PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2023-2

Tarea 3
Problemas de Búsqueda

Fecha de entrega: Lunes 23 de octubre a las 23:59 hrs

Aspectos generales

Formato y plazo de entrega

El formato de entrega son archivos con extensión `.py` con un PDF para las respuestas teóricas. El lugar de entrega es en el repositorio de la tarea, en la branch por defecto, hasta el lunes 23 de octubre a las 23:59 hrs. Para crear tu repositorio, debes entrar en el enlace del anuncio de la tarea en Canvas. Por último, recuerda que los cupones de atraso son días **no hábiles** extra.

Integridad Académica

Este curso se adhiere al Código de Honor establecido por la universidad, el cual tienes el deber de conocer como estudiante. Todo el trabajo hecho en esta tarea debe ser **totalmente individual**. 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. Es fundamental que pongan énfasis en las justificaciones de sus respuestas, cuidando la redacción, ortografía; manteniendo 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.

1. A*: Parte Teórica (1 punto)

Tal como definimos en clases, una heurística h se dice *consistente* si $h(s) \leq c(s, s') + h(s')$ donde s es cualquier estado del espacio de búsqueda, s' es sucesor de s y $c(s, s')$ es el costo de la acción que genera s' a partir de s.

1. Demuestra que cuando A* se usa con una heurística consistente, al agregar un estado s' a Open a partir de s, siempre se cumple que $f(s) \leq f(s')$.

2. Muestra que la propiedad anterior se puede hacer falsa cuando la heurística es multiplicada por un factor $w > 1$; es decir, cuando en vez de usar h usamos otra heurística $h' = wh$.

3. Definimos $\delta(s, s')$ como el costo del camino de menor costo entre s y s'. Muestra que cuando A* es usado con una heurística h que es consistente, entonces cuando un estado s es extraído de Open, podemos asegurar que $g(s) = \delta(s_{init}, s)$, donde s_{init} es el estado inicial del problema. Para demostrar esta propiedad, encuentra una variante del Lema utilizado en la demostración de que A* es óptimo, disponible acá, que en vez de referirse a el estado objetivo, se refiera a todos los estados del grafo.

2. DCCarJam (2.5 puntos)

Introducción al problema

Rush Hour es un puzzle que consiste en mover automóviles estancados en tráfico para lograr que el automóvil rojo pueda salir del mapa (por el lado derecho). El juego sigue la estructura de una grilla en donde cada automóvil puede ser movido en una dirección (eje X o eje Y) siempre y cuando haya espacio libre para ello.

En caso de que desees probar una versión online del juego, puedes acceder al siguiente link para jugarlo.

Figura 1: Versión física del juego Rush Hour

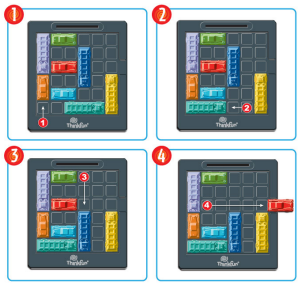

Figura 2: Ejemplo de resolución de un mapa de juego

El objetivo del juego es mover los autos que se encuentren en el camino para que el auto rojo logre su objetivo de salir hacia el lado derecho del tablero, como se muestra en la figura 2.

Implementación entregada

En la implementación del juego con la que trabajaremos esta tarea, el auto rojo será simbolizado con las letras rr. Además, existe un número de otros autos simbolizados con letras mayúsculas, por ejemplo, AA es un auto ubicado horizontalmente que ocupa dos casillas. Estos obstruyen el camino del auto rojo para llegar a su destino (el lado derecho del tablero). Por su parte, el BB es otro auto dispuesto de forma horizontal, mientras que EE es un auto ubicado de forma vertical.

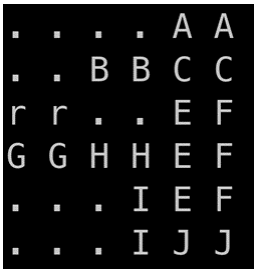

Figura 3: Ejemplo de mapa de juego

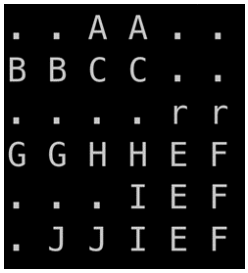

Figura 4: Ejemplo de tablero de juego resuelto

Como puedes ver, los autos `EE` y `FF` actualmente están bloqueando el paso del auto rojo (`rr`). Los autos solo pueden moverse en la dirección en la que están sobre el tablero. En este ejemplo, `AA`, `BB`, `CC`, `GG`, `HH` y `JJ` están posicionados de manera horizontal, mientras que `EE`, `FF` e `I` están posicionados de manera vertical. `rr` siempre está posicionado de manera horizontal.

Una solución es una secuencia de movimientos que permite que el auto `rr` llegue a su destino (lado derecho), de forma satisfactoria.

Archivos del juego

En la carpeta DCCarJam podrás encontrar dos carpetas: `game/` y `tests/`.

Dentro de la carpeta `game/` encontrarás los siguientes archivos:

- `binary heap.py` y `node.py`: Dos estructuras de datos necesarias para trabajar con A*. Es la misma implementación utilizada en clases. No debes modificar este archivo.
- `board.py`: Este archivo se encarga de manejar la lógica del juego. Debes modificarlo y debes considerar los atributos definidos:
  - `name`: Es un carácter.
  - `coord`: Coordenada del auto en el tablero, es un arreglo de arreglos.
  - `length`: Largo del auto
  - `orientation`: Alineamiento horizontal o vertical, tiene dos valores self.Orientation.Vertical o Orientation.Horizontal
  - `is red car`: indica el auto por ser liberado.
- `solver.py`: Este archivo contiene el algoritmo A*. Es la misma implementación vista en clases pero adaptada al problema de DCCarJam. No es necesario modificarlo.
- `solver2.py`: Este archivo contiene la implementación de otro algoritmo que se utilizara más adelante en la tarea. No es necesario modificarlo.
- `run.py`: Este archivo permite correr los algoritmos implementados.

Por otro lado, en la carpeta `tests/` encontrarás mapas de juego sobre cuales ejecutar los algoritmos de búsqueda con que se trabajará dentro de la tarea.

Dentro del archivo `solver.py` encontrarás una implementación del algoritmo de búsqueda A*

Ejecución del programa

Para correr A* debes ubicarte en /DCCarJam y correr el siguiente comando:

```
python game/run.py astar <board> <heuristic> <weight>
```

Por ejemplo:

```
python game/run.py astar tests/test1.txt h0 1
python game/run.py astar tests/test1.txt h1 1.5
```

El algoritmo de A*, estudiado en clases es un algoritmos de búsqueda para obtener soluciones óptimas a un problema de búsqueda. En el archivo `solver.py` podrás encontrar una implementación del algoritmo similar a aquella vista en clases.

2.1. Trabajo con heurísticas, parte 1

1. En el archivo `board.py`, encontrarás 3 heurísticas desarrolladas por el equipo docente del curso, estúdialas y describe brevemente a cada una de ellas, su funcionamiento y qué es lo que computan.

2. Explica por qué las tres heurísticas presentadas anteriormente son consistentes.

2.2. Trabajo con heurísticas, parte 2

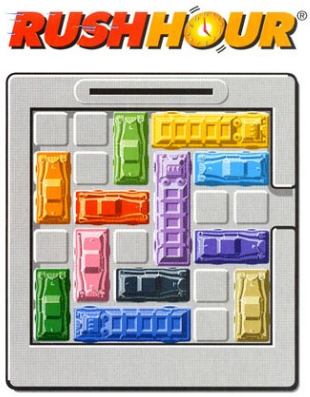

Figura 5: Mapa de ejemplo para el juego Rush Hour

1. Programa una nueva heurística en la función `h3()` que, ojalá, estime el costo mejor que las ya proporcionadas. En el archivo `board.py`, debes dirigirte al método h3, y modificar el apartado. Explica el funcionamiento de ésta mediante comentarios en tu código.

   Recuerda que para ejecutar el algoritmo con la nueva heurística debes ejecutar el siguiente comando:

   python game/run.py astar tests/test7.txt h1 1

2. ¿Es la heurística creada por ti admisible? Demuéstralo.

3. Ejecuta A* (recuerda que para ello `weight = 1`) probando las diferentes heurísticas implementadas, luego, lleva a cabo una comparación empírica para determinar cuál heurística encuentra soluciones de forma menos costosa. Se espera que presentes los resultados en una tabla, reportando número de expansiones y tiempo de ejecución para cada una de las heurísticas entregadas en cada uno de los test entregados.

4. Repite la evaluación anterior usando dos pesos mayores que 1 a tu elección y comenta sobre los resultados obtenidos.

3. SuperGato (2.5 pts.)

Introducción al juego

El juego es el mismo que el explicado por Vsauce en su video sobre *Super Tic Tac Toe*[1]. Este juego puede ser visualizado de mejor manera en la siguiente página web.

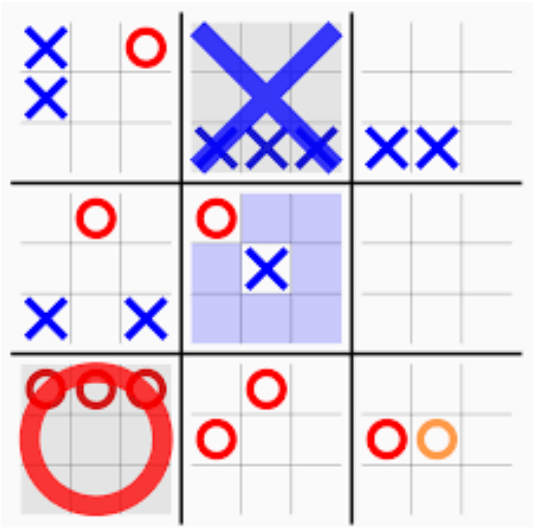

Figura 6: Ejemplo de una partida de SuperGato

El objetivo de esta parte de la tarea es **programar un agente MiniMax** para que juegue inteligentemente a SuperGato.

Archivos mínimos que deben entender y/o manipular

En esta categoría se describen los archivos, clases, métodos y atributos que serán necesarios al momento de trabajar. NO es necesario que comprendas en detalle el resto del código.

- `main.py`: Este archivo contiene la lógica para correr el juego, es decir, los turnos y las instancias de los jugadores.
- `minimax.py`: En este archivo se debe completar el algoritmo minimax (se detalla más adelante lo que se debe hacer). En específico se debe completar la siguiente función:
  - `minimax(tablero, profundidad, alpha, beta, funcion_puntaje, maximizar, ficha)` donde sus parámetros corresponden a:
    - `tablero`: tablero del juego.
    - `profundidad`: profundidad con la que se quiere ejecutar el algoritmo.
    - `alpha` y `beta`: para la poda alpha y beta.
    - `funcion_puntaje`: la función de puntaje que usará el algoritmo para evaluar jugadas.
    - `maximizar`: una variable booleana que indica si se debe maximizar o minimizar el jugador.
    - `ficha`: string de la ficha del jugador de turno.
- `jugador.py`: contiene las clases y lógica de los tipos de jugador (humano o inteligencia artificial).

[1] Explicacion del juego: `https://youtube.com/shorts/_Na3a1ZrX7c?si=W20QUmYFQwq2DEyb`

- `score.py`: contiene las funciones de puntaje. En este archivo deben completar la función `contar_fichas_seguidas` y deben implementar su propia función de puntaje (se detalla más adelante ambas actividades). Todas las funciones de puntaje reciben las siguientes dos variables:
  - `tablero`: tablero actual del juego.
  - `ficha`: ficha del jugador a evaluar.
- `tablero.py`: Este archivo contiene la lógica del tablero y del juego en sí, por lo que se recomienda revisarlo para la implentación de las funciones de puntaje.

  Se recomienda implementar un script que permita ejecutar múltiples veces una partida para el ánalisis solicitado.
  Por último, debes entregar junto a tu código un archivo llamado **`estudio.pdf` que contenga los análisis solicitados** en las siguientes actividades.

3.1. Actividad 1: Minimax (1.5 pts)

En esta actividad, **deberás completar la implementación del algoritmo Minimax** que se encuentra en el archivo `minimax.py`[2]. Para ello, lee bien los comentarios incluídos en el código y trata de seguir la estructura solicitada. Resuerda que no es necesario investigar los archivos, atributos, clases o parámentros del código fuera de los que se mencionaron explícitamente en este enunciado.
Luego, para mostrar que la implementación funciona correctamente, **debes hacer 5 ejecuciones del algoritmo usando la función de puntaje `contar_tableros_diferencia` (ya está implementada), y usar profundidades de 1, 3, 5** [3]. Para mostrar esto debes hacer **una tabla para cada profundidad**, que indique para cada partida la ficha ganadora y el tiempo de ejecución, junto con el tiempo total de las 5 ejecuciones.
Por último, haz un breve **comentario de porque crees que el tiempo de ejecución aumenta** al cambiar la profunidad y como se puede **evidenciar en este juego** en particular.

3.2. Actividad 2: Análisis (3.5 pts)

En esta parte deberás hacer un estudio del efecto que tiene Minimax con sus diferentes parámetros y funciones de puntaje en el desempeño del juego.

3.2.1. Análisis de funciones de puntaje (2 pts)

Junto con el código base, te entregamos una función de puntaje ya implementada:

- `contar_tableros_diferencia(tablero, ficha)`

La cual está explicada con comentarios en el código del archivo `score.py`.

Para esta sección, deberás crear una nueva función de puntaje que llamaremos **`contar_fichas_seguidas(tablero, ficha)`** que cumpla con las siguientes reglas:
(Supongamos que estamos contando el valor para la ficha **X**)

1. Por cada subtablero, si el ganador del subtablero es la ficha **X**, suma 5 puntos.
2. Por cada par seguido de fichas del jugador (en este caso la ficha **X**) en un subtablero que no tenga ganador, se suma 1 punto.
3. Si existen 3 subtableros consecutivos ganados (es decir, si **X** gana el juego), suma 15 puntos.

Luego de implementar esta función, debes incluir los resultados de los siguientes experimentos:

[2] Debes implementar las secciones marcadas con `TODO`
[3] Es decir, 5 ejecuciones para profundidad 1, 5 ejecuciones para profundidad 3 y 5 ejecuciones para profundidad 5

- Al menos 10 juegos en los que el jugador que comienza es un jugador Minimax utilice la función de puntaje **`contar_tableros_diferencia(tablero, ficha)`** y el segundo es un jugador Minimax utilice la función de puntaje **`contar_fichas_seguidas(tablero, ficha)`** ambos con profundidad 3.[4].
- Replicar el exprimento anterior, pero ahora comienza el jugador Minimax con función de puntaje **`contar_fichas_seguidas(tablero, ficha)`** y el segundo jugador Minimax utiliza **`contar_tableros_diferencia(tablero)`** ambos con profundidad 3.

Por último, comenta brevemente que **hace una función de puntaje** en Minimax. Además analiza los **resultados obtenidos y comenta acerca de como se comportan estas funciones de valor**, es decir, si hay algún patrón sobre como juegan y comenta acerca de cual función de puntaje es mejor según lo observado. ¿Cambia algo en los resultados si comienza una o la otra?.

3.2.2. Implementación de nueva función de puntajes (1.5 pts)

En esta actividad debes **implementar tu propia función de puntaje** en base a alguna idea innovadora que creas que puede funcionar bien[5]. Debes **explicar cual es el sentido de esta** y luego hacer los siguientes experimentos:

- Al menos 10 juegos en que un jugador Minimax utilice la **función de puntaje implementada en esta actividad** y el otro jugador Minimax debe usar **la mejor función de valor obtenida en la actividad anterior**, **ambos con profundidad 3**. Queda a tu criterio cual de las dos comienza.
- Al menos 10 juegos en que un jugador Minimax utilice la función de puntaje implementada en esta actividad usando una **profundidad de 1** y el otro jugador Minimax debe usar la mejor función de valor obtenida en la actividad anterior con una **profundidad de 5**. Queda a tu criterio cual de las dos comienza.
- Al menos 10 juegos en que un jugador Minimax utilice la función de puntaje implementada en esta actividad usando una **profundidad de 5** y el otro jugador Minimax debe usar la mejor función de valor obtenida en la actividad anterior con una **profundidad de 1**. Queda a tu criterio cual de las dos comienza.

Analiza los resultados y **comenta acerca de tu función de valor**. ¿Es mejor o peor que la anterior? ¿A qué puede deberse este comportamiento (que gane o pierda)?.
Por último analiza como el **cambio de profundidad afecta en los resultados**. Explica con tus palabras que significa tener una mayor profundidad. ¿Da ventaja o desventaja tener una mayor o menor profundidad? ¿Cómo se ve eso en este caso en particular?

3.2.3. Aclaraciones

Se espera también que muestren toda información de forma amigable. Por ello, para cada uno de los puntos mencionados se les pide al menos un gráfico o recurso (puede ser una tabla o un gráfico que muestre correctamente lo obtenido).
DISCLARIMER: No se esperan demostraciones matemáticas (si las quieres usar eres bienvenido), si no que escribe y explica con tus propias palabras lo que ves y piensas.

[4]Es decir, deben hacer jugar 2 jugadores IA entre ellas.
[5]Se espera que no retorne siempre lo mismo, si no que varie de tablero a tablero