PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2022-1

Tarea 2

Aspectos generales

Formato y plazo de entrega

El formato de entrega son archivos `.py` en los directorios correspondientes a cada pregunta. El lugar de entrega es en el repositorio de la tarea, en la *branch* por defecto, hasta el **lunes 23 de mayo a las 23:59 hrs.** Para crear tu repositorio, debes entrar en el enlace que está en el anuncio de publicación de la tarea en Canvas.

Integridad Académica

Este curso se adhiere al Código de Honor establecido por la universidad, el cual tienes el deber de conocer como estudiante. Se espera que todo el trabajo hecho en esta tarea sea **totalmente individual** en cualquiera de sus aspectos. La idea es que te des el tiempo de aprender estos conceptos fundamentales, tanto para el curso, como para tu formación profesional. Las dudas se deben hacer exclusivamente al cuerpo docente a través de las *issues* en GitHub.

Por otra parte, sabemos que estás utilizando material hecho por otras personas, por lo que es importante reconocerlo de la forma apropiada. Todo lo que obtengas de internet debes citarlo de forma correcta (ya sea en APA, ICONTEC o IEEE). Cualquier falta a la ética y/o a la integridad académica será sancionada con la reprobación del curso y los antecedentes serán entregados a la Dirección de Pregrado.

Comentarios adicionales

El objetivo de esta tarea es que puedan utilizar algoritmos de búsqueda con y sin adversario, como A* y MiniMax, aplicándolos en problemas donde pueden ser de gran utilidad. Por lo tanto, es fundamental que pongan énfasis en las explicaciones de sus respuestas, cuidando la redacción y ortografía, y manteniendo siempre el código ordenado y comentado. Aquellas respuestas que solo presenten resultados o código (sin contexto ni comentarios) no serán consideradas, mientras que tareas desordenadas pueden ser objeto de descuentos.

NOTA: Es de suma importancia que expliques correctamente los archivos de tu tarea con un archivo de explicación en el repositorio. Todos los archivos que crees/modifiques deben estar bien comentados y el README debería **explicar cómo ejecutarlos**, ya que de otra forma no podemos adivinar cómo probar tu tarea, sobre todo si creas nuevos módulos, lo que **podría afectar considerablemente tu nota**. Adicionalmente, puedes dar detalles que faciliten la revisión, como partes logradas, no logradas, errores esperables, etc.

1. DCCubo Rubik (Total: 3 pts.)

Es el año 2131 y eres uno de los cinco humanos que quedan en el mundo, atormentados constantemente por una inteligencia artificial suprema llamada AM, que destruyó la humanidad y ahora cubre la tierra con sus circuitos para terminar su labor. Sin embargo, rendirse no es una opción y estás dispuesto/a/e a combatir con lo que haga falta.

Tras mucha investigación, has encontrado el código fuente de esta malévola máquina, descubriendo que su matriz central consiste en un Cubo Rubik desordenado, el cual se desactiva si lo logras armar correctamente. Para esto, buscas entre tus viejos apuntes del curso Inteligencia Artificial y decides utilizar el algoritmo A* para acabar con esto de una vez por todas.

Modelamiento general del cubo

Para este problema se te entregará un archivo llamado `cube.py` que modela un Cubo Rubik mediante la clase `RCube` (no hace falta que lo modifiques). Este cubo mantiene un sistema de coordenadas con origen en el centro del cubo, donde cada cara del cubo (determinada por la pieza central de una cara) se encuentra en una posición fija. Los movimientos del cubo que cambian de posición alguna de las piezas centrales de las caras no están permitidos. Es decir, solo se pueden hacer los siguientes movimientos:

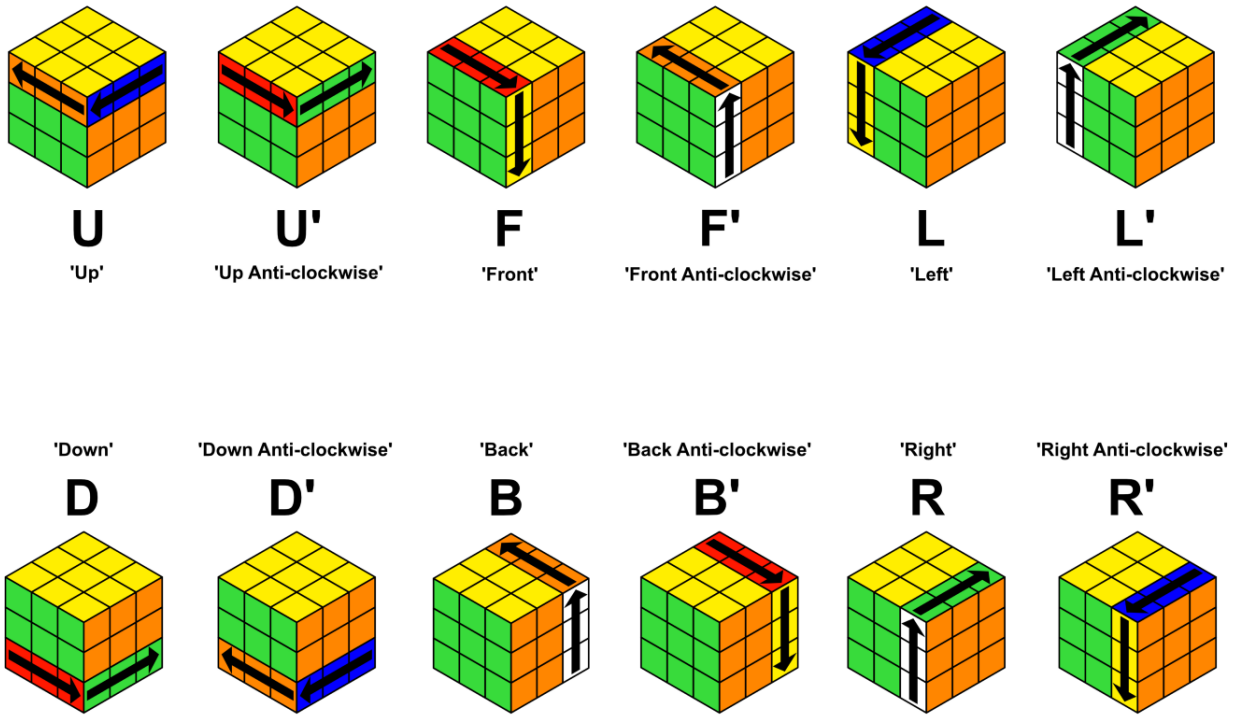

Figura 1: Movimientos Permitidos [1]

[1] https://medium.com/swlh/how-i-learned-to-solve-the-rubiks-cube-in-30-seconds-afff9292b030

Puedes obtener una representación del estado actual del cubo con el método `RCube.flat_str()` y, de forma análoga, se puede crear una instancia de `RCube`[1] a partir de un *string* que represente un estado del cubo usando `RCube(string)`.

Puedes acceder a distintas piezas del cubo mediante indexación de la forma $[x, y, z]$, donde cada uno de estos valores se encuentra entre -1 y 1. También se puede acceder a los atributos `RCube.faces`, `RCube.edges` y `RCube.corners`, que son listas que representan las caras (o piezas de un solo color), `edges` los bordes (o piezas de dos colores) y las esquinas (o piezas de tres colores), respectivamente.

Cada pieza es un objeto de la clase `Piece` y tiene un atributo `pos`, el cual es una tupla (x, y, z) que representa su posición en el cubo; y un atributo `colors`, que es una tupla con las iniciales en inglés de los colores de la pieza según el eje al que sean perpendiculares, en el orden (x, y, z). Puedes ver un ejemplo en la siguiente figura:

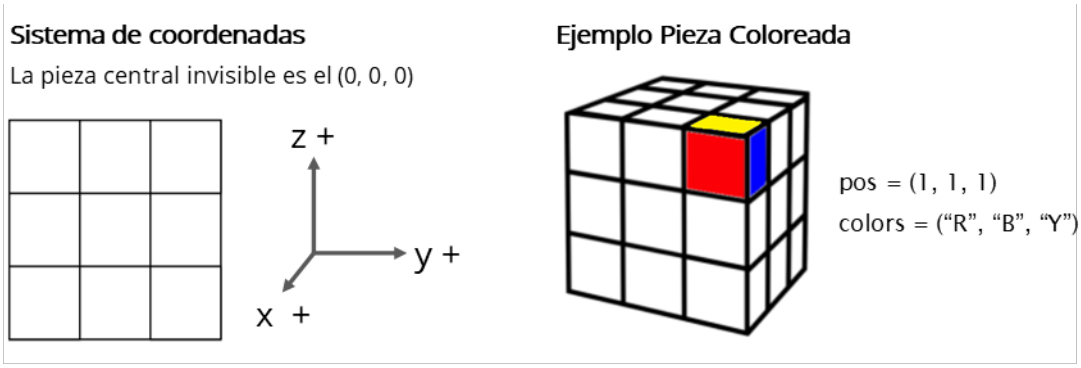

Figura 2: Ejemplo de sistema de coordenadas y atributos

La clase `RCube` tiene un diccionario `moves`, donde se guardan todos los movimientos permitidos y sus inversos como pares *key/value*, respectivamente. Es importante notar que están guardados como funciones de la superclase `Cube` y, para ejecutar un movimiento `move` del diccionario sobre una instancia `cube_1` de `RCube`, se debe llamar de la forma `move(cube_1)`.

Actividad 1: Implementación del algoritmo A* (1 pt.)

Para resolver el cubo, deberás hacerlo implementando completamente el método `search` del algoritmo A* en el archivo `astar.py`. Para esto te sugerimos partir por estudiar el algoritmo, leer el código y aplicarlo al contexto. Se recomienda fuertemente hacer uso de la estructura de datos que se encuentra en los archivos `node.py` y `binary_heap.py` según lo visto en clases/ayudantías[2].

Actividad 2: Implementación de heurísticas (0.5 pts.)

Para esta parte, deberás programar las siguientes heurísticas en el archivo `heuristics.py`:

- `max_manhattan_corners(state)`: Retorna el máximo de las distancias Manhattan que hay entre cada esquina y su posición objetivo; dividido en dos.
- `min_manhattan_corners(state)`: Retorna el mínimo de las distancias Manhattan que hay entre cada esquina y su posición objetivo; dividido en dos.

[1] Deberás instalar la librería rubik-cube. Puedes consultar el repositorio de esta librería si tienes más dudas con el funcionamiento del cubo.

[2] Puedes hacer uso de todo lo visto en clases/ayudantías, aunque recuerda citar correctamente.

- `sum_manhattan_corners(state)`: Retorna la suma de las distancias Manhattan que hay entre cada esquina y su posición objetivo; dividido en dos.

Actividad 3: Admisibilidad de las heurísticas (1 pt.)

1. Demuestra la admisibilidad o no admisibilidad de cada una de las tres heurísticas mencionadas anteriormente y comenta cuál debería ser teóricamente mejor entre ellas (solo entre aquellas que sean admisibles) y porqué. **NOTA:** Para demostrar la admisibilidad, se espera una demostración matemática.

2. Demuestra que toda heurística consistente es también admisible. **Pista:** Considera un problema descrito por un grafo arbitrario, y para un camino desde cualquier estado s hasta un objetivo, escribe la desigualdad de la consistencia. Concluye que $h(s)$ no sobre estima el costo hasta el objetivo.

Actividad 4: Comparación de las heurísticas (0.5 pts.)

Haz una comparación empírica entre cada heurística, tomando en cuenta nodos expandidos y tiempo de ejecución para los ejemplos que se encuentran en el archivo `cubos_ejemplo.txt`. Comenta los resultados observados, analiza si coincide con tu respuesta en la parte 2 y especula sobre los motivos que puedan explicar estos comportamientos. **NOTA:** Toma una muestra de al menos 5 ejecuciones para cada caso.

Bonus (0.5 pts)

Implementa la heurística `max_steps_edges`, que revisa la cantidad mínima de pasos necesaria para llevar cada borde a su posición ideal, y retorna el máximo de estos valores. Demuestra su admisibilidad.

2. DCConecta-4 (3 pts.)

Luego de derrotar a AM con tu implementación de A*, vuelves con los demás supervivientes al refugio a jugar el único juego que les ha permitido mantener la cordura: DCConecta-4. No obstante, luego de tantos años jugando el mismo juego, ya no resulta igual de interesante, por lo que decides modificar una vieja implementación del algoritmo Minimax para que pueda jugar por ti. Eso sí, recuerdas que debes tener en cuenta que la versión que juegan incluye el uso de bombas verticales, horizontales y clásicas, donde estas destruyen una columna completa, una fila completa y una cruz completa, respectivamente.

Archivos del repositorio

En el directorio de la pregunta 2 del repositorio, encontrarás los siguientes archivos:

- **game.py**: Contiene las clases `Connect4` y `Stacks` que controlan la lógica del juego. No debes modificarlo.
- **player.py**: Contiene las clases `HumanPlayer` y `AiPlayer`, las cuales modelan y controlan el juego de jugadores humanos (por si quieres probar/debuguear) o jugadores máquinas (el Minimax), respectivamente. La clase `AiPlayer` debe recibir una función de evaluación al momento de inicializarse. No debes cambiar nada en este archivo.

- **main.py**: Es el archivo de flujo principal del juego, que controla cómo se inicializan las clases y delega las responsabilidades a los demás controladores. Es el que debes ejecutar para probar tu programa. Adicionalmente, tiene las siguientes variables que puedes cambiar para testear tu código:
  - `verbose`: Si su valor es `False`, el programa no imprimirá nada en consola.
  - `player1, player2`: Definen el tipo de cada jugador (humano o máquina). Puedes cambiarlas usando las clases antes mencionadas. Para el correcto funcionamiento del programa, asegúrate que como *token* (tipo de ficha) `player1` tenga `'X'` y `player2` tenga `'O'`.
  - `sleep_time`: Cambia el valor para que los jugadores puedan jugar de forma más rápida o lenta.
- **minimax.py**: Contiene una implementación del algoritmo Minimax como la vista en clases/ayudantías. Deberás modificarlo para la parte 1 de esta pregunta.
- **score.py**: En este archivo se definirán las funciones de evaluación que utilizarán tus implementaciones de Minimax en las partes 2 y 3.

Actividad 1: Poda Alpha-Beta (0.5 pts.)

Para esta parte deberás modificar el algoritmo Minimax que está en `minimax.py`, haciéndolo más eficiente, implementando la poda Alpha-Beta. **NOTA:** La implementación debería ser muy sencilla, con pocas líneas de código.

Actividad 2: Funciones de evaluación (0.7 pts.)

Deberás implementar dos funciones de evaluación en el archivo `score.py` , las cuales reciben un parámetro `board`, que corresponde al tablero actual del juego; y un parámetro `token`, que es un *string* que representa el *token* o tipo de ficha del jugador al que se está evaluando. Las funciones a implementar son:

- `two_next_to(board, player_token)`: Retorna un número n que representa cuántos pares de fichas **pegadas en línea** del jugador `player` hay en el tablero, ya sea vertical, horizontal o diagonalmente. A continuación se muestran algunos ejemplos:

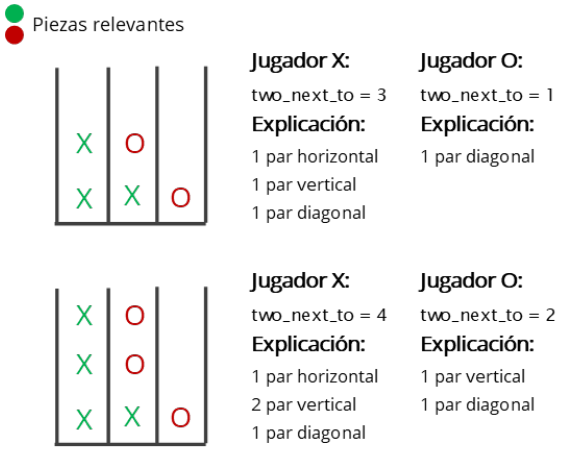

Figura 3: Ejemplo de `two_next_to`

- `three_next_to(board, player_token)`: Retorna un número n que representa cuántos tríos de fichas **pegadas en línea** del jugador `player` hay en el tablero, ya sea vertical, horizontal o diagonalmente. A continuación se muestran algunos ejemplos:

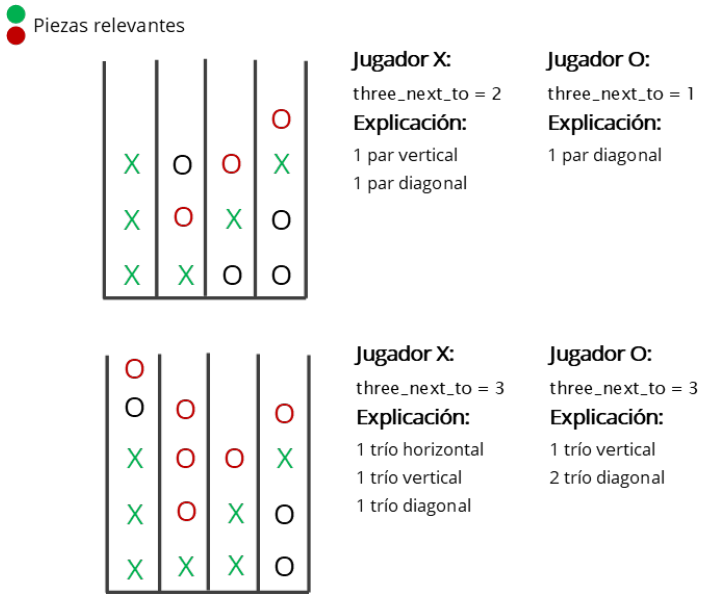

Figura 4: Ejemplo de `three_next_to`

Actividad 3: Comparación de rendimiento (0.8 pts.)

Si bien no quieres jugar por ahora al DCConecta-4, tampoco quieres perder tu prestigio en el juego, por lo que quieres saber cuál de las 2 funciones de evaluación que definiste en la pregunta anterior es mejor. Para esto, deberás crear un archivo llamado `simulate.py`, simulando al menos 100 juegos entre dos jugadores máquinas, donde el jugador 1 ocupe la función de evaluación `two_next_to` y el jugador 2 ocupe `three_next_to`, mostrando la tasa de victorias de cada uno.

En base a los resultados anteriores, deberás hacer un estudio comparativo en un archivo llamado `estudio.pdf`, donde **muestres y expliques** todos estos resultados con gráficos y recursos que aporten a ver si existe algún patrón interesante. Debes mostrar al menos 3 estadísticas o métricas importantes (ej: tiempo de decisión, tasa de victorias, número de jugadas para ganar/perder, etc.). Procura ser ordenado/a/e y explicar lo que estás haciendo. En ningún caso, la respuesta debe tener más de 3 planas en total.

Bonus (0.5 pts.)

Investiga e implementa una nueva función de evaluación, tal que haciendo un estudio comparativo como el de la parte anterior, sea capaz de ganarle al menos el 70 % de las veces a una implementación que use la función de evaluación con mejores resultados en la actividad 2. Este porcentaje de victoria debe ser al menos sobre 100 partidas. Recuerda comentar el código.