PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2022-2

Tarea 2

Aspectos generales

Formato y plazo de entrega

El formato de entrega son archivos con extensión .py con un PDF para las respuestas teóricas. Debes utilizar un PDF diferente para cada pregunta y ubicarlo en su carpeta correspondiente (DCCanicas o DCCarros). El lugar de entrega es en el repositorio de la tarea, en la *branch* por defecto, hasta el **viernes 26 de mayo 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. DCColor Sort (3 pts.)

Funcionamiento del juego

Llevas un buen rato *scrolleando* en tus redes sociales y te das cuenta que hay un anuncio de un juego que se repite mucho. El juego se llama DCColor Sort, un rompecabezas en el que tienes que ordenar bolas de colores en tubos hasta que cada tubo tenga bolas del mismo color. En cada movimiento sólo se puede mover la bola superior de un tubo (no dos al mismo tiempo). Lo descargas y te das cuenta que el juego puede llegar a ser muy desafiante. Piensas un rato y te das cuenta que fácilmente puedes transformar este juego en un problema de búsqueda y resolverlo con A*.

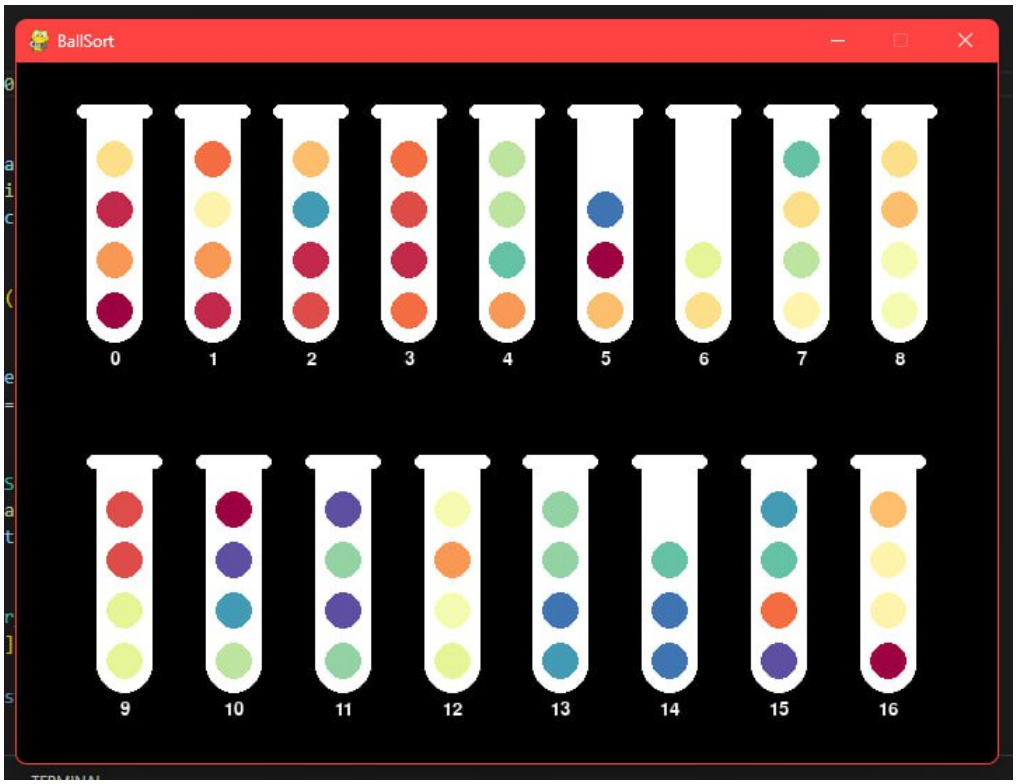


Figura 1: El clásico DCColor Sort

Archivos del Repositorio

En la carpeta DCColorSort podrás encontrar 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.
- `BallSortBack.py`: Este archivo se encarga de manejar la lógica del juego. No es necesario que lo modifiques.
- `BallSortFront.py`: Este archivo se encarga de manejar la visualización del juego. No es necesario que lo modifiques.
- `puzzle_generator.py`: Este archivo permite generar configuraciones iniciales de bolas y tubos. Es muy útil para probar el algoritmo. No debes modificar el archivo excepto por las constantes que se ubican en las primeras líneas. Estas son:

- `NUM_COLORS`: Cantidad de colores a ordenar.
- `NUM_TUBES`: Cantidad de tubos del problema.
- `TUBE_CAPACITY`: La cantidad de bolas que caben en un tubo.
- `RANDOM_MOVES`: Cuánto se desordena la configuración inicial.
- `MAP_NAME`: Nombre con el que se guardará el archivo creado.

- `heuristics.py`: Este archivo contiene las heurísticas que deberás implementar. Los detalles se explicarán más adelante.
- `maps/`: En esta carpeta se guardan los archivos creados por `puzzle_generator.py`.
- `BallSortSolver.py`: Este archivo contiene la implementación de A* que deberás desarrollar más adelante en la tarea.

1.1. Actividad 1: Integrar A* (0.2 pts)

Implementa el método `search` de la clase `AStarSolver` en el archivo `BallSortSolver.py`. Te recomendamos fuertemente utilizar las clases `Node` y `BinaryHeap`, y que utilices el código de A* que vimos en clases.

Implementa correctamente una estadística que cuente cuantos nodos hay en memoria (si ya consideras que esto está implementado, no debes hacer nada adicional).

1.2. Actividad 2: Lazy A* (0.8 pts)

Implementa el método `lazysearch` de la clase `AStarSolver` en el archivo `BallSortSolver.py`, que contendrá Lazy A*, una versión 'floja' de A* que, a diferencia de A* permite que existan dos copias de un mismo estado en la cola de prioridades Open. Así, si un estado s recién ha sido generado, en vez de preguntarse si s ya está en la Open para comparar el valor g nuevo con el anterior, esta implementación simplemente agrega un nuevo nodo con el nuevo g a Open.

Implementa correctamente una estadística que cuente cuantos nodos hay en memoria.

1.3. Actividad 3: Implementación de las heurísticas (0.5 pts)

En esta actividad, deberás implementar las siguientes heurísticas en el archivo `heuristics.py`:

- `wagdy_heuristic(state)`: Esta heurística evalúa el número de pares consecutivos de bolas de diferente color en cada tubo. Por cada par de bolas consecutivas de diferente color, se agrega un costo estimado de 2.
- `repeated_color_heuristic(state)`: Esta heurística evalúa el número de bolas que no tienen el mismo color que el color más repetido en cada tubo. Por cada bola que no tenga el mismo color que el color más repetido, se agrega un costo estimado de 1.

1.4. Actividad 4: Admisibilidad de heurísticas (0.5 pts)

- Demuestra la admisibilidad o no admisibilidad de las dos heurísticas mencionadas anteriormente y comenta cuál debería ser teóricamente mejor entre ellas (solo si ambas son admisibles) y porqué. **NOTA**: Para demostrar la admisibilidad, se espera una demostración matemática.

- Demuestra que cuando A* es usado con una heurística consistente entonces la secuencia de valores f de los estados que son extraídos de open es monótonamente creciente. Dicho de otra forma, si justo después de extraer un nodo n de Open, hicieras un `print` de $f(n)$ notarías que la secuencia de valores impresos es no decreciente. Pista: demuestra primero que si t es hijo de s, entonces siempre ocurrirá que $f(s) \leq f(t)$. Usando eso, argumenta que el teorema es verdadero.

  Lee esto solo si quieres saber algo más: Tal vez te preguntas por qué tiene sentido demostrar esta propiedad. Esta es la respuesta: Si la secuencia de valores f es no decreciente, es fácil ahora demostrar que A*, usado con heurísticas consistentes, nunca puede expandir dos veces el mismo nodo (no, no es necesario que lo hagas!!). Eso es importante, porque cuando las heurísticas no son consistentes A* podría expandir muchas veces el mismo nodo, lo que haría que la ejecución fuera muy, pero muy ineficiente, incluso peor que al usar $h = 0$ (que es una heurística consistente). Podría ser incluso exponencialmente más ineficiente. En la figura 1 de este paper `https://webdocs.cs.ualberta.ca/~holte/Publications/ijcai09.pdf` puedes enconcontrar un ejemplo de mal funcionamiento de A* con heurísticas consistentes.

- Demuestra que Lazy A*, tal como A*, también encuentra soluciones óptimas. Para esto, puedes seguir la línea argumentativa de este video `https://www.youtube.com/watch?v=_41v4I5GTNc`.

1.5. Actividad 5: Comparación de 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 la carpeta `maps/`. Compara ambos algoritmos, A* y Lazy A* en términos de memoria utilizada y tiempo de ejecución. **Si el tiempo de ejecución es mayor a 2 minutos para cualquier mapa, simplemente reportalo como 2 minutos y TIMEOUT (no continúes buscando).**

Comenta los resultados observados, analiza si coincide con tu respuesta en la actividad anterior y especula sobre los motivos que puedan explicar estos comportamientos.

NOTA: Ejecuta cada uno de los 5 mapas para cada combinación algoritmo-heurística.

1.6. Actividad 6: Implementación con Best First Search y comparación (0.5 pts)

Investiga sobre el algoritmo Greedy Best First Search e impleméntalo (Hint: Es un cambio de una línea sobre el código de A*). Explica las diferencias entre ambos algoritmos, compáralos empíricamente al igual que en el inciso anterior y comenta los resultados.

2. Bump DCCheep (3 pts.)

Después de un largo día de trabajo, te encuentras invade el agotamiento y decides tomarte un descanso jugando un juego en tu teléfono móvil. Luego de buscar durante horas en la Play DCCtore, decides descargarte el juego "Bump DCCheep" y te adentras en la historia de dos rebaños de ovejas que compiten por el mejor pasto en las Grandes Llanuras. El juego consiste en desplazar a las ovejas por la pista utilizando gestos táctiles para hacerlas chocar entre sí y así avanzar más rápido.

Logras avanzar en el juego con facilidad hasta que llegas al final y te das cuenta de que las cosas se están poniendo difíciles y necesitas ganar más puntos para vencer a tu oponente. Después de fracasar múltiples veces, te has dado cuenta de que el movimiento de las ovejas en la pista no es completamente aleatorio y existe una forma de prever los movimientos de tu oponente.

Después de muchos intentos, decides utilizar todo tu conocimiento acumulado para poder derrotar el nivel. Luego de mucho sudor y lágrimas, logras completar tu objetivo, para darte cuenta de que quedan otros 5 niveles más. Por lo cual, decides crear un algoritmo para superar estos niveles y acabar con este sufrimiento.

Figura 2: El fabuloso Bump DCCheep

Juego

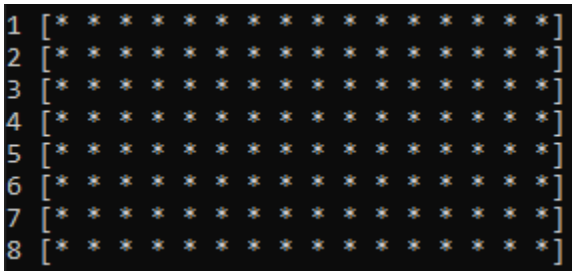
El juego se trata de un tablero formado por varias *Filas* donde puedes colocar *Ovejas* de diferentes tamaños.

Los elementos son :

Tablero (tiles):

Son los espacios donde se pueden colocar a las ovejas. Estas tienen dimensiones arbitrarias que puedes alterar en los parametros del codigo.

Tablero de 5x9



Tablero de 8x16

Ovejas:

Cada jugador utiliza un rebaño de ovejas de un color. El que comienza juega con las ovejas blancas.

Reglas

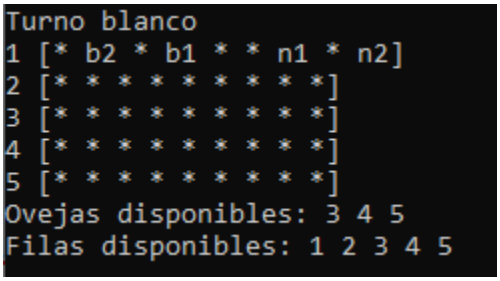
Cada jugador dispone de un rebaño de ovejas de su color y, en cada turno, el jugador deberá colocar una oveja al inicio de alguna fila del tablero:

- Derecha: Ovejas negras
- Izquierda: Ovejas blancas

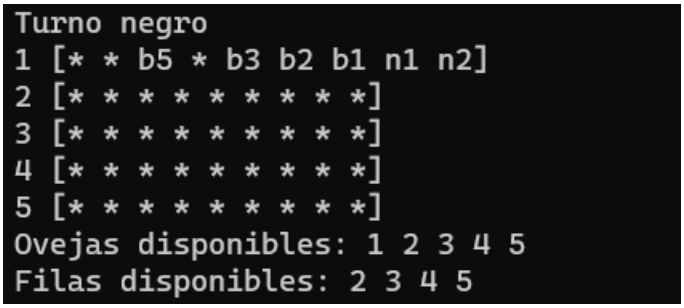
Existen 2 restricciones para para la colocación de las ovejas:

- El enfriamiento que experimentan según su tamaño: a mayor tamaño, mayor será el tiempo de espera.
- La casilla inicial del jugador no puede ser ocupada por otra oveja.

En el ejemplo, estamos en el tercer turno del blanco, podemos ver que ya utilizó 2 ovejas, una de tamaño 1 y otra de tamaño 2, las cuales ahora no están disponibles para lanzar.



En este turno, las ovejas del jugador negro están siendo empujadas por las ovejas de color blanco, ya que la suma de sus tamaños es mayor que el de las ovejas negras. Además, el jugador negro no puede colocar ninguna oveja en la primera fila, porque la casilla inicial no está libre.



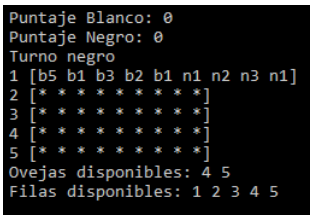
Término del juego

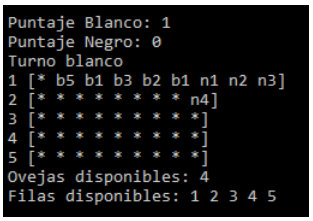
El juego termina cuando en un turno, alguno de los jugadores ha alcanzado el puntaje objetivo.

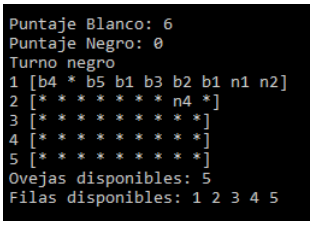
Puntaje

El puntaje de cada oveja se determina por su tamaño. Dependiendo del lado por el que son empujadas, se otorgará el puntaje a uno de los jugadores. Si entran por el lado derecho, ganará puntos el jugador blanco, y si entran por el lado izquierdo, ganará puntos el jugador negro. El puntaje que se otorga por oveja corresponde al doble de su tamaño menos 1.

En este caso el jugador blanco suma 6 puntos:







¡Prueba el Juego!

Antes de pasar a explorar el código y los archivos, te recomendamos fuertemente que juegues un par de partidas contra el robot para familiarizarte con las reglas y la forma de jugar. Para poder jugar, debes instalar previamente la librería `pygame` y luego correr el archivo `main.py`.

Archivos del repositorio

Se les entregará en el repositorio una implementación funcional del juego que utiliza la librería `pygame` para funcionar. La implementación contiene varios archivos, pero no es necesario que revisen y/o manipulen todo. Para ayudarles, a continuación se describen a grandes rasgos las partes que componen la implementación. Solo deberán editar los archivos destacados en amarillo.

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`: archivo que contiene la lógica para correr el juego. No es necesario modificarlo. Solamente deben ejecutarlo en consola con `python main.py`

- `parametros.py`: archivo que contiene las configuraciones del juego. Dependiendo de la actividad deberán modificar ciertos parámetros en este archivo. En particular, los atributos que son de interés son los siguientes:

  - `parametros.MODE_COLOR`: `COLOR` puede ser `BLANCO` o `NEGRO` dependiento de las ovejas con las que juegue el robot. Este atributo es un string que puede ser 'random' si el robot realiza movimientos aleatorios, 'minmax' si el robot usa el algoritmo Minimax para hacer sus jugadas o se puede usar 'player' para que un jugador ejecute los movimientos de ese color.
  - `Settings.IQ_COLOR`: `COLOR` puede ser `BLANCO` o `NEGRO` dependiendo de las ovejas con las que juegue el robot. Este atributo es un entero que determina la profundidad de búsqueda utilizada en Minimax, por lo que solo toma efecto si el atributo anterior es 'minmax' para el robot correspondiente.

- `utils.py`: archivo que contiene funciones utiles que son requeridas para implementar minimax. No se debe modificar. En particular, las funciones de interés son:

  - `utils.disponibilidades`: devuelve las ovejas y filas disponibles para jugar en un juego de mesa, excluyendo las filas no disponibles y ovejas con enfriamiento.
  - `utils.ejecutar_jugada`: avanza las ovejas en el tablero y, si se ingresa una nueva, la coloca y establece su enfriamiento. Luego, cambia el turno y reduce los enfriamientos del jugador del turno.

- `minimax.py`: archivos donde se debe implementar el algoritmo minimax. Contiene las siguientes funciones:

  - `minimax(game, depth, alpha, beta)`: esta función recursiva es la que deberán implementar para que el robot pueda jugar de mejor manera. A partir del estatus del juego, el tablero, una profundidad de iteración, una función de evaluación y los parámetros `alpha` y `beta`, debe retornar una tupla de la forma `((x,y), score)`, donde `(x,y)` es el movimiento recomendado y `score` la evaluación de dicho movimiento.

Descripciones adicionales del resto del código

Aquí pueden revisar qué hace el resto del código por si lo quieren investigar. Nuevamente, NO es necesario modificar o entender esta parte del código para completar tu tarea.

- `entidades.py`: archivo que contiene las clases `Jugador`, `Oveja` y `Game` para controlar el juego. No se debe modificar.

Que no se te olvide

Como podrás notar, son varios archivos los que hacen funcionar el juego, pero que eso no los asuste. Durante su trabajo ustedes solamente deberán modificar los archivos `minimax.py` y `settings.py`.

¿Qué debes implementar?

Más adelante se describirán en más detalle las actividades específicas a realizar, pero esencialmente lo que se te pide es completar la implementación del algoritmo Minimax, agregar la poda alfa-beta y hacer un estudio respecto a los resultados que se obtienen en el juego usando el algoritmo.

Para ello, te recomendamos fuertemente limitarte a utilizar solo las clases, atributos y métodos que se describieron explícitamente en la sección **Archivos mínimos que deben entender y/o manipular**.

2.1. Actividad 1: Minimax (1 pts)

En esta primera actividad, deberás completar la implementación del algoritmo Minimax que se encuentra en el archivo `minimax.py` [1]. Para ello, lee bien los comentarios incluídos en el código y trata de seguir la estructura solicitada. Recuerda que no es necesario investigar los archivos, atributos, clases o parámetros del código fuera de los que se mencionaron explícitamente en este enunciado.

Para probar tu implementación, debes cambiar el tipo de inteligencia del robot del player 2 a 'Minmax' y asignarle una profundidad de búsqueda. Para ello, modifica los atributos correspondientes en `parametros.py`.

IMPORTANTE: Durante esta actividad no debes manipular los parámetros `alpha` y `beta` que recibe `minimax`.

2.2. Actividad 2: Poda Alfa-Beta (0.5 pts)

Ahora, deberás modificar tu implementación de Minimax para que haga una poda alfa-beta y sea más eficiente. Para ello, modifica tu función Minimax para que realmente utilice los parámetros `alpha` y `beta`.

IMPORTANTE: Este cambio no debería agregar muchas líneas de código a tu implementación original de Minimax.

2.3. Actividad 3: Estudio Analítico (1.5 pts)

En esta parte deberás hacer un estudio del efecto que tiene Minimax con sus diferentes parámetros en el desempeño del juego. Para ello, debes entregar junto a tu código un archivo llamado `estudio.pdf` que contenga al menos lo siguiente:

1. Los resultados de 10 juegos en el tablero con 5 filas utilizando Random como jugador negro y Minmax con profundidad 3 jugador blanco.
2. Los resultados de 10 juegos utilizando Minmax en ambos robots, con la misma profundidad, en el tablero con 5 filas, pero con largo variable.

Se espera que muestren esta información de forma amigable. Por ello, para cada uno de los puntos mencionados se les pide al menos un gráfico o recurso [2] (es decir, su informe debería contener al menos 3 gráficos o recursos). Además, comenten brevemente sus resultados, ya sea mencionando posibles patrones o conclusiones. Por ejemplo:

- Es Minimax mejor que Random, ¿por cuánto?

[1] Debes implementar tu código en las secciones marcadas como `TODO`.

[2] Con recurso nos referimos a cualquier figura o tabla que ayude a la visualización de los datos.

- Si se usa Minimax a profundidad igual para ambos jugadores, ¿alguno tiene ventaja por ir primero o segundo?

Su informe no debe tener una extensión mayor a 3 planas.