uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

Heurística y optimización 2025-2026
Grupo 81

Práctica 2

"Satisfacción de Restricciones y Búsqueda Heurística"

Iciar García Izquierdo – 100495789
Anastasiia Sadova – 100494620

Profesor
Carlos Linares

Índice

1. Introducción

En esta práctica se ha desarrollado un modelo de Satisfacción de Restricciones, así como dos modelos de Búsqueda, utilizando tanto la Búsqueda Heusrística como la Búsqueda de Fuerza Bruta.
El modelo CSP tenía como objetivo resolver un problema de colocación, donde los discos blancos y negros debían colocarse en la rejilla cuadringular con ciertas restricciones. El modelo de CSP se resolvió utilizando la biblioteca de restricciones de Python.
Para los modelos de búsqueda se ha empleado el lenguaje C++ y su objetivo es encontrar la ruta con el menor coste.

2. Parte 1: Satisfacción de restricciones

2.1. Modelo del problema

Primero, vamos a modelar nuestro problema como un problema de satisfacción de restricciones (CSP).

- *Definición de dominio y variables*

  Vamos a empezar con una definición de las variables y sus dominios para el problema actual. Estos son los siguientes:

  - *Variables*: $\{pos_{(x,y)}\}$, que representa las coordenadas de la posición del disco en la rejilla cuadrangular.
  - *Dominio*: `{0, 1}`, que representa los colores de los discos disponibles para poner en una posición de la rejilla, donde 0 es blanco y 1 es negro.

- *Restricciones*

  - **Primera restricción:** No debe quedar ninguna posición vacía.
    Esta restricción ya está cubierta por el dominio definido anteriormente, ya que no incluye *vacío* como valor posible.
  - **Segunda restricción:** El número de discos blancos y negros en cada fila, y en cada columna, debe ser el mismo.

    Para cumplir con esta regla, definiremos una restricción para las filas y otra para las columnas.

    - ***Filas:***

$$\sum_{y=0}^{n-1} pos_{(x,y)} = \frac{n}{2}, \quad \forall x \in [0, n-1]$$

- ***Columnas:***

$$\sum_{x=0}^{n-1} pos_{(x,y)} = \frac{n}{2}, \quad \forall y \in [0, n-1]$$

Al definir restricciones de esta manera, aseguramos que como la cantidad de discos blancos debe tomar la mitad de la cantidad total, la cantidad de discos negros también tomará la mitad de la cantidad total. Esto se garantiza gracias al dominio en el que los discos negros se ven como "1", lo que haría posible esta operación matemática.

- **Tercera restricción:** No es posible disponer más de dos discos del mismo color consecutivamente en una fila o columna.

  Para cumplir con esta restricción, inicialmente tuvimos dos ideas: recolectar el número de errores analizando la rejilla completa y luego verificar si es igual a cero; y detener la resolución del problema si se encontraba el error en algún momento.
  Como se puede observar, el falla de la primera idea residía en su complejidad, ya que el problema continuaba ejecutándose a pesar de ser incorrecto, consumiendo tiempo y memoria. Por ello, decidimos avanzar con la segunda idea y modelarla de la siguiente manera:

  - ***Filas:***

$$\neg\big(pos_{(x,y)} = pos_{(x,y+1)} = pos_{(x,y+2)}\big), \quad \forall x \in [0, n-1],\ \forall y \in [0, n-3]$$

  - ***Columnas:***

$$\neg\big(pos_{(x,y)} = pos_{(x+1,y)} = pos_{(x+2,y)}\big), \quad \forall y \in [0, n-1],\ \forall x \in [0, n-3]$$

  Con la restricción escrita de esta manera, si tres posiciones vecinas tienen discos del mismo color, fallará y detendrá el análisis futuro de los otros tríos.

2.2. Implementación del modelo

Utilizamos Python como lenguaje de programación y Python Constraint para implementar y resolver el problema. Repasemos el flujo de trabajo que seguimos.

- ***Leer el archivo de entrada, analizar y validar los datos.***

  Al principio, nos centramos en el procesamiento de una entrada. Para ello, se lee los datos línea por línea del archivo de entrada y se elimina los símbolos de la nueva línea (`\n`) al final de cada una.

  Después, calculamos el número de filas y comprobamos si es par, ya que de lo contrario no podríamos continuar con la resolución del problema. Si el número de

filas es impar, salimos del sistema y no continuamos con el uso de Python Constraint.

También verificamos si el tamaño de la rejilla es realmente cuadrático. De no ser así, salimos del sistema y, como en el caso anterior, no continuamos con la resolución del problema.

Después de eso, decidimos transformar las entradas de la rejilla para que se correspondieran con la lógica de dominio definido previamente. Para ello, en cada línea cambiamos los O (discos blancos) a 0, las X (discos negros) a 1 y los . (espacios vacíos) se marcaron como -1.

- ***Definir funciones para restricciones.***

  Tras el preprocesamiento de los datos, pasamos a definir las funciones que se pasarán al método addConstraint desde la biblioteca Python Constraint en el futuro.

  Al final, solo teníamos dos funciones que se correspondían y utilizaban la misma lógica que las restricciones descritas en el apartado de modelado de CSP.

- ***Python Constraint pipeline.***

  Finalmente, definimos una variable como nuestro problema y creamos variables en las posiciones de la cuadrícula. Posteriormente, creamos restricciones usando las funciones del apartado anterior y, con ello, se construyó el modelo.

  Después, recuperamos soluciones y las visualizamos como se solicitó en el enunciado. Si no se encontraron soluciones, el archivo de salida contendrá una rejilla de entrada y el texto "No se encontraron soluciones". Si se encuentra una solución (puede ser una o más), siempre se mostrará la primera.

2.3. Resolución y análisis de los casos de prueba

- ***Análisis de la complejidad del problema.***

  Analicemos la complejidad del problema analizando el número de variables y restricciones creadas.

  - **Variables**

    Hemos creado *1* variable para cada celda de la rejilla cuadrangular, lo que hace que haya n^2 variables en total, haciéndola de orden $O(n^2)$.

  - **Restricciones**

    En cuanto a las restricciones, las hemos impuesto tanto en filas como en columnas. La restricción que cubre la misma cantidad de discos blancos y negros se invocará *una* vez para toda la fila/columna. Por lo tanto, en total, se invocará $2n$ veces.

    Al mismo tiempo, la restricción que cubre solo dos colores consecutivos se invoca en cada tripleta de la fila/columna, lo que significa que se invocará

$(n-2)$ veces para una fila/columna y $2n(n-2)$ para todas las filas y columnas de la rejilla.

Esto hace que el número total de restricciones que se calcula sea $2n + 2n(n-2) = 2n^2 - 2n$, lo que lo convierte en $O(n^2)$ orden.

- ***Casos de prueba.*** Cubramos todos los casos de uso básicos y aquellos que retan a la implementación de este problema.

  - ***Caso básico de enunciado.*** Primero, verifiquemos si nuestro modelo proporciona el mismo número de soluciones que el propuesto en el enunciado. También comprobaremos la exactitud de la solución elegida.

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

+---+---+---+---+---+---+
|   | X |   |   |   | X |
|   |   | X |   |   | O |
|   | O |   |   | X |   |
| O |   | X |   |   |   |
|   | X |   | O |   |   |
| X |   |   |   |   |   |
+---+---+---+---+---+---+
9 soluciones encontradas
```

parte-1.py input.in example.out

```
1  +---+---+---+---+---+---+
2  |   | X |   |   |   | X |
3  |   |   | X |   |   | O |
4  |   | O |   |   | X |   |
5  | O |   | X |   |   |   |
6  |   | X |   | O |   |   |
7  | X |   |   |   |   |   |
8  +---+---+---+---+---+---+
9  +---+---+---+---+---+---+
10 | O | X | X | O | O | X |
11 | O | X | X | O | X | O |
12 | X | O | O | X | X | O |
13 | O | O | X | X | O | X |
14 | X | X | O | O | X | O |
15 | X | O | O | X | O | X |
16 +---+---+---+---+---+---+
```

Se puede ver que el número de soluciones encontradas es el mismo que en el enunciado, y el resultado que se muestra por la derecha sigue las restricciones definidas.

- ***Caso básico con todas las soluciones.***

  El siguiente ejemplo es una entrada con menor número de soluciones, que podemos analizar manualmente y mostrar todas las soluciones.

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

+---+---+---+---+
|   | X | O |   |
|   |   | X |   |
|   | O |   |   |
| O |   | X |   |
+---+---+---+---+
3 soluciones encontradas
```

parte-1.py input.in example.out

```
+---+---+---+---+
|   | X | O |   |
|   |   | X |   |
|   | O |   |   |
| O |   | X |   |
+---+---+---+---+
+---+---+---+---+
| X | X | O | O |
| O | X | X | O |
| X | O | O | X |
| O | O | X | X |
+---+---+---+---+
| O | X | O | X |
| X | O | X | O |
| X | O | O | X |
| O | X | X | O |
+---+---+---+---+
| X | X | O | O |
| O | O | X | X |
| X | O | O | X |
| O | X | X | O |
+---+---+---+---+
```

Como se puede ver, esta rotación solo tiene 3 soluciones y todas ellas se mostraron correctamente en el fichero de salida.

- ***Caso extremo. No se encuentran soluciones.***

  En esta entrada, ninguna de las filas y columnas contiene rotaciones incorrectas de X y O. Sin embargo, al intentar resolverlo manualmente, observamos rápidamente que la única opción se atasca en una celda inferior izquierda, creando tres O seguidos, lo cual es inaceptable en nuestro problema. Esto también se demuestra en nuestra solución.

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

+---+---+---+---+
| O | O |   |   |
|   |   | X | X |
|   | X |   | O |
| X | O | O |   |
+---+---+---+---+
0 soluciones encontradas
```

parte-1.py input.in example.out

```
+---+---+---+---+
| O | O |   |   |
|   |   | X | X |
|   | X |   | O |
| X | O | O |   |
+---+---+---+---+
No se encontraron soluciones.
```

- ***Caso extremo. Entrada incorrecta [tres discos del mismo color].***

  En este caso nos gustaría cubrir una situación en la que la entrada propuesta ya es incorrecta.

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

+---+---+---+---+
|   | O | X | O |
| O |   |   | X |
| X |   |   | O |
| O | O | O |   |
+---+---+---+---+
0 soluciones encontradas
```

parte-1.py | input.in | example.out

```
+---+---+---+---+
|   | O | X | O |
| O |   |   | X |
| X |   |   | O |
| O | O | O |   |
+---+---+---+---+
No se encontraron soluciones.
```

Como se puede observar no se encuentran soluciones.

- ***Caso extremo. Entrada incorrecta [numero impar de filas/columnas].***
  Con este caso se cubre una situación en la que la entrada tiene una rejilla de tamaño impar (por ejemlo 3x3), lo cual nunca sería satisfacible.

parte-1.py | input.in

```
1  .OX
2  O..
3  X..
```

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

La rejilla debe tener un número par de filas!
```

Como se puede ver, nuestro modelo cubre ese caso extremo correctamente.

- ***Caso extremo. Entrada incorrecta [rejilla no cuadrada].*** En este caso de uso queremos mostrar la forma en que manejamos la rejilla de entrada que no tiene forma cuadrada.

parte-1.py | input.in

```
1  .OX
2  O..
```

```
(venv) anastasiiasadova@Anastasiias-MacBook-Pro parte-1 % python3
parte-1.py input.in example.out

La rejilla debe ser cuadrada!
```

Como se puede ver, no se produce ninguna solución y produce un error.

3. Parte 2: Algoritmos de búsqueda

En esta sección se describe la implementación del algoritmo A* para la resolución del problema de rutas mínimas. El objetivo es encontrar el camino óptimo entre una ubicación inicial y una meta, minimizando el costo acumulado y optimizando el tiempo de cómputo mediante el uso de información heurística. Además también se explicará la implementación empleada para el algoritmo Dijkstra.

3.1. Modelo del problema de Búsqueda Heurística

3.1.1. Espacio de estados

El **espacio de estados** se define como el conjunto de todos los nodos transitables del grafo $G(V, A)$. Un estado S representa la ubicación actual del recorrido, que corresponde a un vértice (o nodo) en el grafo y contiene la información necesaria para la evaluación y reconstrucción de la ruta:

$$S = \{\text{id}, \text{longitud}, \text{latitud}, g, h, f, \text{id}_{\text{padre}}\}$$

Donde:

- **id**: Identificador único del vértice en el grafo.
- **long, lat**: Coordenadas geográficas utilizadas para el cálculo de la heurística.
- **g**: Costo real acumulado desde el estado inicial S_{inicial} hasta el actual.
- **h**: Valor de la función heurística (estimación al objetivo).
- **f**: Función de evaluación total ($f = g + h$).
- **id**_padre: Puntero al estado predecesor para la reconstrucción del camino óptimo.

Estado inicial ($S_{\textbf{inicial}}$): Definido por el vértice de origen, con $g = 0$, $id_{\text{padre}} = -1$ y el cálculo de h correspondiente.
Estado meta ($S_{\textbf{meta}}$): Se alcanza el éxito cuando el nodo extraído de la lista *Abierta* coincide con el identificador de destino, garantizando una solución óptima si la heurística es admisible.

3.1.2. Restricciones y operadores

En este problema, la única transición permitida desde un estado actual (S_{actual}) hacia un nuevo estado (S_{nuevo}) es aquella donde existe un arco directo (Vértice_{actual},Vértice_{nuevo}) definido en el conjunto de aristas del grafo G. Aplicando esta restricción se asegura que todos los estados generados sean válidos. Es decir, se convierte en la precondición del operador del modelo.
Operador de Expansión (expandir_nodos): Es la función encargada de generar el conjunto de sucesores S_i a partir de un estado actual S_{actual}.

- Precondición: Un estado $S_{sucesor}$ es un sucesor válido si y solo si existe un arco directo (Vértice_{actual}, $\text{Vértice}_{sucesor}$) en el grafo G. No se permiten saltos entre nodos no conectados.

3.1.3. Función de coste y función heurística

Función de coste g(n): Representa la suma de los pesos de los arcos desde el origen hasta el nodo n. Es un valor exacto y siempre creciente.

El algoritmo A* utiliza una función de evaluación **f(n)=g(n)+h(n)** por lo que se han explorado varias funciones heurísticas:

Tabla 1: *Comparación de funciones heurísticas (Mapa .BAY, Origen: 1, Destino: 3100)*

Heurística (h)	**Coste Final**	**Expansiones**	**Tiempo (s)**	**¿Es óptimo?**
Dijkstra	687.790	135.192	0,3121	Sí
D. Manhattan	733.781	5.239	0,2154	No
D. euclidiana × 1,0	687.944	24.753	0,2288	No
D. euclidiana × 0,9	687.790	39.352	0,2608	Sí
D. euclidiana × 0,8	687.790	51.054	0,2408	Sí
D. euclidiana × 0,5	687.790	63.375	0,2546	Sí
D. euclidiana × 0,1	687.790	125.872	0,3049	Sí

A partir de los datos experimentales obtenidos, se pueden extraer las siguientes conclusiones técnicas:

1. La **Distancia de Manhattan** es la que menor número de expansiones requiere (5,239), siendo la más rápida en tiempo de ejecución. Sin embargo, el coste final (733,781) es superior al óptimo, lo que confirma que no es una heurística admisible para este grafo. Esto se debe a que sobreestima la distancia real al no permitir desplazamientos diagonales que sí existen en la red de carreteras.
2. En cuando a la **Distancia euclidiana** se han explorado varias variantes:
   - Con α = 1,0, la distancia euclídea pura todavía sobreestima ligeramente el coste real en el mapa .BAY, resultando en una solución no óptima (687,944).
   - Al reducir α a 0,9, el algoritmo alcanza el coste óptimo de Dijkstra (687,790). En este punto, el número de expansiones (39,352) es bastante menor al de Dijkstra (135,192), demostrando la gran eficiencia de A* cuando la heurística es informada y admisible.
   - A medida que reducimos el valor de α (de 0,9 a 0,1), el algoritmo se vuelve "menos informado". Como consecuencia, el número de expansiones aumenta gradualmente ($39k \rightarrow 51k \rightarrow 63k \rightarrow 125k$), relantizando la ejecución. Sin embargo, la solución permanece óptima, cumpliendo con la teoría de que una heurística menor que el coste real ($h(n) < h^*(n)$) siempre garantiza el camino mínimo (es admisible).

Función Heurística h(n): Finalmente, nos decantamos por la **Distancia Euclidiana** multiplicada por $\alpha = 0,5$. Buscamos que sea informada y admisible, por lo que no puede sobrestimar. Si tomáramos un valor menor de $\alpha = 0,5$ sería poco informada y causaría un aumento de tiempo de computación y generar más expansiones.

$$h(x) = \sqrt{(\text{latitud}_n - \text{latitud}_{meta}) + (\text{longitud}_n - \text{longitud}_{meta})}$$

3.2. Modelo del problema de Fuerza Bruta

El algoritmo de Dijkstra es un procedimiento diseñado para encontrar el camino más corto desde un nodo origen al resto de los nodos de un grafo con pesos no negativos. A diferencia de A*, Dijkstra no utiliza información sobre la ubicación del destino; en su lugar, encuentra el camino con menor costo para todos los nodos. Elegimos este algoritmo para fuerza bruta porque es la versión sin búsqueda heurística de A*. Además que garantiza encontrar la solución, si existe (es completo) y encontrará la solución óptima (es admisible).

3.3. Implementación

Todo el código se ha implementado en C++, optando por su capacidad de trabajar a bajo nivel y optimizar el tiempo de ejecución. Tenemos los siguientes archivos que implementan el programa:

- **grafo.cpp** & **grafo.hpp**: Implementan una clase para la lectura de archivos de entrada y almacena el grafo mediante un vector de la Estructura Vértice (con cada vértice y sus coordenadas geográficas) y una lista de adyacencia (vector de vectores de la Estructura Vértice). Contiene otros métodos son auxiliares para comprobar el correcto funcionamiento.

- **abierta.cpp** & **abierta.hpp**: Implementan una clase para la lista abierta del algoritmo A*, que utilizaremos la estructura de datos `std::set` para almacenar los estados por explorar. Se ha implementado un operador de comparación personalizado para ordenar los nodos según su valor f(n), garantizando que el nodo con menor coste estimado siempre sea el primero en ser expandido. Además de los métodos básicos para trabajar con la lista: añadir y eliminar elementos.

- **cerrada.cpp** & **cerrada.hpp**: Implementan la clase para la lista cerrada del algoritmo A*, gestiona en un vector estados explorados. A su vez, para optimizar la búsqueda de nodos para evaluar si se ha encontrado una mejor ruta, se ha implementado un vector indexado por el id del nodo visitado y su valor g(n). Por último, se añade un método para poder buscar nodos en la reconstrucción del camino.

- **algoritmo.cpp** & **algoritmo.hpp**:

- Se ha implementado una clase para el algoritmo A*. Cuenta con métodos para el algoritmo que utiliza las clases ListaAbierta, ListaCerrada y Grafo. Incluye funciones para el cálculo de la heurística y la reconstrucción y escritura del camino óptimo en el fichero.
- Se ha implementado otra clase para el algoritmo Dijkstra, a diferencia de A* este método no utiliza ListaAbierta y ListaCerrada. Se ha empleado una `std::priority_queue` para extraer el nodo con menor coste, y dos vectores auxiliares indexados, el primero almacena el coste mínimo de cada vértice desde el orígen y el otro registra el vértice padre. También incluye una función para escribir la solución en el fichero.
  Mencionar que solo se genera un único fichero de salida, la primera línea se corresponde con el camino de A* y la segunda con el camino de Dijkstra

- **parte-2.cpp**: Actúa como el punto de entrada del programa (`main`). Gestiona el flujo de entrada de datos, la invocación de los algoritmos y la salida de resultados por terminal.

Para compilar hay que utilizar el comando: `g++ parte-2.cpp algoritmo.cpp abierta.cpp cerrada.cpp grafo.cpp -o parte-2`

3.4. Resolución y análisis de los casos de prueba

Para poner a prueba el rendimiento de ambos modelos hemos decidido probarlos en 2 mapas diferentes: USA-road-d.NY y USA-road-d.BAY. Los resultados se pueden ver en la Tabla 1:

Tabla 2: *Comparación de resultados*

Test	mapa	Org → Dest	Algoritmo	Coste Total	Expansiones	Tiempo (s)
1	BAY	1 → 309	A*	10.216	4	0,313387
			Dijkstra	10.216	10	0,313895
2	BAY	321270 → 50976	A*	1.506.939	116.960	0,513263
			Dijkstra	1.506.939	281.823	0,349312
3	BAY	77424 → 285309	A*	2.257.499	253.396	0,798117
			Dijkstra	2.257.499	630.138	0,427228
4	BAY	40397 → 217403	A*	2.605.698	345.936	1,59891
			Dijkstra	2.605.698	799.327	0,450099
5	NY	1 → 264346	A*	750.881	135.174	0,467006
			Dijkstra	750.881	345.320	0,361564
6	NY	178 → 86	A*	42.306	185	0,267494
			Dijkstra	42.306	514	0,2585
7	NY	191500 → 79224	A*	1.246.925	276.289	0,87604
			Dijkstra	1.246.925	716.472	0,383236
8	NY	4362 → 58197	A*	1.291.193	272.484	1,00892
			Dijkstra	1.291.193	675.307	0,373991

*Mapa se refiere a los archivos input que inician con el formato USA-road-d.**

Los tests 1 y 6 prueban que los algoritmos funcionan en rutas más cortas y sin tanta complejidad. Para poder poner al límite nuestra implementación decidimos desarrollar una función que calcula los puntos más extremos basándose en la latitud y longitud, con los tests 3, 4, 7 y 8 pusimos a prueba puntos extremos en el mapa. Los test 2 y 5 fueron seleccionados aleatoriamente.
Podemos sacar varias conclusiones de estas pruebas:

1. El tiempo de ejecución aumenta de forma proporcional al número de nodos expandidos. En Dijkstra se exploran más nodos pues explora en todas las direcciones por lo que aumenta el tiempo pues debe procesar gran parte de los vértices del grafo. De la misma manera sucede con A*. Sobre el crecimiento de las estructuras de datos

   - En Dijkstra las estructuras auxiliares (distancias y predecesores) empleadas tienen un tamaño constante pues son vectores indexados, sin embargo la cola de prioridad aumenta según se va explorando el grafo.
   - En A* el tamaño de ListaCerrada y ListaAbierta va aumentando con el número de nodos expandidos, Lista Cerrada aumenta de manera monótona pero Abierta linealmente.

2. El uso de una función heurística tiene un gran impacto en número de nodos expandidos. Se observa en todos los tests que en Dijkstra se expande más del doble de nodos. Tomemos el como ejemplo el test 7 donde A* tiene 276.289 expansiones frente a 716.472 de Dijkstra, esto supone un 61 % menos de expansiones para encontrar la solución. Mientras que Dijkstra explora toda la supercie, A* con la función heurística consigue acercarse de manera más eficiente al destino.

3. Algo interesante tras los test son los resultados del tiempo. El ejemplo perfecto es el test 4, donde vemos que A* tarda 1,5s, en cambio Dijkstra tarda un tercio de eso (0,45s). El algoritmo Dijkstra es más rápido que A*, a pesar de que hace más expansiones y consume más memoria. Hay otros factores en juego, en A* se realizan cálculos más complejos en todos los nodos expandidos para calcular h(n). Cabe mencionar que en el tiempo está incluido la carga del grafo, lo que explica los tiempos tan altos en casos simples.

Dijkstra resulta más eficiente en esta implementación específica debido a la ligereza de sus estructuras de datos, pero A* demuestra una eficiencia algorítmica superior al reducir drásticamente el número de nodos expandidos, lo cual es crítico en grafos de mayor escala (como el mapa completo de USA).

4. Conclusiones

Esta práctica nos ayudó a mejorar nuestro conocimiento sobre la forma en que el modelado de CSP y la búsqueda heurística funcionan. Fue muy desafiante, pero por otro lado, muy útil para entender cómo funciona la biblioteca Constraint en Python y cómo se puede aplicar a casos prácticos. Desarrollar la segunda parte en C++ ha sido un reto, pues no tenemos mucha experiencia con el lenguaje. Sin embargo, la eficiencia del mismo para calcular los resultados ha sido un beneficio para poder desafiar nuestras implementaciones al límite.