Práctica de Programación II

Curso 2008 - 2009

El Juego de la Vida

UNED
I.T. Informática de Gestión

Juan José Aguililla Martínez
jaguilill1@alumno.uned.es

Creado el: 3 de mayo de 2009
Modificado el: 6 de julio de 2009

Información del Documento

Índice de contenido

Referencias

- Libro de texto de la asignatura....Diseño de Programas: Formalismo y Abstracción
- Página personal de Jerónimo Quesada....http://web.jet.es/jqc/index.html
- Ejercicios resueltos....http://www.lsi.uned.es/p2/
- Life - J. Conway Scientific American....http://ddi.cs.uni-potsdam.de/

Herramientas utilizadas

- OpenOffice 3.1....http://openoffice.org
- VIM (editor de código fuente)....http://www.vim.org
- XDS 2.5 (compilador de Modula-2)....http://www.excelsior-usa.com/xdsx86.html
- Lenguaje LSN....

2. Enunciado de la práctica

2.1. Cuestiones sobre el enunciado

1) **Piense intuitivamente cómo resolvería a mano el problema y exponga claramente sus conclusiones.**

   Resolvería el problema recorriendo todas las celdas del tablero y contando, para cada una de ellas, el número de vecinos vivos. En función del número de vecinos vivos:

   - Si la celda está viva y el número de vecinos vivos está entre dos y tres, sigue viva.
   - Si vive y tiene otro número de vecinos vivos (menos de dos o mas de tres), muere.
   - Si está muerta y tiene exactamente tres vecinos vivos. Vive.

2) **¿Cuáles son los casos extremos del cálculo de vecinos? Clasifique las celdas en categorías atendiendo a su número de vecinos.**

   El número de casos distintos en los extremos coincide con el número mínimo de vecinos de una celda más uno (la posición ocupada por la celda en cuestión).

   El número mínimo de vecinos se da cuando la casilla está situada justamente en una esquina.

   La cantidad de casos en los extremos depende del número de dimensiones (d) del tablero y del radio especificado para los vecinos de una celda (r). Para calcularlos se puede utilizar la formula: $(r + 1)^d$

   Esta formula aplicada a un tablero bidimensional con un radio de uno para los vecinos, nos da cuatro casos distintos.

3) **(IMPORTANTE Y RECOMENDADA) Simplifique las reglas del enunciado de manera que necesite menos reglas para decir lo mismo. Pista: necesitará realizar alguna abstracción.**

   Las reglas que determinan si una célula vive o muere en la siguiente generación son:

   - Si tiene tres vecinos vivos, vive.
   - Si vive y tiene dos vecinos vivos, sigue viva.
   - En otro caso muere.

4) **¿Qué efecto tiene sobre las reglas la definición de los vecinos? ¿Cuántos vecinos tendría una celda en una parrilla de tres dimensiones? Justifíquelo (Pista: ¿qué forma debería tener una celda en una parrilla tridimensional?)**

   Al variar el número de vecinos, el número de casos extremos cambia también.

   Los vecinos se corresponden con la superficie o área (en función del número de dimensiones) que rodea a la célula. Es decir, el cuadrado o el cubo en que está inscrita (en el centro) la casilla.

   Por tanto, para calcular los datos de los vecinos (en función del número de dimensiones -d- y del radio -r-) se pueden utilizar las siguientes formulas:

   El número máximo de vecinos se describe por la formula: $((r * 2) + 1)^d - 1$

   El mínimo (estando la celda justo en una esquina) es: $(r + 1)^d - 1$

   Y el número de posiciones distintas en los bordes es: $(r + 1)^d$

5) **¿Qué sucedería si se considerase el tablero cerrado (o sea, si el borde superior se imaginase unido con el inferior y el izquierdo con el derecho? ¿Qué efecto tendría esta consideración sobre cada generación para su cálculo?**

   En todos los casos (incluso para los bordes) siempre habría el mismo número de vecinos.

   Las casillas fuera del tablero, en vez de ser ignoradas, serán sustituidas por las opuestas.

   Para obtener las coordenadas en el tablero de una casilla que está fuera de los bordes (siendo c cualquier coordenada -x, y, z-):

   - Si la coordenada es negativa, la nueva coordenada será: $c = abs(c) - max\_c \rightarrow c + max\_c$
   - Si la coordenada excede del borde del tablero, se calculará con la siguiente formula: $c = c \% max\_c$
   - Si la coordenada esta dentro del tablero, no se modifica su valor, independientemente de que el resto de coordenadas de la casilla estén dentro o fuera del tablero.

   En el gráfico del punto 3.2 se muestra la relación entre las casillas de los bordes y sus opuestas dentro del tablero: $1 \rightarrow 1'$, $2 \rightarrow 2'$, $3 \rightarrow 3'$, $4 \rightarrow 4'$, $5 \rightarrow 5'$

3. Especificación

3.2. Cuestiones sobre la especificación

1) **Si en la postcondición de una función no se hace referencia alguna a las variables de entrada, ¿qué podemos decir de dicha función?**

   Al no existir relación entre los datos de entrada y los de salida, no se pueden garantizar resultados correctos.

   Los resultados obtenidos serían constantes, o aleatorios.

2) **Expónganse formas alternativas de codificar el estado de las celdas. Sería positivo valorar sus pros y contras.**

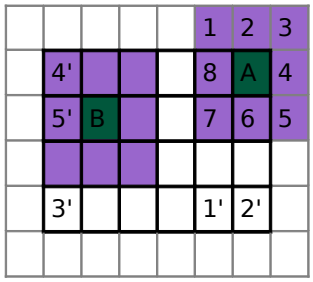
   Podrían especificarse como enteros. En este caso, se tendría la ventaja de poder especificar varios estados distintos para una celda (no solo viva o muerta), se podría almacenar el número de vecinos vivos para dibujar la celda de un color u otro en función de este valor.

   Además, se podría incluir más información acerca de la celda, como por ejemplo, si está en un borde o no (reduciendo el calculo para las celdas en extremos). De este modo se utilizaría un bit para indicar si está viva, otro para indicar si está en el borde izquierdo, etc.

   Por otra parte, utilizar otro tipo de dato implicaría un tamaño mayor en memoria, y mayor tiempo de calculo para procesar cada casilla.

3) **Dibújese un esquema que permita discernir los vecinos de una casilla según su categoría (véase la segunda pregunta del apartado 2.1 en la página 8)**

   En la siguiente tabla se muestra una celda con todos sus vecinos dentro del tablero (B) y una en la esquina (A) las casillas (8, 7, 6 y A) representan los distintos casos que puede tener una celda.



3.3. Especificación de gen

Este punto no será incluido, aunque puede obtenerse sustituyendo las llamadas a funciones por sus postcondiciones dentro de la especificación modular.

La especificación modular de gen se encuentra en el punto 3.5.

3.4. Más cuestiones y algunos ejercicios sobre la especificación

1) **¿Por qué en el perfil de gen no aparecen parámetros para expresar las filas o columnas? ¿Qué ventajas y desventajas tendría incluir como parámetros de entrada las dimensiones del tablero o parrilla (su número de filas y columnas)? Justifíquese.**

   No aparecen porque son constantes para la resolución del problema. El análisis del algoritmo se hace de manera abstracta y para todos los tamaños posibles. Si se pasasen:

   - Pros: se podría utilizar la función para calcular un tablero de cualquier dimensión.
   - Contras: implicaría la comprobación y paso de más parámetros.

2) **Propóngase otra forma diferente (en lugar de un solo vector de booleanos) de representar la parrilla o tablero del Juego de la Vida. Razónense las ventajas e inconvenientes respecto a la representación propuesta en esta práctica.**

   Una matriz de dos dimensiones de booleanos. Complica el calculo recursivo (decrecimiento del tamaño del problema) pero facilita el cálculo de los vecinos. Complica la adaptación del algoritmo a otro número de dimensiones (a una o a tres dimensiones).

3) **Ejercicio recomendado. Especifíquese gen de forma modular. Para ello, deberá proponerse una o más funciones auxiliares y especificarlas completamente declarándose, además, la relación entre gen y sus funciones auxiliares. Sugerimos que se utilice una función vecinos, que calcule el número de celdas adyacentes vivas. Nótese que se debe definir el concepto de ADYACENCIA de manera que resulte consistente con las reglas enunciadas en 2, en la página 7.**

   Adyacencia: celdas entre *-r* y *+r* filas y columnas de una casilla concreta (siendo *-r-* el radio).

   Las funciones propuestas para el diseño modular son: *casilla, fila, columna, vecinos, condiciones* y *gen* (ver el punto 3.5).

4) **Supóngase un tablero tridimensional para el Juego de la Vida. Especifíquense gen y vecinos para adaptarse a esta situación. Compare las especificaciones de gen y vecinos para dos y tres dimensiones.**

   Se añade una nueva constante NZ y se crea una nueva función: *profundidad(i)* para calcular la nueva coordenada de una casilla.

5) **Modifíquese la función vecinos para que calcule los de una casilla dada de tal manera que se consideren no sólo los adyacentes, sino todos los que estén a menos de un cierto número de casillas de distancia? (Pista: defínase una medida de distancia, si se desea, como una función auxiliar). Especifíquese gen utilizando vecinos. Especifíquese nuevamente vecinos para que responda a las reglas del enunciado.**

   Sería necesario cambiar los cuantificadores en la postcondición de vecinos de *{-1, 0, 1}* a *{-r .. r}*.

3.5. Nueva especificación de gen

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0\}$$
$$fun\ gen(v : \text{vector}[0\,..\,NF*NC-1]\ de\ booleano)$$
$$dev(s : \text{vector}[0\,..\,NF*NC-1]\ de\ booleano)$$
$$\{R \equiv \forall\, \alpha \in \{0\,..\,NF*NC-1\}\,.\,s[\alpha] = condiciones(v, \alpha)\}$$

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i > NF*NC\}$$
$$fun\ condiciones(v : \text{vector}[0\,..\,NF*NC-1]\ de\ booleano\ ;\ i : entero)$$
$$dev(s : booleano)$$
$$\{R \equiv s = (vecinos(v, i) = 3) \vee (v[i] \wedge vecinos(v, i) = 2)\}$$

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i > NF*NC\}$$
$$fun\ vecinos(v : \text{vector}[0\,..\,NF*NC-1]\ de\ booleano\ ;\ i : entero)\ dev(s : entero)$$
$$\{R \equiv s = \sum \alpha \in \{-1,0,1\}\,.\,(N\ \beta \in \{-1,0,1\}\,.\,(\begin{array}{l}(\alpha \neq 0 \wedge \beta \neq 0) \\ \wedge (fila(i) + \alpha \geq 0) \wedge (fila(i) + \alpha < NF) \\ \wedge (columna(i) + \beta \geq 0) \wedge (columna(i) + \beta < NC) \\ \wedge (v[casilla(columna(i) + \beta,\ fila(i) + \alpha)])\end{array}))\}$$

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i > NF*NC\}$$
$$fun\ fila(i : entero)\ dev(s : entero)$$
$$\{R \equiv s = i \div NC\}$$

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i > NF*NC\}$$
$$fun\ columna(i : entero)\ dev(s : entero)$$
$$\{R \equiv s = i\ mod\ NC\}$$

$$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i > NF*NC\}$$
$$fun\ casilla(c, f : entero)\ dev(s : entero)$$
$$\{R \equiv s = f*NC + c\}$$

4. Diseño recursivo mediante una inmersión no final de gen

4.2. Cálculo de la inmersión y llamada inicial

El parámetro de inmersión elegido es el índice de la casilla (i). Ha sido elegido como parámetro de inmersión debido a que es el parámetro que especifica el tamaño del problema, y permite reducirlo entre llamadas. El rango admisible para este parámetro está indicado en la precondición, y está relacionado con el tamaño del tablero.

$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i < NF * NC\}$
$fun\ igen(v : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano\ ;\ i : entero)$
$dev\,(s : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano)$
$\{R \equiv \forall\, \alpha \in \{0\,..\,i\}\,.\,s[\alpha] = condiciones\,(v\,,\,\alpha)\}$

$igen(v\,,\,NF * NC - 1)$

4.4. Análisis por casos de la función igen

Caso	Protección	Resultados obtenidos
Caso recursivo	i > 0	Calculamos la casilla *i* y después la combinamos con el resultado de invocar a *igen* para la casilla anterior. El resultado de *igen* para i - 1 supone resueltas las casillas (0 .. i - 1).
Caso trivial	i = 0	Como en este caso la casilla es la primera, calculamos su condiciones y las combinamos con v (que se supone resuelto para las casillas (1 .. NF * NC – 1).

La especificación formal sería:

$caso\ i = 0 \quad \rightarrow sea\ z = condiciones(v\,,i)\ en\ c\,(v\,,i\,,z)$
$\quad [\,]\ \ i > 0 \quad \rightarrow sea\ z = condiciones(v\,,i)\ en\ c\,(igen(v\,,i-1),i\,,z)$

4.5. Composición algorítmica de igen

$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i < NF * NC\}$
$fun\ igen(v : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano\ ;\ i : entero)$
$dev\,(s : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano)$
$\quad caso\ i = 0 \quad \rightarrow sea\ z = condiciones(v\,,i)\ en\ c\,(v\,,i\,,z)$
$\quad\quad [\,]\ \ i > 0 \quad \rightarrow sea\ z = condiciones(v\,,i)\ en\ c\,(igen(v\,,i-1),i\,,z)$
$\quad fcaso$
$ffun$
$\{R \equiv \forall\, \alpha \in \{0\,..\,i\}\,.\,s[\alpha] = condiciones\,(v\,,\,\alpha)\}$

La especificación de la función de combinación es la siguiente:

$\{Q \equiv NF \geq 0 \wedge NC \geq 0 \wedge 0 \leq i < NF * NC\}$
$fun\ c(v : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano\ ;\ i : entero\ ;\ b : booleano)$
$dev\,(s : \text{vector}[0\,..\,NF * NC - 1]\ de\ booleano)$
$\{R \equiv \forall\, \alpha \in \{i\,..\,NF * NC - 1\}\,.\,(\alpha = i \wedge s[\alpha] = b) \vee (\alpha > i \wedge s[\alpha] = v[\alpha])\}$

4.6. Verificación formal de la corrección de igen

Los componentes utilizados para la verificación formal de *igen* son los siguientes:

$x \equiv (v, i)$
$y \equiv (s)$
$Q(x) \equiv 0 \leq i < NF * NC$
$R(x, y) \equiv \forall \alpha \in \{0..i\}.\, s[\alpha] = condiciones(v, \alpha)$
$s(x) \equiv (v, i-1)$
$B_t \equiv i = 0$
$B_{nt} \equiv i > 0$
$triv(x) \equiv sea\ z = condiciones(v, i)\ en\ c(v, i, z)$
$c(y, s(x)) \equiv sea\ z = condiciones(v, i)\ en\ c(igen(v, i-1), i, z)$
$c \equiv \forall \alpha \in \{i..NF * NC - 1\}.(\alpha = i \wedge s[\alpha] = b) \vee (\alpha > i \wedge s[\alpha] = v[\alpha])$

4.6.1. Completitud de la alternativa

$Q(x) \Rightarrow B_t(x) \vee B_{nt}(x)$
$0 \leq i < NF * NC \rightarrow i = 0 \vee i > 0$
$0 \leq i < NF * NC \rightarrow i \geq 0$

4.6.2. Satisfacción de la precondición para la llamada interna

$Q(x) \wedge B_{nt} \Rightarrow Q(s(x))$
$0 \leq i < NF * NC \wedge i > 0 \rightarrow 0 \leq i - 1 < NF * NC$
$0 < i < NF * NC \rightarrow 0 \leq i - 1 < NF * NC$

Como *i* tiene que ser mayor que cero (como mínimo uno) *i – 1* no podrá ser inferior a cero (a lo sumo, cero).

4.6.3. Base de la inducción

$Q(x) \wedge B_t \Rightarrow R(x, triv(x))$
$0 \leq i < NF * NC \wedge i = 0 \rightarrow sea\ z = condiciones(v, i)\ en\ c(v, i, z)$
$i = 0 \rightarrow sea\ z = condiciones(v, 0)\ en\ c(v, 0, z)$
$i = 0 \rightarrow \forall \alpha \in \{i..NF * NC - 1\}.(\alpha = 0 \wedge s[0] = condiciones(v, 0)) \vee (\alpha > 0 \wedge s[\alpha] = v[\alpha])$

Para *i = 0*, se calcula la casilla cero y se asigna al vector resultante, el resto de casillas (que se suponen resueltas por llamadas anteriores) se copian al tablero final.

4.6.4. Paso de inducción

$Q(x) \wedge B_{nt}(x) \wedge R(s(x), y') \Rightarrow R(x, c(y', x))$

$0 \leq i < NF * NC \wedge i > 0 \wedge (\forall \alpha \in \{0..i-1\}.\, s[\alpha] = condiciones(v, \alpha)) \rightarrow$
$c(\forall \alpha \in \{0..i-1\}.\, s[\alpha] = condiciones(v, \alpha), i, condiciones(v, i))$

$0 < i < NF * NC \wedge (\forall \alpha \in \{0..i-1\}.\, s[\alpha] = condiciones(v, \alpha)) \rightarrow$
$c(\forall \alpha \in \{0..i-1\}.\, s[\alpha] = condiciones(v, \alpha), i, condiciones(v, i))$

$0 < i < NF * NC \wedge (\forall \alpha \in \{0..i-1\}.\, s[\alpha] = condiciones(v, \alpha)) \rightarrow$
$\forall \alpha \in \{i..NF * NC - 1\}.(\alpha = i \wedge s[\alpha] = condiciones(v, i)) \vee (\alpha > i \wedge s[\alpha] = condiciones(v, i))$

4.6.5. Elección de una estructura de preorden bien fundado

El preorden bien fundado viene expresado por la siguiente formula: $t(v, i) = i$ esta expresión devuelve siempre valores positivos para los parámetros de entrada que cumplan la precondición (que requiere que *i* sea mayor o igual a cero).

4.6.6. Demostración del decrecimiento de los datos

$t(v,i)=i$
$t(v,i-1)=i-1$
$i>i-1$

4.7. Estudio del coste de igen

Para calcular el coste de *igen*, primero calcularemos el coste de todas las funciones a las que llama:

- **condiciones:** solo invoca a la función vecinos y realiza operaciones simples. Por tanto su coste es constante.
- **vecinos:** a pesar de tener dos bucles anidados, el coste es constante porque el dominio de los bucles no depende del tamaño del problema (es siempre tres, en total nueve iteraciones).
- **fila:** Solo esta compuesta de operaciones matemáticas simples sin relación con el tamaño del problema luego su coste es constante.
- **columna:** Al igual que *fila* tiene un coste constante por los mismos motivos.
- **casilla:** igual que las dos funciones anteriores, su orden es constante.

Para calcular el orden de complejidad de *igen* aplicaremos la formula 1.2 de Peña:

$$T(n)=\begin{cases} cn^k & ,si\ 0\le n<b \\ aT(n-b)+cn^k & ,si\ n\ge b \end{cases}$$

Donde:

- **a:** Número de llamadas recursivas en la función. En este caso es uno.
- **b:** Reducción del problema en cada llamada recursiva. En este algoritmo sería una casilla.
- **n^k:** Coste de las intrucciones no recursivas. Como en este algoritmo dicho coste es uno, *k* debe valer cero.

Y aplicando las reglas de resolución de recurrencias (Peña 1.3):

$$T(n)=\begin{cases} \Theta(n^k) & ,si\ a<1 \\ \Theta(n^{k+1}) & ,si\ a=1 \\ \Theta(n^{n\ \mathrm{div}\ b}) & ,si\ a>1 \end{cases}$$

El coste de *igen* es: $\Theta(n)$

NOTA: La función 'c' teóricamente depende del tamaño del problema. Esto es así debido a la imposibilidad en el LR de modificar una posición de un vector, lo cual implica una operación para copiar los datos (un subconjunto) en otro vector (el resultado).

En la implementación, no se implementará 'c' de manera iterativa, y para que los datos empíricos estén en sintonía con el análisis teórico, se supondrá esta función como constante.

4.8. Cuestiones sobre el diseño y verificación recursivos

1) **¿Que sucedería si la alternativa no fuese completa? ¿Y si hubiese intersección en las protecciones que definen la alternativa?**

   Si la alternativa no fuese completa, habría parámetros válidos para el algoritmo (que cumplen la precondición) que quedarían sin tratar. En el caso de que hubiese intersección en las protecciones, se daría el caso de que los mismos parámetros podrían ser tratados de dos maneras distintas.

2) **Proponga un caso trivial alternativo. ¿Qué consecuencias traería su uso?**

   Un caso trivial alternativo podría ser *i < 0*. Esto supondría una iteración más en el algoritmo, pero un caso trivial mucho más sencillo (devolvería como resultado el mismo vector recibido).

6. Diseño iterativo de genit

6.2. Derivación del invariante desde la postcondición

Obtendremos el invariante (y la condición de salida del bucle) debilitando la postcondición de gen (se ha cambiado el operador relacional *distinto* por el operador *menor que*):

$\{R \equiv \forall \alpha \in \{0..NF*NC-1\}.s[\alpha] = condiciones(v,\alpha)\}$
$\{R \equiv \forall \alpha \in \{0..i\}.s[\alpha] = condiciones(v,\alpha) \wedge i = NF*NC-1\}$
$\{P \equiv \forall \alpha \in \{0..i\}.s[\alpha] = condiciones(v,\alpha)\}$
$\neg B \equiv i = NF*NC-1$
$B \equiv i < NF*NC$

6.3. Inicialización del bucle

Para que después de la inicialización se cumpla el invariante, se inicializará el contador (*i*) a menos uno. Esto hará necesarios algunos artificios dentro del bucle y en la condición de salida. A continuación expongo la demostración:

$\{Q\}\ inic\ \{P\}$
$\{Q \equiv NF \geq 0 \wedge NC \geq 0\}$
$i := -1$
$\{P \equiv \forall \alpha \in \{0..-1\}.s[\alpha] = condiciones(v,\alpha)\}$

El invariante se cumple puesto que el dominio del cuantificador se anula.

6.4. Instrucción avanzar del bucle

El algoritmo procesará todas las casillas del tablero de una en una, luego la instrucción avanzar elegida será: $i := i+1$.

Respecto a la función de cota la solución propuesta es: $t = NF*NC-2-i$ el motivo de que no sea $t = NF*NC-1-i$ es que i es inicializado a menos uno, por tanto en la función de cota para compensar este desplazamiento del rango, se resta uno a la función. De esta manera, en la última iteración, el la que i vale NF * NC – 2 la función de cota devuelve cero.

6.5. Restablecimiento del invariante

Debido a la modificación del rango causada por la inicialización de i. La instrucción que restablece el invariante debe compensar este efecto sumando uno a los valores de i. Este coste adicional se puede eliminar cambiando el orden de las intrucciones avanzar y restablecer. La instrucción que restablece el invariante es la siguiente: $s[i+1] := condiciones(v, i+1)$

6.6. Código del algoritmo iterativo de genit

$\{Q \equiv NF \geq 0 \wedge NC \geq 0\}$
fun $gen(v : \text{vector}[0..NF*NC-1]\ de\ booleano)$
$dev(s : \text{vector}[0..NF*NC-1]\ de\ booleano)$
  var $i : natural$;
  $i := -1$;
  mientras $i+1 < NF*NC$ *hacer* $\{P \equiv \forall \alpha \in \{0..i\}.s[\alpha] = condiciones(v,\alpha)\}$
    $s[i+1] := condiciones(v, i+1)$;
    $i := i+1$;
  fmientras
  dev s ;
ffun
$\{R \equiv \forall \alpha \in \{0..NF*NC-1\}.s[\alpha] = condiciones(v,\alpha)\}$

6.7. Estudio del coste

El coste del algoritmo iterativo es lineal, puesto que las instrucciones internas al bucle son constantes (ver el coste de las funciones en el apartado 4.7) y el número de iteraciones depende del tamaño del problema, en nuestro caso el número de casillas (NF * NC). Luego, aplicando la regla del producto:

$\Theta(n) * \Theta(1)$
$\Theta(n * 1)$
$\Theta(n)$

6.8. Cuestiones sobre la función iterativa

3) ¿Qué sucedería si no se inicializasen las variables antes de comenzar el bucle?

Si no se inicializasen las variables antes de comenzar el bucle, el punto de partida sería indefinido y el número de iteraciones podría ser mayor o menor que las esperadas.

7. Implementación

7.1. Código y juegos de pruebas

7.1.1. Código fuente

El algoritmo ha sido implementado tanto en LSN como en Modula-2, debido a problemas en tiempo de ejecución del primero.

En LSN la función combinar lanza un error en tiempo de ejecución. Al imprimir el parámetro '*i*' (declarado como entero) se obtiene '*FALSO*' (ver documentación del código fuente).

El código con las trazas no se ha suprimido, solo ha sido deshabilitado, por considerarse la penalización en tiempo de ejecución despreciable (según el compilador, posiblemente nula) frente a la ventaja de poder diagnosticar posibles problemas en el futuro.

Implementación LSN:

```
1  modulo vida
2
3  (*
4   * SE PRODUCE UN ERROR EN TIEMPO DE EJECUCIÓN AL PROCESAR LA FUNCIÓN 'c'
5   * QUE PARECE UN DEFECTO DEL COMPILADOR O DE LA MAQUINA PILA.
6   *)
7
8  constantes
9      maxF := 50;
10     maxC := 50;
11     (**
12      * Depuración:
13      * Las sentencias de depuración no se han suprimido, con el * fin de poder
14      * ser utilizadas posteriormente. Para hacerlo se debe modificar esta
15      * constante a 'verdadero'.
16      * El precio que hay que pagar es una pequeña penalización en el rendimiento.
17      *)
18     LOG := falso;
19 fconstantes
20
21 (* Implementacion de funciones auxiliares *)
22 (*****************************************************************************)
23
24 (**
25  * La función de combinación, modifica la posición de un vector con un
26  * valor pasado por parámetro.
27  *
28  * ERROR EN EJECUCIÓN Imprime 'FALSO' para un valor declarado como entero (i)
29  *
30  * @param v Vector que va a ser modificado.
31  * @param i Posición del vector que se va a cambiar.
32  * @param b Nuevo valor de la posición 'i' del vector 'v'.
33  * @return Vector 'v' con su posición 'i' cambiada por 'b'.
34  *)
35 precondicion { 0 <= i y i < maxF * maxC }
36 funcion c (v: vector [0 .. maxF * maxC - 1] de booleano, i: entero, b: booleano)
37 devuelve (s: vector [0 .. maxF * maxC - 1] de booleano)
38 inicio
39     caso si log escribe (">>> c  i -> "); escribeln (i); fcaso;
40     v[i] := b;
41     caso si log escribeln (">>> c (salida)"); fcaso;
42     devuelve v;
43 ffuncion
44 postcondicion {
45     paratodo alfa perteneciente [i .. maxF * maxC - 1].(
46         (alfa = i y s[alfa] = b) o (alfa > i y s[alfa] = v[alfa])
47     )
```

```
 48 }
 49
 50 (**
 51  * Calcula el indice del vector de una celda a partir de sus coordenadas.
 52  *
 53  * @param c Columna de la casilla.
 54  * @param f Fila de la casilla.
 55  * @param nc Número de columnas del tablero.
 56  * @return Indice de la casilla en la fila 'f' y columna 'c'.
 57  *)
 58 precondicion {
 59     0 <= c y c < maxC
 60     y 0 <= f y f < maxF
 61     y nc > 0 y nc <= maxC
 62 }
 63 funcion casilla (c, f, nc: entero) devuelve (s: entero)
 64 inicio
 65     devuelve f * nc + c;
 66 ffuncion
 67 postcondicion { s = f * nc + c }
 68
 69 (**
 70  * Calcula la columna de una celda a partir de su índice 'i'.
 71  *
 72  * @param i Indice de la celda.
 73  * @param nc Número de columnas del tablero.
 74  * @return Columna de la casilla en la posición 'i'.
 75  *)
 76 precondicion { 0 <= i y i < maxF * maxC y nc > 0 y nc <= maxC }
 77 funcion columna (i, nc: entero) devuelve (s: entero)
 78 inicio
 79     devuelve i mod nc;
 80 ffuncion
 81 postcondicion { s = i mod nc }
 82
 83 (**
 84  * Calcula la fila de una celda a partir de su índice 'i'.
 85  *
 86  * @param i Indice de la celda.
 87  * @param nc Número de columnas del tablero.
 88  * @return Fila de la casilla en la posición 'i'.
 89  *)
 90 precondicion { 0 <= i y i < maxF * maxC y nc > 0 y nc <= maxC }
 91 funcion fila (i, nc: entero) devuelve (s: entero)
 92 inicio
 93     devuelve i div nc;
 94 ffuncion
 95 postcondicion { s = i div nc }
 96
 97 (**
 98  * Obtiene el número de vecinos vivos de una celda.
 99  *
100  * @param v Vector que contiene las celdas del tablero.
101  * @param nf Número de filas del tablero.
102  * @param nc Número de columnas del tablero.
103  * @param i Indice de la celda de la que obtener el número de vecinos.
104  * @return Número de vecinos vivos de una celda.
105  *)
106 precondicion {
107     nf > 0 y nf <= maxF
108     y nc > 0 y nc <= maxC
109     y 0 <= i y i < maxF * maxC
110 }
111 funcion vecinos (v: vector [0 .. maxF * maxC - 1] de booleano, nf, nc, i: entero)
```

```
112 devuelve (s: entero)
113 variables
114     ii, jj, contador, fil, col: entero;
115 fvariables
116 inicio
117     contador := 0;
118     ii := -1;
119     invariante { verdadero }
120     mientras (ii <= 1) hacer
121         jj := -1;
122         invariante { verdadero }
123         mientras (jj <= 1) hacer
124             caso si no ( (ii = 0) y (jj = 0) )
125                 fil := fila (i, nc) + ii;
126                 col := columna (i, nc) + jj;
127                 caso si log
128                     escribe (">>> vecinos ["); escribe (i); escribe ("] (");
129                     escribe (fil); escribe (", "); escribe (col);
130                 fcaso;
131 
132                 caso si (fil >= 0
133                     y fil < nf
134                     y col >= 0
135                     y col < nc)
136 
137                     (*
138                      * Condición anidada debido a que la evaluación de
139                      * expresiones lógicas no es por cortocircuito
140                      *)
141                     caso si v [casilla (col, fil, nc)]
142                         contador := contador + 1;
143                         caso si log escribeln (") : VIVA"); fcaso;
144                     si_no
145                         caso si log escribeln (") : MUERTA"); fcaso;
146                     fcaso;
147                 si_no
148                     caso si log escribeln (") : FUERA"); fcaso;
149                 fcaso;
150             fcaso;
151             jj := jj + 1;
152         fmientras;
153         ii := ii + 1;
154     fmientras;
155     devuelve contador;
156 ffuncion
157 postcondicion { s =
158     sumatorio alfa perteneciente [-1 .. 1].
159     (conteo beta perteneciente [-1 .. 1].(
160         (alfa != 0 y beta != 0)
161         y fila (i, nc) + alfa >= 0
162         y fila (i, nc) + alfa < nf
163         y columna (i, nc) + beta >= 0
164         y columna (i, nc) + beta < nc
165         y v [casilla (
166             columna (i, nc) + beta,
167             fila (i, nc) + alfa,
168             nc)]))
169 }
170 
171 (**
172  * Devuelve verdadero si la celda en la posición 'i' debe vivir.
173  *
174  * @param v Vector que contiene las celdas del tablero.
175  * @param i Indice de la celda que se va a comprobar.
```

```
176  * @return Cierto si la celda en la posición 'i' cumple las condiciones para vivir.
177  *)
178 precondicion {
179     nf > 0 y nf <= maxF
180     y nc > 0 y nc <= maxC
181     y 0 <= i y i < maxF * maxC
182 }
183 funcion condiciones (v: vector [0 .. maxF * maxC - 1] de booleano, nf, nc, i: entero)
184 devuelve (s: booleano)
185 variables
186     vivos: entero;
187 fvariables
188 inicio
189     vivos := vecinos(v, nf, nc, i);
190     caso si log escribe (">>> condiciones ["); escribe (i); escribe ("]: ");
escribeln (vivos); fcaso;
191     devuelve (vivos = 3) o (v [i] y vivos = 2);
192 ffuncion
193 postcondicion {
194     s = (vecinos (v, nf, nc, i) = 3) o (v [i] y vecinos (v, nf, nc, i) = 2)
195 }
196
197 (*
198  * Funciones que van a ser llamadas desde el programa principal
199  * La implementacion actual devuelve el mismo vector de entrada
200  * sin realizar ningun calculo adicional
201  *)
202 (*****************************************************************************)
203
204 (**
205  * FUNCIÓN RECURSIVA NO FINAL
206  *)
207 precondicion {
208     nf > 0 y nf <= maxF
209     y nc > 0 y nc <= maxC
210     y 0 <= i y i < maxF * maxC
211 }
212 funcion igen (v: VECTOR [0 .. maxF * maxC - 1] DE BOOLEANO, nf, nc, i: ENTERO)
213 devuelve (s: VECTOR [0 .. maxF * maxC - 1] DE BOOLEANO)
214 variables
215     casilla: booleano;
216 fvariables
217 inicio
218     casilla := condiciones (v, nf, nc, i);
219     caso si (i > 0)
220         (* c (igen (v, nf, nc, i - 1), i, casilla); *)
221         v := igen (v, nf, nc, i - 1);
222         v [i] := casilla;
223     fcaso;
224     (* c (v, i, casilla); *)
225     v [i] := casilla;
226     devuelve v;
227 ffuncion
228 postcondicion {
229     ParaTodo alfa perteneciente [0 .. maxF * maxC - 1].(
230         s [alfa] = condiciones (v, nf, nc, alfa))
231 }
232
233 (**
234  * FUNCIÓN ITERATIVA
235  *)
236 precondicion {
237     nf > 0 y nf <= maxF
238     y nc > 0 y nc <= maxC
```

```
}
funcion genit (v: VECTOR [0 .. maxF * maxC - 1] DE BOOLEANO, nf, nc: ENTERO)
devuelve (s: VECTOR [0 .. maxF * maxC - 1] DE BOOLEANO)
variables
    ii: entero;
    resultado: VECTOR [0 .. maxF * maxC - 1] DE BOOLEANO;
fvariables
inicio
    ii := -1;
    invariante {
        ParaTodo alfa perteneciente [0 .. ii].(
            resultado [alfa] = condiciones (v, nf, nc, alfa))
    }
    mientras ii < nf * nc - 1 hacer
        resultado [ii + 1] := condiciones (v, nf, nc, ii + 1);
        ii := ii + 1;
    fmientras;
    devuelve resultado;
ffuncion
postcondicion {
    ParaTodo alfa perteneciente [0 .. maxF * maxC - 1].(
        s [alfa] = condiciones (v, nf, nc, alfa))
}

fmodulo
```

Implementación Modula-2:

```
IMPLEMENTATION MODULE VIDA;

FROM P2BIOS IMPORT NATURAL,ENTERO;
FROM InOut IMPORT WriteInt, WriteLn, WriteString;

(**
 * Depuración:
 * Las sentencias de depuración no se han suprimido, con el * fin de poder
 * ser utilizadas posteriormente. Para hacerlo se debe modificar esta
 * constante a 'verdadero'.
 * El precio que hay que pagar es una pequeña penalización en el rendimiento.
 *)
CONST LOG = FALSE;

(* Implementacion de funciones auxiliares *)
(*****************************************************************************)

(**
 * La función de combinación, modifica la posición de un vector con un
 * valor pasado por parámetro.
 *
 * @param v Vector que va a ser modificado.
 * @param i Posición del vector que se va a cambiar.
 * @param b Nuevo valor de la posición 'i' del vector 'v'.
 * @return Vector 'v' con su posición 'i' cambiada por 'b'.
 *)
PROCEDURE c (VAR v: TipoVECTOR; i: ENTERO; b: BOOLEAN);
BEGIN
    v[i] := b;
END c;

(**
 * Calcula el indice del vector de una celda a partir de sus coordenadas.
 *
 * @param c Columna de la casilla.
 * @param f Fila de la casilla.
 * @param nc Número de columnas del tablero.
```

```
 38  * @return Indice de la casilla en la fila 'f' y columna 'c'.
 39  *)
 40 PROCEDURE casilla (c, f: ENTERO; nc: NATURAL): ENTERO;
 41 BEGIN
 42     RETURN ORD(f) * nc + ORD(c);
 43 END casilla;
 44
 45 (**
 46  * Calcula la columna de una celda a partir de su índice 'i'.
 47  *
 48  * @param i Indice de la celda.
 49  * @param nc Número de columnas del tablero.
 50  * @return Columna de la casilla en la posición 'i'.
 51  *)
 52 PROCEDURE columna (i: ENTERO; nc: NATURAL): ENTERO;
 53 BEGIN
 54     RETURN ORD(i) MOD nc;
 55 END columna;
 56
 57 (**
 58  * Calcula la fila de una celda a partir de su índice 'i'.
 59  *
 60  * @param i Indice de la celda.
 61  * @param nc Número de columnas del tablero.
 62  * @return Fila de la casilla en la posición 'i'.
 63  *)
 64 PROCEDURE fila (i: ENTERO; nc: NATURAL): ENTERO;
 65 BEGIN
 66     RETURN ORD(i) DIV nc;
 67 END fila;
 68
 69 (**
 70  * Obtiene el número de vecinos vivos de una celda.
 71  *
 72  * @param v Vector que contiene las celdas del tablero.
 73  * @param nf Número de filas del tablero.
 74  * @param nc Número de columnas del tablero.
 75  * @param i Indice de la celda de la que obtener el número de vecinos.
 76  * @return Número de vecinos vivos de una celda.
 77  *)
 78 PROCEDURE vecinos (v: TipoVECTOR; nf, nc: NATURAL; i: ENTERO): ENTERO;
 79 VAR ii, jj, contador, fil, col: ENTERO;
 80 BEGIN
 81     contador := 0;
 82     FOR ii := -1 TO 1 DO
 83         FOR jj := -1 TO 1 DO
 84             IF NOT ( (ii = 0) AND (jj = 0) ) THEN
 85                 fil := fila (i, nc) + ii;
 86                 col := columna (i, nc) + jj;
 87
 88                 IF LOG THEN
 89                     WriteString (">>> vecinos ["); WriteInt (i, 2);
 90                     WriteString ("] ("); WriteInt (fil, 2); WriteString (", ");
 91                     WriteInt (col, 2);
 92                 END;
 93
 94                 IF (fil >= 0)
 95                     AND (ORD(fil) < nf)
 96                     AND (col >= 0)
 97                     AND (ORD(col) < nc)
 98                     AND v [casilla (col, fil, nc)] THEN
 99
100                     contador := contador + 1;
101
```

```
102                     IF LOG THEN
103                         IF v [casilla (col, fil, nc)] THEN
104                             WriteString (") : VIVA");
105                         ELSE
106                             WriteString (") : MUERTA");
107                         END;
108                         WriteLn ();
109                     END;
110                 ELSIF LOG THEN
111                     WriteString (") : FUERA"); WriteLn ();
112                 END;
113             END;
114         END;
115     END;
116     RETURN contador;
117 END vecinos;
118 
119 (**
120  * Devuelve verdadero si la celda en la posición 'i' debe vivir.
121  *
122  * @param v Vector que contiene las celdas del tablero.
123  * @param i Indice de la celda que se va a comprobar.
124  * @return Cierto si la celda en la posición 'i' cumple las condiciones para vivir.
125  *)
126 PROCEDURE condiciones (v: TipoVECTOR; nf, nc: NATURAL; i: ENTERO): BOOLEAN;
127 VAR vivos: ENTERO;
128 BEGIN
129     vivos := vecinos(v, nf, nc, i);
130     IF LOG THEN
131         WriteString (">>> condiciones ["); WriteInt (i, 2); WriteString ("]: ");
132         WriteInt (vivos, 2); WriteLn ();
133     END;
134     RETURN (vivos = 3) OR (v [i] AND (vivos = 2));
135 END condiciones;
136 
137 (*
138  * Funciones que van a ser llamadas desde el programa principal
139  * La implementacion actual devuelve el mismo vector de entrada
140  * sin realizar ningun calculo adicional
141  *)
142 (*****************************************************************************)
143 
144 (**
145  * Implementacion de la funcion recursiva.
146  *)
147 PROCEDURE igen (VAR v: TipoVECTOR; nf, nc: NATURAL; i: ENTERO; VAR s: TipoVECTOR);
148 BEGIN
149     IF i > 0 THEN
150         igen (v, nf, nc, i - 1, s);
151     END;
152     c (s, i, condiciones (v, nf, nc, i));
153 END igen;
154 
155 (**
156  * Implementacion de la funcion iterativa.
157  *)
158 PROCEDURE genit (VAR v: TipoVECTOR; nf, nc: NATURAL; VAR s: TipoVECTOR);
159 VAR ii: NATURAL;
160 BEGIN
161     (*
162      * El algoritmo debería inicializar el bucle a -1, pero se
163      * dejará en 0 por problemas de compatibilidad de tipos.
164      *)
165     FOR ii := 0 TO nf * nc - 1 DO
```

```
166         s [ii] := condiciones (v, nf, nc, ii);
167     END;
168 END genit;
169
170 END VIDA.
```

7.1.2. Juegos de pruebas

Para completar el juego de pruebas, se han añadido casos para: distintos tamaños, tableros cuadrados y rectangulares, el tablero mínimo posible, el tablero máximo posible, todos los casos en los bordes y dos tetraminos (patrones conocidos encontrados en el articulo original de J. Conway).

La estructura de los ficheros de pruebas se ha modificado, moviendo los archivos de entrada y los de salida a sendos directorios, con el fin de facilitar el procesado automático de todas las pruebas.

La estructura de los nombres de fichero para las pruebas se corresponde con el siguiente formato: ***FF*x*CC*-*DESCRIPCION*-*#*.*txt***. Donde: ***FF*** es el número de filas, ***CC*** se corresponde con el número de columnas, ***DESCRIPCION*** es el propósito de la prueba y ***#*** es el número de secuencia, para pruebas de generaciones consecutivas a partir de un tablero.

Por ejemplo: el fichero *salida/20x20-tetraminod-1.txt* representa el resultado de la primera generación de un tablero de 20 por 20 con las casillas en la disposición del tetramino d (ver artículo de J. Conway).

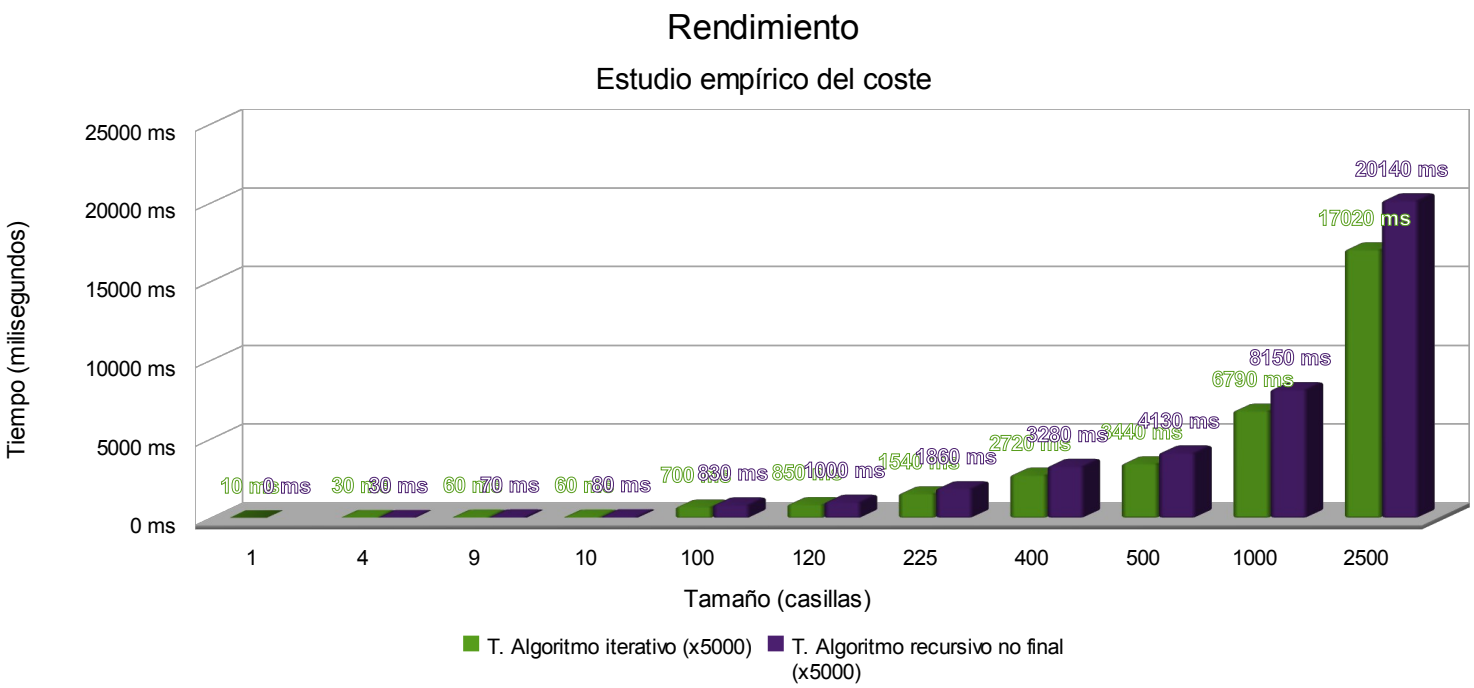
7.6. Estudio empírico del coste

El análisis de rendimiento se realizo sobre la ejecución del juego de pruebas con un **factor multiplicativo de 5000** sin ningún otro proceso en ejecución.

Las pruebas fueron ejecutadas sobre Microsoft Windows XP Profesional Service Pack 3 en una maquina **AMD Athlon 64 2800+ (1800 Mhz)**.

Los resultados se exponen en la tabla y gráfico siguientes:

Fichero	Tamaño (casillas)	T. Algoritmo recursivo no final (x5000)	T. Algoritmo iterativo (x5000)
01X01.TXT	1	0 ms	10 ms
02X02.TXT	4	30 ms	30 ms
03X03.TXT	9	70 ms	60 ms
10X01-1.TXT	10	80 ms	60 ms
10X01-2.TXT	10	80 ms	60 ms
10X01-3.TXT	10	70 ms	70 ms
10X01-4.TXT	10	80 ms	60 ms
10X01-5.TXT	10	80 ms	80 ms
10X10.TXT	100	830 ms	700 ms
06X20-BORDE-1.TXT	120	1000 ms	850 ms
06X20-BORDE-2.TXT	120	980 ms	850 ms
15X15.TXT	225	1860 ms	1540 ms
20X20-TETRAMINOD-1.TXT	400	3280 ms	2720 ms
20X20-TETRAMINOD-2.TXT	400	3280 ms	2720 ms
20X20-TETRAMINOD-3.TXT	400	3280 ms	2720 ms
20X20-TETRAMINOE-1.TXT	400	3280 ms	2720 ms
20X20-TETRAMINOE-2.TXT	400	3280 ms	2720 ms
20X20-TETRAMINOE-3.TXT	400	3290 ms	2720 ms
20X20-TETRAMINOE-4.TXT	400	3280 ms	2720 ms
25X20-BORDE-1.TXT	500	4130 ms	3440 ms
25X20-BORDE-2.TXT	500	4150 ms	3470 ms
25X20-BORDE-3.TXT	500	4170 ms	3470 ms
25X20-BORDE-4.TXT	500	4180 ms	3500 ms
40X25-1.TXT	1000	8150 ms	6790 ms
40X25-2.TXT	1000	8160 ms	6780 ms
40X25-3.TXT	1000	8180 ms	6780 ms
40X25-4.TXT	1000	8160 ms	6810 ms
40X25-5.TXT	1000	8170 ms	6800 ms
50X50.TXT	2500	20140 ms	17020 ms



7.7. Cuestiones sobre la implementación

4) ¿Cómo se podría mejorar (optimizar) el coste de gen?

- Limitando el número de vecinos contados y el número de casillas pendientes. No tiene sentido seguir contando después de cuatro casillas vivas (útil en tableros muy densos). Tampoco es necesario seguir recorriendo los vecinos si la suma de los vecinos vivos y las casillas pendientes de comprobar suman menos de dos (estando la célula viva) o menos de 3 (si la célula está muerta), esta última opción puede significar una gran mejora en tableros poco densos.
- Cambiando la manera en que se reduce el tamaño del problema. Calculando dos casillas en cada iteración (a partir del centro). Esto es: en cada iteración se calcularían las casillas *(N/2) + i* y *(N/2) - i* siendo *N* el número de casillas e *i* el índice del bucle. Esta solución reduce las iteraciones a la mitad (eliminando la penalización que supone la llamada a funciones con arrays (copia de argumentos en la pila, etc.) y las iteraciones del bucle.
- Reduciendo el número de vecinos en los extremos. Ver cuestión 2 en el apartado 3.2.

De estas opciones, habría que estudiar cuales suponen un beneficio, es decir, cuales reducen el tiempo en el caso particular que optimizan más que el tiempo de computación que añaden a **todas** las iteraciones.