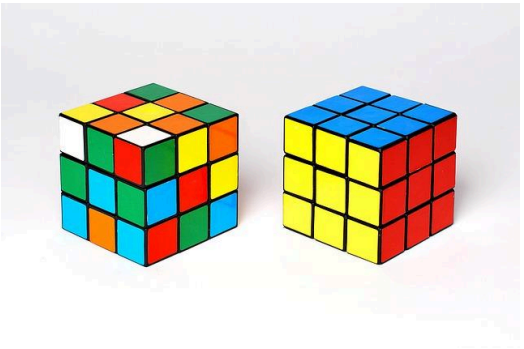
Práctica 1. Búsqueda en espacio de estados

Algoritmos Básicos de Inteligencia Artificial

2023/2024

Descripción de la práctica



El objetivo de la práctica es solucionar el problema del Cubo de Rubik empleando distintos métodos de búsqueda en espacio de estados.

Enlace interesante: Robot LEGO para resolver cubo de Rubik usando IDA*

Objetivos

- Implementar métodos de búsqueda y definir funciones heurísticas.
- Experimentar los problemas de explosión combinatoria.
- Definir métricas para evaluar la eficiencia del proceso de búsqueda.

Tareas a realizar

Se ofrece una implementación de partida en Python con código para modelizar y manejar un cubo de Rubik de tipo 3x3x3, junto con una implementación del método de búsqueda en anchura. Se puede partir del código proporcionado o realizar una implementación propia.

Las tareas concretas a realizar en la práctica serán las siguientes:

1. Implementar el método de búsqueda en anchura.
2. Implementar el método de búsqueda en profundidad y profundidad iterativa.
3. Implementar el método de búsqueda voraz.
4. Implementar el método de búsqueda A*.
5. Implementar el método de búsqueda IDA*.
6. Implementar dos funciones heurísticas distintas para el problema del cubo de Rubik 3x3x3.
7. Definir un conjunto de criterios de evaluación y realizar una batería (lógica y consistente) de experimentos (con 5 y 10 movimientos de mezcla) para comparar el rendimiento de las heurísticas definidas y de los 6 métodos de búsqueda implementados (si es posible).
    - Posibles criterios/medidas a utilizar:

        - Tiempo de búsqueda
        - Núm. total de nodos/estado explorados
        - Tamaño máximo/medio de la lista ABIERTOS
        - Tamaño de la solución encontrada (número de movimientos)
        - otros,...

Nota: Añadir como semilla para la generación de números aleatorios los dos últimos dígitos del DNI de uno de los integrantes del equipo. De esta forma, compararemos los diferentes algoritmos de búsqueda sobre el mismo problema.

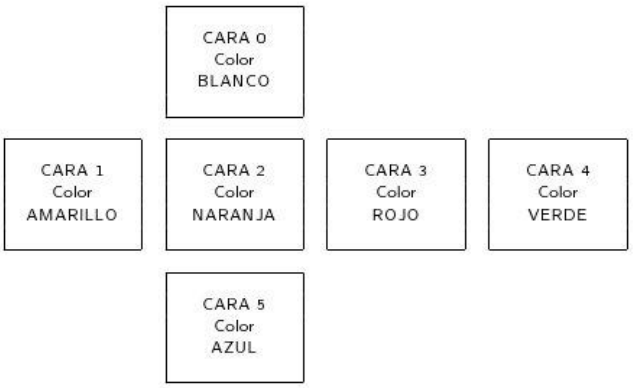
Normas de entrega

- La práctica se realizará de forma individual o en parejas.
- La **fecha límite** de entrega **se publicará a través de Moovi**.
- Se deberá subir a Moovi en un único fichero zip que incluya:
    1. Código fuente de la implementación realizada.
    2. Memoria en PDF con la estructura que se indica más abajo.
- El nombre del fichero zip ha de incluir los apellidos de los integrantes del equipo de la siguiente forma: `fernandez_gonzalez_martinez_rodriguez.zip`
- Estructura de la memoria:
    1. Breve descripción del problema
    2. Descripción de las funciones heurísticas utilizadas
    3. Descripción de la implementación concreta realizada
        1. Detalles de implementación de las heurísticas (si es necesario)
        2. Detalles de implementación de cada algoritmo de búsqueda
            - Explicar las mejoras o adaptaciones que se hayan realizado sobre el algoritmo básico para el caso concreto del cubo de Rubik 3x3x3 (si las hubiera)
            - Explicar las estructuras, funciones, clases, métodos, etc utilizados.
        3. Modificaciones realizadas sobre el código de partida (si las hubiera).
    4. Medidas/criterios usados en la evaluación.
    5. Experimentos realizados y resultados obtenidos.
    6. Conclusiones y problemas encontrados.
- En la semana **indicada a través de Moovi** se defenderá la práctica en el grupo de prácticas correspondiente. En concreto, se comprobará el funcionamiento de la práctica y el modo en que fue desarrollada. Además, se responderán cuestiones formuladas por el profesor sobre la práctica.
- **Nota:** Incluir Nombre, DNI y e-mail en la portada.

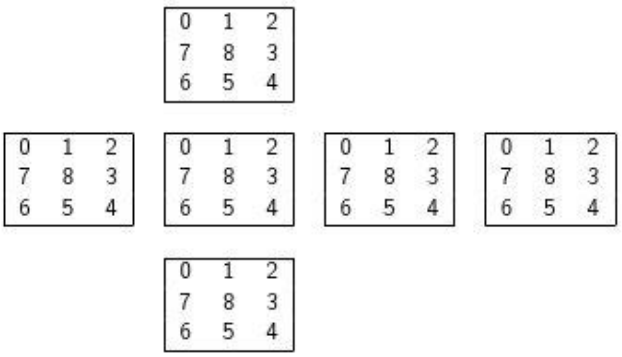
Descripción del problema

Un cubo de Rubik 3x3x3 está formado por 6 caras móviles con 9 casillas en cada una de ellas. Cada casilla está "etiquetada" con el color correspondiente a su cara.

Distribución de las caras



Numeración de las casillas



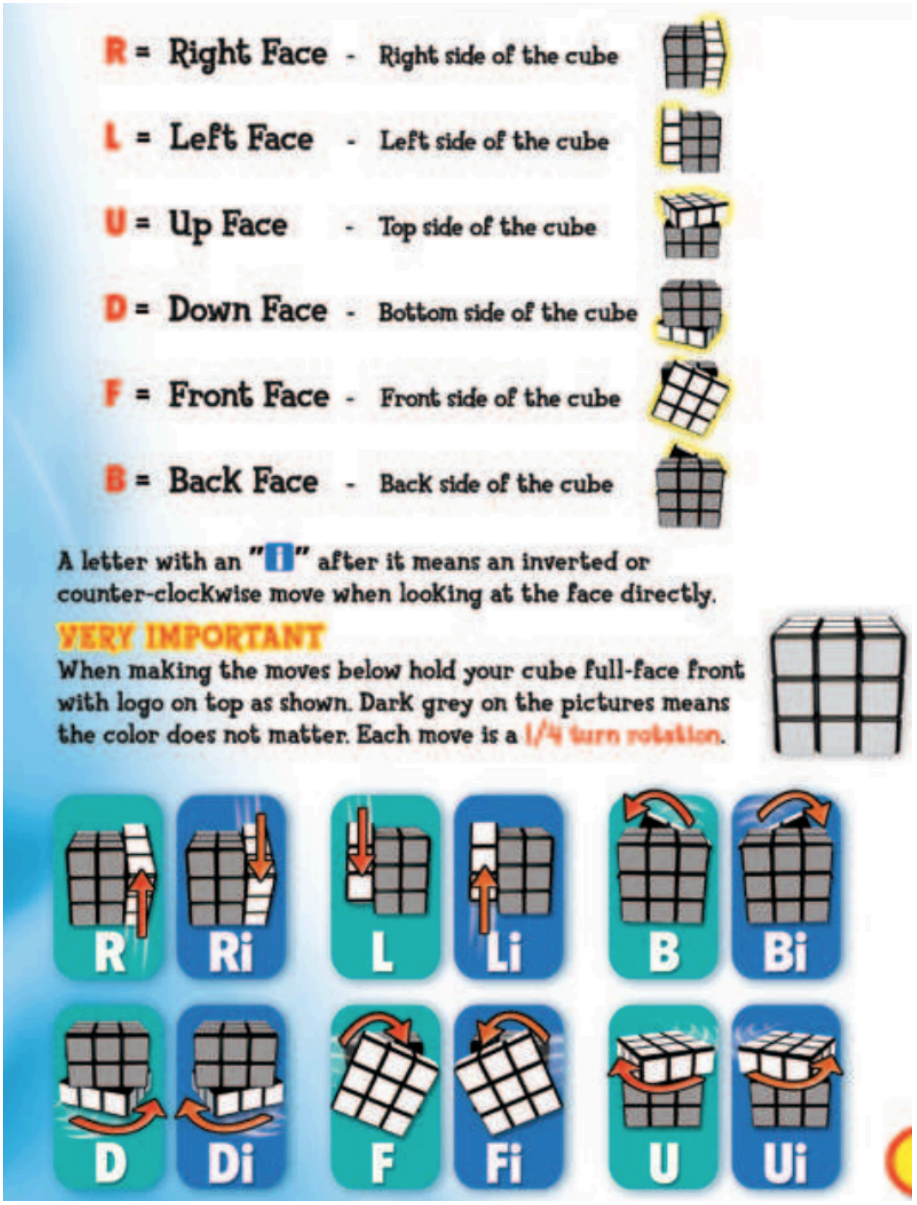
Se sigue la convención de nombrar las 6 caras de acuerdo a su posición desde una perspectiva frontal con la siguiente notación:

UP
cara superior, abreviada con la letra **U** (color BLANCO)
LEFT
cara izquierda, abreviada con la letra **L** (color AMARILLO)
FRONT
cara frontal, abreviada con la letra **F** (color NARANJA)
RIGHT
cara derecha, abreviada con la letra **R** (color ROJO)
BACK
cara trasera, abreviada con la letra **B** (color VERDE)
DOWN
cara inferior, abreviada con la letra **D** (color AZUL)

Dada esta convención, se consideran 12 movimientos posibles en un cubo de Rubik, dos por cada una de las caras anteriores.

- 6 giros de 1/4 de vuelta en el sentido de las agujas del reloj sobre cada una de las 6 caras
    - Notación: **U**, **D**, **L**, **R**, **F**, **B**
- 6 giros de 1/4 de vuelta en el sentido contrario a las agujas del reloj sobre cada una de las 6 caras

- Notación: **Ui**, **Di**, **Li**, **Ri**, **Fi**, **Bi**



Recursos a utilizar

Para desarrollar la práctica se dispone de un código Python de partida que incluye:

- un *framework* de búsqueda en espacio de estados genérico (muy básico), independiente del problema y de los algoritmos de búsqueda.
- una implementación de la búsqueda en Anchura que hace uso de ese *framework.*
- un modelo para gestionar el Cubo de Rubik en memoria (Caras y Casillas).
- clases para integrar el modelo anterior de cubo de Rubik en el *framework* de búsqueda.

El código está estructurado en los siguiente ficheros:

busqueda.py

Incluye:

- Interfaz
    - `Busqueda`: Interfaz a implementar por los métodos de búsqueda.
- Clase
    - `BusquedaAnchura`: Implementación de la búsqueda en anchura haciendo uso de las listas ABIERTOS y CERRADOS.

nodos.py

Incluye:

- Clases:
    - `Nodo`: Superclase con la definición de los nodos (que contienen Estados) a usar en los algoritmos de búsqueda.
        - Para manejar la información extra utilizada por un algoritmo de búsqueda concreto se deberá definir una subclase específica de la clase Nodo.
    - `NodoAnchura`: Definición de los nodos a usar en las búsquedas en anchura. En cada nodo se toma nota del operador empleado para generar el estado asociado (en el nodo raíz es null), para facilitar la generación de la lista de operadores resultante de una búsqueda satisfactoria.

problema.py

Incluye clases abstractas e interfaces genéricos utilizados para la implementación de algoritmos de búsqueda en espacios de estados:

- Interfaz:
    - `Operador`: Interfaz a implementar por los objetos que tomen el papel de operadores.
- Clases:
    - `Estado`: Interfaz a implementar por los objetos que almacenen estados del problema.
    - `Problema`: Problema genérico a resolver mediante búsqueda en espacio de estados. Caracterizado por un Estado inicial y el algoritmo de búsqueda a emplear.

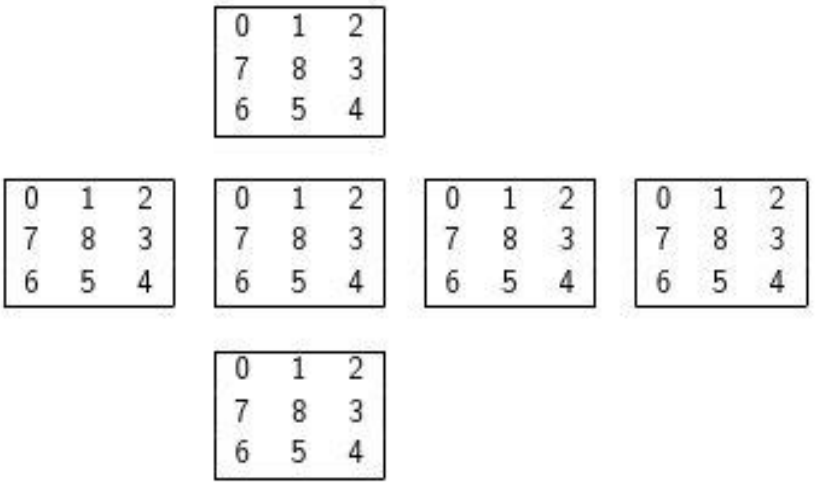
cubo.py

Clases para modelizar y gestionar un Cubo de Rubik

- Clases:
  - `Cubo`: Encapsula la información sobre un cubo de Rubik e información adicional para gestionarlo.
    - Cada Cubo está formado por 6 Caras
    - Incorpora métodos para duplicar y comparar Cubos, determinar si es solución y aplicar movimientos sobre las caras.
    - Incluye arrays estáticos con información sobre la vecindad de cada cara y los índices de las casillas "fronterizas" de cada cara vecina
  - `Cara`: Encapsula la información sobre una cara de un cubo de Rubik e información adicional para gestionarla.
    - Cada Cara almacena en un array de 9 posiciones la situación de sus casillas.
    - La distribución de las Casillas en ese array sigue la distribución *"en espiral"* que se muestra en las gráficas siguientes.
  - `Casilla`: Encapsula la información sobre una casilla de una cara de un cubo de Rubik. Para cada casilla se almacenan dos atributos
    - color: un valor que identifica el color de esa casilla. Es decir, nos indica implíctamente en que cara debería de estar situada.
    - posicionCorrecta: un valor que indica la posición que esa casilla debería ocupar en el array de casillas asociado a la cara a la que pertenece, conforme a la la distribución *"en espiral"*.

Distribución de las caras

Numeración de las casillas



problemaRubik.py

Se incluyen también las clases necesarias para integrar este modelo de Cubo de Rubik 3x3x3 en el *framework* de búsqueda descrio anteriormente.

- Clases:
  - `EstadoRubik`: Implementación del interfaz Estado, contiene un objeto Cubo en el que delega las operaciones del interfaz Estado
    - Por razones de eficiencia, un array con los 12 operadores aplicables se almacenan en un vector estático asociado a la clase para no tener que crearlo cada vez que se le pida al Estado la lista de operadores que son aplicables sobre él.
  - `OperadorRubik`: Implementación del interfaz Operador, que contiene el movimiento aplicado.

Ejecución mediante: **`python main.py <n° movs>`**
