Prácticas de Búsqueda Heurística. Curso 2023-2024. Sesión 1. Algoritmos de búsqueda en aima-python

Ramiro Varela, Francisco J. Gil-Gala. Irene Díaz. Juan Luis Mateo. Sistemas Inteligentes. Grado en Ingeniería Informática. EII. Universidad de Oviedo. Campus de los Catalanes. Oviedo

Resumen. Se trata de hacer, en primer lugar, una experimentación con algunos de los algoritmos de búsqueda implementados en el proyecto aima-python sobre el problema del 8-puzzle. Además, se revisará la implementación del algoritmo A* y se harán algunas ampliaciones para mejorar la visualización de los resultados y añadir heurísticos.

Palabras claves: aima-python, 8-puzzle, búsqueda heurística, A*

• Introducción

El proyecto `aima-python` implementa muchos de los algoritmos que se describen en el texto de S. Russel y P. Norvig [1]. Este es el texto más utilizado en las universidades para las asignaturas básicas del ámbito de la Inteligencia Artificial. El proyecto se puede descargar de https://github.com/aimacode. Actualmente, la versión estable de este código es AIMA3e. Está en desarrollo la rama AIMA4e para adaptarla a la última versión del texto [1], pero todavía no está completa.

En la asignatura de Sistemas Inteligentes vamos a utilizar la parte del proyecto que implementa los algoritmos de búsqueda heurística, en particular A* (AIMA4e), y los algoritmos genéticos (AIMA3e). Para las prácticas de búsqueda en espacios, después de descargar el prototipo maima-java4e, cambiaremos el fichero `search4e` (formato Jupyter Notebook) por este otro que tenemos en el CV: `search4e-Busqueda_2023_2024.ipynb`. Para trabajar con estos ficheros utilizaremos el IDE Visual Studio Code.

• Ejercicios para esta práctica

Como ya hemos adelantado, se trata de experimentar un poco con la implementación actual del algoritmo A*, y a continuación introducir algunas mejoras.

Ejercicio 1. Descargar el código y visualizar el contenido del notebook original `search4e.ipynb`, en particular los bloques Problem and Nodes, Queues, Search algorithms: Best-First, Other Search Algorithms, y 8 Puzzle Problems. El objetivo es entender los detalles de la implementación de las clases Problem y Node, y de los algoritmos de búsqueda y cómo estos algoritmos se adaptan a un problema concreto, en principio el 8 puzzle. Hacer algunos experimentos para resolver instancias del 8-puzzle, sin hacer ningún cambio en el código.

Ejercicio 2. En el problema del 8-puzzle, fijar como objetivo el estado definido por (1, 2, 3, 8, 0, 4, 7, 6, 5) que representa un tablero con las fichas ordenadas alrededor de la casilla central que está vacía. Con este objetivo, es necesario modificar la implementación del heurístico h2 (la que viene en la implementación solamente sirve para el objetivo (0, 1, 2, 3, 4, 5, 6, 7, 8). Además, la implementación de h1 tiene un bug que hay que corregir, ya que considera el 0 como una ficha más y cuenta 1 si está colocado en distintas posiciones en el estado y en el objetivo. Hacer algunas ejecuciones para validar los cambios; en particular, comprobar que los resultados son coherentes con la teoría (admisibilidad, dominancia, consistencia). El fichero `Ejemplos8Puzzle.docx` del CV incluye una batería de ejemplos con costes de 5 a 30.

Ejercicio 4. Modificar la estrategia del algoritmo para que no se expandan nodos dummy, es decir aquellos que permanecen en `frontier` después de insertar un nodo con el mismo estado, pero

representando un camino mejor desde el inicial. Para ello, al extraer un nodo de `frontier`, si tiene un `pathCost` peor que el nodo extraído de `reached`, el nodo se descarta y no se expande.

Ejercicio 5. Modificar el código para que se muestren los nodos expandidos y el valor de `f()` de estos nodos. Así se puede analizar la secuencia de estos valores que nos da información muy importante sobre el heurístico que estemos utilizando. Hacer que se muestren también el número de nodos expandidos, el número de estados reexpandidos (para eso necesitamos una estructura `expanded` que registre los estados que ya fueron expandidos). Asimismo, hacer que el algoritmo cuente el número de nodos dummy descartados. Y por último que se muestre el tiempo de ejecución del algoritmo de búsqueda; ello se puede utilizar el método `time.time()`.

Ejercicio 6. Ahora, se trata de experimentar con los heurísticos que son admisibles y consistentes. Estos son h0, h1 y h2. Al ser admisibles, todos deben encontrar soluciones óptimas, y por ser consistentes ninguno de ellos debe reinsertar estados en frontier, ya que cuando se expande un estado, ya se conoce el mejor camino desde el inicial hasta él, como consecuencia de la consistencia. Lo que sí puede ocurrir es que se encuentren caminos mejores para estados que ya están en frontier (lo que en esta implementación se traduce en que se insertan varios nodos, los dummy, con el mismo estado en frontier).

Además, estos heurísticos tienen las siguientes relaciones de dominancia: $h0(n) < h1(n)$ para todo n no final y $h1(n) \leq h2(n)$ para todo nodo n, que se deben reflejar en el número de nodos expandidos por cada uno de ellos.

Ejercicio 7. Ahora vamos a implementar y experimentar con el heurístico h3 (el que solo considera las fichas que están a distancia 2 de su posición en el objetivo) que es admisible pero no consistente. En este caso, debemos observar que se encuentran soluciones óptimas pero que se reinsertan estados en `frontier`. Además, podemos observar que la secuencia de valores de f() de los nodos expandidos puede no ser no decreciente, a diferencia de lo que ocurre con los heurísticos consistentes.

Ejercicio 8. Por último, se trata de experimentar con la versión de A* con ponderación estática, `weighted_astar_search` (podemos llamarla PEA*). Esta versión se caracteriza por usar una función heurística $f(n) = g(n) + (1 + \text{epsilon})h(n)$, con epsilon > 0. Lo normal es utilizarla con el mejor heurístico admisible, en nuestro caso h2. Obviamente, este algoritmo es equivalente a A* con una función $h'(n) = (1 + \text{epsilon})h(n)$, siendo en general h'() un heurístico no admisible. En consecuencia, las soluciones que encuentra PEA* pueden no ser óptimas y dependerán del valor de epsilon. En este caso, hay que registrar el coste de las soluciones y el número de nodos expandidos para comprobar si se cumple o no lo que nos dice la intuición sobre este algoritmo: que encuentre soluciones bastante buenas expandiendo menos nodos que A* con el heurístico h. Recordemos que la teoría lo único que nos dice es que la solución tendrá un coste $C \leq (1+\text{epsilon})C^*$.

Ejercicio 9. Proponer y programar un heurístico que pueda ser razonable, aunque no sea admisible; hacer algunos experimentos y observar los resultados

Ejercicio 10. Experimentos con los algoritmos de búsqueda a ciegas: búsqueda en anchura y búsqueda en profundidad, y comparar los resultados con los del algoritmo A*.

• Bibliografía

- Russell, S. and Norvig P. Artificial Intelligence, A Modern Approach, Prentice All, New Jersey, 4th ed. 2021.
- Nilsson, N., Principles of Artificial Intelligence, Tioga, Palo Alto, CA, 1980.
- Pearl, J., Heuristics, Morgan Kauffman, San Francisco, CA, 1983.
- Aima-Python (https://github.com/aimacode/aima-python)