UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Algoritmos heurísticos para procesos de decisión de Markov

Heuristic algorithms for Markov Decision Processes

Realizado por
Antonio Ruiz Valverde

Tutorizado por
José Luis Pérez de la Cruz Molina

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2022

Resumen

En la actualidad se conocen un gran cúmulo de aplicaciones y prestaciones para la inteligencia artificial, es innegable la influencia residente en este campo en lo relativo al futuro de la informática. Pero hay que puntualizar que no todos los problemas son iguales, y, por lo tanto, tampoco la metodología más precisa para resolverlos.

En este TFG estudiaremos concretamente un área del aprendizaje automático, conocida como aprendizaje por refuerzo, que trata de optimizar la toma de decisiones mediante la maximización de una recompensa o la minimización de un coste, a través de los procesos de decisión de Markov. Como veremos adelante, este sistema está basado en la idea de que, en problemas concretos, no podemos tener en cuenta la toma de decisiones como un proceso riguroso y sin fallos, si no que el proceso transitorio que arraiga en la toma de decisiones también puede llevar a error. Por ejemplo, aunque el algoritmo de un sistema autónomo móvil decida moverse en una dirección concreta para alcanzar su destino, es necesario tener en cuenta factores ambientales que sugieren un margen de error en el movimiento.

El estudio se llevará a cabo mediante la implementación de una serie de algoritmos heurísticos para procesos de decisión de Markov, y, posteriormente, se hará una comparativa de estos algoritmos a través de la definición de una serie de casos de prueba con el fin de comprobar sus prestaciones y obtener conclusiones acerca de la optimalidad en su uso.

En la memoria se detallará la teoría detrás de los procesos de Markov, así como la referente a los algoritmos implementados con pseudocódigo incluido. También se graficará los datos obtenidos en la aplicación de los algoritmos sobre los casos de prueba y se facilitará una copia del código para el interesado.

Palabras clave: **Aprendizaje automático, heurístico, proceso de decisión de Markov, casos de prueba, aprendizaje por refuerzo.**

Abstract

Nowadays, a lot of applications and benefits of artificial intelligence are well-known, we cannot deny about the influence of this field on the future of computing. But it must be pointed out that not all problems are the same and, therefore, neither is the most precise methodology to solve them.

In this TFG we will study an area of machine learning, known as reinforcement learning, which tries to optimize the decision-making by maximizing a reward or minimizing a cost, through Markov decision processes. As we will see later, this system is based on the idea that, in specific problems, we cannot see decision-making as a rigorous and error-free process, as the transitory process related to the decision-making can also lead to error. For example, even if the algorithm of a mobile autonomous system computes the move in a certain direction, it is necessary to consider environmental factors that suggest a margin of error in the movement.

The study will be carried out through the implementation of a series of heuristic algorithms for Markov decision processes, and subsequently, a comparison of these algorithms will be made through the implementation of a series of test cases to verify its benefits and obtain conclusions about the optimality in its use.

The memory will detail the theory behind the Markov processes, as well as the theory related to the algorithms implemented with pseudocode included. The data obtained in the application of the algorithms on the test cases will also be graphed and a copy of the code will be provided.

Keywords: **Machine learning, heuristic, Markov decision process, test case, reinforcement learning**

Índice

1. Introducción

1.1 Motivación

Como se ha mencionado anteriormente, es destacable la importancia que tiene el campo de estudio de la Inteligencia Artificial en la actualidad, no solo por la cantidad de aplicaciones y posibilidades que ofrece sino también por la gran importancia que supone en el desarrollo de nuevas tecnologías en un amplio abanico de disciplinas: Finanzas, medicina, sistemas de conducción autónoma de vehículos, industria, juegos, arte, etc.

Este TFG está motivado por el estudio de un campo subyacente al de la inteligencia artificial que tiene como fin el desarrollo de algoritmos que facilitan la toma inteligente de decisiones mediante el análisis de patrones encontrados en los datos de entrada, el Machine Learning. Dentro del Machine Learning podemos diferenciar tres tipos de aprendizaje:

- **Aprendizaje supervisado.** Cuentan con una fase de aprendizaje previo donde se establecen ciertos patrones matemáticos que permiten clasificar los datos en función de datos que ya habían sido clasificados anteriormente. Un uso extendido es para detectar si un correo es spam; en este caso, el algoritmo aprende en función de correos que han sido clasificados como spam anteriormente.

- **Aprendizaje no supervisado.** Tratan de clasificar los datos reconociendo patrones en ellos sin haber realizado un estudio previo de datos de entrada ya clasificados. Una de sus aplicaciones es el Marketing, donde se utilizan para reconocer patrones de datos de las redes sociales; esto puede resultar de gran utilidad en el desarrollo de trabajos de investigación o en ámbitos empresariales mediante las campañas de Marketing.

- **Aprendizaje por refuerzo.** Su finalidad es la de realizar la toma de decisiones mediante un proceso matemático de prueba y error en el que se trata de maximizar una recompensa o, en su defecto, de minimizar un coste para encontrar la solución óptima a un problema. Es la que supone mayor similitud con el sistema de aprendizaje humano. Algunas de sus aplicaciones residen en la medicina, puesto que se utilizan en la elaboración de diagnósticos médicos y en la clasificación de secuencias de ADN. También es utilizado en sistemas de reconocimiento facial.

Cada tipo de aprendizaje posee unas características determinadas, y, aunque la intersección de sus aplicaciones no sea el vacío, ya que es posible el uso de distintos tipos de aprendizaje en problemas similares, algunos se adaptan mejor que otros en función de la naturaleza del problema en cuestión. Por ello, no podemos negar la importancia de cada método de aprendizaje dentro del Machine Learning, y, por consiguiente, en el terreno de la inteligencia artificial. El estudio realizado se basa concretamente en el tercer sistema de aprendizaje, el aprendizaje por refuerzo. [1]

1.2 Objetivos

Para este estudio, se realizará la implementación de cinco algoritmos con aplicaciones en el aprendizaje por refuerzo para procesos de decisión de Markov, concepto que se definirá más adelante. Posteriormente, se definirá una serie de casos de uso y se ejecutará cada uno de los algoritmos sobre éstos; en la salida generada, se obtendrán datos relacionados con cada uno de los algoritmos (tiempo utilizado, iteraciones requeridas, estados expandidos, etc). Se variará la complejidad de cada uno de los problemas para establecer una relación entre parámetros de entrada asociados al problema y la salida generada en función al algoritmo utilizado. ¿Cuánto tarda el algoritmo en resolver el problema en función de la complejidad?, ¿Cuántas iteraciones requiere?, …

Finalmente, se graficará los datos obtenidos para sacar conclusiones acerca de cada uno de los algoritmos en lo relativo a las características de cada problema. Mediante este estudio, podremos distinguir las prestaciones de cada uno de los algoritmos en lo que refiere a su optimalidad para resolver un problema en concreto en función de sus características, con el fin de inferir qué algoritmo podría ser más eficiente usar en función del problema que se quiera resolver y por qué.

Para asegurar la rigurosidad de los resultados, cada instancia de un mismo problema será resuelta un número considerable de veces con cada algoritmo para evitar datos desechables a causa de factores externos como el uso excesivo de recursos de la máquina utilizada para generar los datos.

1.3 Tecnologías utilizadas

Estas son las herramientas que se han utilizado para el desarrollo del TFG:

- **Python.** Es un lenguaje de programación de alto nivel y código abierto. Se trata de un lenguaje interpretado por lo que no es necesario realizar el proceso de compilación para poder ejecutar el código, ya que el intérprete traduce las líneas de código de alto nivel a lenguaje máquina en tiempo de ejecución. Implementa los paradigmas imperativos, orientado a objetos y funcional. Su sintaxis simple e intuitiva y la gran cantidad de librerías que ofrece lo convierte en uno de los lenguajes predilectos a la hora de trabajar con Machine Learning, Big Data y Data Science. La programación del código se ha realizado en la versión 3.9.13 y se ha hecho uso de la librería *time* incluida en la propia versión la cual permite gestionar datos relativos a tiempo y fecha, y, por lo tanto, nos brinda la posibilidad de generar información relativa a los tiempos de computación. [2]

- **Git.** Es un software de código abierto destinado al control de versiones, es decir, ofrece herramientas para la gestión de las modificaciones en nuestro código y permite ramificar nuestro código con la posibilidad de destinarlo a distintas funciones, como testing, refactorizaciones[1], generación de datos, etc. Es fácil de usar

[1] método de ingeniería de software para reestructurar el código sin modificar su funcionamiento, con el fin de optimizarlo y obtener un código bien estructurado y legible

y permite el desarrollo de software de forma escalonada y estructurada. El repositorio del código implementado se encuentra en la plataforma GitHub[2]. [3]

- **Visual Studio Code.** Es un entorno de desarrollo y editor de código gratuito. Fácil de usar y con la ventaja de que posibilita trabajar fácilmente con una gran cantidad de lenguajes de programación, entre ellos, Python. Además, ofrece un repertorio amplio de extensiones instalables para sintaxis, depurado, control de versiones… Incluye herramientas para el control de repositorios Git y soporte para la depuración[3] de código. [4]

1.4 Metodología utilizada

La metodología utilizada en el desarrollo del TFG ha sido de prototipado. La metodología de prototipado se basa en un proceso iterativo en el que se establecen una serie de requisitos a implementar, se construye el prototipo con la implementación de esos requisitos y posteriormente se hace una evaluación del prototipo para mejora.

En el caso de este proyecto, la organización se ha realizado en torno a reuniones semanales donde se hacía una definición de las tareas a realizar, y, en la reunión de la semana subsiguiente, se hacía una evaluación de la evolución del proyecto durante la semana para establecer las nuevas funcionalidades a implementar o especificar las características concretas que necesitan corrección.

[2] plataforma de alojamiento de repositorios Git

[3] proceso de ingeniería del software en el que se estudia el código para la identificación y corrección de errores

2. Procesos de decisión de Markov

Un proceso de decisión de Markov, MDP abreviado, está compuesto por:

- Un conjunto de estados S. Cada estado representa una situación concreta del problema a resolver.

- Un conjunto de acciones A. Las acciones permiten transitar entre estados. Cada estado está asociado a un subconjunto de acciones que son las que se pueden realizar desde ese estado.

- Una función de probabilidad p. La probabilidad $p_{ij}(a)$ denota la probabilidad de transitar al estado *j* si aplicamos la acción *a* desde el estado *i*

- Una función de coste c. El coste $c_i(a)$ especifica el coste asociado a realizar la acción *a* desde el estado *i*.

Nuestro objetivo será concretamente abordar una clase de procesos de decisión de Markov, que generaliza los problemas tradicionales de *camino más corto*, es decir, una serie de problemas que son representados mediante grafos cuyos nodos refieren a estados del problema y los cuales definen un estado inicial y uno o más estados objetivos, y cuya finalidad es encontrar el camino más corto entre el estado inicial y un estado objetivo mediante la aplicación de algoritmos de Inteligencia Artificial. La implementación de problemas de *camino más corto* sobre procesos de decisión de Markov se denomina *problemas estocásticos de camino más corto* ya que los MDP son procesos estocásticos, en otras palabras, son procesos que suponen un comportamiento no determinista, aleatorio, regido por la probabilidad.

En un problema estocástico de camino más corto existe un conjunto de estados terminales $T \subseteq S$. La probabilidad de transitar desde un estado terminal a cualquier otro estado realizando cualquier acción es de 0, al igual que el coste de realizar cualquier acción desde el propio estado terminal. El objetivo es alcanzar un estado terminal desde el estado inicial incurriendo en el mínimo coste, lo cual equivaldría a una solución óptima al problema.

Durante el procedimiento de resolución cada estado del problema estará asociado con un valor, definida por una función de evaluación, y una acción, definida por la política.

Una política es una función $\pi: S \rightarrow A$ que asocia a cada estado una acción. Una vez resuelto el problema, se utiliza para representar las soluciones al problema conceptualmente señalando la acción que se debe realizar desde cada estado para alcanzar un estado final desde ese estado incurriendo en el mínimo coste posible.

La función de evaluación es una función $f: S \rightarrow R$ que, dado un estado s, devuelve un valor numérico real que indica el coste esperado de alcanzar un estado objetivo desde el estado s. Se calcula en función de la política. Para una política dada, la función de evaluación se calcula resolviendo el siguiente sistema de |S| ecuaciones con |S| incógnitas.

$$f^{\pi}(i) = \begin{cases} 0 & \text{Si i es un estado terminal,} \\ c_i(\pi(i)) + \sum_{j \in S} p_{ij}(\pi(i)) f^{\pi}(j) & \text{Si no lo es} \end{cases}$$

Fórmula 1. Función de evaluación

Se dice que una política π domina a una política π' cuando se cumple la igualdad $f_{\pi}(i) \leq f_{\pi'}(i)$ para cualquier estado i. Una política es óptima cuando domina a cualquier otra política y su función de evaluación, denotada f^* cumple la ecuación de Bellman de optimalidad, descrita de la siguiente manera:

$$f^*(i) = \begin{cases} 0 & \text{Si i es un estado terminal,} \\ \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^*(j) \right] & \text{Si no lo es} \end{cases}$$

Fórmula 2. Ecuación de Bellman de optimalidad

El valor asociado a un estado terminal siempre va a ser 0. En el caso de un estado no terminal, la evaluación de la política óptima minimiza el valor asociado a cada estado en función de la acción que la política dicte para ese estado.

Para mejorar una función de evaluación estimada en función de las posibles acciones a realizar, se realiza lo que se conoce como actualización de Bellman (*Bellman backup*). Para un estado i, la fórmula de un *Bellman backup* es la siguiente:

$$f(i) := \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f(j) \right].$$

Fórmula 3. Bellman Backup

Los algoritmos de programación dinámica usados en los procesos de decisión de Markov (los cuales estudiaremos en el próximo apartado) aplican esta fórmula sobre el conjunto de estados de forma iterativa hasta converger en la política óptima y en su función de valor asociada. [5]

3. Programación dinámica aplicada a MDP

Existen dos algoritmos característicos usados en programación dinámica, los cuales son iteración de políticas e iteración de valores.

3.1 Iteración de políticas

Consiste en un proceso iterativo que empieza con una política arbitraria para el conjunto de estados del problema a resolver y consta de dos pasos.

- **Evaluación.** Cálculo de la función de evaluación asociada a la política π resolviendo el sistema de |S| ecuaciones con |S| incógnitas especificado anteriormente.

$$f^{\pi}(i) = \begin{cases} 0 & \text{Si i es un estado terminal,} \\ c_i(\pi(i)) + \sum_{j \in S} p_{ij}(\pi(i)) f^{\pi}(j) & \text{Si no lo es} \end{cases}$$

Fórmula 1. Función de evaluación

- **Mejora.** Obtiene la mejor política asociada a la función de valor actual, sin actualizar la función de valor.

$$\pi'(i) := \arg \min_{a \in A(i)} \left[c_i(a) + \sum_{j \in S} p_{ij}(a) f^{\pi}(j) \right]$$

Fórmula 4. Mejora de política

Este proceso iterativo de evaluación y mejora de política asegura que, tras un determinado número de iteraciones, el algoritmo converja en la política óptima. Esto es así ya que el número de posibles políticas para un conjunto de estados S finito es también finita, y en cada paso el algoritmo mejora la política actual, hasta llegar a convergencia, lo que implicaría que en ese paso la política no cambia. Como en el último paso, el paso en el que ocurre la convergencia, la política no se ha modificado, se infiere que la política no se puede mejorar, por lo tanto, tenemos una política óptima. El pseudocódigo del algoritmo aparece especificado en la siguiente tabla:

Tabla 1. Iteración de políticas

1. Inicializar π de forma arbitraria.
2. Inicializar π'
3. Mientras π sea distinto a π'
3.1. π ← π'
// Paso de evaluación. Se aplica la fórmula de evaluación de política sobre π
3.2. evaluar_Politica(π)
// Paso de mejora. Se obtiene la mejor política para la función de valor actual
3.3. mejorar_Politica(π)
4. Devolver π

De esta forma, lo que se hace en cada paso es una copia de la política actual, la cual se guarda en una variable distinta, posteriormente se procede a aplicar el proceso de evaluación y mejora sobre la política actual y, finalmente, si la política se ha modificado con el proceso de evaluación y mejora, seguimos iterando; de lo contrario, implicaría que hemos llegado a convergencia y que la política no se puede mejorar, por lo tanto, la política π es óptima.

3.2 Iteración de valores

Este algoritmo simplifica el proceso descrito en iteración de políticas mediante la sola aplicación de *Bellman Backups* de forma iterativa. En cada iteración, el procedimiento mejora la función de evaluación actual mediante la ejecución de un *backup*. En este caso, la convergencia no se alcanza en el momento en el que la política no se modifica, si no cuando la función de valor no se puede mejorar en mayor medida que un margen de error *ε,* en ese caso, diremos que hemos alcanzado una política *ε*-óptima. El error en la función de valor se calcula según la siguiente fórmula:

$$r = \max_{i \in S} \left| f'(i) - f(i) \right|$$

Fórmula 5. Error en la función de evaluación

Conceptualmente es la máxima diferencia de valores entre el mismo estado para la función de evaluación anterior *f'* y la función de evaluación actual *f*.

Cuando la función de valor no se pueda mejorar en mayor medida que el margen de error que decidamos, entonces será cuando el algoritmo converja. A mayor margen de error, más rápido convergerá el algoritmo, pero menos certeza tendremos de que el algoritmo haya convergido en la solución óptima. El pseudocódigo de iteración de valores aparece especificado en la siguiente tabla:

Tabla 2. Iteración de valores

```
1. Inicializar función de evaluación y parámetro ε
2. f' ← f
// Mejorar función de valor
3. Para cada estado i en S
        3.1. f ← bellman_Backup(i)
// Calcular error
4. error ← calcular_error(f, f')
5. Si error > ε ir al paso 2
6. Extraer política ε-óptima y devolver
```

Al acabar el algoritmo, cuando se obtiene una función de valor que no se puede mejorar en mayor medida que ε, es necesario extraer la política ε-óptima. Para ello, solo hay que aplicar el paso de mejora descrito en iteración de políticas sobre la función de valor óptima.

Para las pruebas que se han hecho en la implementación de los algoritmos, se ha utilizado un valor $\varepsilon = 0.01$.

En este proyecto, solo se ha realizado la implementación del algoritmo iteración de valores, porque, aunque el algoritmo de iteración de políticas se adapte mejor a la condición de optimalidad (ya que su naturaleza le permite computar siempre una política óptima), iteración de valores es computacionalmente más eficiente, ya que supone una complejidad de orden cuadrático, en comparación con iteración de políticas, que asume una complejidad de orden cúbico. [5]

4. Representación del problema

Cuando tratamos con problemas de camino más corto en un ámbito determinista, también tenemos un conjunto de estados, un conjunto de acciones que causa transiciones entre estados, y una función de coste que especifica el coste requerido de transitar de un estado a otro al aplicar una acción concreta desde el estado origen. Los algoritmos utilizados en este ámbito son bien conocidos: Dijkstra, búsqueda primero en profundidad, búsqueda primero en amplitud, A*… Cuando aplicamos estos algoritmos en problemas de búsqueda del camino más corto (shortest-path problem), se aplican sobre una representación del problema en forma de grafo, en el que los nodos del grafo representan los estados y las aristas representan las transiciones de un estado a otro, así, las aristas pueden tener asociada un coste y una acción que provoca la transición, y el objetivo es encontrar el camino más corto o el que incurra en un menor coste (en caso de que se estén utilizando costes) desde un estado inicial hasta un estado objetivo.

Aunque en este caso no trabajemos en un proceso determinista, si no estocástico, como son los procesos de decisión de Markov, esta es una condición que se mantiene; los algoritmos de búsqueda del camino más corto para procesos estocásticos también reciben como entrada una representación del problema en forma de grafo, concretamente, en una generalización de estos, llamadas hipergrafos.

4.1 Hipergrafos

Un hipergrafo está compuesto por un conjunto de nodos, habitualmente etiquetados, conectados mediante un conjunto de aristas; la diferencia de estos con los grafos convencionales es que una misma arista puede conectar a un nodo origen con un conjunto de nodos destino, y no solamente a un nodo destino. Estas aristas reciben el nombre de hiperarista, hiperarco o k-conector, donde k indica el número de estados sucesores para esa arista.

En este caso, un nodo representaría un estado y las transiciones serían representadas mediante hiperarcos, pero, como ya sabemos, al ser un proceso estocástico, el realizar una acción desde un nodo origen supone transitar a uno de los posibles nodos destino (con mayor o menor probabilidad), en otras palabras, cada nodo, para una acción concreta, puede tener más de un sucesor.

En resumidas cuentas, un hiperarco está asociado con un nodo origen *s*, que representa el estado desde el que se transita, una acción *a*, la cual provoca la transición, un coste *c*, el cual refiere al coste de realizar la acción *a* desde el estado *s* y un conjunto de nodos destino, los cuales representan al conjunto de estados sucesores a los que se pueden transitar al realizar la acción *a* desde el estado *s*, cada sucesor asociado con una probabilidad, que es la probabilidad de transición, formalmente $p_{ij}(a)$

Veamos un ejemplo simple:

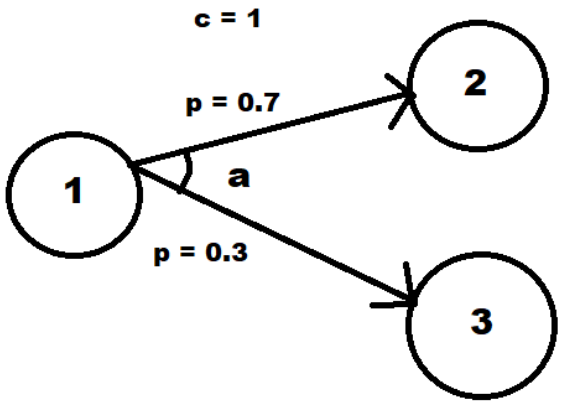

Figura 1. Ilustración de un hiperarco con 3 nodos

En este ejemplo podemos ver un hiperarco en el que el nodo origen es el nodo 1, y mediante la realización de la acción a desde el estado representado por el nodo 1, es posible transitar a dos estados, el representado por el nodo 2, con probabilidad 0.7, y el representado por el nodo 3, con probabilidad 0.3. El coste asociado de realizar la acción desde el estado 1 es de 1.

Una vez que la búsqueda en grafo ha terminado, se genera lo que se conoce como hipergrafo solución. Un hipergrafo solución cumple las siguientes características:

- El estado inicial se encuentra en el grafo hipergrafo solución.
- De cada estado que se encuentra en el hipergrafo solución sale exactamente un hiperarco, que se corresponde con una acción (la acción óptima a realizar desde el estado origen del hiperarco), y cada uno de los sucesores también se encuentran en el hipergrafo solución.
- En el hipergrafo solución, todo camino dirigido acaba en un estado objetivo.

El objetivo, por lo tanto, será el de, mediante la aplicación de distintos algoritmos, actualizar los valores asociados a los nodos, así como la política hasta generar un grafo solución al problema generado. Como una política óptima asegura alcanzar el estado objetivo desde cualquier otro estado incurriendo en el mínimo coste, y, teniendo en cuenta que los caminos en un hipergrafo solución están dirigidos por la política, el hipergrafo asociado a la política óptima es solución (en caso de que la haya). [5]

5. Algoritmos heurísticos para MDP

Como ya se ha especificado, en los problemas de búsqueda del camino más corto para procesos de decisión de Markov, tenemos un estado inicial y al menos un estado objetivo. La aplicación de los algoritmos de iteración de valores e iteración de políticas a estos problemas aseguran siempre encontrar una solución, pero no necesariamente de la forma más eficiente; esto es así ya que son algoritmos que requieren computar sobre todo el espacio de estados. Esto puede resultar ineficaz teniendo en cuenta que buscamos un camino óptimo desde un estado inicial a un estado final, y solo un subconjunto de estados en el espacio de estados puede ser alcanzable desde el estado inicial; esto implica que, para los algoritmos de programación dinámica estaríamos haciendo cálculos sobre estados que nunca pueden pertenecer a la solución.

Ahora vamos a estudiar una serie de algoritmos para SSP (Stochastic shortest-path problem) que, además de no requerir visitar todo el espacio de estados para computar una solución óptima, hacen uso de los heurísticos.

Los heurísticos son métodos cuya finalidad es la de proporcionar más información sobre el problema con el objetivo de encontrar una solución en el espacio de estados de forma más eficiente. Concretamente, en los algoritmos que han sido implementados, sirven como guías que indican qué estados son más prometedores, acelerando el proceso de búsqueda. Como ya hemos comentado, en los procesos de decisión de Markov existe una función de valor que asocia a cada estado un valor que se define como el coste esperado que requiere un camino desde ese estado hasta un estado objetivo siguiendo una política determinada (la política asociada a esa función de valor). Pues bien, para hacer uso de la heurística, utilizaremos las funciones heurísticas. Una función heurística es una función de evaluación que inicializa los valores de los estados, es decir, les da un valor antes de que sean visitados por primera vez por el algoritmo que se esté utilizando. Los valores de la función heurística pueden concebirse como el coste mínimo esperado requerido para alcanzar un estado objetivo desde cada estado. De esta forma, tenemos una estimación inicial de cómo de prometedor puede ser un estado.

Los efectos que tienen el uso de una función heurística adecuada es reducir en gran medida el número de estados que son necesarios expandir antes de encontrar una solución al problema, de manera que solo se computa una fracción más reducida del espacio de estados.

Un heurístico es admisible cuando se cumple que para todo estado s, $h(s) \leq V^*(s)$, en otras palabras, la estimación del coste requerido para alcanzar un estado objetivo desde el estado s no puede superar al coste mínimo real. Una forma de verlo es que la función heurística debe suponer una estimación optimista, ¿cuál es el mínimo coste que sé que va a requerir alcanzar un estado objetivo desde el estado actual?

Cabe destacar la importancia de que, aunque el heurístico tenga que ser admisible para que funcione adecuadamente, es necesario que sea capaz de proveer la mayor cantidad de información posible. En el hipotético caso de que usemos costes positivos, el heurístico cero (heurístico que inicializa los valores de todos los nodos a 0) es admisible, porque no sería posible alcanzar un estado objetivo desde cualquier otro estado con coste menor a 0, por lo tanto, sería válido, pero no sería útil, ya que no nos ofrece ninguna información sobre el problema; según esa función heurística, todos los nodos son igual de prometedores.

Ahora vamos a estudiar una serie de algoritmos de búsqueda heurística para problemas estocásticos de camino más corto.

5.1 LAO*

La idea principal de LAO* es la de buscar estados para expandir dentro de un subconjunto restringido del espacio de estados, este subconjunto de estados está formado por los estados del grafo "solución parcial". Anteriormente se ha especificado que un grafo solución es aquel que contiene el estado inicial, en el que todo estado tiene exactamente un hiperarco que sale de él (el hiperarco asociado a la mejor acción a realizar desde ese estado), en el que todos los sucesores de ese hiperarco también se encuentran en el grafo solución y en el que cualquier camino dirigido desemboca en un estado objetivo. Pues bien, un grafo solución parcial cumple las mismas características, con la diferencia de que todo camino puede desembocar o en un estado objetivo o en un estado no objetivo pero que aún no han

sido expandido; podría ocurrir incluso que no hubiera ningún estado objetivo en el grafo solución parcial. Además, LAO* mantiene un supergrafo del grafo solución parcial llamado grafo *envelope* o cubierta; este grafo contiene todos los estados que alguna vez han formado parte del grafo solución parcial en iteraciones anteriores.

En cada iteración, LAO* actualiza los valores de los estados, así como la política. Al modificar la política, el grafo solución parcial cambia, ya que éste se construye en función de la política de la siguiente forma:

- El estado inicial se introduce en el grafo solución parcial. Se añade la hiperarista cuyo nodo origen sea el estado inicial y cuya acción asociada sea la que dicte la política actual para ese estado. A esa acción la denominaremos acción voraz (*greedy*).
- Recursivamente se aplica la misma lógica con todos los sucesores *greedy* del estado inicial, es decir, con todos los sucesores de la hiperarista con origen en el estado inicial y asociado a su acción *greedy*.
- Si un estado no ha sido expandido aún o es un estado terminal, se añade el estado al grafo solución parcial pero no se añade ningún hiperarco que sale de él. Si consideramos la construcción del grafo solución parcial como la construcción de un árbol cuya raíz es el estado inicial, estos serían los nodos hoja. Como ya hemos comentado antes, todo camino en el grafo solución acaba en un estado no expandido o en un estado terminal. [5]

Tabla 3. Pseudocódigo para construir grafo solución

```
// La primera llamada se hace con el estado inicial
function construir_grafo_solucion(estado, politica, grafo):
// Añadimos el estado el grafo
1. grafo.add(estado)
// Si el estado ha sido expandido y no es terminal
2. Si estado ha sido expandido y estado no es terminal
// Añadimos el hiperarco asociado a su mejor acción con origen en el propio estado
    2.1. grafo.añadir_hiperarco(estado, politica(estado))
// Para cada uno de los sucesores greedy del nodo
    2.2. Para s en estado.sucesores(politica(estado))
// Si el sucesor no se encuentra ya en el grafo
        2.2.1. Si s no está en grafo.estados()
// Llamada recursiva sobre el sucesor
            2.2.1.1. grafo ← construir_grafo_solucion(s, politica, grafo)
3. Devolver grafo
```

En cada paso, los valores de los nodos que han sido expandidos, y, por lo tanto, han sido actualizados, tenderán a aumentar (ya que se inicializan al heurístico, que es una estimación optimista). El efecto que esto provoca en el algoritmo es que prioriza expandir áreas del espacio de estados que no han sido visitadas y que son prometedoras según la función heurística para encontrar el camino óptimo de forma tentativa. Es decir, durante la ejecución, el algoritmo prueba distintos caminos que pueden ser óptimos hasta encontrar el óptimo.

Eventualmente, LAO* acaba con una política en el que todos los estados del grafo solución han sido expandidos y no se puede mejorar. Esta política es óptima.

LAO* empieza con el estado inicial, y construye sucesivamente el grafo cubierta añadiendo nuevos estados al grafo. El conjunto de estados del grafo cubierta está dividido en estados interiores y estados *fringe*. El conjunto de estados interiores está formado por los estados

que ya han sido expandidos, y los estados fringe son un subconjunto de estados que no han sido expandidos y que son nodos sucesores de estados interiores, por lo tanto, también se encuentran en el grafo cubierta. LAO* también almacena el grafo solución parcial, el cual se modifica en cada iteración a causa de la actualización de la política.

Cuando LAO* empieza a computar, el grafo cubierta solo contiene el estado inicial, y, al no haber sido expandido todavía, es un estado fringe. En cada iteración, LAO* escoge un estado fringe que se encuentra en el grafo solución parcial para expandir. Expandir un estado implica añadir todos sus sucesores bajo cualquier acción al grafo cubierta, excepto por aquellos que ya estén; por consiguiente, los sucesores también deben ser añadidos al conjunto de estados fringe. El estado que se expande puede escogerse arbitrariamente, pero también existe la posibilidad de dar prioridad a algunos estados a la hora de seleccionar cuál se va a expandir con el fin de expandir primero los estados más prometedores, en otras palabras, los que tengan un menor valor heurístico; esto puede mejorar la eficiencia del algoritmo. Una vez expandido, un estado fringe pasa a ser un estado interior. Posteriormente LAO* actualiza el estado expandido *s*, así como cualquier estado desde el cual haya un camino hasta *s* siguiendo las acciones greedy dictaminadas por la política actual. Este conjunto de estados, llamado Z, es el conjunto de estados cuyos valores pueden haber cambiado a causa de la expansión. Para la actualización de los estados del conjunto Z se puede usar tanto iteración de valores como iteración de políticas. En nuestro caso, se ha utilizado iteración de valores porque, como se ha explicado antes, es computacionalmente más eficiente. Finalmente, se reconstruye el grafo solución parcial ya que puede haber cambiado a causa de la actualización de la política. Este proceso acaba cuando el grafo solución parcial no tenga más estados fringe que no sean objetivo.

Tabla 4. Pseudocódigo LAO*

```
// Se inicializa la función de valor V, el conjunto de estados fringe F, el conjunto de estados interiores I,
// el grafo cubierta y el grafo solución parcial
1. V ← heuristico
2. politica ← π0
3. F ← {estado_inicial}
4. I ← {}
5. grafo_cubierta ← {estados: {estado_inicial}, hiperarcos: {}}
6. grafo_solucion ← {estados: {estado_inicial}, hiperarcos: {}}
7. Mientras F.interseccion(grafo_solucion.estados()) tenga estados no terminales
        7.1. s ← estado no terminal de F.interseccion(grafo_solucion.estados())
        7.2. F.eliminar(s)
// Se añaden al conjunto F todos los sucesores del estado s que no estén en el conjunto de estados
// interiores
        7.3. F.union(s.sucesores() - I)
        7.4. I.añadir(s)
        7.5. grafo_cubierta ← {nodos: I.union(F), hiperarcos: todas las acciones desde todos los estados de I}
        7.6. Z ← {s y todos sus antecesores en el grafo cubierta siguiendo las acciones greedy}
// Se actualizan el valor de los estados y la política actual
        7.7. Ejecutar Iteración de Valores sobre Z
// El grafo solución se construye recursivamente partiendo desde el estado inicial
        7.8. grafo_solucion ← construir_grafo_solucion(estado_inicial, politica, grafo)
8. Devolver política óptima
```

El conjunto Z también se puede construir recursivamente de forma similar al grafo solución. Partiendo desde el estado recién expandido *s*, y buscando los antecesores de *s* según la política greedy en el grafo cubierta; posteriormente se añade cada antecesor al conjunto Z y, de forma recursiva, se aplica el mismo procedimiento para ese antecesor. [6]

5.2 ILAO*

La desventaja respecto a la eficiencia de LAO* es la gran cantidad de estados que tiene que evaluar mediante programación dinámica. Aunque el algoritmo en sí no requiera evaluar todo el espacio de estados, ocurre que hay una cantidad considerable de estados que van a ser evaluados más de una vez. Recordemos que, cada vez que se expande un estado, es necesario actualizar un subconjunto de los estados que ya han sido expandidos, que son los estados cuyos valores pueden haber sido modificados a causa de la expansión del nuevo estado. El tiempo de cómputo requerido por LAO* aumenta según lo hace el número de estados evaluados multiplicado por el número medio de veces que se evalúa cada uno de esos estados, por lo tanto, puede ocurrir que el evaluar menos estados del espacio de estados se compense con el hecho de que el algoritmo debe evaluarlos un mayor número de veces hasta llegar a convergencia.

Existen métodos que pueden limitar el número de veces que se expande cada estado, como, por ejemplo, expandir más de un estado fringe del grafo solución parcial en vez de uno solo en cada paso. Otros métodos se basan en limitar el número de iteraciones máximas en iteración de valores, así como limitar la cantidad de estados del conjunto Z. Estos métodos suelen mejorar el rendimiento de LAO* ya que la parte más computacionalmente costosa del algoritmo es la de actualizar los costes, ya sea mediante iteración de valores o de políticas.

Teniendo en cuenta esto, estudiaremos una versión más eficiente de LAO*, ILAO*. Este algoritmo combina la expansión del grafo solución parcial con la actualización de los valores de los estados.

ILAO* difiere de LAO* en tres aspectos:

- En cada expansión del grafo solución parcial, en vez de seleccionar un único estado del conjunto fringe para expandir, se seleccionan todos.
- En cada paso en vez de aplicar iteración de valores o de políticas sobre los estados del grafo solución parcial, solo se ejecuta un Bellman backup.

- Los backups se ejecutan siguiendo un recorrido primero en profundidad[4] del grafo solución, de manera que el valor de cada estado se actualiza antes que el valor de sus antecesores.

Tabla 5. Pseudocódigo función búsqueda en profundad del grafo solución parcial

```
function busqueda_primero_en_profundidad(estado, I, F, actualizados, V, politica)
// I -> Conjunto de estados interiores
// F -> Conjunto de estados fringe
1. actualizados.añadir(estado)
2. Si estado está en I
    2.1. Para suc en estado.sucesores(politica(estado)), suc no es terminal ni actualizado
        2.1.1. busqueda_primero_en_profundidad(suc, I, F, V, politica)
3. Si no
    3.1. I.añadir(estado)
    3.2. F.eliminar(estado)
    3.3. Para suc en estado.sucesores(), suc no está en I
        F.añadir(suc)
4. Bellman_Backup(estado, V, politica)
```

Esta es una versión simplificada del código necesario para realizar la búsqueda del grafo solución mientras se actualizan los estados con un solo backup. Si el estado ya ha sido expandido, es decir, se encuentra en el conjunto de estados interiores, se obtienen sus sucesores y se llama de forma recursiva a la función sobre los sucesores del estado, a excepción de los que sean estados objetivo y de los que ya hayan sido actualizados. Este punto es importante ya que el hipergrafo que representa el problema permite bucles, por lo tanto, resulta necesario controlar que cada estado no sea actualizado más de una vez. En el caso en que el estado aún no haya sido expandido, es decir, que sea un estado fringe, se expande; para ello, se añade al conjunto de estados interiores, posteriormente, se elimina del conjunto de estados fringe y se inserta en este conjunto todos los sucesores del estado

[4] algoritmo de búsqueda en árbol que recorre primero todos los nodos de un camino concreto de forma ordenada.

que no hayan sido expandidos y, por ende, se encuentren ya en el conjunto de estados interiores. Finalmente, se realiza un backup sobre el estado. Es importante realizar el backup después de la expansión porque, como se ha comentado anteriormente, al ser un recorrido primero en profundidad, los sucesores de un estado son actualizados antes que el propio estado.

Esta expansión del grafo solución se hace mientras éste tenga estados no terminales sin expandir. Finalmente, se realiza la prueba de convergencia, aplicando iteración de valores sobre los estados del grafo solución parcial. En el caso de que el grafo solución parcial se modifique en la prueba de convergencia de forma que contenga estados sin expandir, este procedimiento debe repetirse, hasta obtener un grafo solución con todos los estados interiores o terminales.

Intuitivamente podemos concebir la idea de que el algoritmo busca una posible solución al problema mediante la búsqueda primero en profundidad, una vez que la ha encontrado, aplica la prueba de convergencia para ver si realmente esa es la mejor solución. En caso de que lo sea, el algoritmo devuelve la solución óptima; de lo contrario, el algoritmo itera para expandir estados fringe que hayan quedado sin expandir en búsqueda de una solución mejor.

Tabla 6. Pseudocódigo ILAO*

```
1. V ← heuristico
2. politica ← π0
// I -> Conjunto de estados interiores
// F -> Conjunto de estados fringe
3. I ← {}
4. F ← {estado_inicial}
5. grafo_solucion ← {nodos: {estado_inicial}, hiperarcos: {}}
// Construir grafo solución mientras se actualizan los valores de los estados
6. Mientras F.interseccion(grafo_solucion.estados()) no sea vacío
        6.1. actualizados ← {}
        6.1. busqueda_primero_en_profundidad(estado_inicial, I, F, actualizados, V, politica)
        6.2. grafo_solucion ← construir_grafo_solucion(estado_inicial, politica, grafo)
// Prueba de convergencia
7. Ejecutar Iteración de Valores sobre grafo_solucion.estados()
8. Si F.interseccion(grafo_solucion.estados()) no es vacío, volver al paso 6
9. Devolver política óptima
```

5.3 RLAO*

Durante el proceso de ejecución de LAO*, la expansión del grafo cubierta depende del heurístico. Existe la posibilidad de que LAO* requiera un número excesivo de iteraciones antes de alcanzar un estado final durante el proceso de expansión, dependiendo en gran medida de las características del heurístico. La importancia de esta circunstancia reside en la idea de que, aunque es verdad que el heurístico pueda proveer información sobre lo prometedor que es un estado, la información que provee al algoritmo conocer la posición de los estados terminales es mucho más sólida y menos hipotética, es decir, una vez que la posición de los estados terminales es conocida, la actualización de los valores va a ser más rigurosa porque no se basará en una idea optimista de lo "bueno" que es cada estado, si no en una noción concisa atribuida por el conocimiento de la posición de los estados terminales, de forma que, los estados que se encuentren más cerca de los estados terminales tendrán un valor menor en comparación a los estados que se encuentren más

lejos. El algoritmo RLAO* (Reverse LAO*) implementa una solución a este problema. [7]

La idea principal en la que se basa RLAO* es en aplicar la búsqueda en dirección contraria a LAO*, es decir, desde los estados terminales hasta el estado inicial, de esta forma, la actualización de los valores asociados a cada estado será más precisa, porque en el conjunto de estados a actualizar siempre se encuentra el estado terminal. Su implementación es similar a ILAO*, con la diferencia de que, el grafo cubierta tiene su raíz en el estado terminal, y, por lo tanto, la expansión se hará desde el estado terminal hacia atrás, obteniendo los antecesores de cada estado a expandir.

Tabla 7. Pseudocódigo función búsqueda hacia atrás

```
function busqueda_hacia_atras(estado, I, F, actualizados, V, politica)
1. actualizados.añadir(estado)
// En este caso hacemos el backup primero, porque los sucesores se actualizan antes que
// sus respectivos predecesores
// I -> Conjunto de estados interiores
// F -> Conjunto de estados fringe
2. Bellman_Backup(estado, V, politica)
3. Si estado está en I
    3.1. Para pred en estado.predecesores(), pred no está en actualizados
        3.1.1. busqueda_hacia_atras(pred, I, F, actaulizados, V, politica)
4. Si no
    4.1. I.añadir(estado)
    4.2. F.eliminar(estado)
    4.3. Para pred en estado.predecesores(), pred no está en I
        F.añadir(pred)
5. Bellman_Backup(estado, V, politica)
```

Como queda descrito en el pseudocódigo el método de búsqueda hacia atrás es muy similar al utilizado en ILAO* para realizar la búsqueda hacia adelante, con la principal diferencia de que la búsqueda se realiza obteniendo los predecesores en lugar de los sucesores. El efecto de expandir y actualizar los valores partiendo desde el estado final provoca que el conocimiento de la situación del estado final con respecto del resto de estados se "propague" obteniendo una actualización de los valores más fiable.

Tabla 8. Pseudocódigo RLAO*

```
1. V ← heuristico
2. politica ← π0
3. I ← {}
4. F ← {estado_final}
5. grafo_solucion ← {nodos: {estado_final}, hiperarcos: {}}
// Construir grafo solución mientras se actualizan los valores de los estados
6. Mientras F.interseccion(grafo_solucion.estados()) no sea vacío
        6.1. actualizados ← {}
        6.1. busqueda_hacia_atras(estado_final, I, F, actualizados, V, politica)
        6.2. grafo_solucion ← construir_grafo_solucion(estado_inicial, politica, grafo)
// Prueba de convergencia
7. Ejecutar Iteración de Valores sobre grafo_solucion.estados()
8. Si F.interseccion(grafo_solucion.estados()) no es vacío, volver al paso 6
9. Devolver política óptima
```

Es posible implementar RLAO* como una versión de LAO* en la cual la expansión de los nodos del conjunto fringe empieza por el estado final y, en cada iteración, se expande un predecesor de un estado interior que se encuentra en el grafo solución, pero, como ya se ha comentado en el apartado dedicado a ILAO*, el rendimiento del algoritmo aumenta notablemente cuando en cada iteración se expanden todos los estados del conjunto fringe y se hace un solo backup en vez de ejecutar iteración de valores o de políticas completamente. Como en cada paso se expanden todos los estados del conjunto fringe, y el

conjunto fringe aumenta añadiendo en cada iteración todos los antecesores de todos los estados expandidos que no hayan sido expandidos previamente, el rendimiento general del algoritmo depende del *fan-in*, esto es, el número medio de estados desde los cuales es posible transitar a un estado concreto. Cuanto mayor sea, peor será el rendimiento del algoritmo.

5.4 BLAO*

BLAO* (Bidirectional LAO*) es un algoritmo derivado de LAO* que implementa la expansión hacia atrás y hacia adelante simultáneamente. La búsqueda hacia adelante empieza en el estado inicial y expande los estados hasta llegar a un estado objetivo; de forma análoga, la búsqueda hacia atrás comienza desde un estado objetivo y se propaga hasta alcanzar el estado inicial. En cierto punto, la búsqueda hacia atrás y hacia adelante coincidirán, estableciendo un "puente" entre el estado inicial y el estado final. La búsqueda prosigue hasta que se obtiene un grafo solución parcial sin estados fringe no terminales. Entonces, se realiza la prueba de convergencia aplicando iteración de valores o iteración de políticas. Este proceso se repite hasta encontrar una solución óptima.

El algoritmo puede concebirse como una combinación de los algoritmos ILAO* y RLAO*, de lo que se infiere que hay una correlación entre el rendimiento de estos dos algoritmos y el rendimiento de BLAO*, que se estudiará en los próximos capítulos.

La implementación original de este algoritmo solo expande un estado de cada búsqueda en cada iteración, es decir, expande un estado durante la búsqueda hacia atrás y otro estado en la búsqueda hacia adelante. Para este proyecto, se ha modificado la implementación original aplicando el principio que afirma que suele dar mejores resultados expandir todos los nodos del conjunto fringe en cada iteración que solamente uno (lo que ocurre en LAO*). [8]

Tabla 9. Pseudocódigo BLAO*

```
1. V ← heuristico
2. politica ← π0
// I -> Conjunto de estados interiores
// F -> Conjunto de estados fringe
3. I ← {}
4. F ← {estado_inicial, estado_final}
5. grafo_solucion ← {nodos: {estado_inicial}, hiperarcos: {}}
6. Mientras F.interseccion(grafo_solucion.estados()) no sea vacío
    6.1. actualizados ← {}
// Búsqueda hacia adelante
    6.1. busqueda_primero_en_profundidad(estado_inicial, I, F, actualizados, V, politica)
// Búsqueda hacia atrás
    6.2. busqueda_hacia_atras(estado_final, I, F, actualizados, V, politica)
// Construcción grafo solución
    6.3. grafo_solucion ← construir_grafo_solucion(estado_inicial, politica, grafo)
// Prueba de convergencia
7. Ejecutar Iteración de Valores sobre grafo_solucion.estados()
8. Si F.interseccion(grafo_solucion.estados()) no es vacío, volver al paso 6
9. Devolver política óptima
```

6. Definición del problema

En este capítulo haremos una definición concisa del problema que ha sido implementado y para el que se obtendrán estadísticas asociadas a cada algoritmo. Es importante especificar que las características del problema influyen en el rendimiento del algoritmo, y, en función de las parámetros de entrada, se obtendrán unos resultados concretos. Por ello, las conclusiones que se obtendrán finalmente a partir de los datos obtenidos estarán relacionadas y justificadas con aspectos concretos del problema implementado, y servirá de utilidad a la hora de especificar qué algoritmo nos puede ser más útil aplicar en función de sus características.

6.1 Componentes del problema

La idea genérica del problema se basa en encontrar el camino más corto desde una posición inicial hasta una posición final en un tablero de dos dimensiones. Cada posición o celda del tablero está asociado con un estado. Las componentes del problema son las siguientes:

- Estado inicial. Hace referencia a la posición inicial del tablero. Se establece en la mitad del tablero.
- Estado objetivo. Hace referencia a la posición final del tablero. Se establece en una de las 4 esquinas de forma aleatoria.
- Número de filas del tablero.
- Número de columnas del tablero.
- Sumideros. Son posiciones del tablero que no deben ser alcanzadas por el agente. El grafo solución solo será válido si, aparte de las condiciones mencionadas anteriormente, está libre de sumideros, indistintamente de la probabilidad. En ese sentido, difiere de los obstáculos; en un problema de SSP con obstáculos si existe la posibilidad de un grafo solución con estados que son obstáculos (aunque teóricamente con baja probabilidad de intentar acceder a ellos).
- Acciones. El conjunto de acciones que se puede realizar desde cada estado es avanzar un paso en cualquier dirección (norte, sur, este, oeste, noreste, noroeste, sureste, suroeste); aparte, existe una acción que hace transitar desde cualquier estado a ese

mismo estado. En total, tenemos 9 acciones posibles.

- Costes. El coste de realizar cualquier acción desde cualquier estado en esta implementación es de 1.

El problema se genera aleatoriamente a partir de un conjunto de parámetros de entrada. Los parámetros de entrada son el número de filas y de columnas del tablero, el porcentaje de celdas que van a ser sumideros en el problema generado, y el sistema de transiciones, que aparece explicado en el siguiente apartado.

6.2 Sistemas transitorios

Como ya se ha explicado previamente, los procesos de decisión de Markov son procesos estocásticos, no deterministas. Ello implica que realizar una acción desde un estado no ofrece una certeza total de que se aplique la transición adecuada desde ese estado al aplicar esa acción, es decir, aplicar la acción norte desde un estado concreto no asegura necesariamente transitar al estado que se encuentra al norte.

Mediante los sistemas de transiciones definimos las probabilidades de transitar desde un estado a otro estado realizando una acción concreta.

En este proyecto y, para el problema definido anteriormente, se han implementado los siguientes sistemas transitorios.

- Sistema 1. En este sistema, al intentar realizar una acción *a* se aplicará la transición correcta con una probabilidad de 0.8, y se transitará a cualquiera de los dos estados adyacentes al estado al que queremos transitar con una probabilidad de 0.1, por ejemplo, si desde un estado concreto aplicamos la acción 'Norte', la probabilidad de transitar al estado que está al norte será de 0.8, mientras que las probabilidades de transitar a los estados que se encuentran adyacentes al estado norte, es decir, el estado noreste y el estado noroeste, será de 0.1 cada una.
- Sistema 2. En este sistema, al intentar realizar una acción *a* se aplicará la transición correcta con una probabilidad de 0.9, y se transitará al estado adyacente al estado que queremos transitar en sentido horario con una probabilidad de 0.1, por ejemplo, si aplicamos la acción 'Sur' desde un estado concreto, la probabilidad de transitar al estado que se encuentra al sur es de 0.9, mientras que la probabilidad de transitar al estado que se encuentra al suroeste (sentido horario) es de 0.1

- Sistema 3. En este sistema, al intentar realizar una acción *a* se aplicará la transición correcta con una probabilidad de 0.9 y se transitará al mismo estado con una probabilidad de 0.1, por ejemplo, si realizamos la acción 'Este' desde un estado se transitará al estado que se encuentra al este con una probabilidad de 0.9, pero habrá una probabilidad de 0.1 de mantenerse en el mismo estado.

También es necesario especificar que si el agente trata de realizar una acción que conlleva una transición a un posición fuera de los límites del tablero, la transición que se aplicará será al mismo estado, es decir, si el agente se encuentra en el límite sur del tablero, e intenta realizar la acción sur, la transición se realizará al mismo estado. Además, indiferentemente del sistema de transiciones que se esté aplicando, el realizar la acción que mantiene al agente en el mismo estado provocará una transición al mismo estado con una probabilidad de 1 y coste unitario.

6.3 Heurístico utilizado

En el apartado de teoría de heurísticos se afirmó que, para que un heurístico fuese efectivo, tenía que ser admisible, es decir, que para todo estado *s*, $h(s) \leq V^*(s)$. Como ya se aclaró, un heurístico es una aproximación optimista del coste que le requiere a cada estado alcanzar un estado objetivo, y que, por lo tanto, no debe superar el coste real óptimo. En esta implementación, el valor heurístico que se le otorga a cada estado es el relativo al número mínimo de pasos que necesitaría para alcanzar un estado objetivo en el caso de que no hubiera sumideros. Es trivial afirmar que este heurístico es admisible, ya que, en el caso de que no haya sumideros, el heurístico coincide con el coste real requerido, por el contrario, si hay sumideros, el valor del heurístico siempre va a ser igual o menor, porque la presencia de sumideros no puede reducir los valores de los estados, solo aumentarlos, es decir, los sumideros no facilitan alcanzar un estado objetivo desde un estado concreto, sino al contrario.

Como se permiten los desplazamientos diagonales, el número de pasos que requiere el estado *s* para alcanzar el estado *s'* es el máximo entre la diferencia de filas y la diferencia de columnas en valor absoluto. Es fácilmente entendible mediante la ilustración de un ejemplo: si, por ejemplo, tenemos un estado *s* que se encuentra a 6 filas del estado final y a 4

columnas, el número de pasos mínimos requeridos para alcanzar el estado final desde el estado s es 6; esto es así ya que el estado *s* va a necesitar 4 pasos diagonales para situarse en la misma columna que el estado final, como hemos dado 4 pasos diagonales, ahora nos encontramos a 2 filas del estado final (ya que el dar un paso diagonal en la dirección correcta nos acerca una fila y una columna), por lo tanto, el número total de pasos es de 4 pasos para situarnos en la misma columna y 2 pasos más para situarnos en la misma fila.

El cálculo del heurístico para el estado *s* es el siguiente:

h(s) = max(|s_terminal.fila() - s.fila()|, |s_terminal.columna() - s.columna()|)

Este cálculo es aplicable porque estamos utilizando costes unitarios tanto para los desplazamientos diagonales como para el resto de los desplazamientos, de lo contrario, sería necesario tener en cuenta el coste de cada tipo de desplazamiento.

7. Estudio del rendimiento

En este capítulo se van a graficar los datos relativos al rendimiento de cada algoritmo para finalmente, sacar conclusiones a partir de estos datos. Concretamente, para cada algoritmo se estudiará:

- Tiempo de cómputo
- Número de iteraciones
- Número de expansiones (Número de estados del grafo cubierta)

Estos datos se estudiarán en función de:

- Número de estados
- Número de sumideros
- Sistema de transiciones utilizados

7.1 Comparativa sin sumideros

Empezamos comparando el rendimiento de los algoritmos para casos sin sumideros.

7.1.1 Comparativa entre distintos algoritmos para sistema 1

En este apartado se hará una comparativa de los distintos algoritmos con el primer sistema transitorio para problemas sin sumideros.

En las gráficas ilustradas, el eje de abscisas se corresponde con el número de estados y el eje de ordenadas se corresponde con la variable que se quiera medir, ya sea tiempo de cómputo, número de iteraciones o número de nodos expandidos.

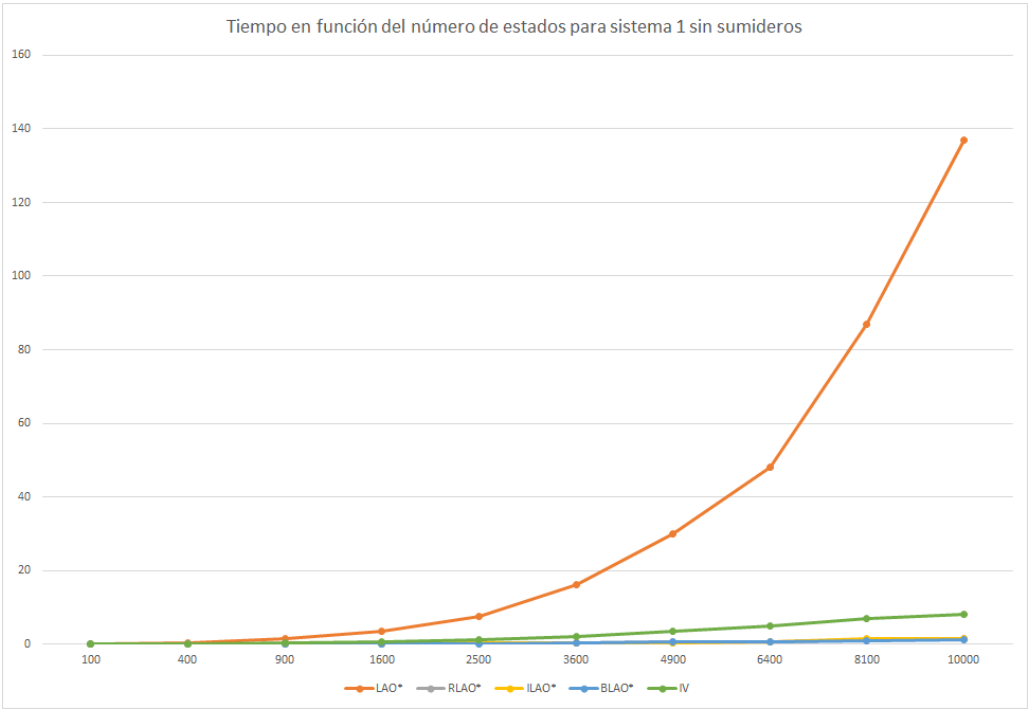

Figura 2. Gráfica de tiempo en función de número de estados para sistema 1 sin sumideros

Como se aprecia en la figura, LAO* es el algoritmo menos eficiente y asume un crecimiento exponencial de tiempo de cómputo en función del número de estados. Esta es una circunstancia que depende de la implementación concreta, así como de las características del problema. En este caso, al estar utilizando el sistema de transiciones 1, el rendimiento de LAO* es peor, ya que es el sistema que presenta un mayor número de sucesores por acción realizada desde cada estado, por lo tanto, es el que va a inducir a mayores tamaños del conjunto Z. El rendimiento del algoritmo, como se ha explicado anteriormente, depende del número de estados evaluados multiplicado por el número promedio de veces que es evaluado cada estado; esto confiere a LAO* una naturaleza en cuanto a su rendimiento más circunstancial que el resto de los algoritmos. Los algoritmos que muestran un mejor rendimiento son los derivados de LAO*; seguido de iteración de valores (línea verde).

Procedemos a estudiar el número de iteraciones y el de expansiones de cada algoritmo para un problema con las mismas características.

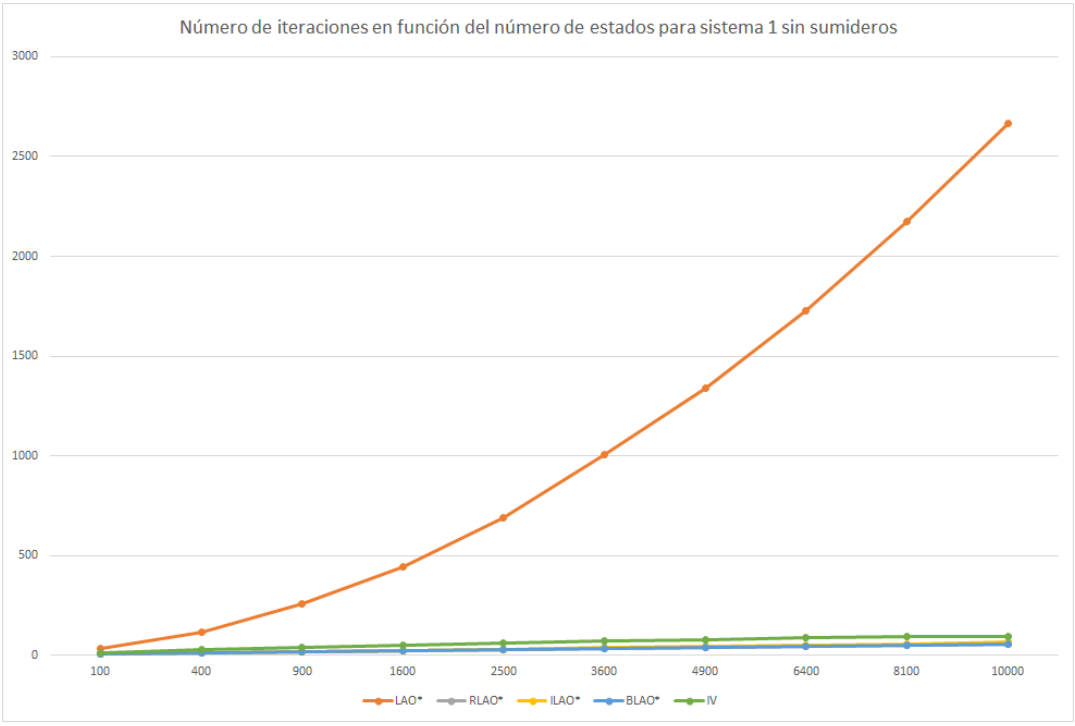

Figura 3. Gráfica de número de iteraciones en función de número de estados para sistema 1 sin sumideros

En este ejemplo podemos apreciar una de la ventaja de los algoritmos derivados de LAO* sobre el propio LAO*: no requieren tantas iteraciones hasta converger en la política óptima ya que, a diferencia de LAO*, estos algoritmos expanden todos los estados del conjunto fringe en cada iteración. En contraparte, cada iteración de LAO* supone una sola expansión.

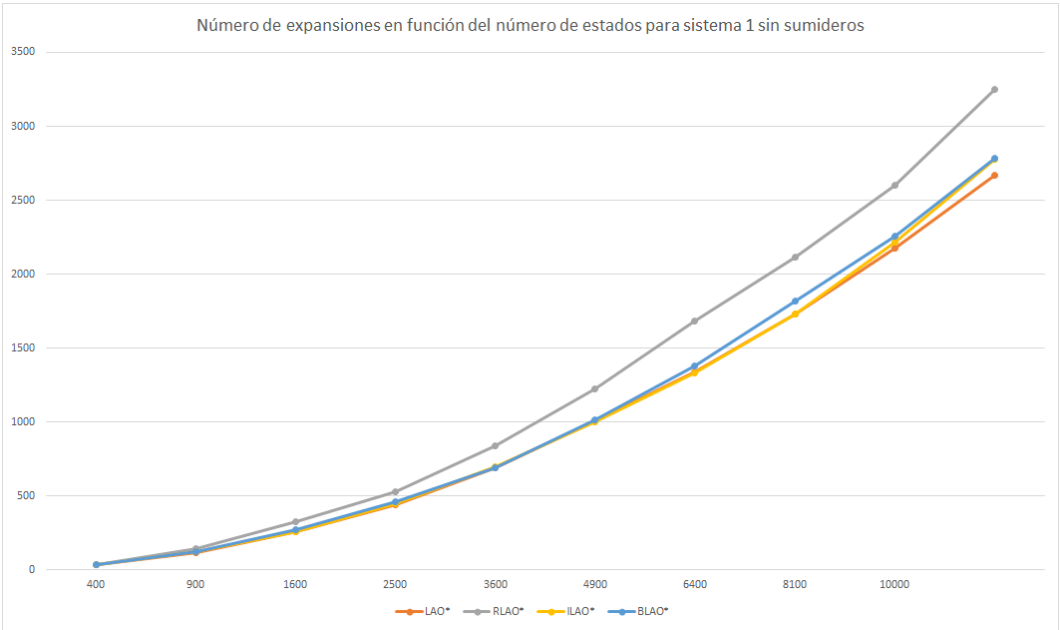

Figura 4. Gráfica de número de expansiones en función de número de estados para sistema 1 sin sumideros

Como se puede apreciar en la gráfica, según crece el número de estados, LAO* requiere un menor número de expansiones en comparación. Esto es así por la misma condición descrita anteriormente: en cada iteración solo expande un estado del conjunto fringe; esto le permite evitar expandir estados con poca probabilidad de pertenecer al grafo solución. La condición de expandir todos los estados fringe en cada iteración sacrifica expandir estados innecesarios a favor de requerir menos iteraciones hasta convergencia. Por otra parte, RLAO* es el algoritmo que requiere más expansiones, esto es así porque, a diferencia de ILAO*, expande todos los estados fringe del grafo cubierta, y no solamente los del grafo solución.

7.1.2 Comparativa entre sistemas

En este apartado se comparará el rendimiento de cada algoritmo individualmente en función del sistema que se esté utilizando. En las gráficas ilustradas, el eje de abscisas se corresponde con el número de estados de cada algoritmo y el eje de ordenadas se corresponde con la variable que se quiera medir. Las rectas refieren a los distintos sistemas transitorios.

LAO*:

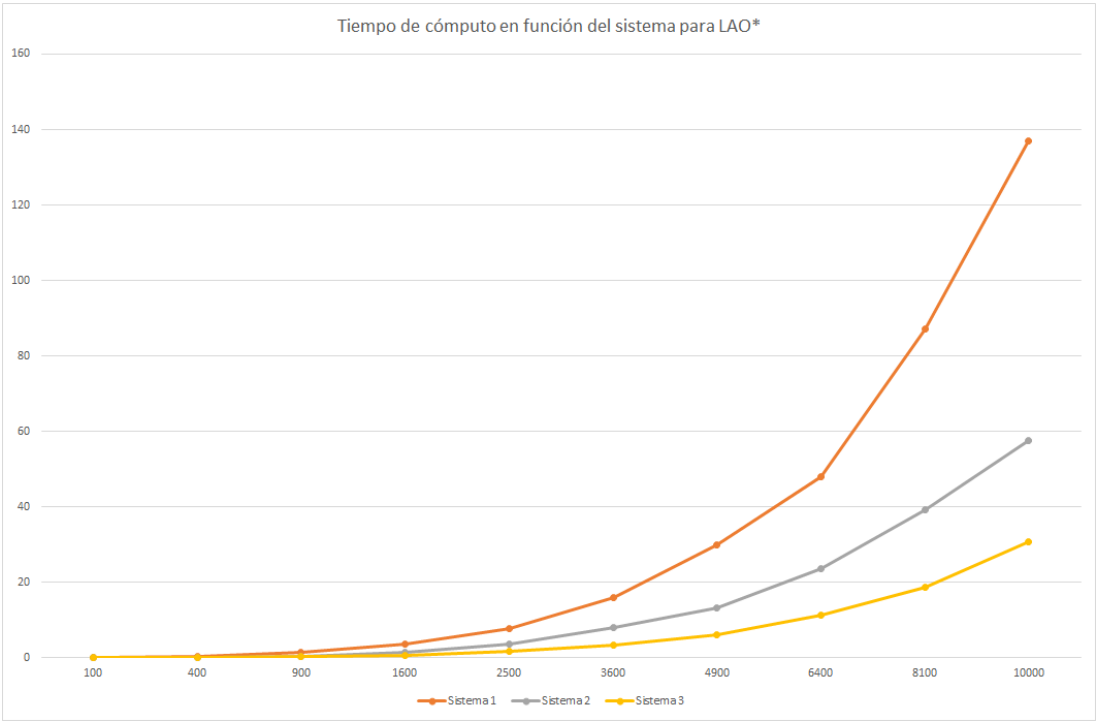

Figura 5. Gráfica de tiempo de cómputo en función del sistema utilizado para LAO*

Como se puede apreciar en la gráfica, el tiempo de cómputo varía notablemente según el sistema que se esté utilizando. Además, es perceptible que mientras que el crecimiento en el tiempo requerido para el sistema 1 es exponencial, para los sistemas 2 y 3 podemos apreciar un crecimiento lineal de distinto orden.

Esto se puede explicar gracias a la información facilitada por la gráfica que mide el número de iteraciones para los 3 sistemas. En el caso de LAO* el número de estados expandidos coincide con el número de iteraciones que utiliza el algoritmo, ya que en LAO* se expande un estado en cada iteración, así que es suficiente con graficar una de las variables.

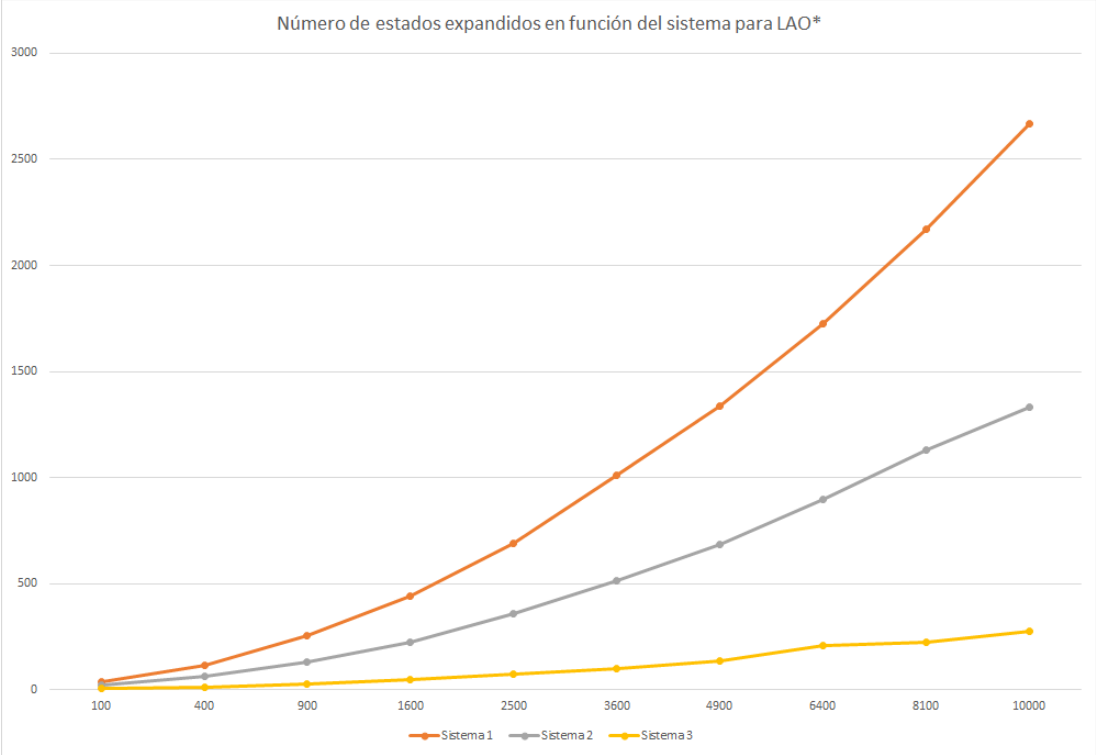

Figura 6. Gráfica de número de expansiones por sistema para LAO*

Como el sistema 3 es el más conservativo, es decir, es el sistema asociado a un menor número de sucesores por acción realizada desde un estado concreto, ya que solo permite transitar a un estado sucesor o a el propio estado mediante una acción, el crecimiento del conjunto de estados por expandir "fringe" es mucho menor (ya que este conjunto aumenta añadiendo en cada iteración los sucesores bajo cualquier acción del último estado expandido), y, por lo tanto, requiere expandir un número considerablemente menor de estados en comparación con los otros dos sistemas para obtener una solución óptima. Además, como el conjunto de antecesores de un estado es menor, ya que para el sistema 3

cada estado solo tiene un antecesor por acción, el conjunto Z, el cual se conforma con los antecesores del último estado expandido según la política greedy, también tiene menos estados, por lo tanto, como se aplica iteración de valores sobre un número menor de estados, el rendimiento del algoritmo mejora notablemente, ya que, como se ha explicado anteriormente, la parte más pesada del algoritmo es la de actualización de los valores mediante programación dinámica, ya sea iteración de valores o de políticas.

ILAO*:

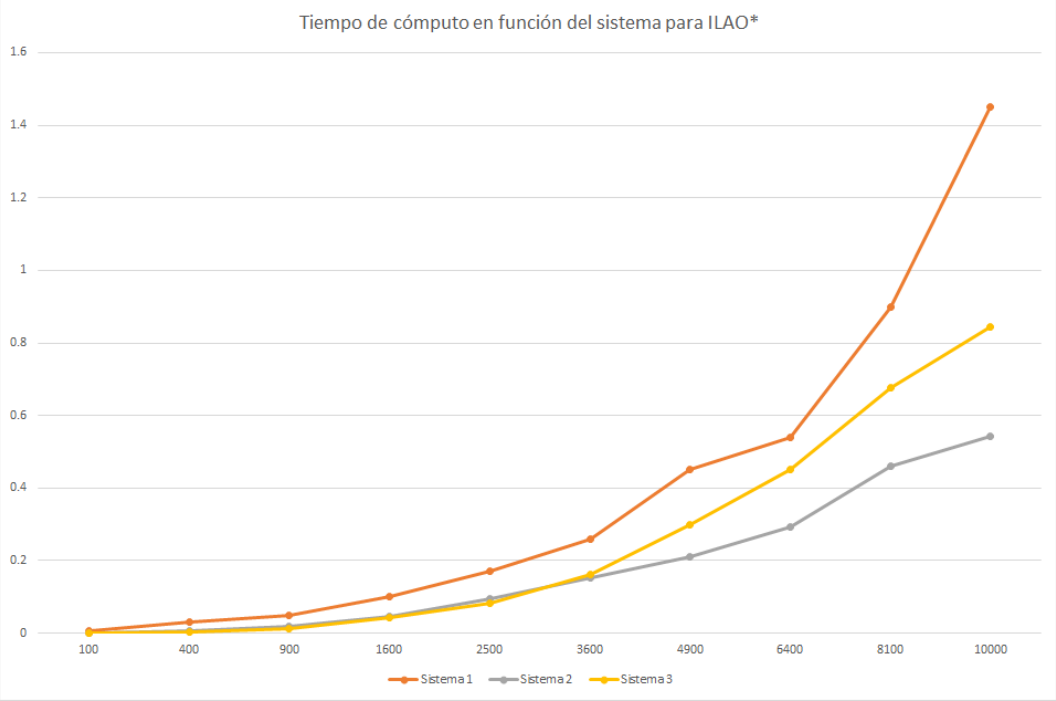

Figura 7. Gráfica de tiempo de cómputo por sistema en ILAO*

En este caso, existe una notable diferencia entre ILAO* y LAO* con respecto al rendimiento del algoritmo en función del sistema. Para estudiar esa circunstancia, necesitaremos el apoyo de los datos que grafican la información acerca del número de expansiones y de iteraciones requeridas por el algoritmo en función del sistema utilizado.

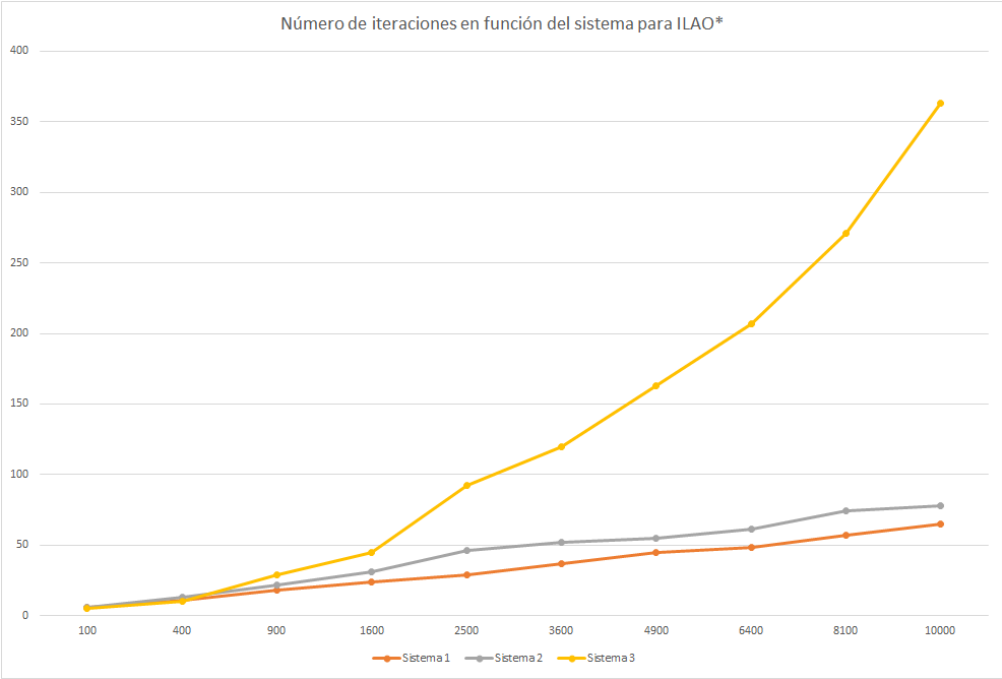

Figura 8. Gráfica de número de iteraciones por sistema en ILAO*

Para este caso, el número de iteraciones que requiere ILAO* hasta convergencia asume un crecimiento mayor para el sistema 3 en comparación con el sistema 1 o el sistema 2.

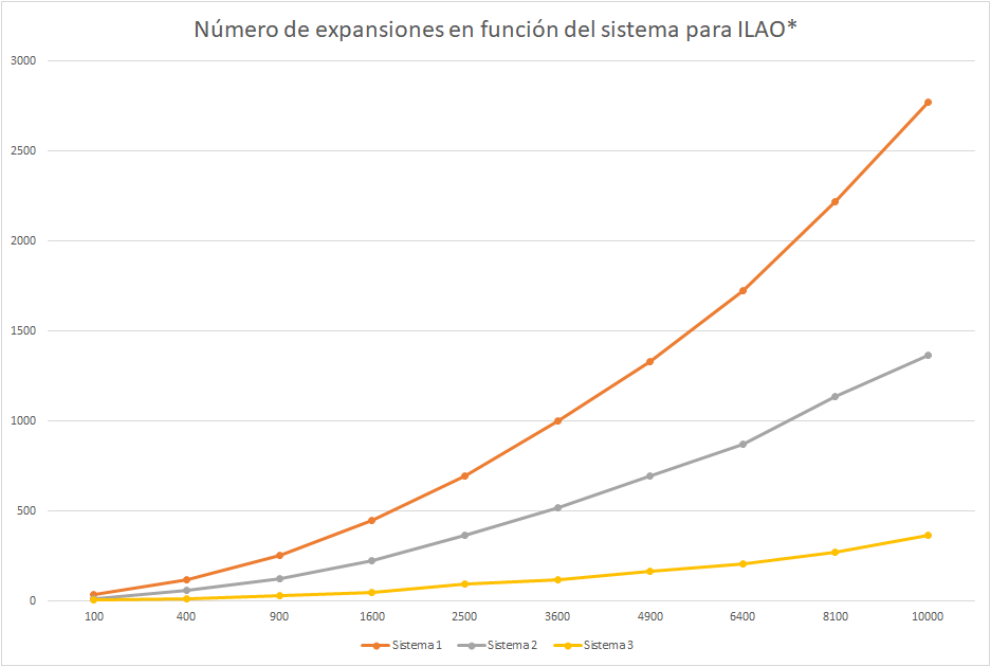

Figura 9. Gráfica de número de expansiones por sistema en ILAO*

Si bien podemos apreciar, aunque el sistema 3 sea el que requiere más iteraciones hasta convergencia para ILAO*, también es el que requiere realizar menos expansiones. Esto es así por la naturaleza del algoritmo: en cada iteración, ILAO* expande todos los estados del conjunto fringe; como el sistema 3 es el que produce menor número de sucesores por cada estado, es el que expande menos estados en cada iteración. Por lo tanto, aunque se requiera más iteraciones que el resto de los algoritmos para el sistema 3, se requiere expandir menos estados. Precisamente es lo que reduce el rendimiento de ILAO*; es cierto que se necesitan realizar un número sumamente bajo de expansiones hasta encontrar una solución, pero al realizar menos expansiones por iteración, el algoritmo necesita más iteraciones hasta encontrar una posible solución. Por lo tanto, al final el resultado es que se expandirán menos estados, pero cada uno de esos estados tendrá que ser actualizado un número considerablemente mayor de veces, afectando al rendimiento del algoritmo.

Para asegurar un rendimiento óptimo para ILAO* sería necesario encontrar el "balance" perfecto en un sistema de transiciones en el que no se produzcan demasiados sucesores, para evitar expandir estados de forma innecesaria (cosa que ocurre con el sistema 1) pero en el que tampoco se produzcan demasiado pocos, porque ralentiza el proceso de encontrar una posible solución e incrementa el número de veces que debe ser actualizado cada esto. En este caso, el mejor balance lo aplica el sistema 2 de transiciones.

RLAO*:

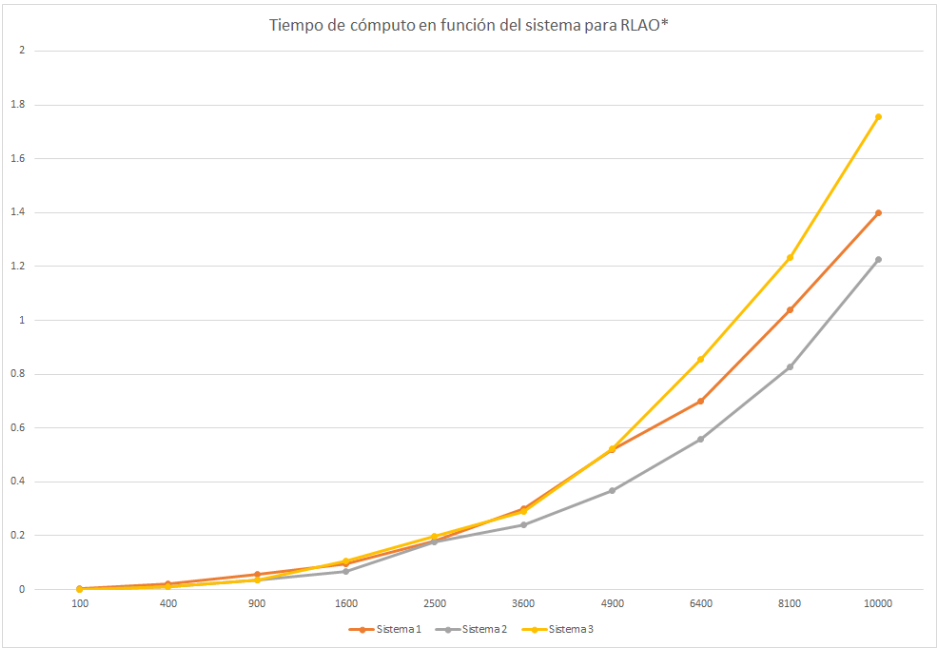

Figura 10. Gráfica de tiempo de cómputo en función del sistema utilizado para RLAO*

En este caso, pasa una circunstancia similar que con ILAO* pero de forma más discreta y suponiendo una diferencia menor; esto es así porque RLAO* expande todos los estados del conjunto fringe del grafo cubierta en cada iteración, a diferencia de ILAO* que solo expande los estados no expandidos del grafo solución parcial. Además, como ya se comentó anteriormente, el rendimiento del algoritmo RLAO* es regido por el fan-in (número medio de estados que pueden transitar a un estado concreto), y el sistema 3 es el que asume un fan-in menor.

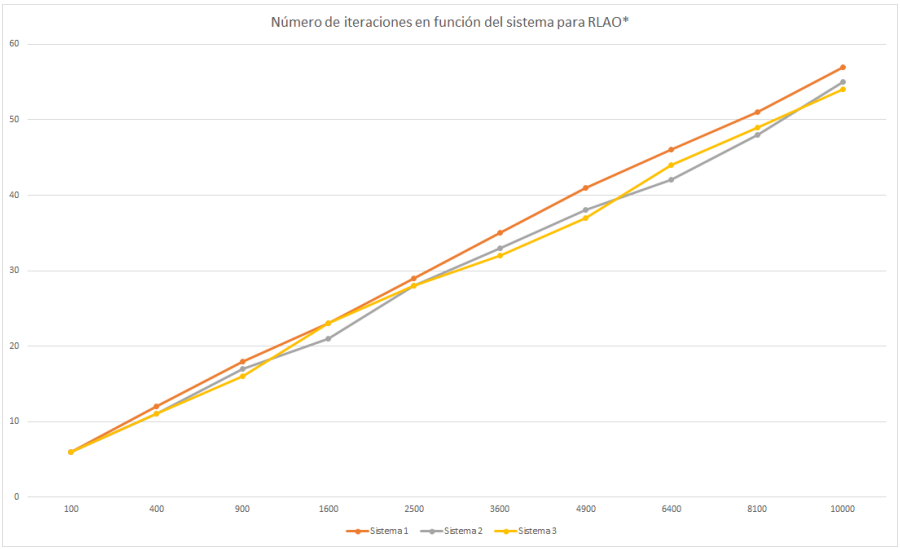

Figura 11. Gráfica de número de iteraciones en función del sistema utilizado para RLAO*

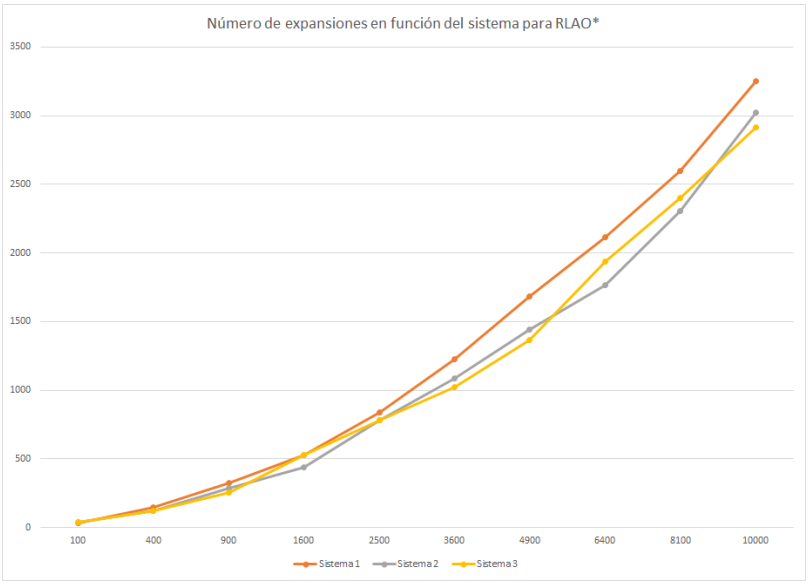

Figura 12. Gráfica de número de expansiones en función del sistema utilizado para RLAO*

Como se puede comprobar, la diferencia entre el número de iteraciones y de expansiones para cada sistema en RLAO* es más discreta en comparación a ILAO* por el motivo descrito anteriormente. Esto hace a RLAO* un algoritmo menos circunstancial, capaz de acomodarse mejor al sistema transitorio utilizado, siempre y cuando no suponga un fan-in demasiado alto, que es cuando el rendimiento del algoritmo puede verse mermado.

BLAO*:

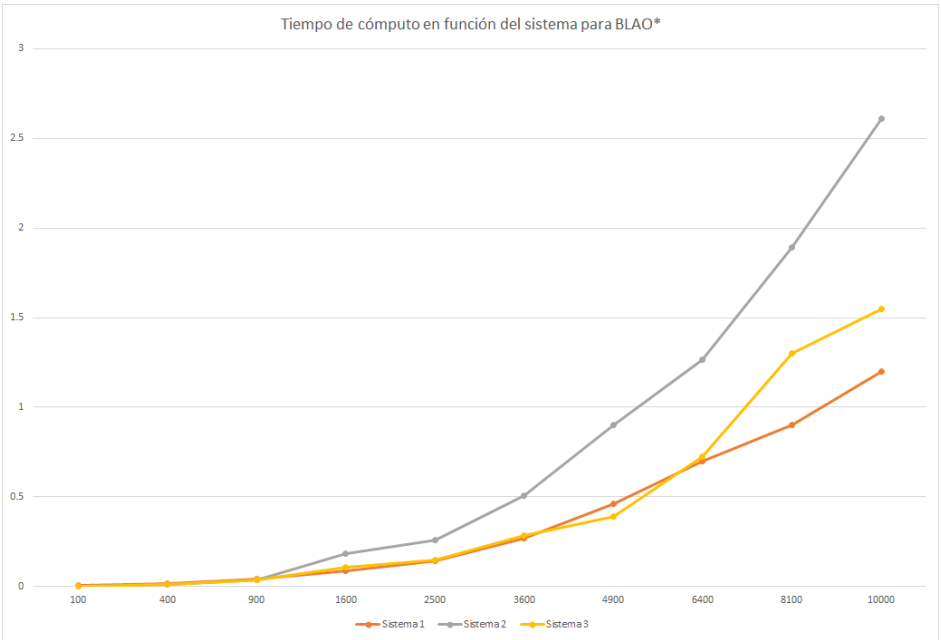

Figura 13. Gráfica de tiempo de cómputo en función del sistema utilizado para BLAO*

Para BLAO* es el sistema 2 el que supone un peor rendimiento. Para hacernos una idea más precisa, vamos a estudiar el número de expansiones y de iteraciones en función del sistema.

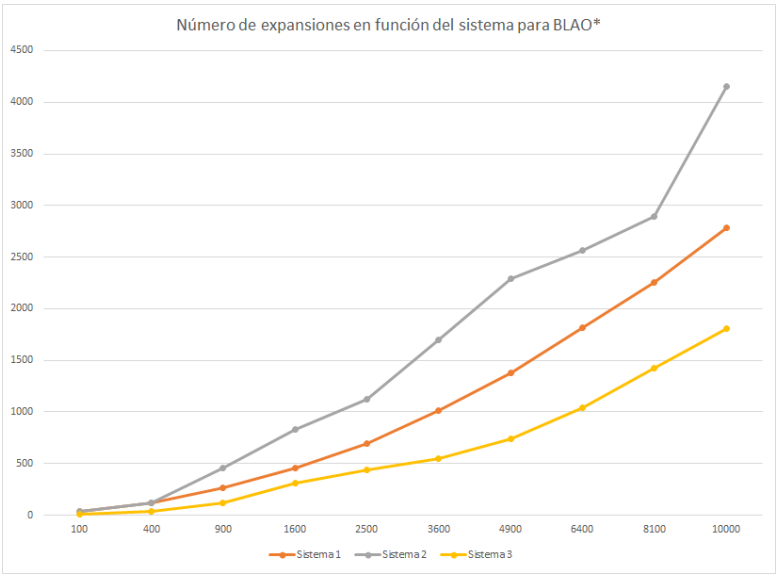

Figura 14. Gráfica de número de expansiones en función del sistema utilizado para BLAO*

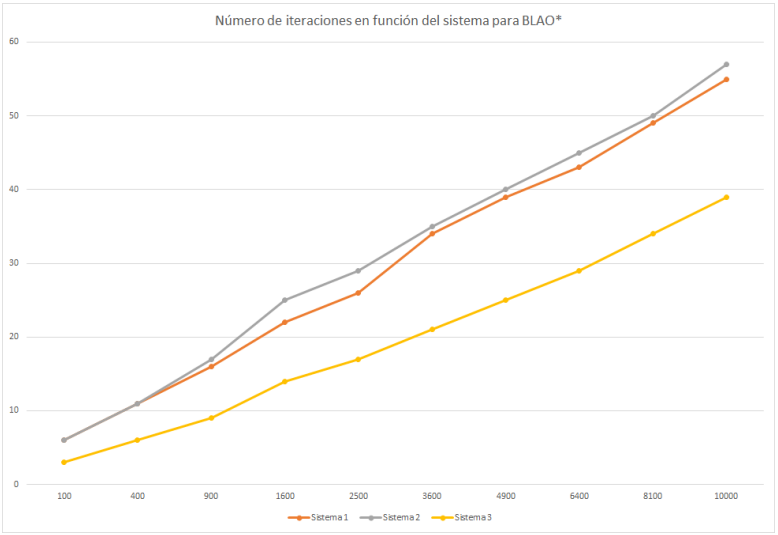

Figura 15. Gráfica de número de iteraciones en función del sistema utilizado para BLAO*

Como es apreciable en las distintas gráficas, el sistema 2 es el que le requiere a BLAO* un mayor número de iteraciones, superando incluso al que requiere el estado 1. Como, además, en cada iteración BLAO* expande más estados para el sistema 2 que para el sistema 1 (porque el sistema 2 hace crecer el conjunto fringe más rápidamente), es trivial afirmar que para el sistema 2, BLAO* necesita expandir una proporción mayor del espacio de estados y, claro está, esta es una condición que disminuye el rendimiento del algoritmo.

Como ya se comentó anteriormente, esta no es la implementación original del algoritmo; en esta implementación se expanden todas los estados del conjunto fringe en cada una de las búsquedas en vez de solamente 1. Si se hubiera utilizado la implementación original, la comparativa de rendimiento hubiera sido similar a la de LAO*: mejor rendimiento cuanto menor número de sucesores por acción aplicada tenga cada estado en el sistema utilizado. En comparación, esta implementación es más eficiente para sistemas de transiciones en los que cada estado tiene un mayor número de sucesores.

Iteración de valores:

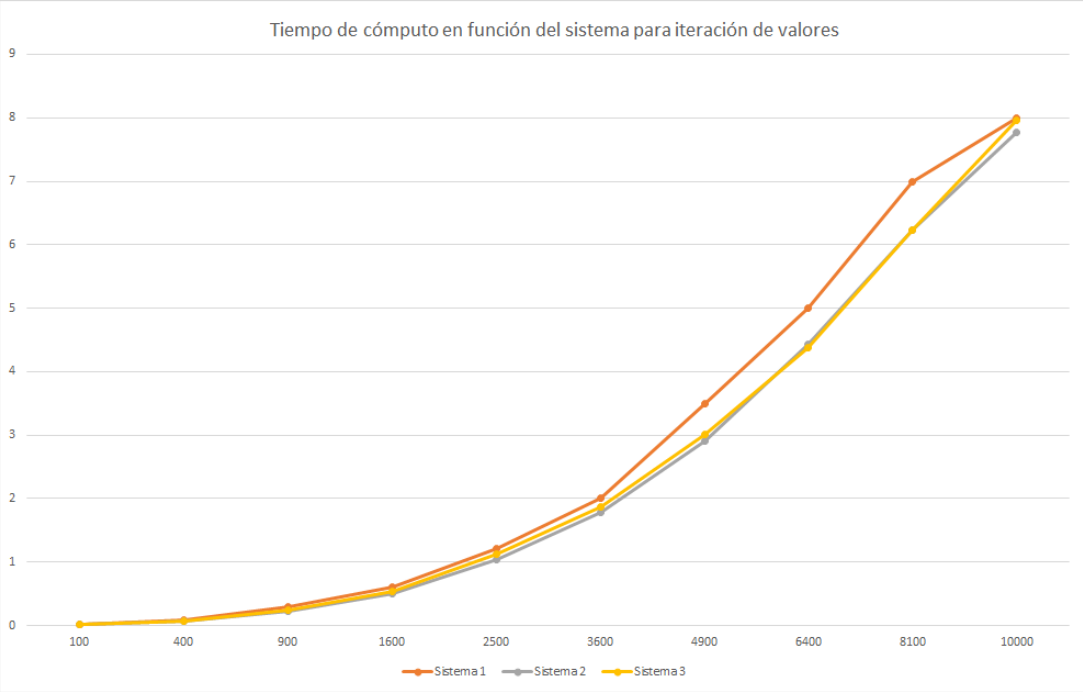

Figura 16. Gráfica de tiempo de cómputo en función del sistema utilizado para iteración de valores

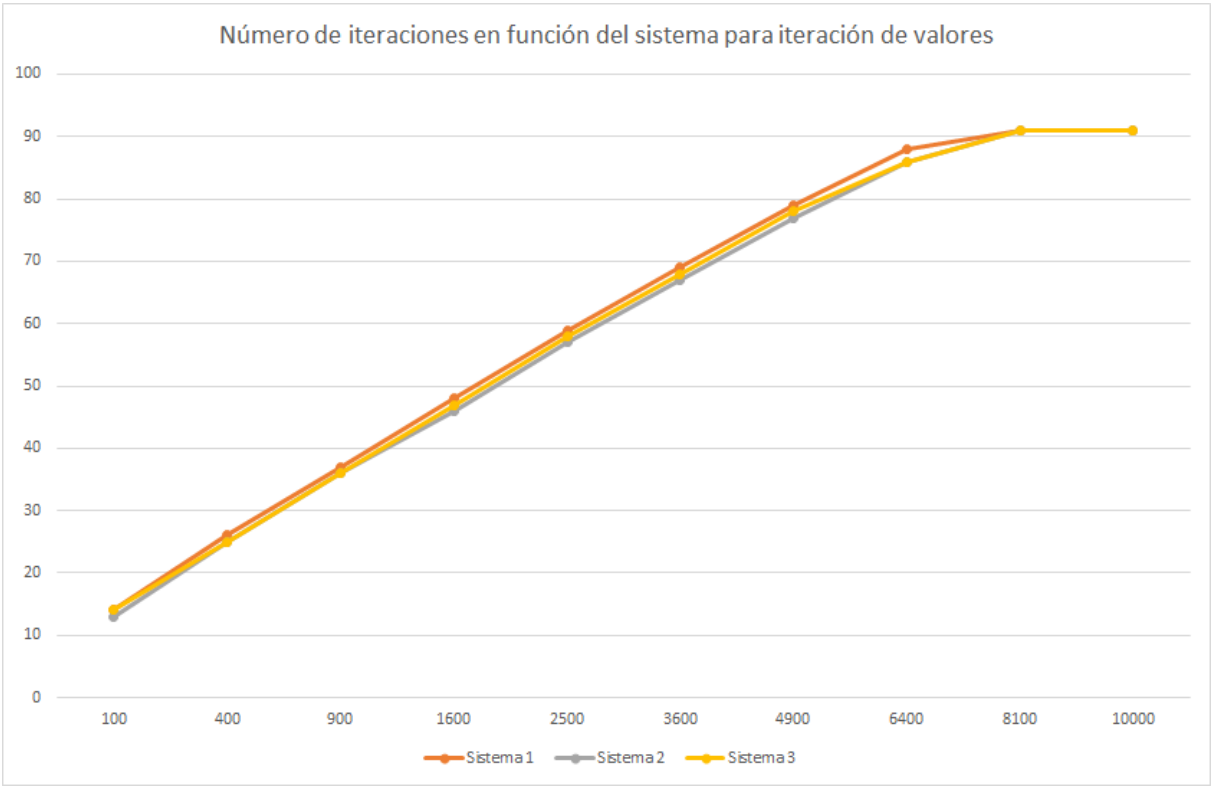

Figura 17. Gráfica de número de iteraciones en función del sistema utilizado para iteración de valores

En este caso la explicación es bastante simple; iteración de valores expande todos los estados y requiere un número similar de iteraciones para llegar a convergencia sin importar cuál sea el sistema que se esté utilizando. El motivo por el que para el sistema 1 se requiere un tiempo de cómputo ligeramente superior es porque en cada iteración tiene que hacer más cálculos, ya que el valor de cada estado se calcula en función de sus sucesores asociados a su acción greedy; cuantos más sucesores (caso del sistema 1) más cálculos y, por ende, más tiempo.

El crecimiento en el número de iteraciones sigue un crecimiento logarítmico a causa de que se está utilizando una tasa de descuento. La tasa de descuento es un valor inferior a 1 que se multiplica por el valor de un estado sin incluir el coste en el momento de la actualización de su valor y que asegura la convergencia en caso de problemas sin solución y, además, lo hace en menos tiempo. Es importante usar una tasa de descuento que no sea excesivamente baja porque de lo contrario el algoritmo podría no asegurar converger en la política óptima.

Para este proyecto se está utilizando una tasa de descuento de 0.95

7.1.3 Comparativa entre distintos algoritmos para sistemas 2 y 3

Una vez ilustrado y explicado el rendimiento de cada algoritmo en función del sistema, procedemos a mostrar el resultado de una comparativa del tiempo de cómputo de los distintos algoritmos para cada sistema.

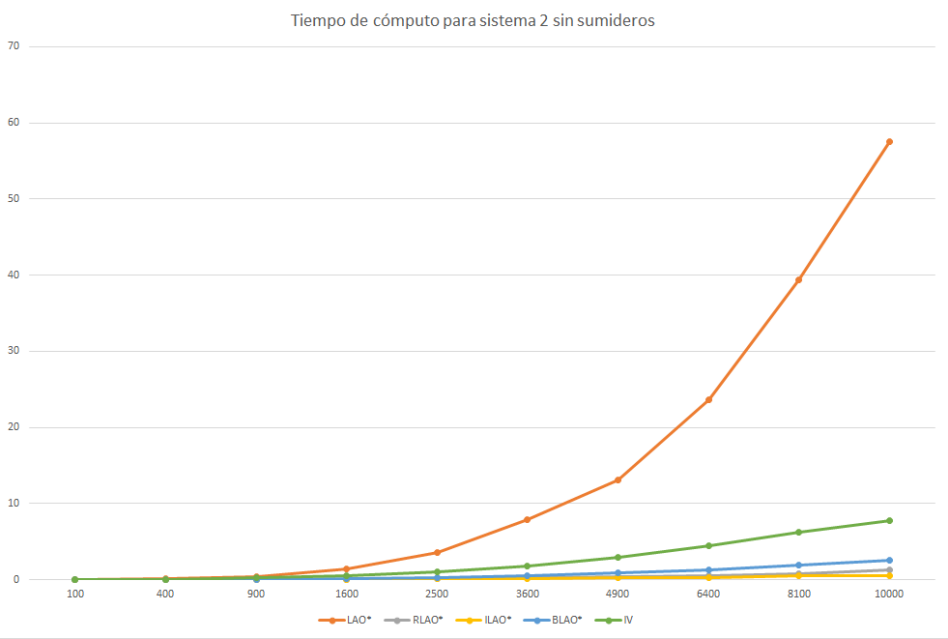

Figura 18. Gráfica de tiempo de cómputo para sistema 2 sin sumideros

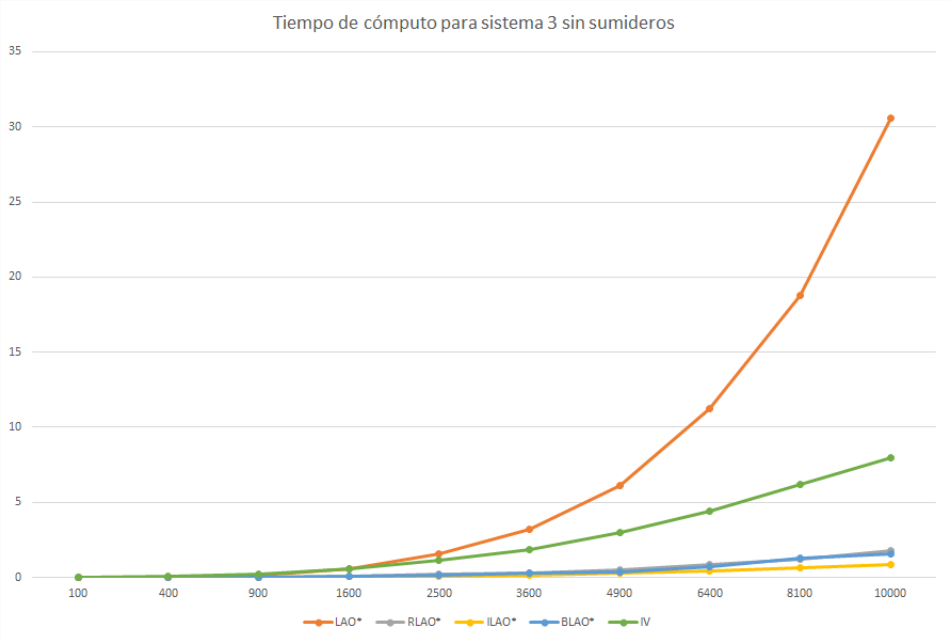

Figura 19. Gráfica de tiempo de cómputo para sistema 3 sin sumideros

7.2 Comparativa con sumideros

7.2.1 Comparativa con distinto porcentaje de sumideros

En este apartado estudiaremos el rendimiento de cada algoritmo en función del porcentaje de sumideros del problema.

Para obtener los datos graficados se ha realizado una media de los datos obtenidos en ejecuciones de problemas con las mismas características, ya que la presencia de sumideros puede provocar una gran variabilidad en los resultados a causa de la distribución aleatoria de los sumideros en el tablero.

Para la comparativa utilizaremos tres porcentajes de sumideros: 30%, 50% y 80%. La recta para el 30% aparecerá etiquetada como "P30", la recta para el 50% como "P50" y para el 80% como "P80"

LAO*:

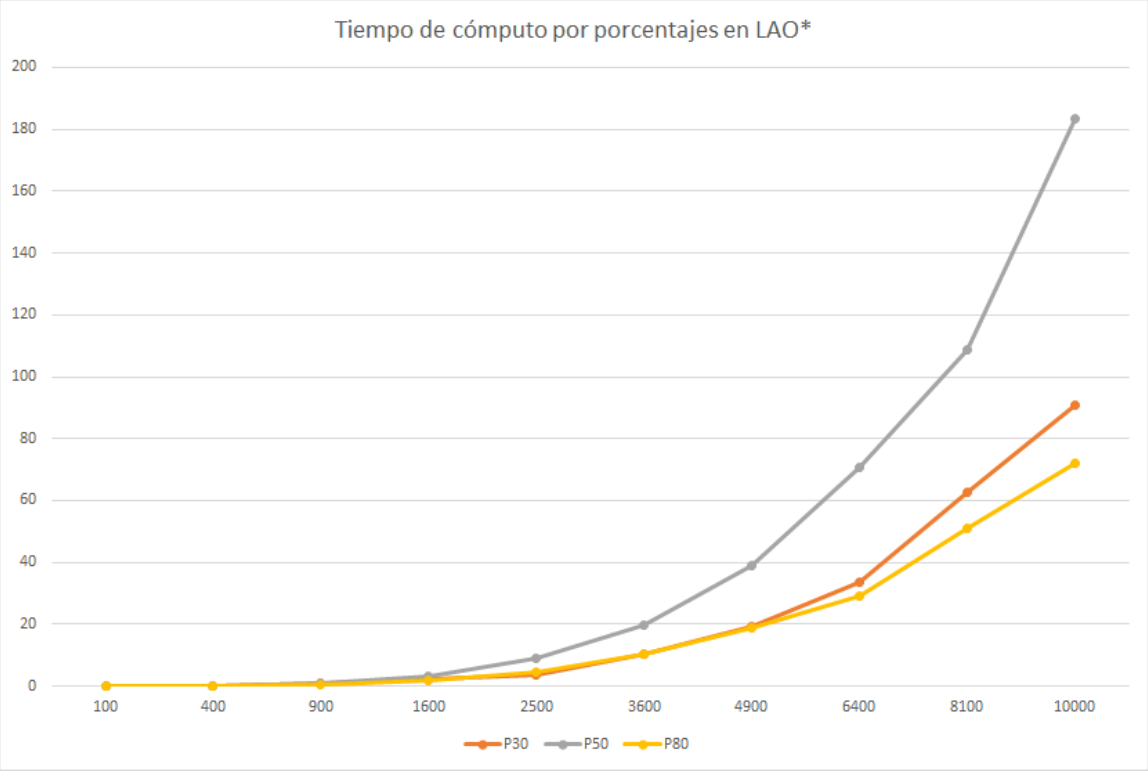

Figura 20. Gráfica de tiempo de cómputo por porcentajes en LAO*

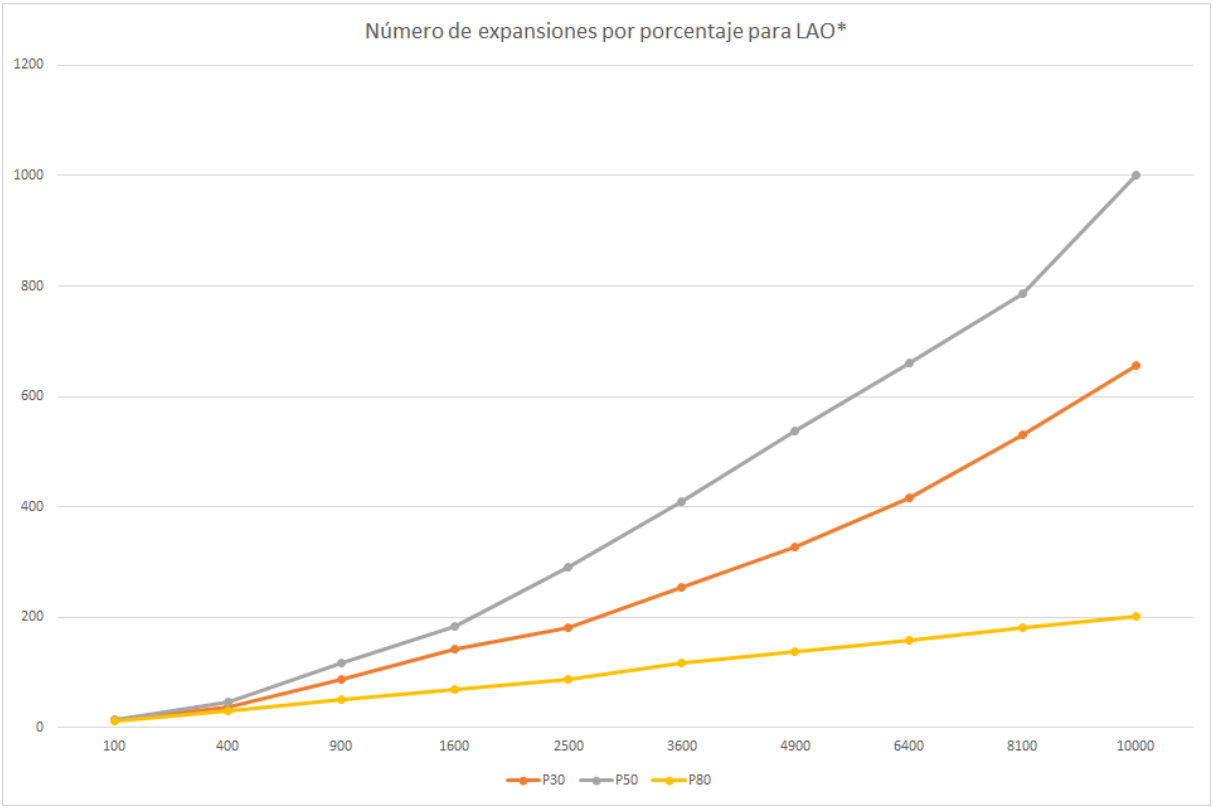

Figura 21. Gráfica de número de expansiones por porcentajes en LAO*

Como bien se puede apreciar en las gráficas, los problemas con un 50% de sumideros son los que requieren un mayor número de expansiones del algoritmo y, por lo tanto, más tiempo de cómputo. La causa que provoca esta situación es la siguiente: intuitivamente un mayor número de sumideros debería estar relacionado con un mejor rendimiento del algoritmo porque hay menos estados que expandir y más transiciones descartables desde cada estado, y esto es así, en parte, porque cuando el número de sumideros no es lo suficientemente grande puede dar lugar a que se expandan muchos estados que no son válidos y sobre los que hay que actualizar los valores, incrementando el tiempo de cómputo. Esto es así porque el heurístico define como "prometedores" a estados que no lo son, ya que el heurístico se rige por la cercanía sin tener en cuenta la distribución de los sumideros en el tablero. ¿Qué pasa si un estado está a dos pasos del estado objetivo, pero está rodeado por sumideros? Según el heurístico, ese estado, por cercanía, sería muy prometedor pero la realidad sería que ese estado no se encuentra en un camino óptimo desde el estado inicial hasta el estado terminal.

Por otra parte, cuando el número de sumideros es demasiado alto, esta situación se compensa, porque se elimina la posibilidad de existencia de estos estados falsamente prometedores, y, en su lugar, se genera un mapa en el que el número de posibles soluciones se reduce considerablemente, a su vez que la cantidad de estados que el algoritmo debe explorar.

ILAO*:

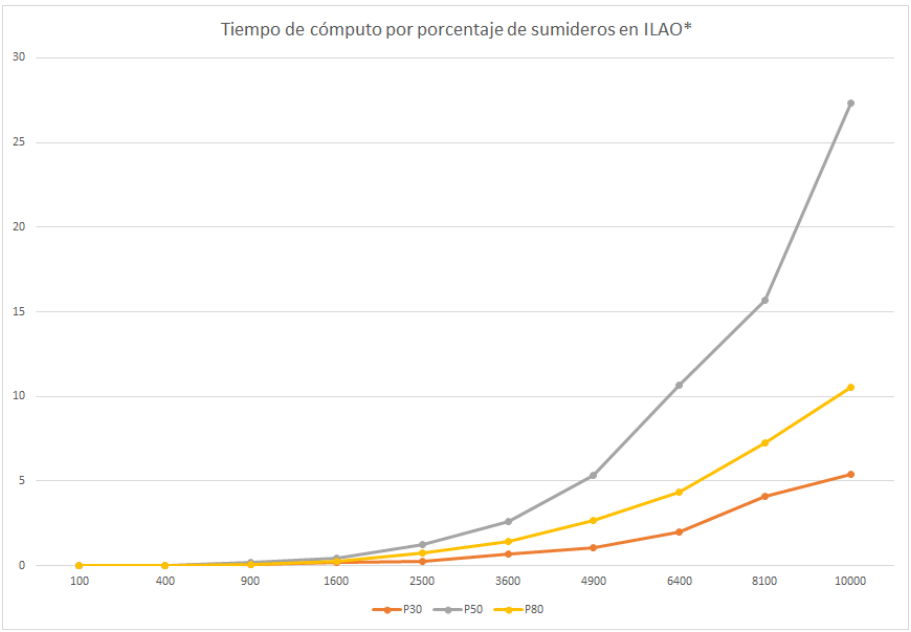

Figura 22. Gráfica de tiempo de cómputo por porcentajes en ILAO*

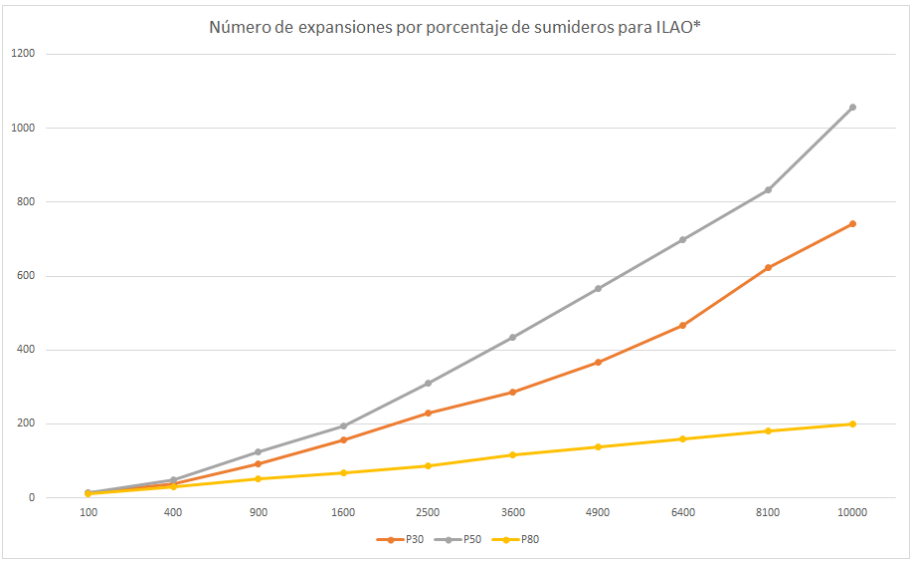

Figura 23. Gráfica de número de expansiones por porcentajes en ILAO*

Para ILAO*, ocurre una cosa similar con LAO*, un porcentaje intermedio de sumideros puede dar lugar a una gran cantidad de estados descartables que van a ser expandidos porque son prometedores. Además, se puede observar cómo, aunque para un 80 por ciento de sumideros el tiempo de cómputo sea mayor, el número de estados expandidos es menor, lo cual parece no resultar intuitivo.

Lo que está ocurriendo es lo siguiente: como ya se aclaró, la parte más pesada computacionalmente de ILAO* es la prueba de convergencia, ya que tiene que ejecutar iteración de valores sobre los estados del grafo solución parcial, pues bien, lo que ocurre es que aunque para un 80 por cierto de sumideros se expandan menos estados, ocurre que el algoritmo tiene que realizar la prueba de convergencia más veces que en el caso de un 30 por cierto de sumideros; esto es así porque la prueba de convergencia se aplica cada vez que el algoritmo encuentra un grafo solución parcial y, a cuanto mayor sea el porcentaje de sumideros, más fácilmente se genera un grafo solución parcial al que aplicar la prueba de convergencia. Conceptualmente lo podemos entender como que aumentar el porcentaje de sumideros incita al algoritmo a estudiar un mayor número de caminos que son posibles soluciones pero que en realidad acaban en callejones sin salidas.

RLAO*:

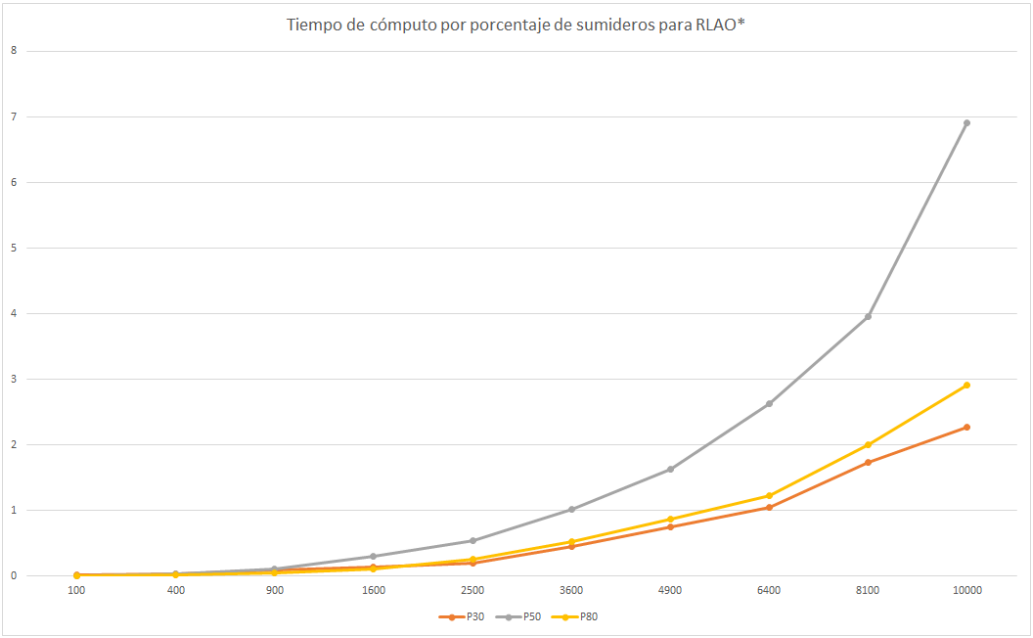

Figura 24. Gráfica de tiempo de cómputo por porcentajes en RLAO*

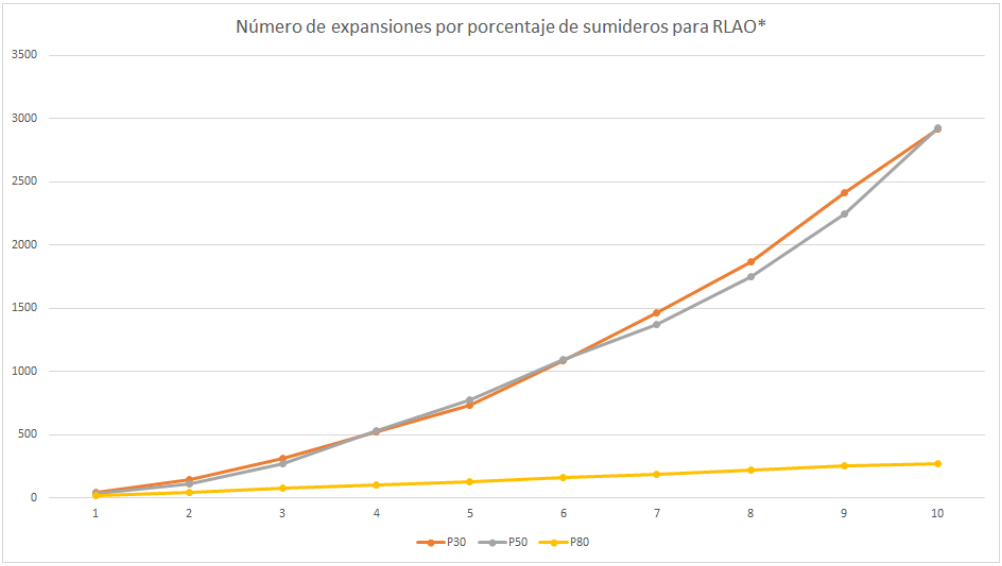

Figura 25. Gráfica de número de expansiones por porcentajes en RLAO*

En este caso pasa algo similar a lo que ocurre con ILAO* ya que ambos algoritmos implementan un procedimiento muy similar. El efecto que tiene un alto porcentaje de sumideros no afecta tanto a RLAO* como lo hace con ILAO*. Esto es así porque el rendimiento del algoritmo se rige principalmente por el fan-in, y, además, como la expansión se realiza desde el estado objetivo hacia el estado inicial y el grafo solución parcial se construye a partir del estado inicial, en todo grafo solución parcial al que se le aplique la prueba de convergencia estará tanto el estado objetivo como el estado inicial, descartando una gran cantidad de caminos que desembocan en sumideros.

BLAO*:

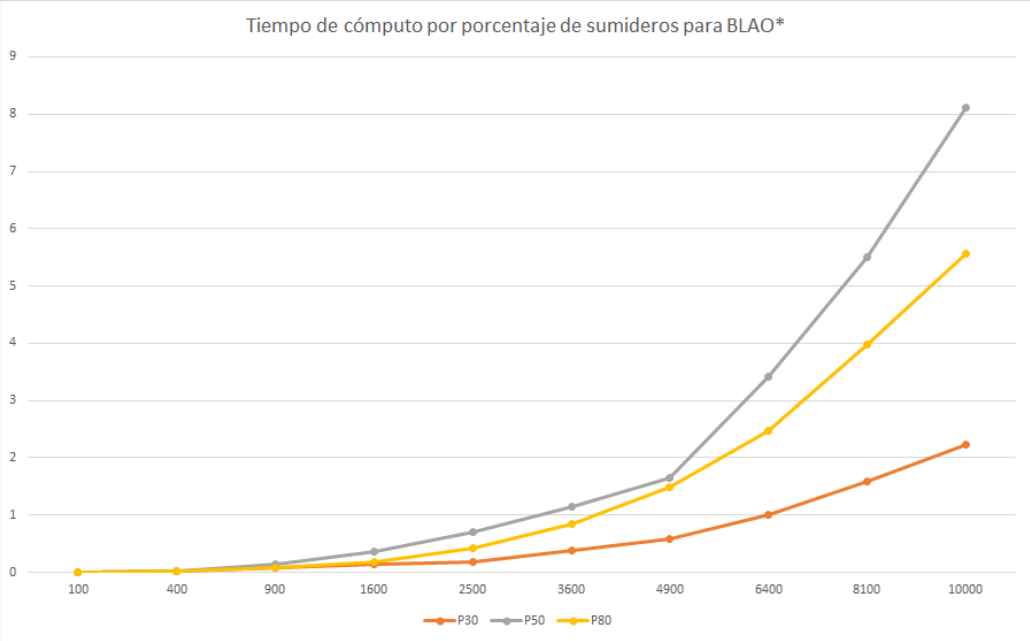

Figura 26. Gráfica de tiempo de cómputo por porcentajes en BLAO*

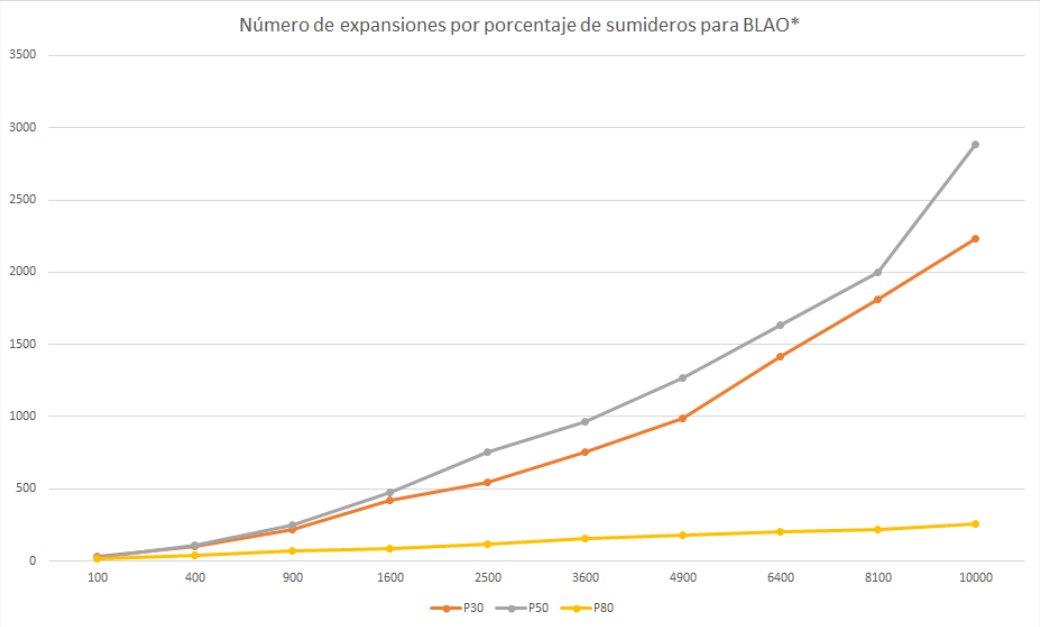

Figura 27. Gráfica de número de expansiones por porcentajes en BLAO*

Al ser BLAO* una combinación de los algoritmos ILAO* y RLAO* en esta implementación, es justificable que se obtengan resultados similares a la de ambos algoritmos.

Iteración de valores:

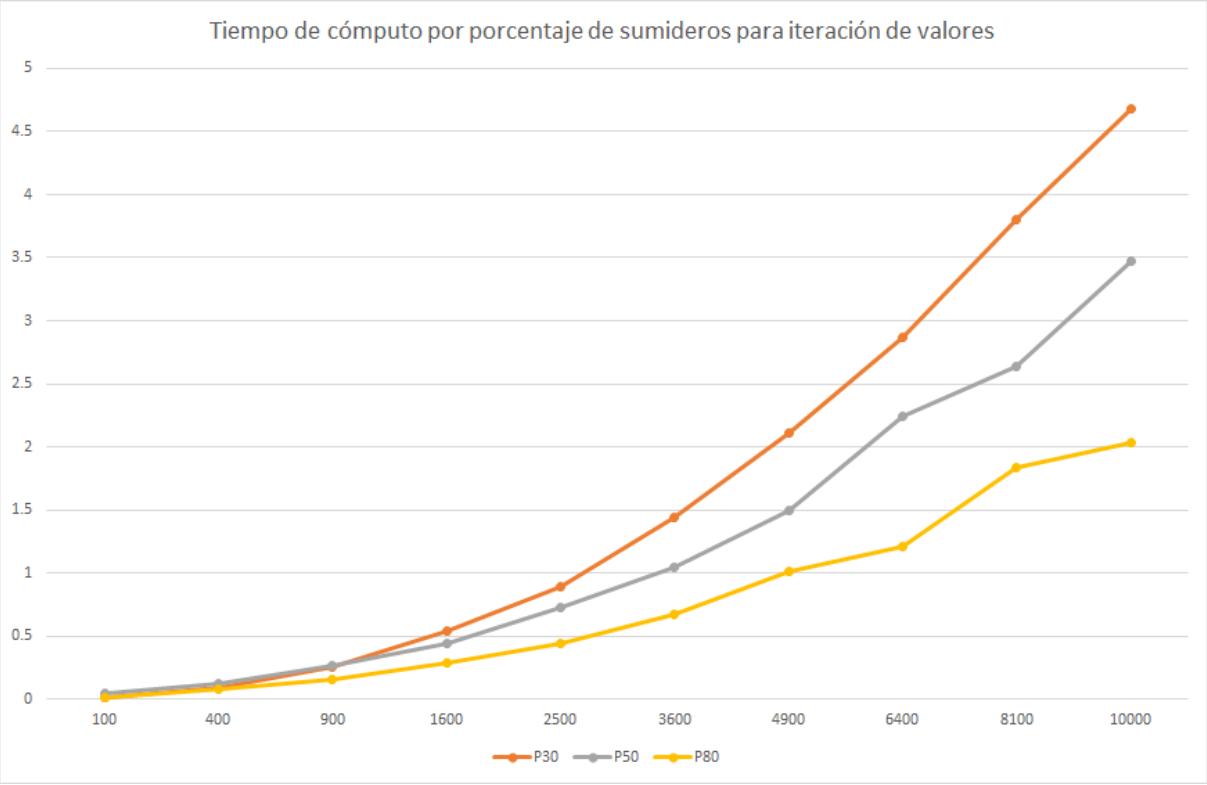

Figura 28. Gráfica de tiempo de cómputo por porcentajes en iteración de valores

Como el algoritmo explora todo el espacio de estados, a cuanto mayor porcentaje de sumideros, menor número de estados que son necesarios expandir y, por lo tanto, menor tiempo de computación. Es un algoritmo notablemente estable a la hora de resolver problemas con sumideros.

7.2.2 Comparativa con mismo porcentaje de sumideros

Una vez estudiado el rendimiento de cada algoritmo en función del porcentaje de sumideros, vamos a ilustrar una comparativa de los algoritmos para cada uno de los tres porcentajes. En cada una de las gráficas el eje de abscisas se corresponde con el número de estados, el eje de ordenadas con el tiempo de cómputo requerido y cada una de las rectas se corresponde con uno de los algoritmos.

30%:

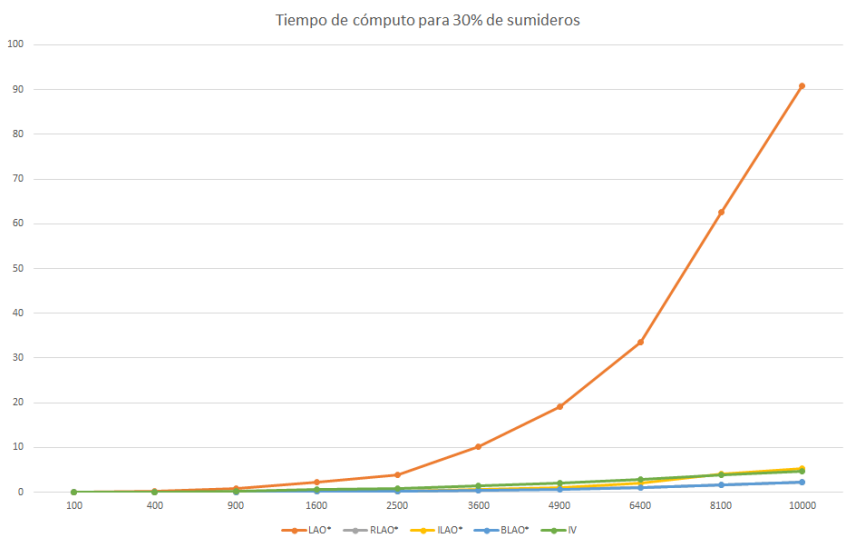

Figura 29. Gráfica tiempo de cómputo por algoritmo para 30% de sumideros

50%:

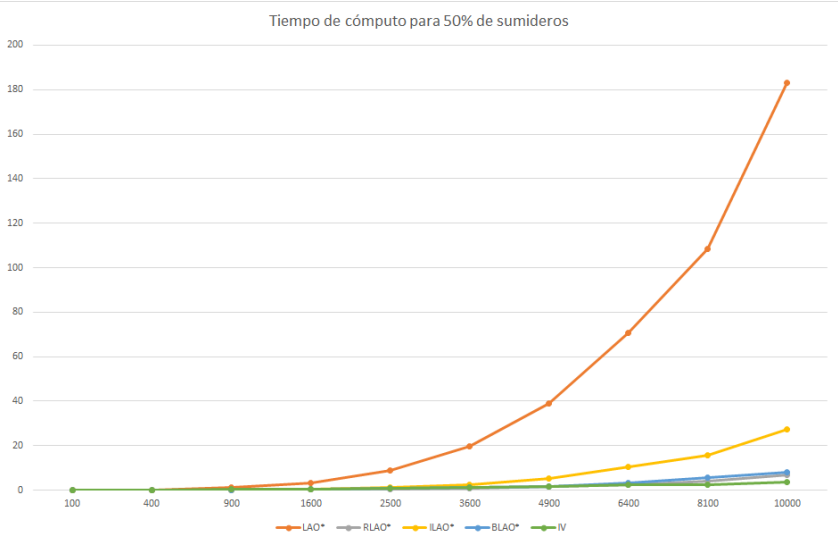

Figura 30. Gráfica tiempo de cómputo por algoritmo para 50% de sumideros

80%:

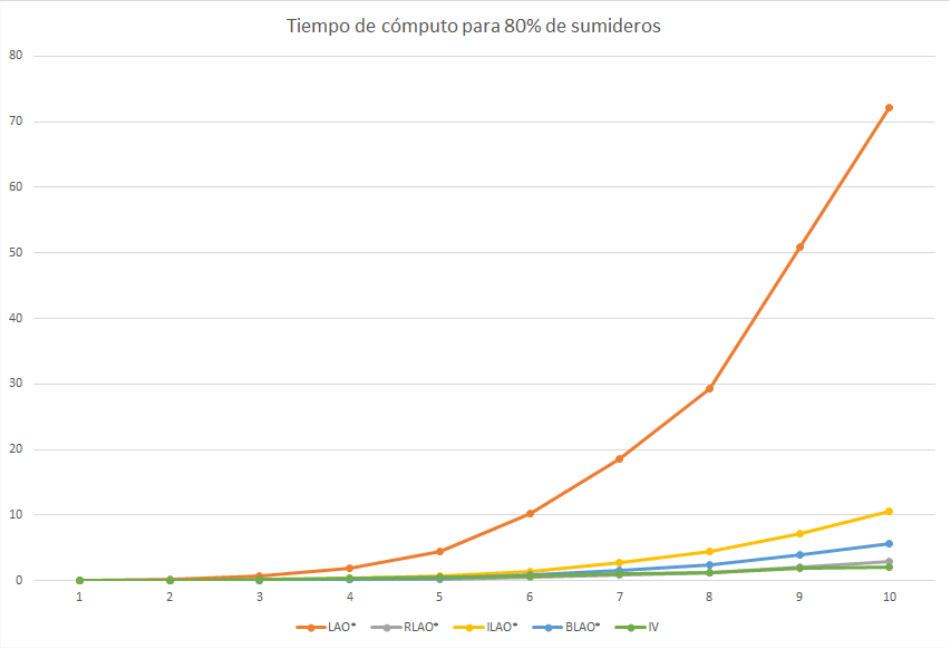

Figura 31. Gráfica tiempo de cómputo por algoritmo para 80% de sumideros

Como bien se puede apreciar, existe cierta variación en la optimalidad de cada algoritmo en función del porcentaje de sumideros. Mientras que BLAO* es el algoritmo que mejor se adapta a un porcentaje bajo de sumideros, iteración de valores suele trabajar mejor con porcentajes medios o altos. También podemos apreciar una decaída en el rendimiento de ILAO* en su aplicación a problemas con sumideros mientras que RLAO* ofrece resultados bastante consistentes. Por otra parte, LAO* sigue siendo el algoritmo que ofrece un peor rendimiento.

8. Conclusiones

En este capítulo obtendremos conclusiones de forma individual sobre cada uno de los algoritmos a partir de los datos obtenidos y graficados en el capítulo anterior.

Iteración de valores:

El algoritmo de iteración de valores por sí solo suele dar buenos resultados en cuestión de tiempo de cómputo y el valor de las variables asociadas a su rendimiento tienen un crecimiento lineal en función del número de estados. Además, tiene la ventaja de que suele dar resultados bastante estables, sin importar la aleatoriedad en la generación del problema. La desventaja principal del algoritmo, como ya se explicó en capítulos anteriores, es que requiere explorar todo el espacio de estados. Además, para problemas con sumideros es necesario utilizar una tasa de descuento para asegurar la convergencia del algoritmo.

En esta implementación el rendimiento del algoritmo es mejor que en una implementación en la que se aplica la tasa de descuento, porque la tasa de descuento limita el número de iteraciones que requiere el algoritmo para converger; a cambio, su uso puede no asegurar converger en la política óptima para problemas con características específicas, por ejemplo, si el espacio de estados es muy amplio.

LAO*:

Este algoritmo no ha dado buenos resultados en lo que refiere a tiempo de cómputo para el problema en el que se ha aplicado. Aunque la cantidad de expansiones sí eran razonables, la necesidad de ejecutar iteración de valores en cada iteración sobre un subconjunto de los estados interiores perjudica notablemente el rendimiento del algoritmo. En general, como se ha observado, es preferible realizar un solo backup por iteración a aplicar iteración de valores completamente.

LAO* se adapta mejor a problemas en los que el ratio entre el número medio de sucesores para una acción concreta y el número total de sucesores por estado es menor, porque este ratio es el que condiciona la porción del espacio de estado que se actualiza en cada iteración y, como ya sabemos, la actualización de valores es la parte más costosa del algoritmo.

ILAO*:

En general ILAO* es un algoritmo que ha dado buenos resultados para problemas sin sumideros pero que no se ha adaptado tan bien a problemas en los que sí había. La presencia de sumideros le transfiere una naturaleza más circunstancial. El sistema de transiciones utilizado también afecta al rendimiento del algoritmo. Para un rendimiento óptimo, debe tener la posibilidad de, en cada iteración expandir un número de estados que no sea excesivamente alto para la expansión de demasiados estados que no van a pertenecer a la solución, pero que tampoco sea excesivamente bajo para poder encontrar un grafo solución parcial en una cantidad razonable de iteraciones.

RLAO*:

Aunque el algoritmo no de buenos resultados para casos sin sumideros como ILAO*, para casos con sumideros ofrece un mejor rendimiento. Como ya se ha comentado, RLAO* depende del número medio de estados que pueden transitar a un estado concreto, llamado fan-in, por lo tanto, cualquier condición que disminuya el fan-in puede mejorar la eficacia del algoritmo, como, por ejemplo, la presencia de sumideros o de obstáculos en el mapa. En general, RLAO* puede adaptarse mejor a este tipo de añadidos a los problemas. Su desventaja está en problemas en los que el fan-in sea muy alto, pero este estudio se ha basado en problemas con un sistema de transiciones en el que el fan-in es bajo, por lo tanto, se han obtenido buenos resultados.

BLAO*:

Como ya se comentó antes, para esta implementación modificada, existe una correlación entre los rendimientos de ILAO* y RLAO* y el rendimiento de BLAO*, ya que éste se ha implementado como una combinación de ambos algoritmos. El algoritmo suele dar buenos resultados y no se aleja mucho del rendimiento del resto de variantes de LAO*. Aunque si es verdad que hay determinados problemas para los que BLAO* funciona mejor, al combinar la búsqueda hacia atrás y hacia delante, ofrece un rendimiento más sólido, menos condicionado por las características del problema, sin ser excesivamente perjudicado por el fan-in ni por el fan-out. Su desventaja principal en esta implementación es que debe

explorar una porción más alta del espacio de estados en comparación con ILAO* ya que implementa las dos búsquedas simultáneamente y tiene que aplicar la prueba de convergencia (iteración de valores) en más ocasiones que RLAO*.

Referencias

[1] *¿Qué es el Machine Learning?* https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico (2019)

[2] *Python (lenguaje de programación).* https://es.wikipedia.org/wiki/Python (2018)

[3] *Gustavo B. ¿Qué es GitHub y cómo usarlo?* https://www.hostinger.es/tutoriales/que-es-github#%C2%BFQue_es_GitHub (2022)

[4] *Visual Studio Code.* https://es.wikipedia.org/wiki/Visual_Studio_Code (2020)

[5] *Eric A. Hansen, Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops*. Computer Science Department, University of Massachusetts, pp. 37-39 (2001)

[6] *S. Sutton, G. Barto. Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massachusetts, London, England. (2014, 2015)

[7] *Peng Dai, Judy Goldsmith. LAO*, RLAO*, or BLAO*?* Computer Science Dept., University of Kentucky (2006)

[8] *Venkata D. Bidirectional LAO* Algorithm (A Faster Approach to Solve Goal-directed MDPs)*. University of Kentucky, pp. 14-16 (2004)

Apéndice A

Manual de usuario

Para poder hacer pruebas con el código, es imprescindible descargar Python. La descarga puede realizarse desde la página web oficial https://www.python.org/downloads/

No existe la necesidad de instalar la última versión, el código solamente utiliza las librerías *time* y *system*, así que servirá cualquier versión de Python que tenga ambas librerías incluidas y actualizadas. Al solo utilizar librerías implementadas en el propio lenguaje, no es necesario la instalación de librerías externas.

Una vez instalado el software, descargamos el fichero con el código y lo descomprimimos donde nos resulte conveniente. Posteriormente, desde un terminal, utilizamos el comando cd para acceder al directorio donde hemos descomprimido el código.

Para hacer uso del comando escribimos cd seguido de la ruta donde se encuentra el código. Si, por ejemplo, descomprimimos el código en el escritorio, el comando se utilizaría de la siguiente forma:

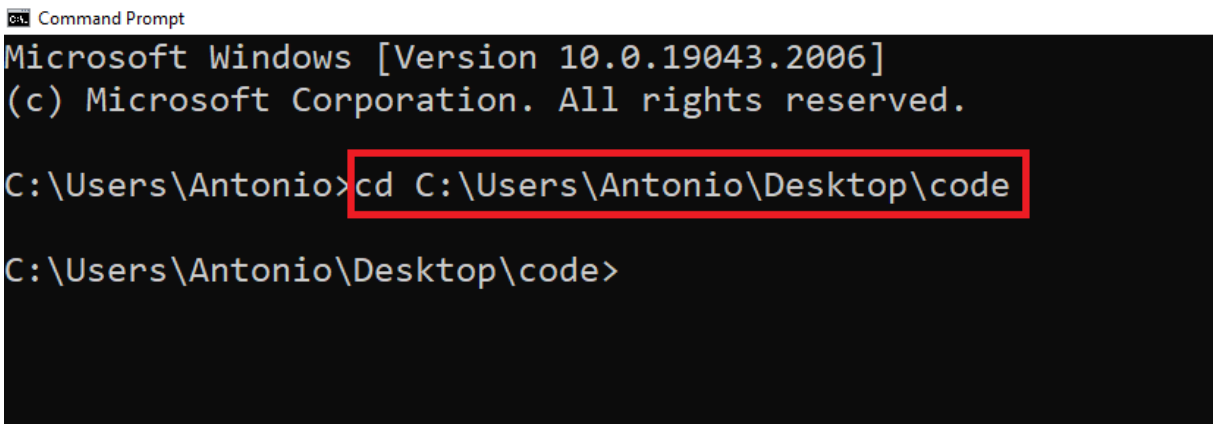

Figura A.1. Uso del comando cd

Una vez situado en el directorio donde se encuentra el código, ejecutamos el comando *python Main.py* para poder ejecutar el código del fichero Main.py

Después, se nos solicitirá escribir por teclado los parámetros de entrada para la generación y resolución del problema. Los datos son los siguientes:

- Número de filas. Filas del tablero donde se realizará la búsqueda.
- Número de columnas. Columnas del tablero donde se realizará la búsqueda.
- Porcentaje de sumideros (0-100). Porcentaje de celdas que queramos que sean sumideros. Es necesario tener cuidado con los porcentajes altos de sumideros, ya que puede provocar que el problema generado no tenga solución.
- Sistema a utilizar (1-3). Sistema transitorio a utilizar. Para porcentajes de sumidero altos, es recomendable utilizar el sistema 3, ya que es el único que asegura que el problema generado tenga solución.
- Algoritmo a utilizar (1-5). Es el algoritmo que queremos que se utilice para resolver el problema.

Además, también se nos ofrecerá la posibilidad de volcar el resultado en un fichero de texto; en caso de que seleccionemos esa opción se generará un fichero con el nombre *"res.txt"* sobre el que se escribirá la solución al problema, además de por consola.

```
C:\Users\Antonio\Desktop\code>python Main.py
Ingrese el número de filas: 30
Ingrese el número de columnas: 30
Ingrese el porcentaje (%) de sumideros: [0-100) 50
SI HAY SUMIDEROS SE RECOMIENDA USAR SISTEMA 3, YA QUE ES EL ÚNICO SISTEMA QUE ASEGURA
Ingrese el sistema que quiera utilizar: (1-3) 3
1 -> LAO*
2 -> RLAO*
3 -> ILAO*
4 -> BLAO*
5 -> VI
¿Qué algoritmo desea utilizar? (1-5) 3
¿Desea volcar resultado en un fichero? (S/N) S
```

Figura A.2. Parámetros de entrada para la ejecución del programa

Aclarar que, en el caso de que demos un valor no válido a alguno de los parámetros, por ejemplo, si concretamos un porcentaje de sumideros mayor de 100, el programa nos volverá solicitar que escribamos un valor válido de forma reitarada hasta que el valor que introduzcamos lo sea.

Una vez especificados los parámetros, se nos generará un cuadro de texto donde se informará de la celda inicial y la celda objetivo en el tablero del problema generado, y, posteriormente, se generará una solución. La solución es un tablero en dos dimensiones con un par de carácteres en cada celda. Para cada tipo de estado su representación en la solución es la siguiente:

- Estado terminal. Se denota con los carácteres 'TT'.
- Estado sumidero. Se denota con los cáracteres '##'.
- Estado no terminal ni sumidero que no pertenece a la solución. Es el conjunto de estados que no se encuentran en el grafo solución. Se denota con los carácteres '..'.
- Estados que pertenecen a la solución. Conjunto de estados que se encuentran en el grafo solución. Especifican la mejor acción a realizar desde ese estado, es decir, la acción que lleva al estado final incurriendo en el menor coste según la siguiente asociación.

Norte	NN
Sur	SS
Este	EE
Oeste	OO
Noreste	NE
Noroeste	NO
Sureste	SE

Suroeste	SO
Ninguno	--

Además, el programa añade una última línea que especifica el tiempo que ha requerido el algoritmo para resolver la instancia del problema.

En el caso del algoritmo Iteración de valores, como no existe un grafo solución propiamente dicho, ya que se expanden todos los estados, la salida generada mostrará la mejor acción a realizar desde cada estado expandido, es decir, la acción que minimice la función de evaluación.

Puede darse el caso en el que, para un problema concreto, no haya solución (esto puede ocurrir si el número de sumideros es muy alto para cualquier sistema transitorio que no sea el 3), en ese caso, los estados hojas del grafo solución no serán necesariamente terminales, y, por lo tanto, en la representación de la solución, el camino solución no desembocará en un estado terminal.

Aquí podemos ver un ejemplo de la solución generada para los parámetros de entrada que han sido ilustrados en la figura anterior.

Figura A.3. Ejemplo representación de una solución

En este ejemplo, como hemos especificado que queremos volcar el resultado en un fichero, se ha generado un fichero con la extensión *.txt* que almacena la solución que aparece ilustrada en la última figura. Es importante aclarar que en caso de que ya que exista un fichero con ese nombre, ese fichero será sobreescrito.