Programación de espacios de estados con Java en asignaturas de inteligencia artificial

Lawrence Mandow*
Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga. Campus de Excelencia Internacional Andalucía Tech
lawrence@lcc.uma.es

Resumen

La representación mediante espacios de estados es un elemento central en los cursos sobre inteligencia artificial simbólica. Las prácticas de programación resultan muy importantes para la asimilación de este concepto. En este trabajo presentamos un software orientado a objetos en Java diseñado para facilitar la asimilación del concepto de espacio de estados y lo comparamos con las clases proporcionadas para la misma tarea en el software aima-java. A continuación se resumen las ventajas e inconvenientes de ambas propuestas, así como la evaluación realizada sobre un grupo de alumnos.

Abstract

State-space representations are a key element in symbolic artificial intelligence courses. Programming assignments are important for students learning this concept. This paper presents an object-oriented Java software designed to facilitate learning of the state-space concept. This is compared with the classes provided by the aima-java software for the same task. This work also summarizes pros and cons of both software frameworks, and presents an evaluation over a group of students.

Palabras clave

Inteligencia artificial, espacios de estados, programación Java, aima-java, sia-java.

1. Introducción

En este trabajo abordamos diversas cuestiones relacionadas con la realización de prácticas de programación de espacios de estados y resolución de problemas en asignaturas de inteligencia artificial. Si atendemos al Computer Science Curricula 2013 (CSC2013) [3], la materia que nos ocupa se encuentra en el área de conocimiento «Intelligent Systems», con un número de 10 horas Core-Tier2, que se mantiene desde la edición de 2001. Más concretamente, los «espacios de problema» y la «resolución de problemas mediante búsqueda» se encuentran en el bloque «Basic Search Strategies» (p. 122), con 4 horas Core-Tier2, junto con otros temas relacionados como los juegos de dos jugadores o la satisfacción de restricciones. En las programaciones docentes que se presentan como ejemplo en el CSC2013 el número de horas dedicadas a este bloque varía entre 2,5 y 18. El libro de texto recomendado como primera opción en todos ellos es «Artificial Intelligence. A modern approach» (AIMA) [5], que dispone de implementaciones en código abierto en diversos lenguajes (Java, Phyton y Lisp) de la mayoría de los algoritmos descritos en el libro.

Se trata por tanto de una materia central en cualquier curso introductorio sobre inteligencia artificial. En la ETSI Informática de la Universidad de Málaga esta materia se cubre en la asignatura «Sistemas Inteligentes», obligatoria en segundo curso en los grados de Ingeniería de Computadores, Ingeniería del Software, e Ingeniería Informática. La resolución de problemas también es un tema central en la asignatura «Inteligencia Artificial para Juegos», ofertada como optativa en los tres grados para alumnos de tercer y cuarto curso. Durante la elaboración de los nuevos grados se eligió Java como lenguaje de programación para ambas asignaturas, al estimarse que sería el más familiar para los alumnos de segundo curso. La experiencia descrita en este trabajo se centra por tanto en dicho lenguaje, si bien algunas de las cuestiones suscitadas son hasta cierto punto independientes del mismo.

Muchos textos docentes de IA inician al alumno en los conceptos de representación y búsqueda resolviendo problemas en entornos conocidos, observables, deterministas y de un solo agente. Bajo estas suposiciones se hace posible la elaboración de *planes* (secuencias de acciones), con la seguridad de que su ejecución

*La presentación de este trabajo está subvencionada por el Plan Propio de Investigación de la Universidad de Málaga - Campus de Excelencia Internacional Andalucía Tech.

posterior tendrá siempre los efectos esperados. Dicho de otro modo, resolver el problema pasa por encontrar una secuencia de acciones que transforme la situación inicial en otra objetivo. Ejemplos como el puzzle-8 o los mundos de bloques son ubicuos en la literatura para este tipo de entornos [5][2][1][4].

Las tres representaciones más habituales en este contexto son:

- Mediante un *grafo abstracto*, donde la estructura interna de cada nodo es desconocida para el algoritmo resolutor del problema. Estos problemas pueden ser resueltos por algoritmos de búsqueda en grafos como amplitud, BID, A* o IDA*.
- Mediante un *vector de variables*, cuyas restricciones y valores posibles son accesibles al algoritmo resolutor del problema.
- Mediante *lenguajes declarativos* basados en la lógica de predicados.

Dichas representaciones se denominan en AIMA *atómicas*, *factorizadas* y *estructuradas* respectivamente (p.57). Desde un punto de vista práctico, la primera es la que require un mayor esfuerzo de programación. Los algoritmos de búsqueda que la utilizan requieren operaciones sobre los estados para consultar si un estado es final así como sus sucesores (es decir, aquellos estados directamente accesibles mediante una única acción) y, en algunos casos, el coste de una transición, o una estimación heurística del coste restante para alcanzar el estado final. La construcción de clases y métodos que implementen esta representación queda en manos del programador. El enfoque basado en la lógica, por el contrario, pasa idealmente por definir lenguajes declarativos que permitan describir de manera homogénea (usando fórmulas de la lógica) cualquier espacio de estados sin necesidad de programar. En el caso de la satisfacción de restricciones la representación emplea variables, valores y restricciones de uso común que se pueden expresar fácilmente, si bien las restricciones específicas de muchos problemas deben programarse también explícitamente.

En este trabajo analizamos la representación basada en grafos abstractos, que es normalmente la primera que suele estudiarse.

La organización de las prácticas de programación es un problema en el que intervienen muchos factores, como el número de horas a dedicar tanto a la teoría como a la práctica, la intensidad de la programación en las prácticas, los objetivos concretos a perseguir o la experiencia en programación de los alumnos. No existe por tanto una solución universal. Una alternativa sencilla es dejar completamente en manos del alumno la programación tanto de los espacios de estados como de los algoritmos de búsqueda. Otra alternativa, que permite guiar mejor el trabajo del alumno, es proporcionar un marco de trabajo software (*software framework*) o más concretamente una arquitectura de aplicación (*application framework*). En este trabajo proponemos una de estas arquitecturas con objeto de iniciar una reflexión sobre distintos aspectos de la misma, señalando sus ventajas e inconvenientes, y describir nuestra experiencia. Más concretamente, comparamos la arquitectura proporcionada por el proyecto aima-java con nuestra propuesta.

Ambas propuestas proporcionan tanto la arquitectura para la definición de espacios de estados como los algoritmos de búsqueda necesarios para resolver problemas. El diseño de estos es relativamente independiente. Nos centraremos aquí exclusivamente en la parte dedicada a la definición de espacios de estados.

La sección 2 introduce algunas reflexiones sobre los conceptos de problema y espacio de estados. Las secciones 3 y 4 describen las dos arquitecturas de aplicación analizadas. En la sección 5 se presentan algunas reflexiones sobre ambas arquitecturas. Una evaluación preliminar de su uso en el aula aparece en la sección 6. Por último se presentan algunas conclusiones y trabajos futuros.

2. Espacios de estados y representación de problemas

En la tercera edición de AIMA [5] (más conocido como *AIMA3e*) se propone una definición formal de *problema* basada en los siguientes elementos (p. 66):

- Un estado inicial.
- La descripción del conjunto de acciones disponibles para el agente, accesibles a través de una función $ACTIONS(s)$ aplicable a un estado s.
- Una descripción de lo que hace cada acción (formalmente, el *modelo de transición*) especificada por una función $RESULT(s, a)$ aplicable a un estado s y una acción a.
- Una prueba para determinar si un estado dado es objetivo o no.
- Una función de coste de caminos que asigna un coste numérico a cada camino.

Los autores aclaran, en una nota al pie, que anteriores ediciones de AIMA así como otros autores utilizan una *función sucesor* en lugar de las funciones $ACTIONS$ y $RESULT$, ya que la primera hace difícil describir «un agente que sabe qué acciones puede probar, pero no lo que consiguen». Se refieren, por tanto, al caso de entornos *desconocidos* (p. 44), o más precisamente a casos en los que el agente no conoce las «leyes físicas» del entorno. Se citan como ejemplos un nuevo videojuego en el que no sabemos qué hacen los botones hasta que los probamos, o una persona que

empieza a jugar al póker sin conocer las reglas.

En nuestra experiencia, una de las dificultades habituales del alumno a la hora de realizar una formalización es la dificultad para distinguir entre la resolución de una instancia de problema concreta y la concepción abstracta de un espacio de estados. Por este motivo introducimos en la medida de lo posible ambos conceptos de manera *separada*. La formalización de un espacio de estados persigue la caracterización de un determinado aspecto del mundo: tanto las situaciones en que se puede encontrar, como las acciones que se pueden realizar en el mismo y los efectos que tendrán. Una vez acotada la parcela formalizada del mundo se puede introducir el concepto de problema: pasar de una situación dada a otra deseada (conocida o no, pero reconocible). De este modo intentamos evitar un error recurrente en muchos alumnos: formalizar el espacio de estados pensando únicamente en cómo resolverían ellos una instancia de problema concreta. Concretamente, un *espacio de estados* vendrá definido por:

- Un conjunto de estados, que representan todas las situaciones posibles en que puede encontrarse una determinada parcela del mundo.
- El conjunto de acciones disponibles para el agente y que le permiten pasar de un estado a otro. Cada una de ellas viene caracterizada por una descripción de su precondición (es decir, cuándo es aplicable) y otra de sus efectos (es decir, los cambios que produce en el estado del mundo). Para su implementación optamos por la función sucesor. Opcionalmente, cada acción puede tener un coste asociado.

Conceptualmente, un agente tiene un *problema* cuando se encuentra en un estado, pero desearía encontrarse en otro. Para un mismo espacio de estados pueden definirse muchas instancias de problema, definidas por:

- Un estado inicial.
- Una condición objetivo que determina qué estados quieren alcanzarse.

3. El software aima-java

El software *aima-java* se encuentra disponible actualmente a través de GitHub [1]. En este trabajo nos referiremos a la versión 1.7.0-Chp4-Complete, que es la última disponible en la página de descargas. Se trata de un conjunto de proyectos Java que incluyen la programación de diversos algoritmos y representaciones utilizados en inteligencia artificial. Concretamente, el proyecto denominado *aima-core* contiene la implementación de numerosos algoritmos descritos en AIMA3e. Entre ellos se encuentran las clases y métodos para la representación de problemas, y algoritmos de búsqueda tales como A*.

[1] `https://github.com/aima-java/`

3.1. Clases y métodos

La figura 1 muestra el diagrama de clases del proyecto aima-core necesarias para programar el espacio de estados del N-puzzle. Por simplicidad no mostramos las clases relativas a la implementación de los algoritmos de búsqueda. Se trata de un conocido puzle correspondiente a un entorno observable, conocido, determinista y de un sólo agente. La representación utilizada para resolver este problema con A* suele ser un grafo abstracto.

El paquete aima.core.environment.eightpuzzle contiene 5 clases, de las cuales al menos 4 son necesarias para poder resolver el problema (*EightPuzzleBoard*, *EightPuzzleFunctionFactory*, *EightPuzzleGoalTest* y uno de los heurísticos *ManhattanHeuristicFunction* o *MisplacedTilleHeuristicFunction*). Estas clases utilizan clases y/o interfaces de otros cuatro paquetes:

- Del paquete aima.core.agent la interfaz *Action*.
- Del paquete aima.core.agent.impl la clase *DynamicAction*.
- Del paquete aima.core.util la clase *XYLocation*.
- Del paquete aima.core.search.framework las interfaces *ActionsFunction*, *ResultFunction*, *GoalTest* y *HeuristicFunction*, y la clase *Problem*.

Para resolver el problema pueden utilizarse las clases incluidas en el paquete aima.core.search.informed (por ejemplo, AStarSearch si se desea utilizar el algoritmo A*) o en aima.core.search.uninformed.

3.2. Ejemplo de uso

El cuadro 1 (arriba) muestra el código necesario para resolver el problema empleando el algoritmo A*. Cuantitativamente, se deben programar 4 clases, utilizar otras 8 clases o interfaces de 4 paquetes diferentes, instanciar 7 objetos y realizar 3 llamadas a métodos.

4. Propuesta software

El software propuesto se ha desarrollado y utilizado en los dos últimos cursos académicos para algunas de las prácticas de la asignatura «Inteligencia artificial para Juegos». Su origen está en el software desarrollado para la asignatura de inteligencia artificial del anterior plan de estudios, en la que se utilizaba como lenguaje de programación Common Lisp. Se encuentra dis-

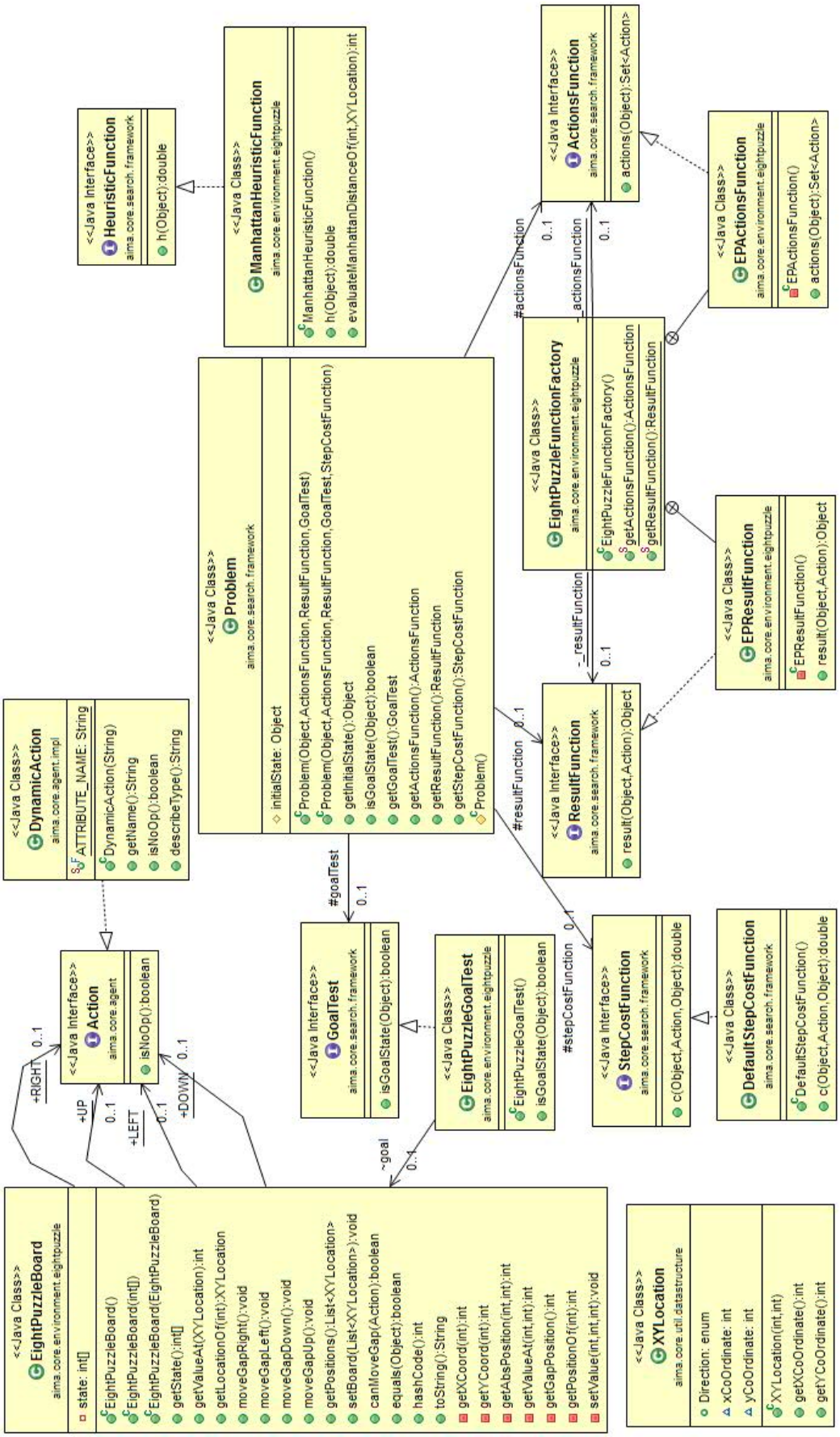


Figura 1: Diagrama de clases para la implementación del espacio de estados del N-puzzle en el framework aima-java.

```
static EightPuzzleBoard p8 = new EightPuzzleBoard(new int[] { 1, 4, 2, 7,
    5, 8, 3, 6, 0 });
Problem p = new Problem(p8,EightPuzzleFunctionFactory.getActionsFunction(),
    EightPuzzleFunctionFactory.getResultFunction(), new EightPuzzleGoalTest
    ());
Search search = new AStarSearch(new GraphSearch(), new
    MisplacedTilleHeuristicFunction());
SearchAgent agent = new SearchAgent(p, search);
printActions(agent.getActions());
```

```
Puzzle s = new Puzzle(3,3);
ProbPuzzleH0 problema = new ProbPuzzleH0(s);
List<Puzzle> solucion = problema.aMono();
```

Cuadro 1: Creación y resolución de una instancia de problema con el software aima-java (arriba) y con el software propuesto (abajo).

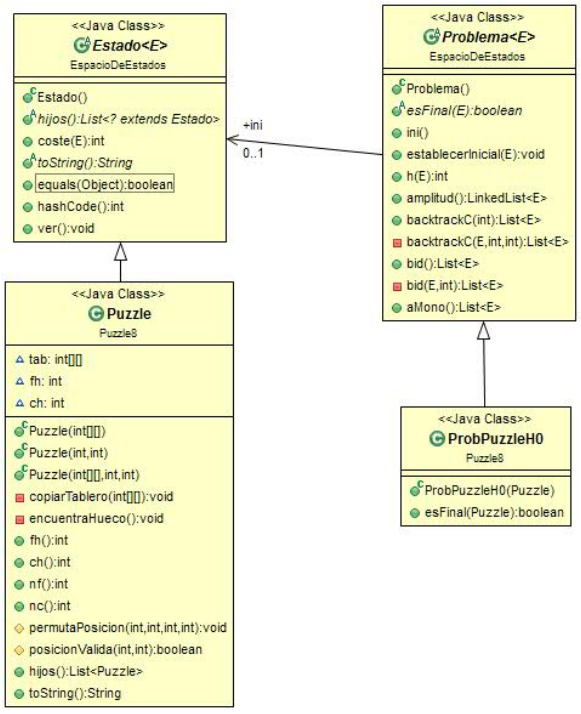


Figura 2: Diagrama de clases para la implementación del espacio de estados del N-puzzle en el framework propuesto.

ponible como código abierto[2]. La figura 2 muestra el diagrama de clases del framework propuesto.

4.1. Clases y métodos

Nuestra propuesta define un paquete *EspacioDeEstados* donde se encuentran 2 clases abstractas relevantes para la definición de un nuevo espacio de estados:

- *Estado*. Incluye los métodos para la representación del espacio de estados: *hijos* (la función sucesor), *toString* y, en su caso, *coste*.
- *Problema*. Incluye los métodos que definen una instancia de problema: *establecerInicial*, *esFinal* y, en su caso, *h* (el heurístico). En esta clase abstracta están definidos algunos de los métodos habituales para la resolución de problemas: amplitud, backtrack acotado, BID y A*. De este modo, cualquier subclase de Problema heredará estos métodos, y sus instancias podrán ser resueltas llamando al método elegido.

La utilización de clases abstractas en lugar de interfaces permite proporcionar código a los alumnos. Por ejemplo, cualquier implementación eficiente de algoritmos como amplitud o A* precisa del uso de tablas hash para el control de estados repetidos. Por ello siempre deben definirse los métodos *equals* y *hashCode* para cada nueva clase de estados. La clase *Estado* proporciona una implementación genérica de estos métodos, siempre y cuando los alumnos definan el método *toString* de modo que a estados diferentes le correspondan cadenas diferentes. Esto debería permitir concentrar aún más la atención del alumno en la programación de los métodos *hijos*, *esFinal*, *h* y *coste*. Si más adelante se considera necesario, pueden definirse métodos *equals* y *hashCode* específicos para cada clase. Algunos alumnos realizan esta tarea espontáneamente.

Uno de los ejemplos que se proporcionan con el código es también el del N-puzzle. Un ejercicio interesante es intentar resolver el problema empleando distintas funciones heurísticas. Entre ellas destacamos $h(n) = 0$, el número de piezas descolocadas, la distancia Manhattan y el uso de bases de datos de patrones. El software proporciona la implementación del primero y el último, ya que habitualmente la programación de los otros dos se propone a los alumnos. Una ventaja

[2] https://github.com/lawrencecode/sia-java

es que se puede definir una clase básica *ProbPuzzleH0* con el heurístico nulo, de modo que las siguientes aprovechen el mecanismo de herencia y sólamente deban redefinir el método correspondiente al heurístico.

4.2. Ejemplo de uso

El código mostrado en el cuadro 1 (abajo) muestra cómo puede crearse y resolverse una instancia de problema. Cuantitativamente, el alumno debe utilizar 2 clases de un mismo paquete, instanciar 2 objetos y realizar una única llamada al método que resuelve el problema. El método $aMono$ corresponde al algoritmo A* (suponiendo heurístico monótono). La lista $solucion$ contendrá el camino solución. Comparativamente, la cantidad de elementos software que deben programarse y emplearse es mucho menor que la mostrada en la sección 3.2 para aima-java.

5. Comparativa

Aún sirviendo una misma finalidad, las dos arquitecturas de aplicación analizadas presentan características muy diversas. Las de aima-java provienen de la caracterización concreta del concepto de problema ya señalada en la sección 2, así como de la precisión adicional al distinguir entre la función de acciones y la de resultados. Además, en aima-java la clase *EightPuzzleBoard* no tiene en principio relación con la clase *Problem*. La arquitectura define una *interfaz* para cada operación necesaria sobre los estados. El problema se construye a partir de las diversas clases, programadas por el alumno, para implementar estas interfaces. Las operaciones están definidas sobre objetos de la clase *Object*, por lo que el programador debe realizar un *casting* al comienzo de cada una a la clase de estados concreta. Nótese que la función heurística no forma parte de la clase *Problem*, sino que se proporciona directamente al algoritmo de búsqueda. La clase *EightPuzzleFunctionFactory* implementa un patrón de diseño Factory. Finalmente, podemos destacar que para la resolución de distintas instancias de problemas es necesario proporcionar nuevamente todos los elementos del espacio de estados a la nueva instancia de *Problem*.

En el caso del software propuesto, la clase que define los estados hereda directamente de la clase *Estado*. En ella se establece como método obligatorio *hijos* (que implementa la ya mencionada *función sucesor*). Para la resolución de problemas se pueden definir subclases de la clase *Problema*, parametrizadas con la clase de estados adecuada. En ellas se define obligatoriamente el método *esFinal* (que implementa la condición objetivo), así como la función heurística empleada. A continuación se pueden crear instancias de problema estableciendo únicamente el estado inicial.

Otra diferencia notable entre ambas alternativas es el tratamiento de los algoritmos de búsqueda. En el caso de aima-java existe una clase para cada algoritmo, que debe ser instanciado para poder resolver el problema. En el software propuesto cada algoritmo es un método de la clase abstracta *Problema* y puede ser invocado para resolver cualquier instancia.

6. Opiniones de los alumnos

Respecto a la evaluación en el aula, se realizó una encuesta voluntaria y anónima entre los alumnos de la asignatura «Inteligencia Artificial para Juegos» (optativa de 3º y 4º, en la que se usa el software propuesto), que a su vez ya hubiesen cursado la asignatura «Sistemas Inteligentes» (obligatoria de 2º, en la que se usa el software aima-java).

El primer dia de clase se presentó una encuesta a los alumnos pidiéndoles, entre otras cosas, que evaluaran su experiencia previa con el software de AIMA en la asignatura «Sistemas Inteligentes». Dicha encuesta fue completada por 21 alumnos. El último dia de clase, se presentó nuevamente una encuesta a los alumnos, pidiéndoles en esta ocasión que evaluaran su experiencia con el software propuesto y valorasen cuál de los dos encontraban más intuitivo. Esta fue realizada por 16 alumnos. El cuadro 2 muestra la distribución de las preguntas y respuestas sobre facilidad de uso, observándose claramente que el software propuesto fue mejor valorado por los alumnos. El cuadro 3 muestra los resultados de una pregunta directa sobre las preferencias de los alumnos, que se realizó a final de curso. Ninguno de ellos manifestó su preferencia por el software aima-java, 3 de ellos consideraron indiferente usar uno u otro, y 11 manifestaron algun grado de preferencia por el software propuesto.

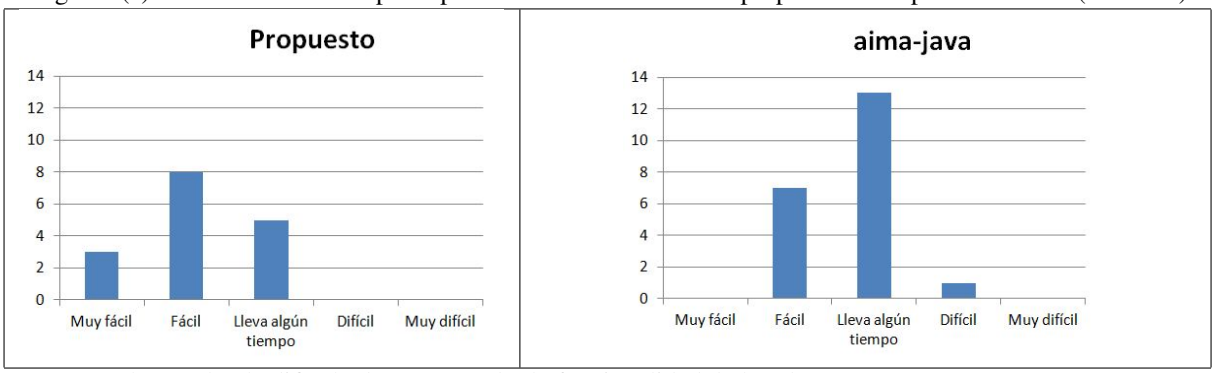
El cuadro 4 recoge algunos de los motivos manifestados por los alumnos para su preferencia por el software propuesto.

7. Conclusiones y trabajo futuro

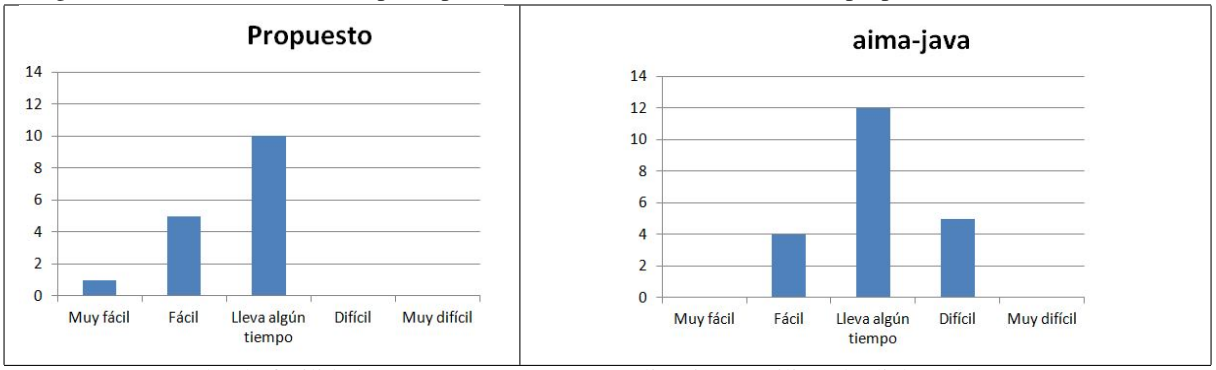
En este trabajo partimos de la premisa de que el concepto de representación y su formalización mediante espacios de estados deben ser elementos centrales en una asignatura de inteligencia artificial simbólica. Por ese motivo, consideramos muy importante su asimilación mediante el desarrollo de prácticas adecuadas.

El tiempo suele ser un recurso escaso en las asignaturas introductorias de IA, que deben tradicionalmente abarcar muy diversas representaciones, métodos y aplicaciones. El software aima-java proporciona una formalización de los espacios de estados en la que se distinguen las funciones de acciónes y resultados. Es-

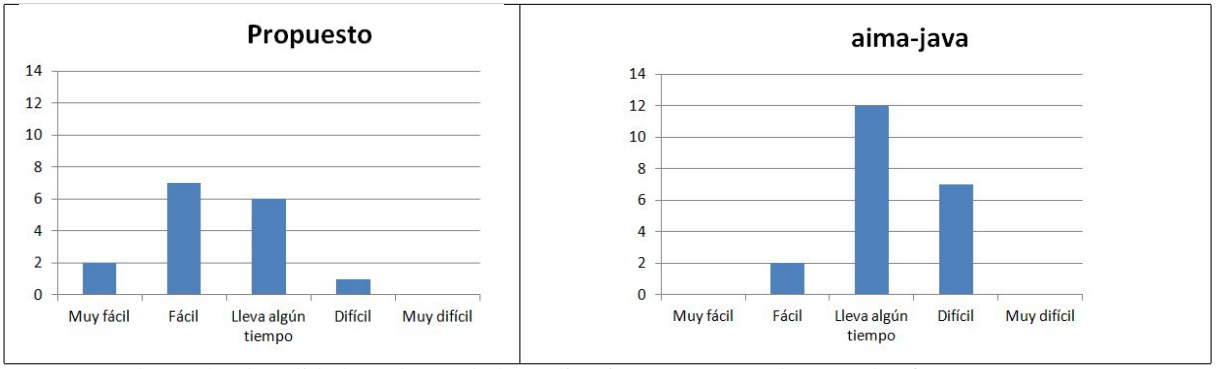
Pregunta (a). Evalúa la dificultad para aprender a utilizar las demos proporcionadas por el software (Puzzle-8):



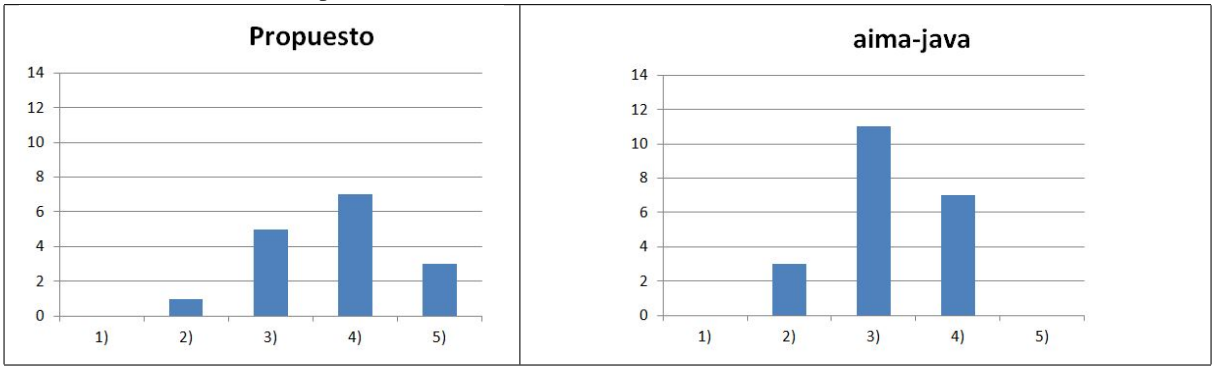
Pregunta (b). Evalúa la dificultad para aprender la funcionalidad de las clases propuestas:



Pregunta (c). Evalúa la facilidad para programar nuevas aplicaciones utilizando dichas clases:

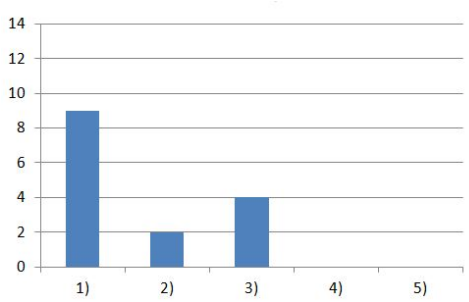


Pregunta (d). Evalúa la calidad resultante de las aplicaciones programadas con el software (valora de 1 a 5, siendo 1 la peor calidad):



Cuadro 2: Resultados de la encuesta (preguntas (a)-(d)).

Pregunta (e). Valora qué software te parece más intuitivo para organizar y programar espacios de estados (1 - Mejor el software proporcionado en la asignatura; 3 - Diferentes, pero ninguno mejor que otro; 5 - Mejor el software aima-java):



Cuadro 3: Resultados de la encuesta (pregunta (e)).

Mucho más legible y fácil de utilizar.
Más flexible. AIMA no lo utilizaría en aplicaciones futuras.
Es más fiel a los contenidos en clase.
Es mucho más abarcable y fácil de entender.
Te anima más a aprender y «trastear».

Cuadro 4: Algunos motivos por los que los alumnos dijeron preferir el software propuesto frente a aima-java.

to puede permitir reutilizar algunas interfaces y clases en prácticas de temas posteriores, aunque incrementa considerablemente el número de interfaces y clases a tener en cuenta para programar un espacio de estados. La elección de aima-java hace necesario valorar hasta qué punto es beneficioso para el alumno aprender a utilizar las clases proporcionadas en aima-java para la programación de espacios de estados y en qué grado se podrían reutilizar en otras prácticas de la asignatura.

El software propuesto, por el contrario, introduce los conceptos de espacio de estados y problema de una manera separada, permitendo asimilar mejor la diferencia entre ambos conceptos. La arquitectura de aplicación es también más sencilla, permitiendo al alumno desarrollar nuevos programas con la definición de pocos métodos en dos únicas clases. La evaluación realizada, aunque preliminar, muestra claramente que el software propuesto es más fácil de entender para los alumnos y les motiva más para la realización de las prácticas de programación de espacios de estados. Estas incluyen normalmente la programación de tres nuevos espacios de estados (dos nuevos puzles y mapas de videojuego).

Normalmente la mayor curva de aprendizaje impuesta por la utilización de un framework se compensa cuanto más crece el trabajo de desarrollo. Por ese motivo será conveniente, antes de plantear las prácticas, evaluar cuidadosamente el tiempo disponible para las mismas, así como su intensidad en programación por parte de los alumnos. Posiblemente proporcionar un framework es una buena manera de organizar y estructurar las prácticas, si bien siempre existirá un compromiso entre la generalidad del marco propuesto y la sencillez de uso del mismo.

Respecto a los trabajos futuros, debemos mencionar que las implementaciones eficientes del N-puzzle normalmente no utilizan *arrays* para representar los estados. Aunque no es habitual a nivel introductorio, se puede proporcionar una representación más eficiente que permita resolver puzles de tamaños mayores. Otra línea de trabajo interesante es la ampliación de esta arquitectura de aplicación para la programación de juegos con adversario y prácticas de aprendizaje por refuerzo.

Referencias

[1] Stefan Edelkamp and Stefan Schrödl. *Heuristic search. Theory and applications*. Morgan Kaufmann, 2012.

[2] Nils J. Nilsson. *Inteligencia artificial. Una nueva síntesis*. McGraw Hill, 2001.

[3] The Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*. ACM and IEEE, 2013.

[4] José Palma and Roque Marín. *Inteligencia artificial. Técnicas, métodos y aplicaciones*. McGraw Hill, 2008.

[5] Stuart Russell and Peter Norvig. *Artificial intelligence. A modern approach*. Pearson, 2010.