David Pérez Sampedro 48774775D

# Análisis y Diseño de Algoritmos

## Práctica Final

# Índice

# 1. Estructura de datos

## 1.1. Nodos

La estructura de los nodos en la práctica está construida utilizando únicamente las coordenadas de estos en la matriz, creada a partir del mapa de pesos. Para ello utilizo el tipo de dato *pair<T, T>*; sustituyendo la clase genérica por el tipo de dato *int*, donde la primera componente de la pareja es la fila en la que se sitúa, y la segunda, la columna.
La *Figura 1* muestra un ejemplo de cómo se inicializaría.

```
pair<int, int> inicial( x: 0,  y: 0);   //nodo inicial (primera casilla)
```

*Figura 1*

## 1.2. Lista de nodos vivos

Para poder trabajar con los valores que acompañan a los nodos correctamente, he creado un *struct* llamado *estructura_cola*. Esta estructura guarda el nodo (aquí llamado casilla), un entero con el valor de la solución del propio nodo, y un vector de nodos, que son los que se han recorrido hasta llegar a dicho nodo, llamado *casillas_usadas*; por último, se guarda el valor da la cota optimista, para poder reutilizarla cuando se va a expandir el nodo.

```
struct estructura_cola {
    pair<int, int> casilla;
    int valor;
    vector<pair<int, int>> casillas_usadas;
    int optimista;
};
```

*Figura 2*

La lista que he utilizado para almacenar los nodos vivos se trata de una *priority_queue<T, Container, Compare>*; sustituyendo la clase genérica por la estructura de *estructura_cola*, el contenedor por un vector de *estructura_cola*, y la función de comparación por *Comparar*, una clase propia con un *operator()* para ordenar la cola de prioridad.
Para ordenar la cola de prioridad, el operador *operator()* ordena el almacén de forma ascendente, utilizando la solución actual del nodo como parámetro a comprar, y así operar antes con los nodos que tengan una mejor solución hasta el momento.

```
class Comparar {
public:
    //ordena de menor a mayor según el pesos total del camino hasta la casilla
    bool operator()(const estructura_cola &abajo, const estructura_cola &arriba) {
        return abajo.valor > arriba.valor;
    }
};
```

*Figura 3*

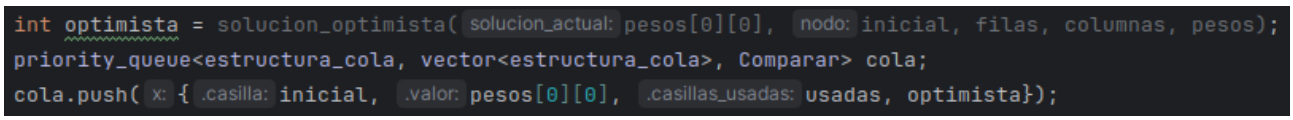



*Figura 4*

Otros criterios que se ha probado para la ordenación de la cola de prioridad en el *operator()* de *Comparar* han sido:
- La utilización de la cota pesimista y de la optimista, sin embargo, esta forma de ordenar la cola de prioridad no daba la solución correcta del problema.
- Otros criterios como una ordenación descendente, o por el nodo en vez del valor de su solución, pero estos criterios, al igual que las cotas optimistas y pesimistas, dan resultados erróneos, tanto si se ordena de forma ascendente o descendente.

# 2. Mecanismos de poda

## 2.1. Poda de nodos no factibles

En mi práctica, un nodo es factible cuando no ha sido utilizado todavía, por lo tanto, cuando ya se encuentra en la lista *casillas_usadas* asociada al nodo que se está intentando expandir, se descarta y se continua con el siguiente. Sin embargo, no realizo podas de estos nodos, simplemente los descarto de las posibilidades de expansión de su nodo padre, y continúo probando con el resto de hijos.

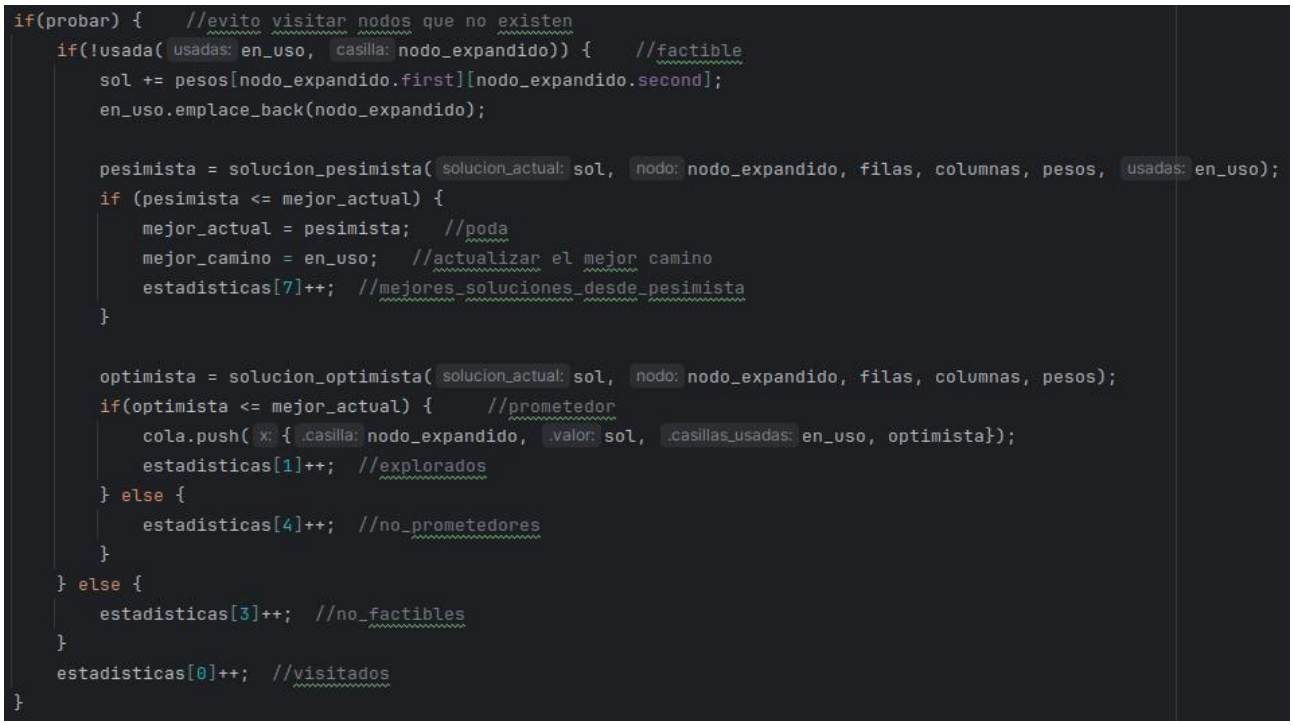



*Figura 5*

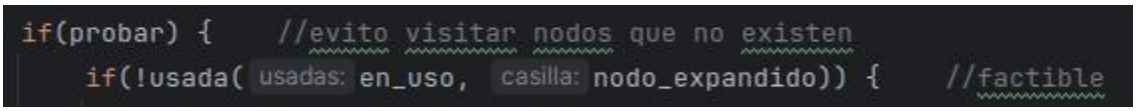



*Figura 6*

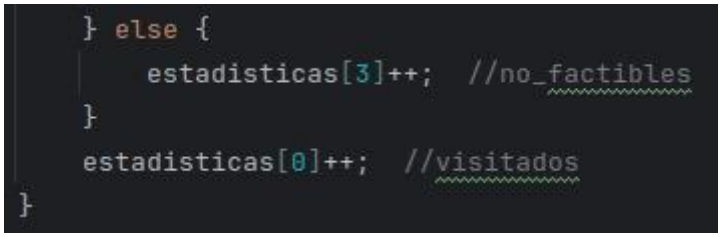



*Figura 7*

## 2.1. Poda de nodos no prometedores

Para descartar a los nodos no prometedores utilizo la cota optimista, que viene dada por la función *solución_optimista(…)*. Los nodos expandidos que son factibles son pasados por esa función, que, en caso de devolver true, los añade a la cola de prioridad para ser expandidos más adelante, pero en caso de devolver false, se descartan y se sigue con el resto de nodos de expansión del nodo actual de ese momento.

No obstante, al igual que con los nodos no factibles, no existe una poda propiamente dicha.

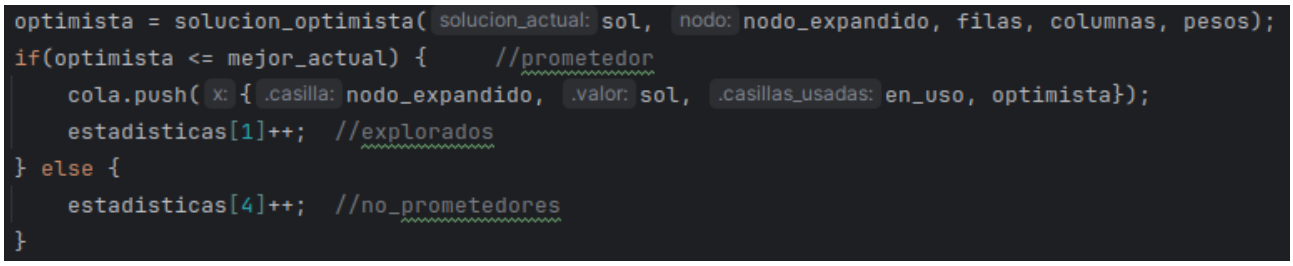



*Figura 8*

# 3. Cotas pesimistas y optimistas

Las cotas pesimistas y optimistas se calculan dentro de una función cada una, siendo la de la cota pesimista, *solucion_pesimista(…)*; y la de la cota optimista, *solucion_optimista(…)*. Ambas devuelven un entero con la solución de la función.

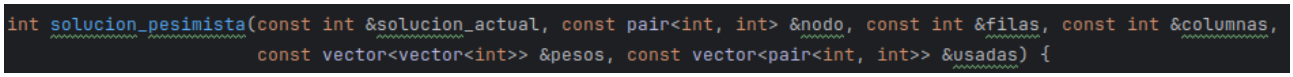



*Figura 9*

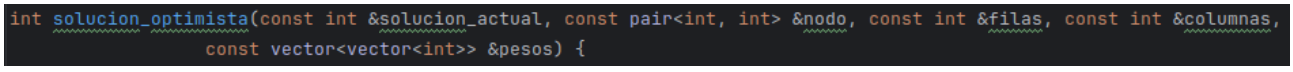



*Figura 10*

## 3.1. Cota pesimista inicial (inicialización)

La forma de calcular la cota pesimista del nodo inicial y la del resto de nodos es la misma, en este caso, el nodo sería la posición (0, 0), la solución del nodo es el valor de la posición (0, 0) de la matriz de posiciones, y dentro de la lista de nodos usados sólo se encuentra él mismo. El valor de la cota optimista que se guarda es el que devuelve la función *solución_optimista(…)* de ese nodo. Como la función es la misma, la explicaré en el siguiente apartado.

```
pair<int, int> inicial( x: 0,  y: 0);   //nodo inicial (primera casilla)
usadas.emplace_back(inicial);   //añadr el nodo inicial a usadas
int mejor_actual = solucion_pesimista( solucion_actual: pesos[0][0],  nodo: inicial, filas, columnas, pesos, usadas);
```

*Figura 11*

## 3.2. Cota pesimista del resto de nodos

Para la cota pesimista utilizo la función solucion_pesimista(...), que devuelve un entero con una solución voraz iterativa del problema. La función comienza declarando las variables que se van a utilizar.

```
vector<pair<int, int>> en_uso(usadas);  //copia para añadir casillas usadas temporalmente
int sol = solucion_actual;  //copia paraa  sobreecribir la solución
int i = nodo.first;
int j = nodo.second;
int S1, S2, S3, S4, S5, maximo;
vector<int> num;    //vector donde guardar todas las opciones de expansión
```

*Figura 12*

Le sigue el cuerpo de la función, un bucle *while* que termina cuando se alcanza la última posición de la matriz. Dentro de ella, primero se inicializan todas las posibilidades de expansión, al número más pequeño posible. Después, si la expansión es posible (el nodo es factible), se le asigna un nuevo valor, que será la solución del nodo anterior más su propio peso.

Tomar en cuenta que la lista de movimientos posibles se compone de las 8 posiciones a las que se puede acceder desde un nodo, pero sin contar los movimientos hacia arriba (diagonal arriba-derecha, arriba, diagonal arriba-izquierda).

Una vez comprobados todos los movimientos posibles, se selecciona el máximo de todos ellos, y se comprueba el valor del máximo, si es -inf (*numeric_limits<int>::min()*), entonces se devuelve +inf (*numeric_limits<int>::max()*); esto se hace porque así, si no se ha encontrado una solución factible, se devuelve un número muy grande para que nunca sea menor o igual que el valor de la mejor solución de ese momento.

El bucle while termina añadiendo el nodo seleccionado a usados para no caer en bucles, e igualando la solución actual a ese valor mínimo.

```
while(i + 1 < filas || j + 1 < columnas) {
    S1 = S2 = S3 = S4 = S5 = numeric_limits<int>::min();

    if(j + 1 < columnas && !usada( usadas: en_uso,  casilla: make_pair( &: i,  y: j + 1))) {     //derecha
        S1 = sol + pesos[i][j + 1];
    }

    if(i + 1 < filas && j + 1 < columnas && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  y: j + 1))) {     //diagonal abajo derecha
        S2 = sol + pesos[i + 1][j + 1];
    }

    if(i + 1 < filas && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  &: j))) {     //abajo
        S3 = sol + pesos[i + 1][j];
    }

    if(i + 1 < filas && j - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  y: j - 1))) {     //diagonal abajo izquierda
        S4 = sol + pesos[i + 1][j - 1];
    }

    if(j - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( &: i,  y: j - 1))) {     //izquierda
        S5 = sol + pesos[i][j - 1];
    }
```

*Figura 13*

```
maximo = max(max(max(S1, S2), max(S3, S4)), S5);
```

*Figura 14*

```
        if(maximo == numeric_limits<int>::min()) {
            return numeric_limits<int>::max();
        }

        if(maximo == S1) {
            j++;
        } else if(maximo == S2) {
            i++;
            j++;
        } else if(maximo == S3) {
            i++;
        } else if(maximo == S4) {
            i++;
            j--;
        } else if(maximo == S5) {
            j--;
        }

        en_uso.emplace_back( a: i,  b: j);
        sol = maximo;
    }

    return sol;
}
```

*Figura 15*

## 3.3. Cota optimista

La cota optimista se calcula dentro de la función *solucion_optimista(…)*, que es una solución voraz iterativa con menos restricciones que la cota pesimista.
La función empieza con la declaración de las variables que se van a usar.

```
int sol = solucion_actual;
int i = nodo.first;
int j = nodo.second;
int S1, S2, S3, minimo;
```

*Figura 16*

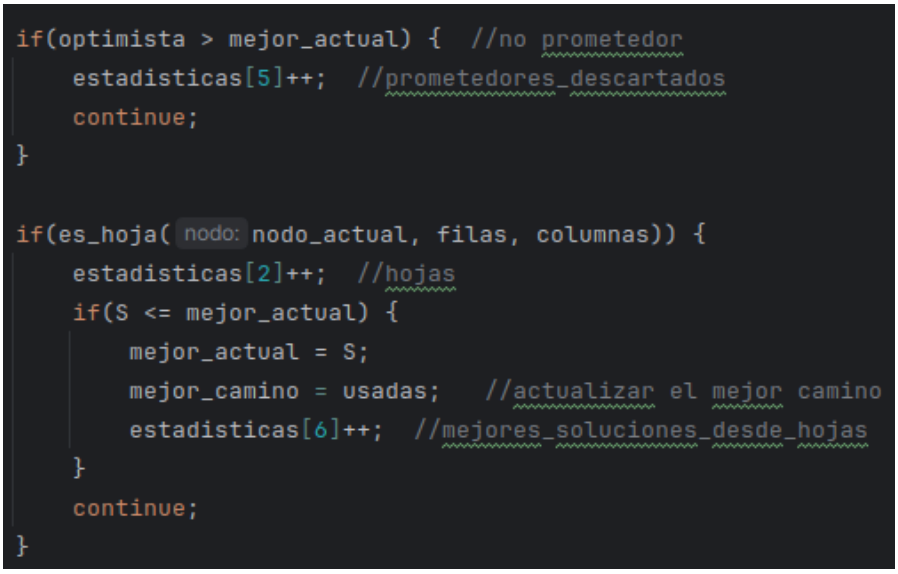
El cuerpo de la función consiste en bucle *while* que termina cuando se alcanza el último nodo de la matriz, dentro de esta matriz se realiza la solución voraz, que consta de 3 posibles movimientos: derecha, abajo y diagonal derecha-abajo. Se selecciona el mínimo de estos 3 y se actualiza la solución añadiendo su valor.

```
    while(i + 1 < filas || j + 1 < columnas) {
        if(i + 1 == filas) {
            sol += pesos[i][j + 1];  //Movimiento hacia la derecha
            j++;
        } else if(j + 1 == columnas) {
            sol += pesos[i + 1][j];  //Movimiento hacia abajo
            i++;
        } else {
            S1 = sol + pesos[i + 1][j + 1];  //Movimiento diagonal
            S2 = sol + pesos[i][j + 1];  //Movimiento hacia la derecha
            S3 = sol + pesos[i + 1][j];  //Movimiento hacia abajo

            minimo = min(S1, min(S2, S3));

            if(minimo == S1) {
                sol = S1;
                i++;
                j++;
            } else if (minimo == S2) {
                sol = S2;
                j++;
            } else {
                sol = S3;
                i++;
            }
        }
    }

    return sol;
}
```

*Figura 17*

# 4. Otros medios empleados para acelerar la búsqueda

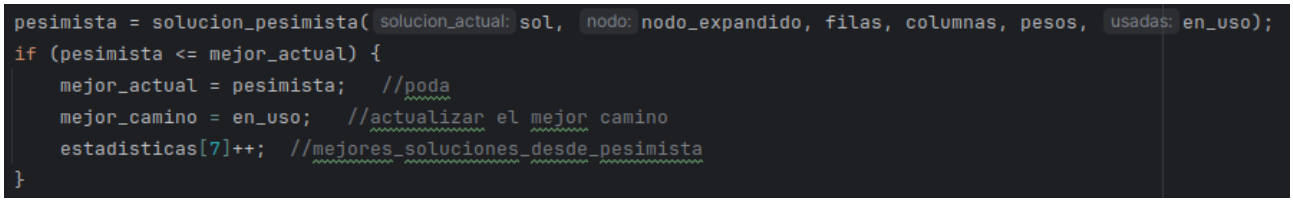
Con la finalidad de acelerar la búsqueda, cada vez que se tiene la intención de expandir un nodo, se comprueba nuevamente si sigue siendo prometedor, porque en la fase de exploración donde se consideró a este nodo prometedor, pudo actualizarse la mejor solución hasta el momento y que este nodo ya no lo sea; en dicho caso se ignora y directamente no se expande. Lo mismo pasa cuando el nodo es una hoja (filas – 1, columnas – 1).



```
if(optimista > mejor_actual) {  //no prometedor
    estadisticas[5]++;  //prometedores_descartados
    continue;
}

if(es_hoja( nodo: nodo_actual, filas, columnas)) {
    estadisticas[2]++;  //hojas
    if(S <= mejor_actual) {
        mejor_actual = S;
        mejor_camino = usadas;   //actualizar el mejor camino
        estadisticas[6]++;  //mejores_soluciones_desde_hojas
    }
    continue;
}
```

*Figura 18*

Además, al expandir un nodo y corroborar que es factibles, también se comprueba si su solución pesimista puede ser mejor o igual que la mejor solución actual, y en caso de serlo se asigna como mejor solución, podando posibles nodos que ya hubieran sido considerados prometedores.



```
pesimista = solucion_pesimista( solucion_actual: sol,  nodo: nodo_expandido, filas, columnas, pesos,  usadas: en_uso);
if (pesimista <= mejor_actual) {
    mejor_actual = pesimista;   //poda
    mejor_camino = en_uso;   //actualizar el mejor camino
    estadisticas[7]++;  //mejores_soluciones_desde_pesimista
}
```

*Figura 19*

Para acelerar la búsqueda, he probado distintas formas de hallar el siguiente movimiento en la cota pesimista, en la cual probé varias versiones

1. Inicializando todos los movimientos a -inf y metiéndolos en un vector ordenado ascendentemente, para después sacar el número intermedio de este (la cuarta posición debido a que son pares).
2. Igual que en la primera versión, pero en esta versión los movimientos que se desplazan a una posición superior (diagonal arriba-derecha, arriba, diagonal arriba-izquierda) no han sido recalculados y mantienen siempre el valor de -inf, pero se mantienen y añaden al vector.
3. Inicializando todos los movimientos a -inf y seleccionando el mayor de todos ellos.
4. Igual que en la tercera versión, pero en esta versión los movimientos que se desplazan a una posición superior (diagonal arriba-derecha, arriba, diagonal arriba-izquierda) no han sido recalculados y mantienen siempre el valor de -inf, pero se mantienen y añaden al vector.

Otras opciones que he valorado, como invertir la selección asignando +inf al principio en vez de -inf, dan resultados erróneos que no se adecuan a la solución.

```
while(i + 1 < filas || j + 1 < columnas) {
    S1 = S2 = S3 = S4 = S5 = S6 = S7 = S8 = numeric_limits<int>::min();

    if(i - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( x: i - 1,  &: j))) {     //arriba
        S1 = sol + pesos[i - 1][j];
    }

    if(i - 1 >= 0 && j + 1 < columnas && !usada( usadas: en_uso,  casilla: make_pair( x: i - 1,  y: j + 1))) {     //digonal arriba derecha
        S2 = sol + pesos[i - 1][j + 1];
    }

    if(j + 1 < columnas && !usada( usadas: en_uso,  casilla: make_pair( &: i,  y: j + 1))) {     //derecha
        S3 = sol + pesos[i][j + 1];
    }

    if(i + 1 < filas && j + 1 < columnas && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  y: j + 1))) {     //diagonal abajo derecha
        S4 = sol + pesos[i + 1][j + 1];
    }

    if(i + 1 < filas && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  &: j))) {     //abajo
        S5 = sol + pesos[i + 1][j];
    }

    if(i + 1 < filas && j - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( x: i + 1,  y: j - 1))) {     //diagonal abajo izquierda
        S6 = sol + pesos[i + 1][j - 1];
    }

    if(j - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( &: i,  y: j - 1))) {     //izquierda
        S7 = sol + pesos[i][j - 1];
    }

    if(i - 1 >= 0 && j - 1 >= 0 && !usada( usadas: en_uso,  casilla: make_pair( x: i - 1,  y: j - 1))) {     //diagonal arriba izquierda
        S8 = sol + pesos[i - 1][j - 1];
    }
```

*Figura 20*

```
num = {S1, S2, S3, S4, S5, S6, S7, S8};
sort( first: num.begin(),  last: num.end());

maximo = num[3];
```

*Figura 21*

Aunque he intentado reducir los tiempos en la cota optimista, no lo he conseguido, sólo me daban resultados erróneos al cambiar el método.
De la misma forma me pasaba al cambiar el *operator()* de la clase *Compara* que ordena la cola de prioridad, las otras opciones que he comprobado, como utilizar la cota optimista o la pesimista, dan directamente resultados erróneos.

# 5. Estudio comparativo de distintas estrategias de búsqueda

En este apartado voy a realizar una pequeña comparación de las 4 versiones que he probado para la realización de la cota pesimista.
Todos los tiempos de la tabla de comparación han sido probados con el mapa 002 de los mapas pertenecientes a los ficheros de test.

| Forma de hallar el nodo | Movimientos disponibles | Tiempo (ms) |
|---|---|---|
| El elemento intermedio | Todos | 55.816 |
| El elemento intermedio | Todos menos los superiores | 34.472 |
| El máximo elemento | Todos | 519.228 |
| El máximo elemento | Todos menos los superiores | 2.328 |

Como se puede observar en la tabla, escoger el elemento intermedio da resultados muy parecidos, no existe un gran cambio al realizar todos lo movimientos en la búsqueda o no, aunque sigue siendo más rápido no considerar los desplazamientos hacia arriba.
Mientras tanto cuando se selecciona el máximo elemento el resultado es totalmente distinto, en el caso de considerar todos los movimientos el tiempo se dispara, dando un resultado muy superior a cualquier otra versión. Al mismo tiempo, realizar sólo los movimientos que no se desplazan hacia arriba da un resultado mucho menor que cualquier otro método.

# 6. Tiempos de ejecución

Por desgracia, no he conseguido que mi programa sea lo suficientemente rápido para poder completar ninguno de los siguientes mapas en un tiempo inferior a 30 segundos.

| Fichero de test | Tiempo (ms) |
|---|---|
| 060.map | ? |
| 090.map | ? |
| 201.map | ? |
| 301.map | ? |
| 501.map | ? |
| 700.map | ? |
| 900.map | ? |
| 1K.map | ? |
| 2K.map | ? |
| 3K.map | ? |
| 4K.map | ? |