmo, punto 3.3, existe un parámetro ε (error límite). Al cambiar el valor de este parámetro no se ha observado ninguna diferencia con respecto a mantenerlo a cero.

  El tiempo que tarda en calcular los TT de todas las celdas se mantiene constante con un error u otro. Además, el error producido en el valor de los TT tampoco varía.

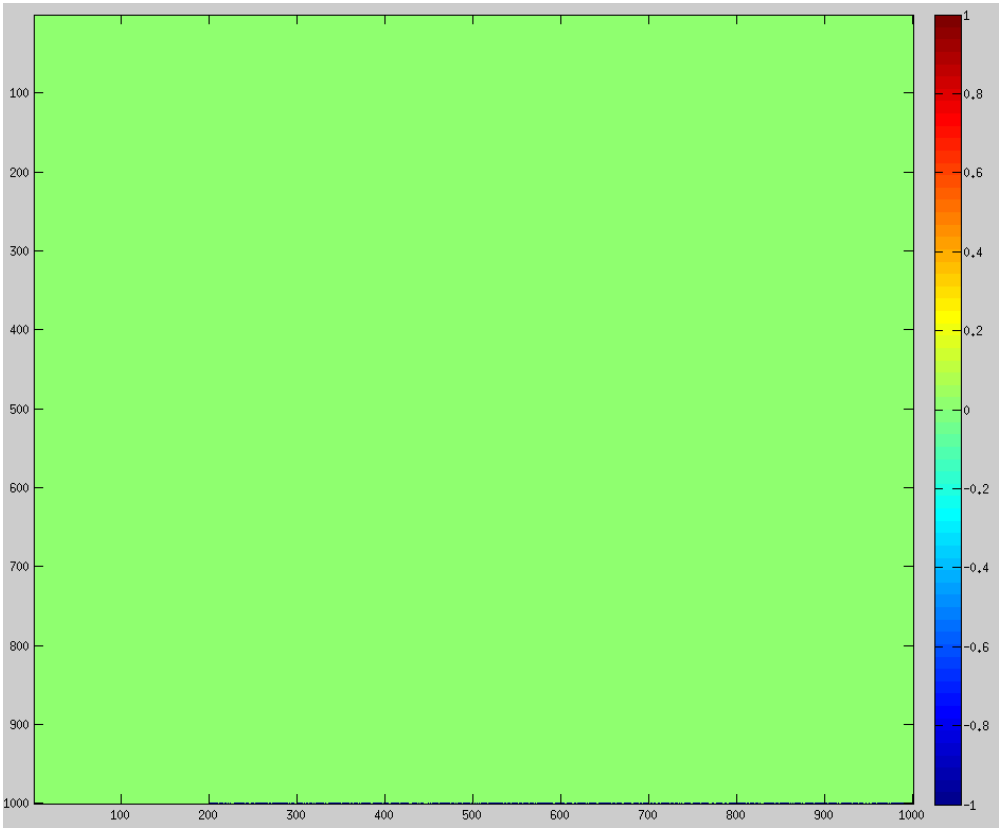


Figura 4.22: Visualización gráfica de los errores en milisegundos del algoritmo FMM_Dary.

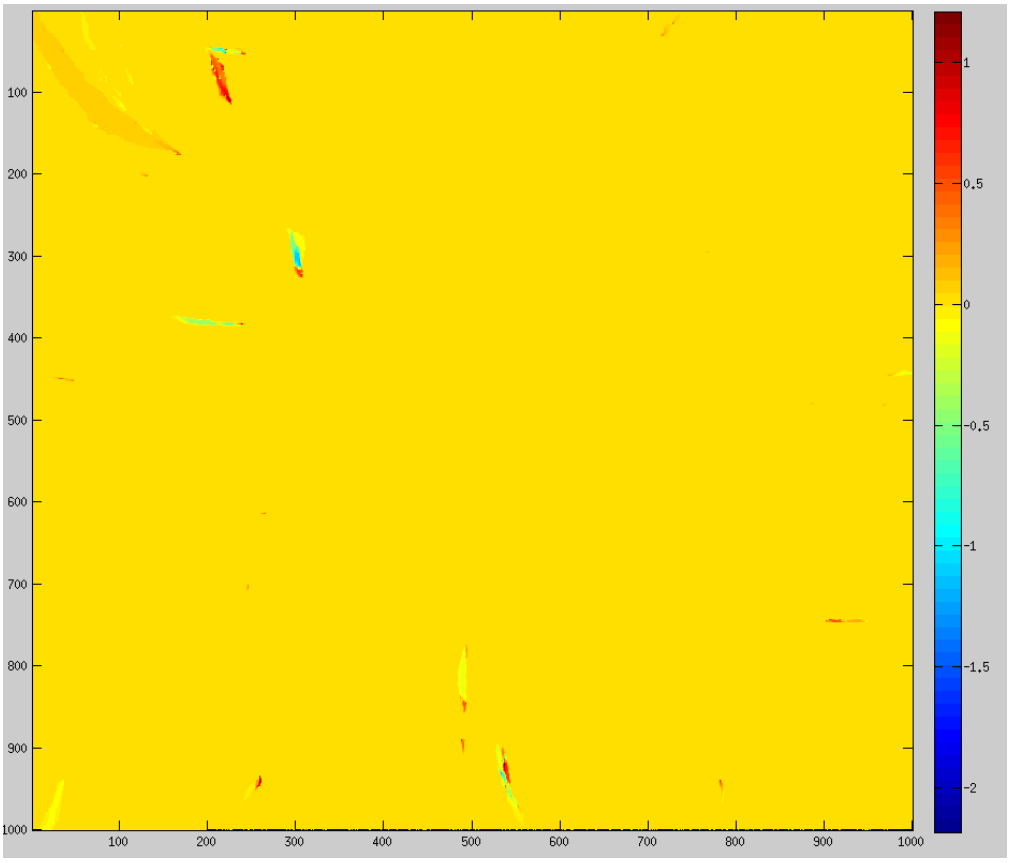


Figura 4.23: Visualización gráfica de los errores en milisegundos del algoritmo SFMM.

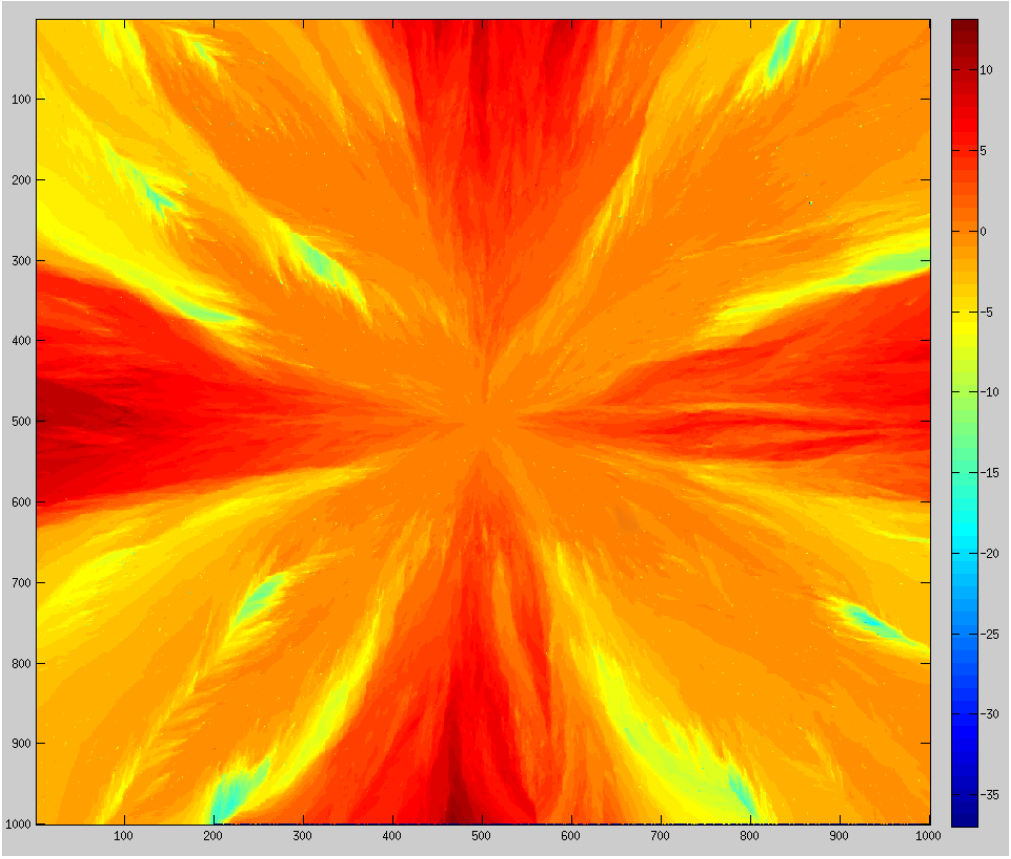


Figura 4.24: Visualización gráfica de los errores en milisegundos del algoritmo GMM.

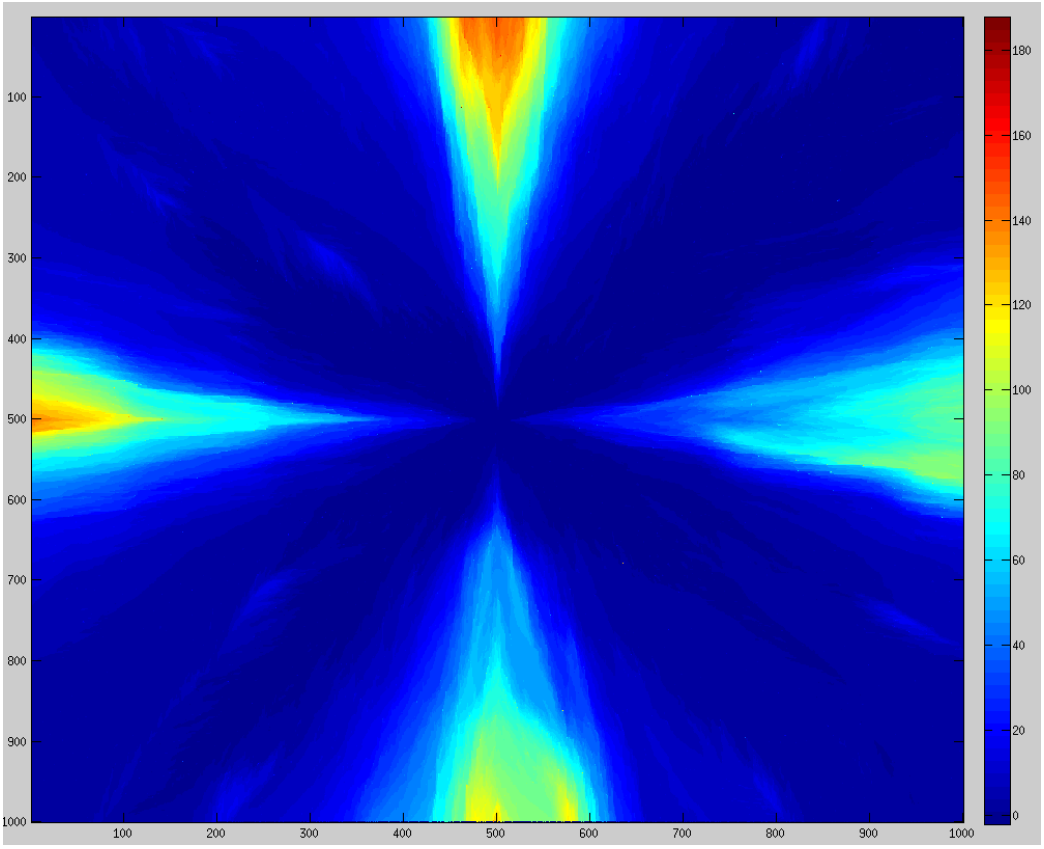


Figura 4.25: Visualización gráfica de los errores en milisegundos del algoritmo FIM.

Experimento Nº 8:

Este experimento servirá para comprobar las consecuencias, en el tiempo de cálculo y la precisión, de cambiar una serie de parámetros en el algoritmo UFMM.

Los creadores de la *Untidy Priority Queue* comprueban, en la documentación del algoritmo [12], los errores que se producen al cambiar el número de discretizaciones de tiempo en la cola de prioridad. La conclusión del estudio fue que, poca cantidad de discretizaciones es suficiente para que los errores se puedan considerar despreciables.

En la documentación, además, se relaciona la cantidad de operaciones en la cola de prioridad con la cantidad de discretizaciones utilizadas. Concluye con la observación de que, a mayor cantidad de discretizaciones, mayor es el número de operaciones. Pero los autores no especifican las consecuencias en el tiempo de cálculo de los *tiempos de llegada*(TT). En este experimento se modificará el valor de número de discretizaciones y se comparará con el tiempo que tarda el algoritmo en calcular los TT y la precisión alcanzada. Para este experimento se usará el mapa generado aleatoriamente del experimento Nº7.

En la Tabla 4.6 se observa la relación entre el tiempo de cálculo y el número de discretizaciones de tiempo usadas. Además, en las Figuras 4.26, 4.27 y 4.28 se puede comparar con la precisión conseguida, ya que se trata de la representación gráfica del error cometido respecto a FMM.

Discretizaciones	Tiempo de cálculo(ms)
1.000	337
2.000	317
4.000	295
7.000	291
10.000	290
50.000	295
70.000	297
100.000	300
250.000	325
500.000	366

Tabla 4.6: Resultado en milisegundos, del tiempo usado por UFMM cuando el número de discretizaciones temporales cambia.

El tiempo de cálculo aumenta según aumenta las discretizaciones. La precisión alcanzada, como era de esperar, mejora según aumenta el número de discretizaciones hasta alcanzar un valor del error despreciable.

Se obtiene un resultado anormal, en el tiempo de cálculo, cuando las discretizaciones están entre 1000 y 4000, ya que el resultado esperado es que el tiempo se redujera y aumentara el error. Dado el que los autores usan en sus experimentos entre 512 y 2048 discretizaciones, se comprueba, a continuación, el origen de este fenómeno.

Usando la herramienta de depuración de Ubuntu llamada Valgrind, se comprueba el tiempo empleado en cada función y el número de veces que se ha usado dicha función. Los resultados son visibles en la Tabla 4.7. En la tabla, el número de usos de la función *"Incrementar prioridad"*, una función que actualiza el valor TT de una celda, disminuye cuando las discretizaciones aumentan, tal y como se expone en la documentación[12].

Sin embargo, el tiempo usado, en porcentaje del total, por la función no es coherente con las veces que es usada. Cuando el número de discretizaciones es de 500, el tiempo usado en esa función es de 9%, mientras que cuando el número de discretizaciones es 10.000, el tiempo usado solamente corresponde al 1.7%.

Discretizaciones	Usos de la función Incrementar Prioridad	Tiempo usado por la función(% del total)
500	54.509	9.29
1.000	58.830	5.91
10.000	63.983	1.77

Tabla 4.7: Resultados de la herramienta Valgrind. Porcentajes de uso de tiempo de la función de incrementar clave.

Sabiendo que una *double linked list*, como la usada en la *untidy priority queue*, tiene las siguiente complejidad computacional:

	Buscar	**Insertar**	**Borrar**
Double linked list	O(n)	O(1)	O(1)

Tabla 4.8: Complejidad computacional de las operaciones de una double linked list[21].

En la *untidy priority queue*, la operación incremento de prioridad se realiza de la siguiente manera:

- Buscar el valor del dato que se quiere actualizar. Operación **O(n).**
- Eliminar el dato. Operación **O(1).**
- Añadir el dato con el valor actualizado al principio de la lista. Operación **O(1).**

La eliminación e inserción tienen complejidad computacional constante, una vez se sabe dónde hay que borrar o insertar. Por lo que para borrar un valor determinado, de una *double linked list*, realmente es una operación *O(n)*.

La razón de por qué invierte tanto tiempo, cuando el número de discretizaciones es bajo, es debido a la alta densidad de celdas por cada discretización. Suponiendo que todas las discretizaciones tienen un número igual de celdas, en el mapa usado de 1000 x 1000 celdas, en cada discretización existirían una lista de 2000 celdas. En cambio, cuando existe un número mayor de discretizaciones(10 000), el número de celdas por cada una de las discretizaciones se reduce a 100.

Para comprobar el impacto del uso de esta función dependiendo de la densidad de celdas, se ha diseñado un pequeño experimento. En el experimento, se realiza la búsqueda para eliminar un número mediante la función de *borrado*, usada por la lista de prioridad. La lista de números puede cambiar de tamaño para simular la densidad de celdas por discretización de tiempo.

En la Tabla 4.9 se puede comprobar el resultado obtenido para dos densidades de celdas, la densidad de 2000 celdas corresponde a un número de discretizaciones de 500 en un mapa de 1000 x 1000 celdas, mientras que una densidad de 100 corresponden al uso de 10 000 discretizaciones en ese mismo mapa.

Los resultados del experimento, indican que una alta densidad de celdas afecta al tiempo de cálculo cuando la función se repite varias veces. En el caso de usar 500 discretizaciones de tiempo, la repetición de la función "*incrementar prioridad*" se repite hasta 54.509 veces. En el caso expuesto en la Tabla 4.9, una repetición de 55 000 veces de la función de *borrado* de una *double linked list*, con alta densidad de celdas, necesita un tiempo de 220 milisegundos. En caso de que la lista tenga una baja densidad de celdas, el tiempo usado, para una cantidad mayor de repeticiones, solamente es de 19 milisegundos.

Número de repeticiones del programa	Tiempo (ms) en recorrer la lista de 2000 celdas.	Tiempo (ms) en recorrer la lista de 100 celdas.
1	0	0
1.000	8	0
25.000	98	7
55.000	220	18
65.000	260	19

Tabla 4.9: Tiempo, en milisegundos, que tarda en recorrer y borrar un número en una lista cuando sus tamaños varían y el programa se repite un número determinado de veces.

En conclusión, en la documentación de la *untidy priority queue* aparece que el número de operaciones disminuye según disminuye el número de discretizaciones. Pero la disminución del número de operaciones no implica, necesariamente, menor tiempo de cálculo. Ya que el tiempo usado en las operaciones de la cola de prioridad puede aumentar cuando la densidad de celdas, por discretización de tiempo, aumenta. Esto es debido a que la complejidad computacional de estas operaciones es O(n), y se ve afectado por el número de datos que tiene que controlar.

Disminuir la densidad de discretizaciones no hace mejorar el tiempo de cálculo para siempre. Como se ha visto, un mayor número de discretizaciones temporales hace aumentar el tiempo de cálculo, elevando la precisión.

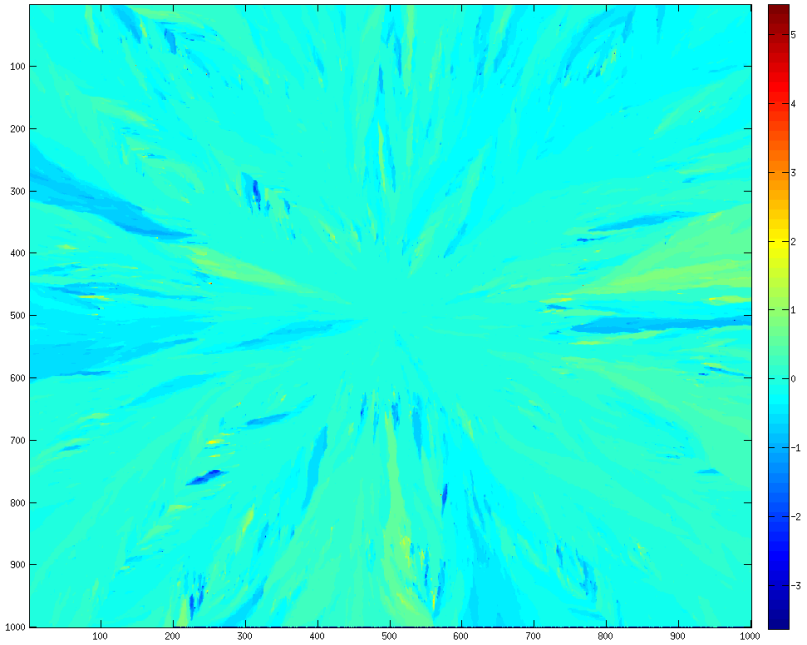


Figura 4.26: Visualización gráfica de los errores en milisegundos del algoritmo UFMM con un número de discretizaciones temporales de 1 000.

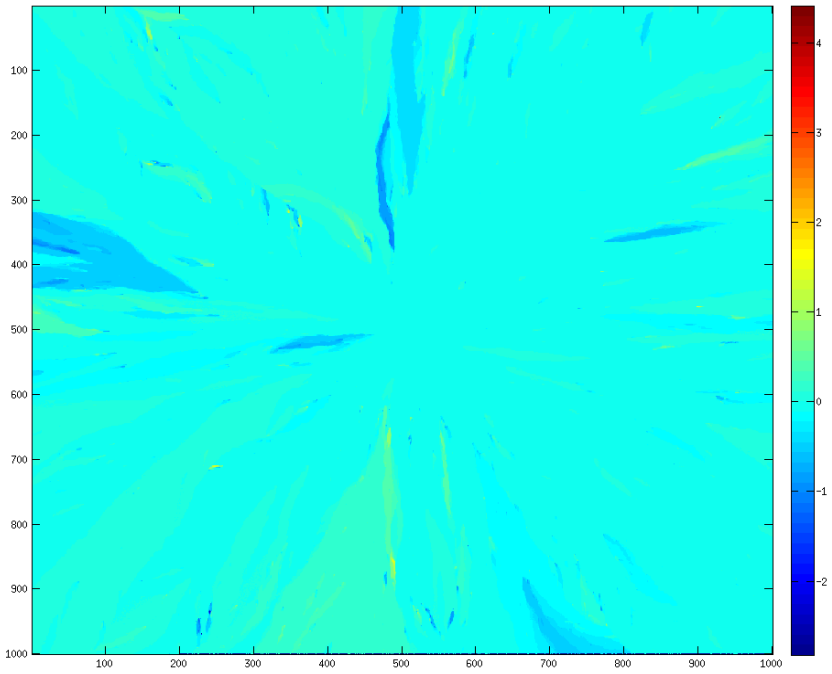


Figura 4.27: Visualización gráfica de los errores en milisegundos del algoritmo UFMM con un número de discretizaciones temporales de 10 000.

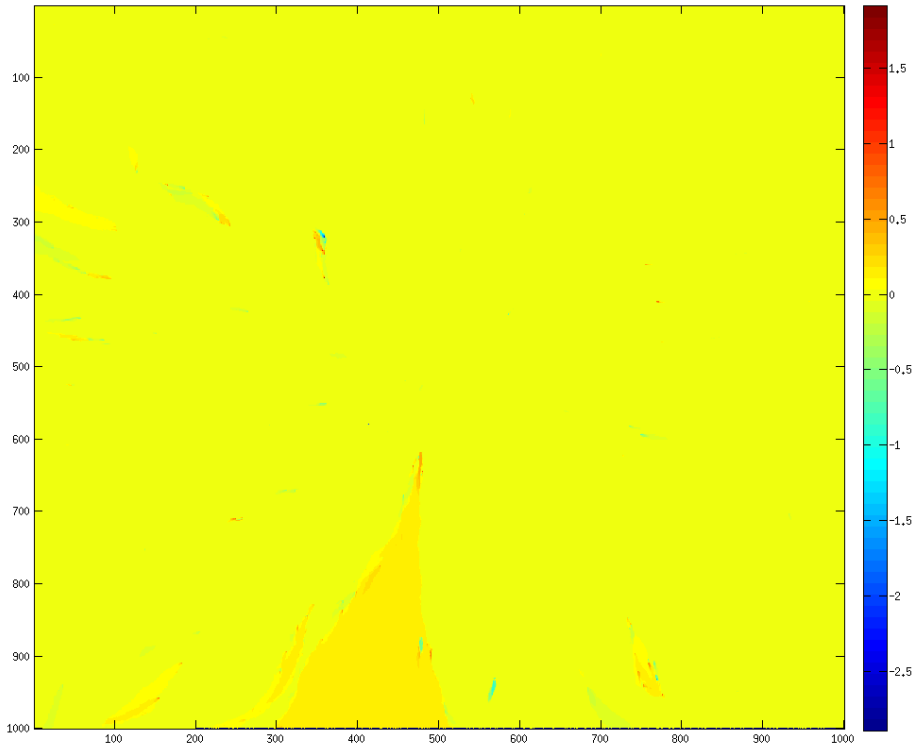


Figura 4.28: Visualización gráfica de los errores en milisegundos del algoritmo UFMM con un número de discretizaciones temporales de 70 000.

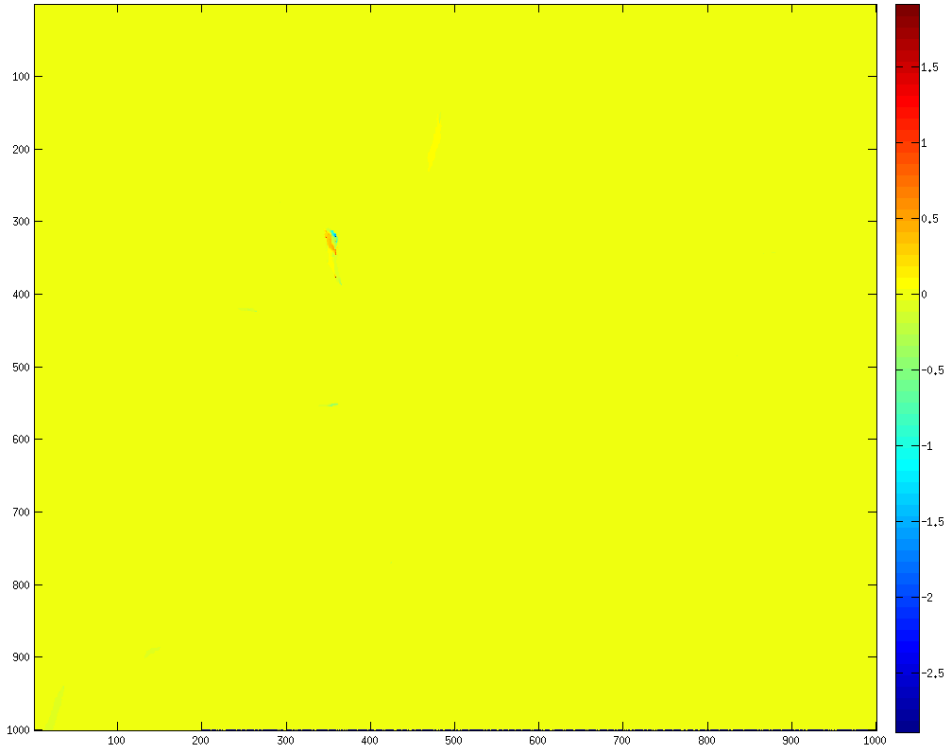


Figura 4.29: Visualización gráfica de los errores en milisegundos del algoritmo UFMM con un número de discretizaciones temporales de 500 000.

Experimento Nº 9:

Se han realizado diferentes pruebas en dos dimensiones, comprobando las propiedades de cada uno de los algoritmos. En este experimento se realizará un test en tres dimensiones, para observar si los algoritmos mantienen las tendencias anteriores.

Para este experimento se ha diseñado un mapa con velocidad de propagación constante máxima (F=1) en tres dimensiones y el inicio de la onda en el centro. En la Figura 4.30 se puede observar la forma esférica generada por la onda.

En la Figura 4.31 se puede comprobar que las tendencias generales de la propagación de la onda en dos dimensiones se mantienen cuando estas se aumentan a tres.

En la gráfica se puede observar que hay dos líneas correspondientes al algoritmo UFMM. Cada una de ellas corresponde al tiempo de cálculo con distintas discretizaciones que, como se comprueba en el experimento Nº8, afecta al resultado. Se han usado 1000 y 50000 discretizaciones "UFMM 1000 D" y "UFMM 50000 D" respectivamente.

Se comprueba que usando pocas discretizaciones la diferencia con el algoritmo GMM aumenta considerablemente. En cambio, usando más discretizaciones los métodos se mantienen muy parejos, arrojando prácticamente el mismo resultado.

El algoritmo FIM sigue siendo líder en el apartado de tiempo de cálculo.

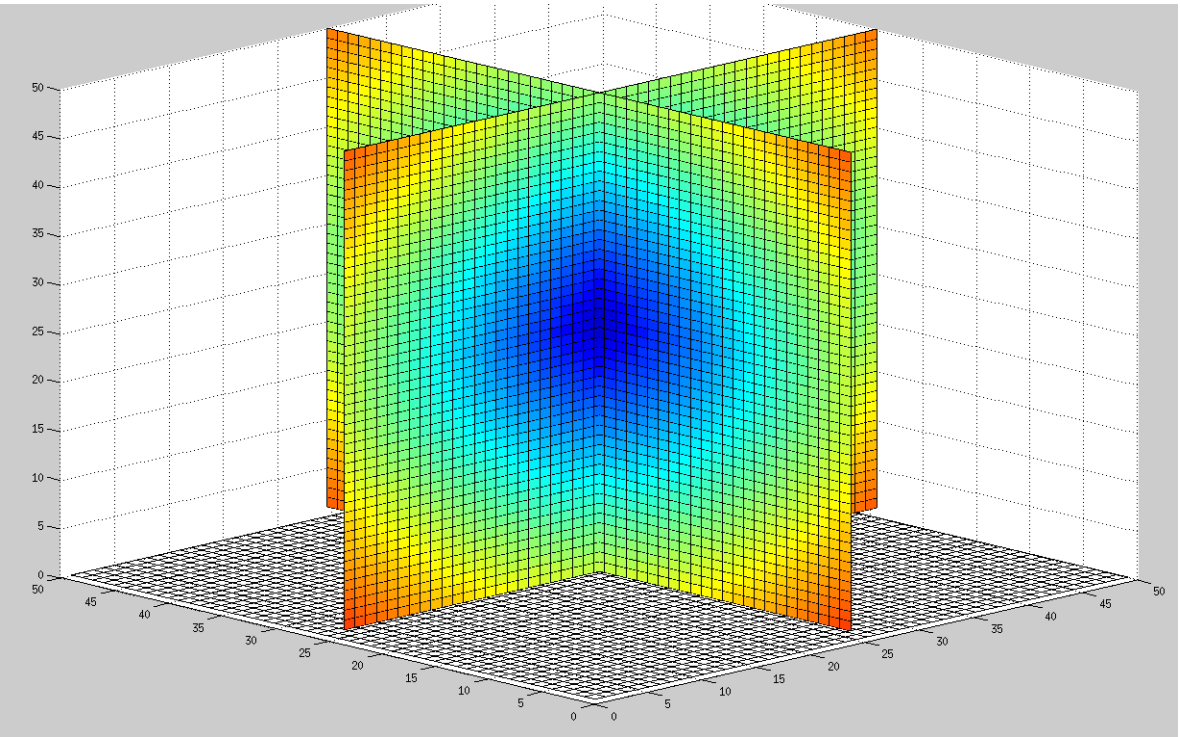


Figura 4.30: Generación de una onda esférica en el experimento Nº9. Debido a que las velocidades de propagación de la onda son constantes.

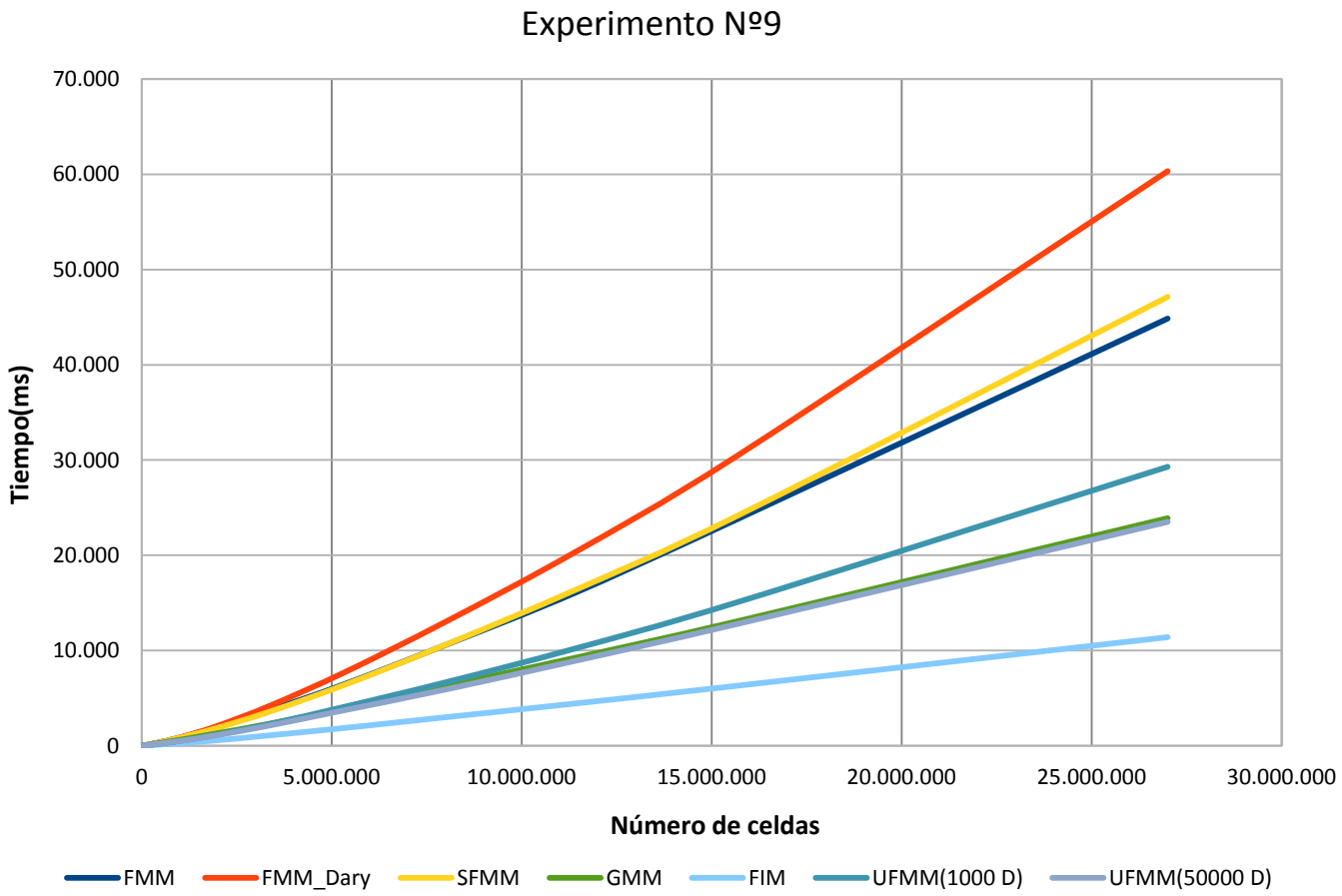


Figura 4.31: Resultado gráfico del experimento Nº9. Mapa con velocidades constantes y con una sola fuente en el centro en tres dimensiones.

En la Tabla 4.5 se expone un resumen de cómo manteniendo constante el número de celdas, han evolucionado los algoritmos en dos dimensiones y tres dimensiones, según los datos recogidos en los experimentos Nº1 y Nº9.

	Número celdas	**2D**	**3D**
FMM	1.000.000	462,2	861,4
FMM_Dary	1.000.000	450.7	868
SFMM	1.000.000	346.7	847,6
GMM	1.000.000	337.6	631,6
UFMM(1000 D)	1.000.000	241.4	510,6
FIM	1.000.000	203	275

Tabla 4.5: Resultado en milisegundos, de la comparación entre los algoritmos en 2D y 3D con el mismo número de celdas.

Capítulo 5:

CONCLUSIONES Y TRABAJO FUTURO

En los experimentos del capítulo 4, se han ido obteniendo una serie de conclusiones al interpretar los resultados. Realizando una síntesis de estas deducciones, que se detallan a continuación:

- *Fast Marching Method*(FMM): entre los métodos de complejidad computacional *O(n log n)* tiene unos resultados más lentos para mapas con poca cantidad de celdas. Con el aumento del número de celdas esta diferencia se acorta. En cambio, en tres dimensiones es el algoritmo, de esa complejidad computacional, más rápido.

- *Fast Marching Method* con pila binaria (FMM_Dary): rivaliza en lentitud con FMM para mapas con pocas celdas. En el caso de trabajar con mapas más extensos, el algoritmo se vuelve muy lento con diferencia. Además, una *narrow band* grande hace que pierda eficiencia muy rápidamente. Estas dos razones hacen de este algoritmo, una opción inviable para el uso en tres dimensiones o más. Este algoritmo no presenta errores respecto a FMM.

- *Fast Marching Method Simplificado* (SFMM): de los algoritmos *O(n log n)*, es el que mejor resultados ofrece. Para mapas con pocas celdas, rivaliza con Group Marching Method, debido a que, a pesar de su complejidad computacional, es el algoritmo que menos se ve afectado por el tamaño de la *narrow band*. En tres dimensiones ofrece una eficiencia similar que FMM. Tiene errores de cálculo en los *tiempos de llegada*(TT), pero son ínfimos comparado con la velocidad que aporta.

- *Group Marching Method* (GMM): de los algoritmos *O(n)*, es el más lento en dos dimensiones y para mapas con pocas celdas es peor que SFMM. Aunque en tres dimensiones su eficiencia mejora apreciablemente.
  El principal problema de este algoritmo es cuando existen contrastes de velocidad de propagación de la onda. Se ve muy afectado por esta circunstancia, empeorando su eficiencia de manera notable y haciéndolo poco aconsejable para situaciones en las que las condiciones del mapa son desconocidas. También presenta errores moderados en el cálculo de los TT.

- *Untidy Fast Marching Method* (UFMM): este algoritmo ofrece buenas prestaciones tanto en tiempo como en precisión. En todas las pruebas realizadas ha sido el segundo algoritmo más rápido. Además, los errores generados han sido muy leves comparados con GMM y sobre todo FIM. Además, cambiando el número de discretizaciones es capaz de aumentar la precisión, a costa del aumento del consumo de tiempo, haciéndolo muy adaptable.
  Pierde eficiencia respecto a GMM cuando se ejecuta el algoritmo en tres dimensiones y la densidad por discretización de tiempo es alta, pero sigue siendo la mejor opción en la relación rapidez de cálculo y precisión.

- *Fast Iterative Method* (FIM): este algoritmo es el que mejor tiempo de cálculo ha obtenido en todos los experimentos realizados, en dos y tres dimensiones o con grandes contrastes de velocidad de propagación. El problema de este algoritmo es que genera errores notables en el cálculo de los TT.

También se han obtenido las siguientes conclusiones generales:

- Los algoritmos *O(n)*, no se ven prácticamente afectados por el tamaño de *narrow band*. En cambio, los de orden *O(n log n*) si son afectados en diferente grado, empeorando o mejorando según el tamaño de la *narrow band*.

- Todos los algoritmos estudiados en este trabajo son inmunes a los contrastes en la velocidad de propagación de la onda excepto el algoritmo GMM.

- Los cambios de sentido en la dirección de propagación de la onda, no intervienen, de ninguna manera, en el tiempo de cálculo de los algoritmos basados en frente de onda.

- Cuando los mapas tienen velocidades de propagación constantes (F=cte), los errores entre los algoritmos son nulos. Pudiendo ser buena opción elegir el algoritmo más rápido, FIM.

-Trabajo futuro

Este trabajo se ha centrado en comparar seis algoritmos basados en frentes de onda. Como trabajo futuro, sería necesario comparar con los algoritmos basados en "*Fast Sweeping Methods"*, ya que aportan una nueva perspectiva ante el mismo problema o soluciones para otras circunstancias. Por ejemplo, los *"sweeping methods"* solamente

funcionan para mapas con velocidad de propagación de onda constante. Pero podrían arrojar mejores resultados en este tipo de mapas que los algoritmos basados en frentes de onda aunque sean menos versátiles.

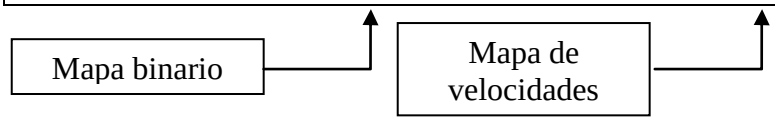
También se podría comparar con nuevos algoritmo que se desarrollen en un futuro, tanto los basados en frente de ondas como en *"sweeping methods"*.

APÉNDICES

a. Tutorial de uso de código

Para iniciar el programa en Linux, hay que dirigirse a la carpeta fastmarching y a la subcarpeta /build:

```
cd fastmarching/build
$ cmake ..
$ make
$ ./fmm -map ../data/map.png -vel ../data/velocitymap.png
```



Para cargar mapas en el programa, se pueden colocar los archivos en la carpeta /data. Es posible cargar dos clases de mapas:

- Mapas binarios: el color negro se toma como obstáculo y el color blanco como celda en la que se puede propagar la onda a velocidad máxima.

- Mapas de velocidades: se tomará la imagen en escala de grises, en el que cada uno de los 256 niveles corresponde a una velocidad de propagación de la onda en esa celda. Siendo 256 el color blanco y máxima velocidad de propagación F=1.

A continuación, se explicará el código:

```
1  int main(int argc, const char ** argv)
2  {
3  constexpr int ndims2 = 2;
4  constexpr int ndims3 = 3;
5
6  typedef nDGridMap<FMCell, ndims2> FMGrid2D;
7  typedef nDGridMap<FMUntidyCell, ndims2> FMGrid2DUntidy;
8
9  time_point<std::chrono::system_clock> start, end;
10 double time_elapsed;
11
12 string filename, filename_vels;
13 console::parseArguments(argc,argv, "-map", filename);
14 console::parseArguments(argc,argv, "-vel", filename_vels);
15
```

Obtención de los mapas que se pasan por argumento al iniciar el programa. El mapa binario pasa a llamarse "*filename*" y el mapa de velocidades "*filename_vels*" **(13) y (14)**.

```
16  nDGridMap<FMCell, ndims2> grid_fmm;
17  nDGridMap<FMCell, ndims2> grid_fmm_dary;
18  nDGridMap<FMCell, ndims2> grid_sfmm;
19  nDGridMap<FMCell, ndims2> grid_gmm;
20  nDGridMap<FMUntidyCell, ndims2> grid_ufmm;
21  nDGridMap<FMCell, ndims2> grid_fim;
22
```

Creación de cada una de las rejillas discretizadas del espacio de cada algoritmo.
Cada rejilla usa una celda "*Fast marching Cell*", que contiene las funciones que controlan la información de la celda, como por ejemplo, la velocidad de propagación o el tiempo de llegada.

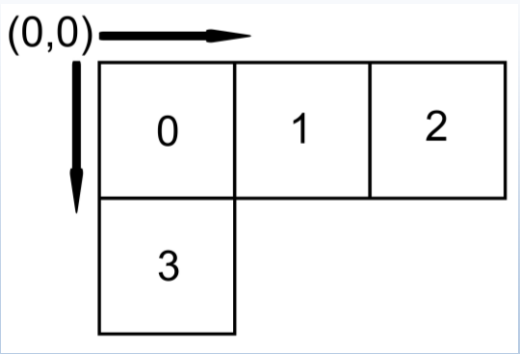
UFMM usa una "Untidy Fast Marching Cell", que es prácticamente igual que la FMCell pero con algunas funciones adicionales para el control de la cola de prioridad.
El segundo parámetro es el número de dimensiones.

```
23  MapLoader::loadMapFromImg(filename.c_str(), grid_fmm);
24  MapLoader::loadMapFromImg(filename.c_str(), grid_fmm_dary);
25  MapLoader::loadMapFromImg(filename.c_str(), grid_sfmm);
26  MapLoader::loadMapFromImg(filename.c_str(), grid_gmm);
27  MapLoader::loadMapFromImg(filename.c_str(), grid_ufmm);
28  MapLoader::loadMapFromImg(filename.c_str(), grid_fim);
29  GridPlotter::plotMap(grid_fmm,0);
30
```

Se crea una rejilla, para cada uno de los algoritmos, a partir del mapa binario.
(29) permite la representación por pantalla del mapa que se acaba de crear.

```
31  vector<int> init_points;
32  array<int,ndims2> coords = {44,40};
33          int idx;
34  grid_fmm.coord2idx(coords, idx);
35  init_points.push_back(idx);
36
```

Creación del punto o puntos de inicio desde donde se va a lanzar la onda.
Cada rejilla creada tiene la función **(34)**, que permite guardar en una variable la conversión de las coordenadas. La conversión se realiza de la siguiente manera:



```
37  console::info("Testing algorithms");
38  ////////////FMM////////////
39  console::info("Testing FMM");
40  FastMarching<FMGrid2D, FMFibHeap<FMCell> > fmm;
41  fmm.setEnvironment(&grid_fmm);
42  start = system_clock::now();
43  fmm.setInitialPoints(init_points);
44  fmm.computeFM();
45          end = system_clock::now();
46          time_elapsed = duration_cast<milliseconds>(end-start).count();
47          cout << "\tElapsed FMM time: " << time_elapsed << " ms" << endl;
48  GridPlotter::plotArrivalTimes(grid_fmm,0);
49  GridWriter::saveGridValues("test_fmm.txt", grid_fmm);
50
```

(37) y (39) muestran por pantalla caracteres con un formato más vistoso que el comando 'cout' .

(40) crea un objeto que representa el algoritmo *"Fast Marching Method"*. **(41)** Se encarga de inicializar el algoritmo con las condiciones iniciales, ver Capitulo 2.2.

(44) genera la expansión de la onda en el mapa y mediante **(42)** y **(45)** permiten calcular el tiempo que le ha tomado al algoritmo calcular todos los tiempos de viaje.

Con **(48)** se representa por pantalla los resultados de los tiempo de viaje mediante un código de colores, el parámetro '0', permite la inversión vertical, ya que el mapa creado está traspuesto.

(49) permite escribir los valores de los tiempos de viaje en un fichero '.txt' para poder representar en otros programas como Matlab.

```
51  ///////////FMM_dary////////////
52  console::info("Testing FMM_Dary ");
53  FastMarching<FMGrid2D> fmm_dary;
54  fmm_dary.setEnvironment(&grid_fmm_dary);
55  start = system_clock::now();
56  fmm_dary.setInitialPoints(init_points);
57  fmm_dary.computeFM();
58          end = system_clock::now();
59          time_elapsed = duration_cast<milliseconds>(end-start).count();
60          cout << "\tElapsed FMM_Dary time: " << time_elapsed << " ms" << endl;
61  GridWriter::saveGridValues("test_fmmdary.txt", grid_fmm_dary);
62
63  ///////////SFMM////////////
64  console::info("Testing SFMM ");
65  FastMarching<FMGrid2D, FMPriorityQueue<> > sfmm; //Choosing the default cell class.
66  sfmm.setEnvironment(&grid_sfmm);
67          start = system_clock::now();
68  sfmm.setInitialPoints(init_points);
69  sfmm.computeFM();
70          end = system_clock::now();
71          time_elapsed = duration_cast<milliseconds>(end-start).count();
72          cout << "\tElapsed SFMM time: " << time_elapsed << " ms" << endl;
```

```
73  GridWriter::saveGridValues("test_sfmm.txt", grid_sfmm);
74  ////////////GMM/////////////
75  console::info("Testing GMM ");
76  GroupMarchinglist<FMGrid2D> gmm;
77  gmm.setEnvironment(&grid_gmm);
78          start = system_clock::now();
79  gmm.setInitialPoints(init_points);
80  gmm.computeFM();
81          end = system_clock::now();
82          time_elapsed = duration_cast<milliseconds>(end-start).count();
83          cout << "\tElapsed GMM time: " << time_elapsed << " ms" << endl;
84  GridWriter::saveGridValues("test_gmm.txt", grid_gmm);
85
86  ///////////UFMM/////////////
87  console::info("Testing UFMM ");
88  FMM_Untidy<FMGrid2DUntidy> ufmm;
89  ufmm.setEnvironment(&grid_ufmm);
90          start = system_clock::now();
91  ufmm.setInitialPoints(init_points);
92  ufmm.computeFM();
93          end = system_clock::now();
94          time_elapsed = duration_cast<milliseconds>(end-start).count();
95          cout << "\tElapsed UFMM time: " << time_elapsed << " ms" << endl;
96  GridWriter::saveGridValues("test_fmm_untidy.txt", grid_ufmm);
97
98  ///////////FIM/////////////
99  console::info("Testing FIM ");
100 FastIterativeMethod<FMGrid2D> fim;
101 fim.setEnvironment(&grid_fim);
102         start = system_clock::now();
103 fim.setInitialPoints(init_points);
104 fim.computeFM();
105         end = system_clock::now();
106         time_elapsed = duration_cast<milliseconds>(end-start).count();
107         cout << "\tElapsed FIM time: " << time_elapsed << " ms" << endl;
108 GridWriter::saveGridValues("test_fim.txt", grid_fim);
```

Se expande una onda en cada mapa, por cada uno de los cinco algoritmos restantes. Además de crearse los archivos con los tiempos de llegada correspondientes a cada celda.
En el terminal de Linux puede verse el tiempo que tarda cada uno de los algoritmos:

```
[INFO] Testing algorithms
[INFO] Testing FMM
       Elapsed FMM time: 540 ms
[INFO] Testing FMM_Dary
       Elapsed FMM_Dary time: 449 ms
[INFO] Testing SFMM
       Elapsed SFMM time: 326 ms
[INFO] Testing GMM
       Elapsed GMM time: 419 ms
[INFO] Testing UFMM
       Elapsed UFMM time: 306 ms
[INFO] Testing FIM
       Elapsed FIM time: 274 ms
```

```
111 console::info("Computing gradient descent ");
112
113 int goal;
114 grid_fmm.coord2idx(std::array<int, ndims2>{1426,1121}, goal);
115
116 typedef typename std::vector< std::array<double, ndims2> > Path; // A bit of short
117 hand.
118 Path path_fmm;
119 Path path_fmm_dary;
120 Path path_sfmm;
121 Path path_gmm;
122 Path path_ufmm;
123 Path path_fim;
124
125 GradientDescent< nDGridMap<FMCell, ndims2> > grad_fmm;
126 GradientDescent< nDGridMap<FMCell, ndims2> > grad_fmm_dary;
127 GradientDescent< nDGridMap<FMCell, ndims2> > grad_sfmm;
128 GradientDescent< nDGridMap<FMCell, ndims2> > grad_gmm;
129 GradientDescent< nDGridMap<FMUntidyCell, ndims2> > grad_ufmm;
130 GradientDescent< nDGridMap<FMCell, ndims2> > grad_fim;
131
```

Se crea un punto de llegada de la onda, mediante el mismo mecanismo que cuando se crea un punto de inicio de la onda **(113)** y **(114)**.

Se crea un vector de puntos, con sus correspondientes número de dimensiones **(118)-(123)**, que guardarán el camino más corto, temporalmente hablando, desde el inicio de la onda hasta el punto de llegada. Se crea uno para cada algoritmo
(125) - (130) crea objetos de la clase GradientDescent, que permite encontrar el camino más corto a partir de los tiempos de llegada hallados en los pasos anteriores.

```
132 grad_fmm.apply(grid_fmm,goal,path_fmm);
133         GridWriter::savePath("path_fmm.txt", grid_fmm, path_fmm);
134 grad_fmm_dary.apply(grid_fmm_dary,goal,path_fmm_dary);
135         GridWriter::savePath("path_fmm_dary.txt", grid_fmm_dary, path_fmm_dary);
136 grad_sfmm.apply(grid_sfmm,goal,path_sfmm);
137         GridWriter::savePath("path_sfmm.txt", grid_sfmm, path_sfmm);
138 grad_gmm.apply(grid_gmm,goal,path_gmm);
139         GridWriter::savePath("path_gmm.txt", grid_gmm, path_gmm);
140 grad_ufmm.apply(grid_ufmm,goal,path_ufmm);
141         GridWriter::savePath("path_ufmm.txt", grid_ufmm, path_ufmm);
142 grad_fim.apply(grid_fim,goal,path_fim);
143         GridWriter::savePath("path_fim.txt", grid_fim, path_fim);
144
145 GridPlotter::plotMapPath(grid_fmm,path_fmm,0);
146
147
```

Se calcula el camino **(132)**, según los tiempos de llegada calculados anteriormente, por el que la onda tarda menos tiempo en llegar desde el inicio hasta el destino. Mediante **(133)** se escribe la lista de celdas, por sus coordenadas, que simboliza el camino calculado y lo guarda en un archivo '.txt'.
(145) Representa por pantalla el camino calculado situándolo encima de la representación en colores de los tiempos de llegada.

```
148 console::info("Testing algorithms in different velocities");
149
150 init_points.clear();
151 init_points.push_back(58571);
152 int goal_vels;
153 grid_fmm.coord2idx(std::array<int, ndims2>{1426,1121}, goal_vels);
154
155 nDGridMap<FMCell, ndims2> grid_fmm_vels;
156 nDGridMap<FMCell, ndims2> grid_fmm_dary_vels;
157 nDGridMap<FMCell, ndims2> grid_sfmm_vels;
158 nDGridMap<FMCell, ndims2> grid_gmm_vels;
159 nDGridMap<FMUntidyCell, ndims2> grid_ufmm_vels;
160 nDGridMap<FMCell, ndims2> grid_fim_vels;
161 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_fmm_vels);
162 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_fmm_dary_vels);
162 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_sfmm_vels);
163 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_gmm_vels);
164 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_ufmm_vels);
165 MapLoader::loadVelocitiesFromImg(filename_vels.c_str(), grid_fim_vels);
166 GridPlotter::plotMap(grid_fmm_vels, 0);
167
```

Se vuelven a crear las rejillas para cada algoritmo pero usando el mapa de velocidades y **no** el mapa binario.

```
168 ///////////FMM/////////////
169 FastMarching<FMGrid2D, FMFibHeap<FMCell> > fmm_vels;
170 fmm_vels.setEnvironment(&grid_fmm_vels);
171         start = system_clock::now();
172 fmm_vels.setInitialPoints(init_points);
173 fmm_vels.computeFM();
174         end = system_clock::now();
175         time_elapsed = duration_cast<milliseconds>(end-start).count();
176         cout << "\tElapsed FM time: " << time_elapsed << " ms" << endl;
177 GridWriter::saveGridValues("vel_grid_fmm.txt", grid_fmm_vels);
178 GridPlotter::plotArrivalTimes(grid_fmm_vels,0);
179 ////////////FMM_dary//////////////
180 FastMarching<FMGrid2D> fmm_dary_vels;
181 fmm_dary_vels.setEnvironment(&grid_fmm_dary_vels);
182         start = system_clock::now();
183 fmm_dary_vels.setInitialPoints(init_points);
184 fmm_dary_vels.computeFM();
```

```
185         end = system_clock::now();
186         time_elapsed = duration_cast<milliseconds>(end-start).count();
187         cout << "\tElapsed FMMdary time: " << time_elapsed << " ms" << endl;
188
189 GridWriter::saveGridValues("vel_grid_fmm_dary.txt", grid_fmm_dary_vels);
190 ////////////SFMM/////////////
191 FastMarching<FMGrid2D, FMPriorityQueue<> > sfmm_vels;
192 sfmm_vels.setEnvironment(&grid_sfmm_vels);
193         start = system_clock::now();
194 sfmm_vels.setInitialPoints(init_points);
195 sfmm_vels.computeFM();
196         end = system_clock::now();
197         time_elapsed = duration_cast<milliseconds>(end-start).count();
198         cout << "\tElapsed SFMM time: " << time_elapsed << " ms" << endl;
199 GridWriter::saveGridValues("vel_grid_sfmm.txt", grid_sfmm_vels);
200 ////////////GMM/////////////
201 GroupMarchinglist<FMGrid2D> gmm_vels;
202 gmm_vels.setEnvironment(&grid_gmm_vels);
203         start = system_clock::now();
204 gmm_vels.setInitialPoints(init_points);
205 gmm_vels.computeFM();
206         end = system_clock::now();
207         time_elapsed = duration_cast<milliseconds>(end-start).count();
208         cout << "\tElapsed GMM time: " << time_elapsed << " ms" << endl;
209 GridWriter::saveGridValues("vel_grid_gmm.txt", grid_gmm_vels);
210 ////////////UFMM/////////////
211 FMM_Untidy<FMGrid2DUntidy> ufmm_vels;
212 ufmm_vels.setEnvironment(&grid_ufmm_vels);
213         start = system_clock::now();
214 ufmm_vels.setInitialPoints(init_points);
215 ufmm_vels.computeFM();
216         end = system_clock::now();
217         time_elapsed = duration_cast<milliseconds>(end-start).count();
218         cout << "\tElapsed UFMM time: " << time_elapsed << " ms" << endl;
219 GridWriter::saveGridValues("vel_grid_ufmm.txt", grid_ufmm_vels);
220
221 ///////////FIM/////////////
222 FastIterativeMethod<FMGrid2D> fim_vels;
223 fim_vels.setEnvironment(&grid_fim_vels);
224         start = system_clock::now();
225 fim_vels.setInitialPoints(init_points);
226 fim_vels.computeFM();
227         end = system_clock::now();
228         time_elapsed = duration_cast<milliseconds>(end-start).count();
229         cout << "\tElapsed FIM time: " << time_elapsed << " ms" << endl;
230 GridWriter::saveGridValues("vel_grid_fim.txt", grid_fim_vels);
231
```

```
232  console::info("Computing gradient descent of different velocities ");
233  Path path_fmm_vels;
234  Path path_fmm_dary_vels;
235  Path path_sfmm_vels;
236  Path path_gmm_vels;
237  Path path_ufmm_vels;
238  Path path_fim_vels;
239
240  GradientDescent< nDGridMap<FMCell, ndims2> > grad_fmm_vels;
241  GradientDescent< nDGridMap<FMCell, ndims2> > grad_fmm_dary_vels;
242  GradientDescent< nDGridMap<FMCell, ndims2> > grad_sfmm_vels;
243  GradientDescent< nDGridMap<FMCell, ndims2> > grad_gmm_vels;
244  GradientDescent< nDGridMap<FMUntidyCell, ndims2> > grad_ufmm_vels;
245  GradientDescent< nDGridMap<FMCell, ndims2> > grad_fim_vels;
246
247  grad_fmm_vels.apply(grid_fmm_vels,goal_vels,path_fmm_vels);
248          GridWriter::savePath("path_fmm_vels.txt", grid_fmm_vels, path_fmm_vels);
249  grad_fmm_dary_vels.apply(grid_fmm_dary_vels,goal_vels,path_fmm_dary_vels);
250          GridWriter::savePath("path_fmm_dary_vels.txt", grid_fmm_dary_vels
251  ath_fmm_dary_vels);
252  grad_sfmm_vels.apply(grid_sfmm_vels,goal_vels,path_sfmm_vels);
253          GridWriter::savePath("path_sfmm_vels.txt", grid_sfmm_vels, path_sfmm_vels);
254  grad_gmm_vels.apply(grid_gmm_vels,goal_vels,path_gmm_vels);
255          GridWriter::savePath("path_gmm_vels.txt", grid_gmm_vels, path_gmm_vels);
256  grad_ufmm_vels.apply(grid_ufmm_vels,goal_vels,path_ufmm_vels);
257          GridWriter::savePath("path_ufmm_vels.txt", grid_ufmm_vels, path_ufmm_vels);
258  grad_fim_vels.apply(grid_fim_vels,goal_vels,path_fim_vels);
259          GridWriter::savePath("path_fim_vels.txt", grid_fim_vels, path_fim_vels);
260
261  GridPlotter::plotMapPath(grid_fmm_vels,path_fmm_vels,0);
262
263
```

Se siguen los pasos anteriores en la resolución de los tiempos de llegada y posterior cálculo del camino más corto, pero usando el mapa de velocidades.

```
[INFO] Testing algorithms in different velocities
        Elapsed FM time: 1127 ms
        Elapsed FMMdary time: 915 ms
        Elapsed SFMM time: 660 ms
        Elapsed GMM time: 430 ms
        Elapsed UFMM time: 304 ms
        Elapsed FIM time: 269 ms
[INFO] Computing gradient descent of different velocities
```

```
264 console::info("Testing algorithms in 3D");
265
266 typedef nDGridMap<FMCell, ndims3> FMGrid3D;
267 typedef nDGridMap<FMUntidyCell, ndims3> FMGrid3DUntidy;
268 typedef array<int, ndims3> Coord3D;
269
270 int x=100;
271 int y=100;
272 int z=100;
273 Coord3D dimsize = {x,y,z};
274 Coord3D init_point = {x/2, y/2, z/2};
275 init_points.push_back(idx);
276
277 FMGrid3D grid_fmm3D (dimsize);
278 FMGrid3D grid_fmm_dary3D (dimsize);
279 FMGrid3D grid_sfmm3D (dimsize);
280 FMGrid3D grid_gmm3D (dimsize);
281 FMGrid3DUntidy grid_ufmm3D (dimsize);
282 FMGrid3D grid_fim3D (dimsize);
283
284 ///////////FMM/////////////
285 FastMarching<FMGrid3D, FMFibHeap<FMCell> > fmm3D;
286 fmm3D.setEnvironment(&grid_fmm3D);
287         start = system_clock::now();
288 fmm3D.setInitialPoints(init_points);
289 fmm3D.computeFM();
290         end = system_clock::now();
291         time_elapsed = duration_cast<milliseconds>(end-start).count();
292         cout << "\tElapsed FMM time: " << time_elapsed << " ms" << endl;
293 GridWriter::saveGridValues("test_fmm3D.txt", grid_fmm3D);
294
295 ///////////FMM_dary/////////////
296 FastMarching<FMGrid3D> fmm_dary3D;
297 fmm_dary3D.setEnvironment(&grid_fmm_dary3D);
298         start = system_clock::now();
299 fmm_dary3D.setInitialPoints(init_points);
300 fmm_dary3D.computeFM();
301         end = system_clock::now();
302         time_elapsed = duration_cast<milliseconds>(end-start).count();
303         cout << "\tElapsed FMM_Dary time: " << time_elapsed << " ms" << endl;
304 GridWriter::saveGridValues("test_fmm_dary3D.txt", grid_fmm_dary3D);
305
306 ///////////SFMM/////////////
307 FastMarching<FMGrid3D, FMPriorityQueue<> > sfmm3D;
308 sfmm3D.setEnvironment(&grid_sfmm3D);
309         start = system_clock::now();
310 sfmm3D.setInitialPoints(init_points);
311 sfmm3D.computeFM();
```

```
312         end = system_clock::now();
313         time_elapsed = duration_cast<milliseconds>(end-start).count();
314         cout << "\tElapsed SFMM time: " << time_elapsed << " ms" << endl;
315 GridWriter::saveGridValues("test_sfmm3D.txt", grid_sfmm3D);
316
317 ///////////GMM/////////////
318 GroupMarchinglist<FMGrid3D> gmm3D;
319 gmm3D.setEnvironment(&grid_gmm3D);
320         start = system_clock::now();
321 gmm3D.setInitialPoints(init_points);
322 gmm3D.computeFM();
323         end = system_clock::now();
324         time_elapsed = duration_cast<milliseconds>(end-start).count();
325         cout << "\tElapsed GMM time: " << time_elapsed << " ms" << endl;
326 GridWriter::saveGridValues("test_gmm3D.txt", grid_gmm3D);
327
328 ///////////UFMM/////////////
329 FMM_Untidy<FMGrid3DUntidy> ufmm3D;
330 ufmm3D.setEnvironment(&grid_ufmm3D);
331         start = system_clock::now();
332 ufmm3D.setInitialPoints(init_points);
333 ufmm3D.computeFM();
334         end = system_clock::now();
335         time_elapsed = duration_cast<milliseconds>(end-start).count();
336         cout << "\tElapsed UFMM time: " << time_elapsed << " ms" << endl;
337 GridWriter::saveGridValues("test_fmm_untidy3D.txt", grid_ufmm3D);
338
339 ///////////FIM/////////////
340 FastIterativeMethod<FMGrid3D> fim3D;
341 fim3D.setEnvironment(&grid_fim3D);
342         start = system_clock::now();
343 fim3D.setInitialPoints(init_points);
344 fim3D.computeFM();
345         end = system_clock::now();
346         time_elapsed = duration_cast<milliseconds>(end-start).count();
347 cout << "\tElapsed FIM time: " << time_elapsed << " ms" << endl;
348 GridWriter::saveGridValues("test_fim3D.txt", grid_fim3D);
```

Finalmente se crea un mapa en tres dimensiones con ciertas medidas especificadas en **(270)-(272)** con un punto de inicio **(274).** Se calculan los tiempos de llegada para todas las celdas, guardando sus valores en archivos '.txt'.

```
[INFO] Testing algorithms in 3D
        Elapsed FMM time: 871 ms
        Elapsed FMM_Dary time: 844 ms
        Elapsed SFMM time: 845 ms
        Elapsed GMM time: 664 ms
        Elapsed UFMM time: 4513 ms
        Elapsed FIM time: 299 ms
```

b. Ejemplos

Mediante el proceso desarrollado en el apéncide *a. "Tutorial de uso de código"*, se va a resolver un pequeño laberinto como el que se puede ver en la Figura b.1, a modo de ejemplo. Siendo el punto verde donde se inicia la onda y el punto rojo la meta.

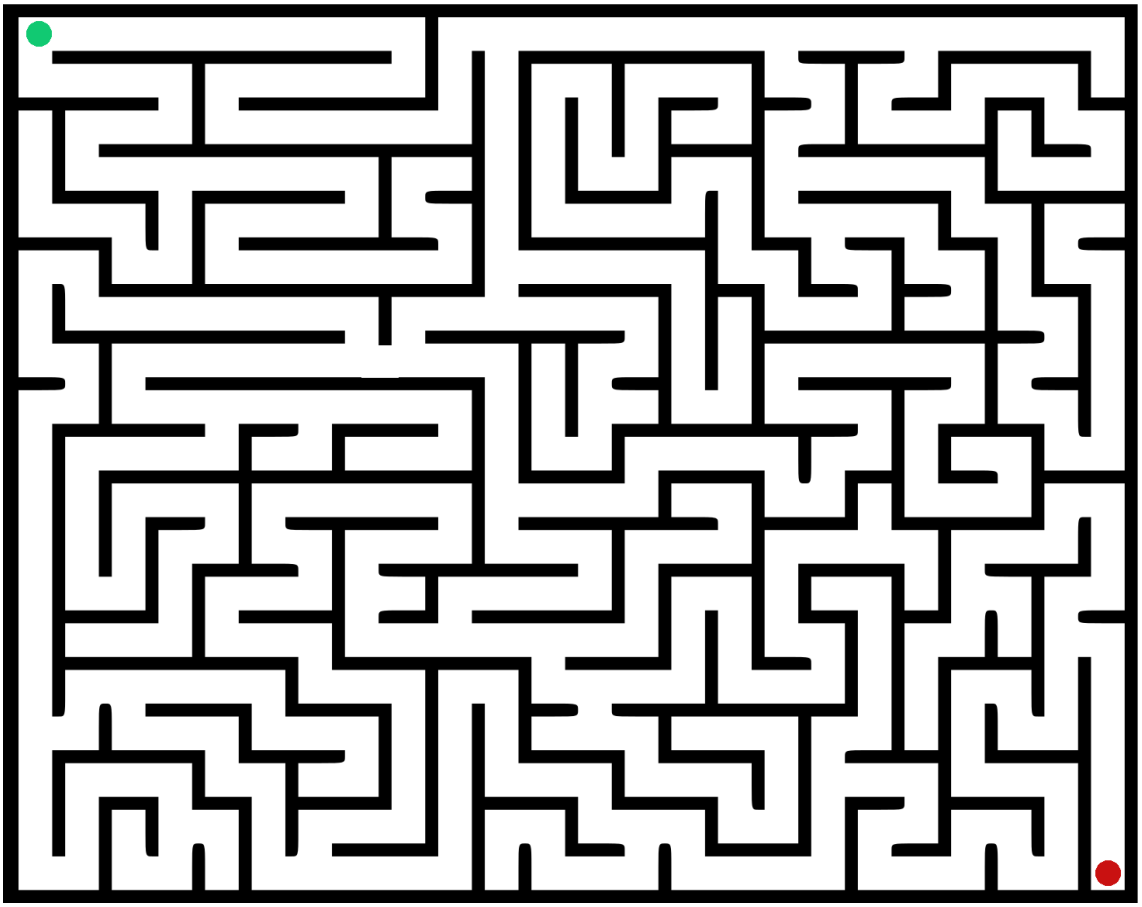


Figura b.1: Laberinto usado como ejemplo, mapa con velocidades constantes.

El laberinto usado tiene velocidad de propagación de la onda constante siendo esta igual a la velocidad máxima (F=1).
El resultado de la expansión de una onda desde el punto verde a través de todo el mapa se puede ver en la Figura b.2. Comparando el resultado del mapa con los milisegundos, se puede comprobar que la salida se encuentra en el punto más alejado del mapa ya que es donde más tiempo le toma a la onda llegar.
La Figura b.3 es la representación del camino más corto espacial y temporalmente hablando hasta la meta.

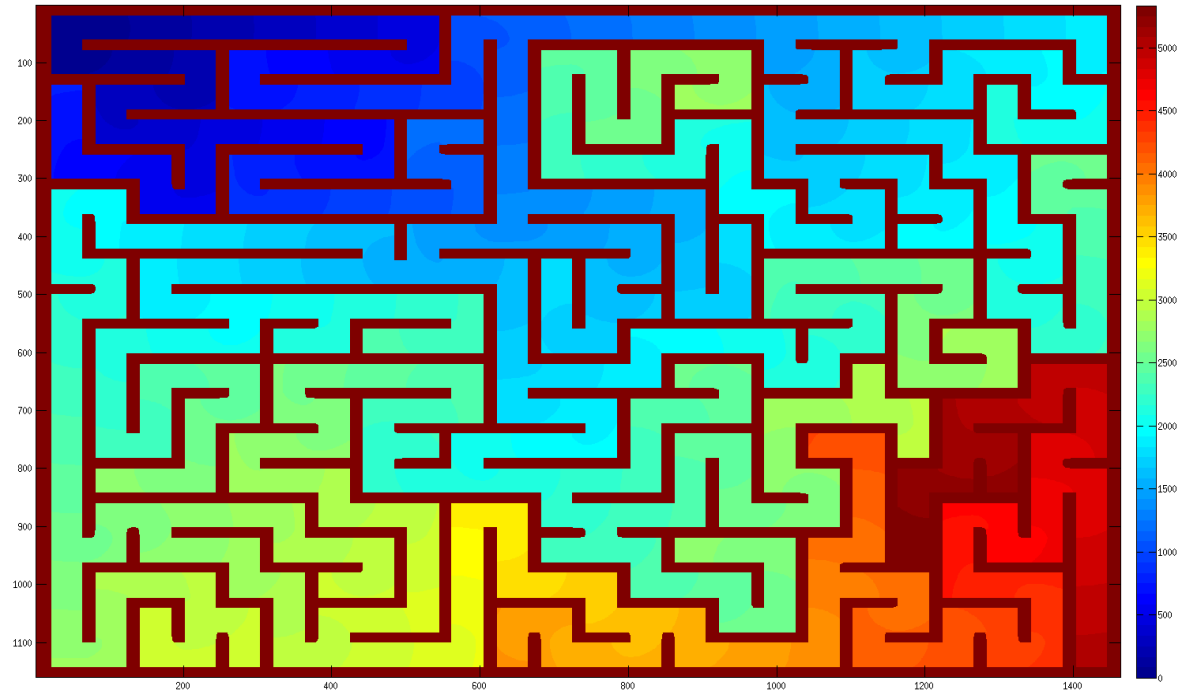


Figura b.2: Resultado de generar una onda en el punto de inicio y su posterior expansión por todas las calles del laberinto. Resultado en milisegundos.

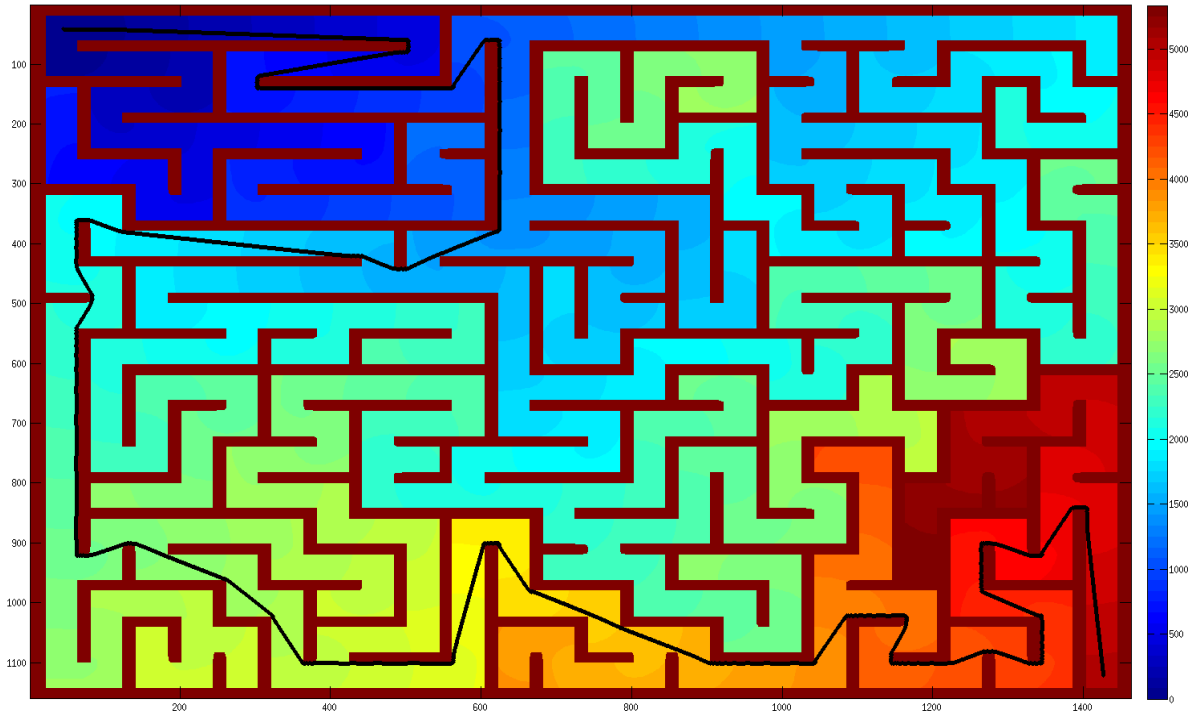


Figura b.3: Representación del camino más corto, temporal y espacialmente, desde el inicio hasta la meta del laberinto.

La otra aplicación del tutorial es usar diferentes velocidades de propagación de onda. Para probar este función, se crea el laberinto de la Figura b.4. Este laberinto tiene el mismo trazado que el anterior, pero tiene una zona oscura en la que la velocidad de propagación se ha reducido a un 10% de la velocidad máxima.

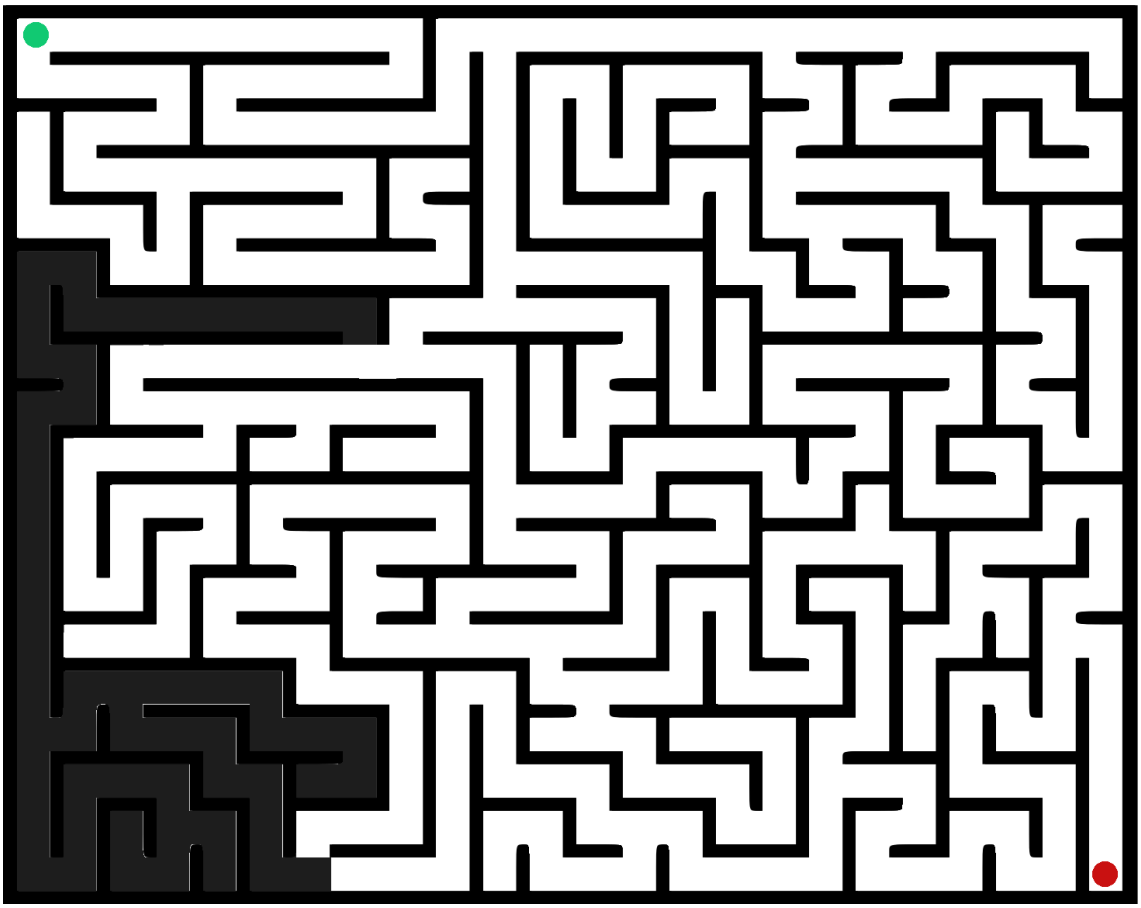


Figura b.4: Laberinto usado como ejemplo, mapa con velocidades no constantes. La zona más oscura representa un 10% de velocidad respecto a la zona blanca.

Los resultados de la propagación de la onda desde el punto de inicio del laberinto, se pueden observar en la Figura b.5. Se puede comprobar un aumento del tiempo considerable en la zona oscura.
La Figura b.6 es la representación del camino más corto temporalmente, pero no espacialmente. Al cambiar la velocidad de propagación del camino más corto, los resultados temporales empeoran, esto provoca que se tarde menos en llegar por otros caminos más largos espacialmente, causando un cambio de trazado con respecto al anterior laberinto.

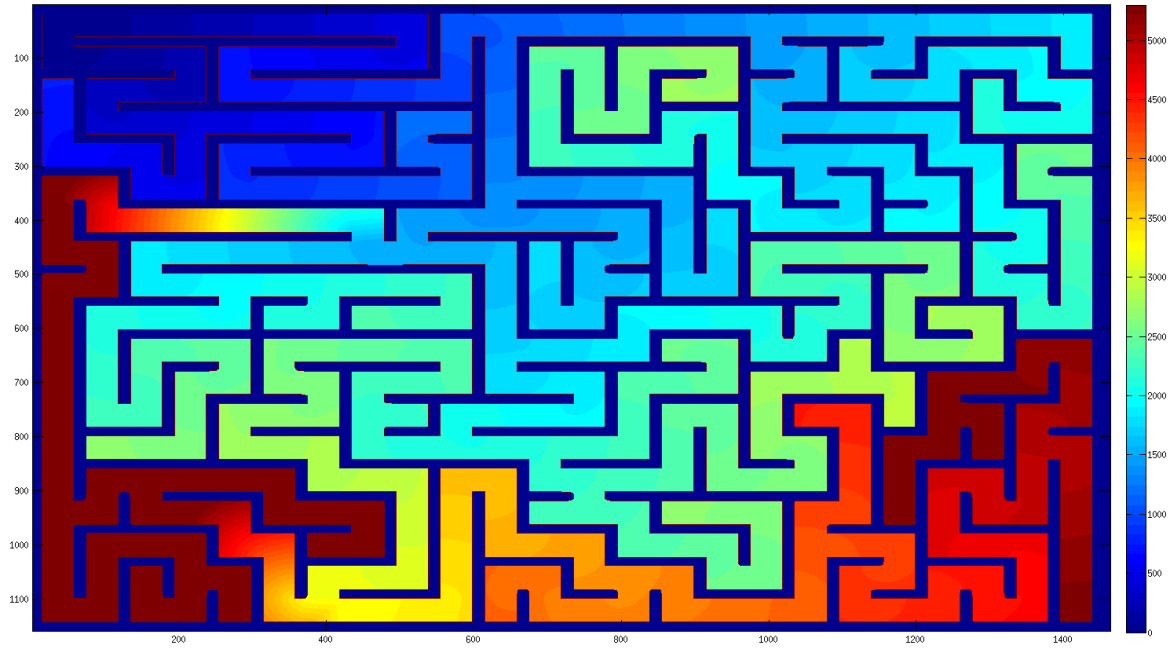


Figura b.5: Resultado de generar una onda en el punto de inicio y su posterior expansión por todas las calles del laberinto. Resultado en milisegundos.

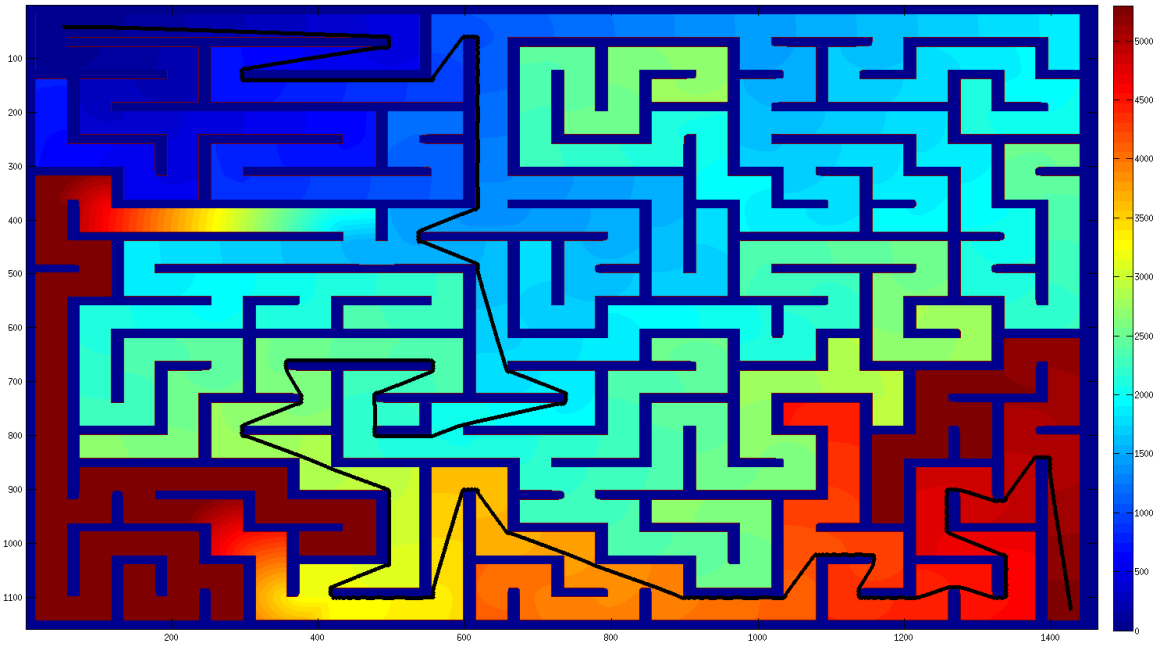


Figura b.6: Representación del camino más corto, temporal pero no espacialmente, desde el inicio hasta la meta del laberinto.

c. Resultados números

En este apéndice se proporcionan todos los datos que se han usado para representar las gráficas de los experimentos del capítulo 4.

- **Experimento Nº1**

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM	FIM
10x10	100	0	0	0	0	0	0
100x100	10.000	3,8	2	2	3,4	2	1
250x250	62.500	25	20	16	20	12,8	11
375x375	140.625	59,4	49,8	39,7	45,9	31,3	25,7
500x500	250.000	109,5	94,6	75,4	83,8	58,3	49
750x750	562.500	259,4	233,6	183,3	189,3	133	114,7
1000x1000	1.000.000	462,2	450,7	346,7	337,6	241	203
1250x1250	1.562.500	733,7	749,7	571,2	527,5	382	321,5
1500x1500	2.250.000	1076	1137,4	858,9	761,6	554	464,4
1750x1750	3.062.500	1473,7	1623,2	1219	1042	764	630,3
2000x2000	4.000.000	1968,1	2205,7	1659	1368	1004	814,6
2250x2250	5.062.500	2526,3	2886,5	2174	1728	1275	1041
2500x2500	6.250.000	3164,7	3669,3	2768	2142	1583	1290
2750x2750	7.562.500	3878,1	4555,6	3431	2602	1927	1569
3000x3000	9.000.000	4651,5	5558,5	4178	3088	2294	1861
3250x3250	10.562.500	5485,2	6663,7	5011	3664	2706	2189
3500x3500	12.250.000	6419,3	7883,9	5929	4241	3146	2551
3750x3750	14.062.500	7444,5	9220	6928	4863	3631	2931
4000x4000	16.000.000	8558,3	10671,8	8032	5491	4142	3330
4250x4250	18.062.500	9763,6	12237,8	9196	6233	4685	3752
4500x4500	20.250.000	11050	13936,8	10480	7006	5276	4223
4750x4750	22.562.500	12433	15773,4	11824	7819	5908	4698
5000x5000	25.000.000	13917	17737,5	13261	8667	6578	5438
5250x5250	27.562.500	15488	19853,5	14799	9457	7288	5791
5500x5500	30.250.000	17164	21962,7	16432	10533	8049	6346
5750x5750	33.062.500	18969	24334,2	18138	11552	8846	6956

Tabla c.1: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº1. Datos recogidos en mapas de 10x10 hasta 5750x575 celdas.

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM	FIM
10x10	100	0	0	0	0	0	0
50x50	2.500	1	0	0	1	0	0
100x100	10.000	3	2	2	3,1	2	1
150x150	22.500	8	6,3	5	7	4	3,4
200x200	40.000	15	12	10	12,6	8	7
250x250	62.500	24	20	16	20	13	11
300x300	90.000	35,7	30	24	29	19	16
350x350	122.500	50,3	42	34	40,5	26,8	22,3
400x400	160.000	66,1	57,5	45,9	53,1	36,1	29,1
450x450	202.500	84,6	74,5	59	68	46,9	39
500x500	250.000	107,6	94,7	75,2	84,4	58,1	49
550x550	302.500	130,2	116,1	93	102,1	70,9	60,2
600x600	360.000	157,9	141,2	114	122,1	84,3	72,5
650x650	422.500	187,7	169,1	134,4	143,6	99,9	85,8
700x700	490.000	217,6	199,7	158,4	166,7	115,8	99,2
750x750	562.500	251	233,2	183,4	190,8	133,1	113,4
800x800	640.000	282,9	270,1	212	214,9	152,4	129,3
850x850	722.500	320,6	310,2	241,8	244,9	173	146
900x900	810.000	362,2	354	273,9	274,5	194,3	165,4
950x950	902.500	406,1	400,1	309,9	307,2	216,9	184,1
1000x1000	1000.000	453,6	449,6	350,5	339,7	241,2	199,3
1050x1050	1.102.500	503,8	503,2	387,3	374,6	266	225,5
1100x1100	1.210.000	558	561,3	430,8	410,5	292,9	244,4
1150x1150	1.322.500	608,3	617,5	474,8	439	320,2	266
1200x1200	1.440.000	666,8	680	523,1	487,6	350,3	296,8
1250x1250	1.562.500	728,4	746,6	572,1	531,9	381,3	320,2

Tabla c.2: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº1. Datos recogidos en mapas de 10x10 hasta 1250x1250 celdas.

- **Experimento Nº2**

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM	FIM
10x10	100	0	0	0	0	0	0
100x100	10.000	3,7	2	2	3	2	1,7
250x250	62.500	24	18	14	19,8	13,8	10,3
375x375	140.625	57	43,8	34,6	45,7	32,7	25,4
500x500	250.000	105,1	81,7	65,2	82,4	59,7	48,5
750x750	562.500	247,8	194,9	155,6	184,1	132,5	111,3
1000x1000	1.000.000	463,1	372,1	289	324,9	236,8	200,6
1250x1250	1.562.500	735,3	614,2	473,7	507,6	371,7	311,9
1500x1500	2.250.000	1089,4	934,5	707,6	728,1	542,6	449,6
1750x1750	3.062.500	1524,4	1337	1001,2	996,6	742,9	614,9
2000x2000	4.000.000	2037,8	1815,7	1359,7	1311	976,1	808,4
2250x2250	5.062.500	2633,5	2377,8	1774,9	1659,5	1237,4	1022,6
2500x2500	6.250.000	3303,8	3029,2	2254,4	2060	1540,1	1271,1
2750x2750	7.562.500	4061,3	3771,4	2805,9	2507,3	1872,2	1545,3
3000x3000	9.000.000	4903,8	4597,1	3405,4	2990,1	2229,9	1811,4
3250x3250	10.562.500	5840,6	5528,3	4086,6	3519,3	2631	2152,1
3500x3500	12.250.000	6861,8	6551,9	4843,8	4089	3054,3	2516,6
3750x3750	14.062.500	7971,2	7679,5	5671,6	4705,8	3531,2	2894,5
4000x4000	16.000.000	9169,7	8893,2	6562,8	5374,5	4028,8	3168
4250x4250	18.062.500	10470,7	10218,8	7543,5	6047,7	4553	3573,7
4500x4500	20.250.000	11842,3	11633,6	8610,9	6771,3	5115,7	4165,3
4750x4750	22.562.500	13308,1	13174,9	9734,5	7566,6	5719	4642,5
5000x5000	25.000.000	14837,6	14777,4	10877,1	8402,8	6356,6	4626,2
5250x5250	27.562.500	16478,1	16474,9	12143,4	9283,2	6989,5	5196,5
5500x5500	30.250.000	18285,7	18374,9	13571,4	10170,7	7733,3	6267,6
5750x5750	33.062.500	20143,2	20337,2	15016,5	11116,6	8465,7	6827

Tabla c.3: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº2. Datos recogidos en mapas de 10x10 hasta 5750x5750 celdas.

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM	FIM
10x10	100	0	0	0	0	0	0
50x50	2.500	0,7	0	0	0	0	0
100x100	10.000	3	2	2	3	2	1,6
150x150	22.500	8	6	4,6	7	5	3
200x200	40.000	15	11,2	8,9	12,1	9	6,5
250x250	62.500	24	18,7	14	20	13,9	10,4
300x300	90.000	35,3	27,9	21,5	28,9	20,2	15,5
350x350	122.500	49,9	37,9	30,8	39,8	28	21,8
400x400	160.000	67,7	51,8	41,9	52	37,7	29,2
450x450	202.500	84,3	65,5	51,6	65,4	48	38,6
500x500	250.000	106,4	83,6	65,2	82,5	59,9	48,8
550x550	302.500	130,9	101,8	80,1	99,6	71,9	59,4
600x600	360.000	156,2	121,8	97,1	118,8	84,1	71
650x650	422.500	184,2	146,1	114,6	138,8	100	84
700x700	490.000	214	172,3	134,3	161,4	115,3	97,3
750x750	562.500	246,9	197,7	154,7	183,9	132,4	112,1
800x800	640.000	283,5	235,5	182,7	209,1	151,4	127,6
850x850	722.500	321,3	263,6	201,9	236	170,6	143,8
900x900	810.000	367,2	295	224,7	261,9	191,8	144,9
950x950	902.500	414,3	336,7	256,9	294,3	213,7	171,4
1000x1000	1000.000	460,1	376,7	287,8	325,1	236,9	200,3
1050x1050	1.102.500	513,5	411,6	317	357,5	260,9	198,5
1100x1100	1.210.000	564,3	462,4	354,2	390,7	286,9	241,1
1150x1150	1.322.500	624,6	519,6	396,4	426,9	313,9	263,9
1200x1200	1.440.000	684,3	582,7	439,6	466,9	342,6	282
1250x1250	1.562.500	741,3	622,1	474	505,8	373,1	312,2

Tabla c.4: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº2. Datos recogidos en mapas de 10x10 hasta 1250x1250 celdas.

- **Experimento Nº3**

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM	FIM
4000x4000	16.000.000	9570	9646,3	7183,9	5461,9	4095,2	3401,7
8000x2000	16.000.000	8579,1	8114,9	6066,3	5389,1	3928,1	3414,6
16000x1000	16.000.000	7440,7	6678,4	5248,2	5294,7	3761	3361,7
40000x400	16.000.000	6453,7	5563,6	4535,6	5233,1	3751,7	3273,7
80000x200	16.000.000	5959,1	5152,1	4302,1	5199,7	3848	3258,1
160000x100	16.000.000	5515,4	4894	4077	5292,2	4083,6	3255,6

Tabla c.5: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº3.

- **Experimento Nº5**

Franjas	90%	10%
FMM	467,3	487,5
FMM_Dary	392	382,4
SFMM	305,8	308,6
GMM	340	435
UFMM	244,7	245,4
FIM	196,4	195,9

Tabla c.6: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos contrastes en la velocidad de expansión de la onda, en el experimento Nº5.

- **Experimento Nº9**

Tamaño	Nº Celdas	FMM	FMM_Dary	SFMM	GMM	UFMM (1000 D)	UFMM (5000 D)	FIM
10x10x10	1.000	1	0	0	0	0	3	0
30x30x30	27.000	17,6	14	14	16	10,2	16,5	6
50x50x50	125.000	89,2	79,2	79,6	77	54,2	61,5	32
75x75x75	421.875	329	315,2	316	264,2	198,4	206,5	114,4
100x100x100	1.000.000	861,4	868	847,6	631,6	510,6	505	275
125x125x125	1.953.125	1918,4	2041,2	1839,2	1291,6	1144,4	1100	571
150x150x150	3.375.000	3742,4	4211,6	3601,2	2329	2294,4	2140,5	1103
175x175x175	5.359.375	6511,4	7761	6438	3934,6	4118,6	3758,5	1885,4
200x200x200	8.000.000	10594,4	13032,6	10615,2	6279,6	6668,4	5915,5	3001
225x225x225	11.390.625	16041	20304	16332,2	9228	10196,2	8905,5	4436,2
250x250x250	15.625.000	23702,4	30332,4	24052,6	13027,6	15015,2	12744,5	6284,6
300x300x300	27.000.000	44854,4	60329,2	47128,8	23894,2	29290,8	23506,5	11405,6

Tabla c.7: Tiempo de cálculo, en milisegundos, de cada uno de los algoritmos en distintos tamaños en el experimento Nº9. Datos recogidos en mapas de 10x10x10 hasta 300x300x300 celdas.

BIBLIOGRAFÍA

[1] Zhu, J., & Sethian, J. A. (1992). Projection Methods Coupled to Level Set Interface Techniques. *Journal of Computational Physics*, *102*, 128–138.

[2] Yu, J-D., Sakai, S., & Sethian, J.A. (2003). A Coupled Level Set Projection Method Applied to Ink Jet Simulation. *Interferences and Free Boundaries, 193*(1), 275–305.

[3] Hogea, C.S., Murray, B.T., & Sethian, J.A. (2005). Computational Modeling of Solid Tumor Evolution via a General Cartesian Mesh/level set method. *Fluid Dynamics \& Materials Processing Vol., 1, 2*.

[4] Hogea, C.S., Murray, B.T., & Sethian, J.A. (2005). Simulating Complex Tumor Dynamics from Avascular to Vascular Growth using a General Level Set Method. *Mathematical Biology, 51*(1).

[5] Gómez, J. V., Lumbier, A., Garrido, S., & Moreno, L. (2012). Planning Robot Formations with Fast Marching Square including Uncertainty Conditions. *Robotics and Autonomous Systems*(Accepted).

[6] Cameron, M. K., Fomel, S. B., & Sethian, J. A. (2006). Seismic Velocity Estimation using Time Migration Velocities. *Inverse Problems, submitted for publication.*

[7] Malladi, R., & Sethian, J.A. (1995*).* Image Processing via Level Set Curvature Flow. *Proceedings of the National Academy of Sciences, 92(15),* 7046–7050.

[8] Malladi, R., Sethian, J.A. & Vemuri, B. (1993). A Topology Independent Shape Modeling Scheme. *Proceedings of SPIE Conference on Geometric Methods in Computer Vision II, 2031,* 246–258.

[9] Sethian, J.A . Segmentation in Medical Imaging. [ONLINE] Disponible en: http://math.berkeley.edu/~sethian/2006/Applications/Medical_Imaging/artery.html. [Last Accessed 01/09/2014].

[10] Osher, S., & Sethian, J. A. (1988). Fronts Propagating with Curvature dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics, 79*(1), 12–49.

[11] Kim, S., & Folie, D. The Group Marching Method: An O(N) Level set Eikonal Solver.

[12] Yatziv, L., Bartesaghi, A., & Sapiro, G. (2005). O(N) Implementation of the Fast Marching Algorithm. *Journal of Computational Physics, 212*(2), 393–399.

[13] Rosenfeld, A., & Pfaltz, J. L. (1996). Sequential operations in digital picture processing. *Journal of the ACM, 13*(4), 471–494.

[14] Jones M.W. (2006). 3d distance fields: A survey of Techniques and Applications. *Visualization and Computer Graphics, 12*(4).

[15] Borgefor, G. (1984). Distance Transformations in Arbitrary Dimensions. *Computer Vision, Graphics and Image Processing, 27*(3), 321–345.

[16] Satherley, R., & Jones, M. W. (2001). Vector-City Vector Distance Transform. *Computer Vision and Image Understanding, 82*(3), 238–254.

[17] Kao, C. Osher, S. & Qian , J.(2004). Lax-friedrichs sweeping scheme for static hamilton-jacobi equations. *Journal of Computational Physics, 196*(1), 367–391.

[18] Valero, A., Gómez, J. V., Garrido, S., & Moreno, L. (2013). Fast Marching Method for Safer, More Efficient Mobile Robot Trajectories. *IEEE Robotics & automation magazine.*

[19] dws. (2003). An informal introduction to O(N) notation. [ONLINE] Disponible en: http://www.perlmonks.org/?node_id=227909. [Last Accessed 23/08/2014].

[20] Astrachan, O. (2003). Bubble Sort: An Archaeological Algorithmic Analysis. *SIGCSE.*

[21] -, Big(O) cheat sheet. [ONLINE] Disponible en: http://bigocheatsheet.com/. [Last Accessed 23/08/2014].

[22] Boost C++ Libraries(2010). Data Structure. [ONLINE] Disponible en: http://www.boost.org/doc/libs/1_55_0/doc/html/heap/data_structures.html. [LastAccessed 23/08/2014].

[23] Michael, L. (1987). Fibonacci Heaps and their uses in improved network. *Journal of the ACM, 34*(3), 596–615.

[24] Edelsbrunner , H. (2008). Design and analysis of algorithms. *CPS 230 Chapter 3: Fibonacci Heaps*, 38–40.

[25] Louis R.,Jesse.(2013). Data structure: pairing heap. [ONLINE] Available at: http://n-e-r-v-o-u-s.com/blog/?p=3552. [Last Accessed 25/08/2014].

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, & Clifford Stein. (2001). Introduction to Algorithms, Second Edition. *MIT Press and McGraw-Hill, Chapter 12: Binary Search Trees*, 286–299.

[27] Tarjan, R. E. (1983). 3.2. d-heaps. Data Structures and Network Algorithms. *CBMS-NSF Regional Conference Series in Applied Mathematics, 44, 34–38*.

[28]Siangsukone, T., Aswakul, C., & Wuttisittikulkij, L. (2014). Study of Optimised bucket widths in Calendar Queue for Discrete Event Simulator. *Thailand's Electrical Engineering Conference EECON-26*.

[29] Jeong,W., & Whitaker, R.(2007*).* O(N) Implementation of the Fast Marching Algorithm. *SIAM Journal of Scientific Computing, 30*(5), 2512–2534.

[30] Stewart, J. (2005). An Investigation of SIMD instruction sets. *University of Ballarat School of information Technology and Mathematical Sciences.*