PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2025-1

Tarea 3
Problemas de Búsqueda

Fecha de entrega: viernes 16 de mayo a las 23:59 hrs

Aspectos generales

Formato y plazo de entrega

El formato de entrega son archivos con extensión `.py` y un PDF para las respuestas teóricas. El lugar de entrega es en el repositorio de la tarea, en la branch por defecto, hasta el viernes 16 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 **reales** (hábiles o no) 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. DCC Puzzle 15 (2.5 puntos)

Introducción

En una ciudad del futuro, donde la inteligencia artificial ha revolucionado hasta las actividades más cotidianas, existe un pequeño laboratorio dirigido por un excéntrico ingeniero llamado Jorge Baier. A diferencia de sus colegas, Jorge no estaba obsesionado con crear asistentes domésticos ni autos autónomos. Su verdadera pasión era revivir antiguos juegos de lógica y transformar cada uno en un desafío para las máquinas.

Una noche, Jorge desempolvó un viejo rompecabezas mecánico que había pertenecido a su abuelo: el famoso puzzle de 15, un cuadrado con 16 espacios —15 de ellos ocupados por fichas numeradas del 1 al 15, y uno vacío— que requiere deslizar las piezas hasta ordenarlas correctamente. Observando la maraña desordenada de números, se preguntó: "¿Cómo enseñaría a una IA a resolver este caos del modo más eficiente posible?"

Así nació su nuevo proyecto: entrenar a una inteligencia artificial para que resolviera el puzzle de 15 utilizando el algoritmo A* (A estrella), conocido por su capacidad de encontrar caminos óptimos en mapas, laberintos... y, por qué no, rompecabezas.

El problema es que Jorge necesita la ayuda de un ingeniero que pueda ayudarlo a resolver este problema, ya que de tanto jugar con el puzzle de olvido como funciona el algoritmo A*. Como tú eres un experto en este algoritmo, decides ayudarlo en su misión.

Explicación Juego

El juego *Puzzle 15* consiste en una cuadrícula de 4×4 con fichas deslizantes dispuestas sobre una bandeja. Hay 15 fichas numeradas del 1 al 15 y un espacio vacío. Antes de comenzar el juego, las fichas se mezclan aleatoriamente. El objetivo es deslizarlas, una por una, utilizando el espacio vacío, hasta ordenarlas en secuencia numérica ascendente, como se muestra en la siguiente imagen:

1	2		10
4	6	3	12
7	13	5	8
15	9	11	14

Figura 1: Puzzle de 15 desordenado

1	2	3	4
5	6	7	8
9	10	11	12
14	13	15	

Figura 2: Puzzle de 15 ordenado

El objetivo de esta tarea es resolver el *Puzzle 15* utilizando distintos algoritmos de búsqueda, y comparar su rendimiento frente a un mismo problema. Para ello, se entrega la carpeta `DCCpuzzle`, que contiene los siguientes archivos:

Archivos que NO debes editar:

- `problemas/`: Carpeta que contiene archivos `.txt` con configuraciones iniciales del *Puzzle 15*, separadas por dificultad en las carpetas `easy`, `medium` y `hard`. Estos serán los tests con los que evaluarás

tus algoritmos.

- `15puzzle_solver_model.h5`: Red neuronal preentrenada para resolver el *Puzzle 15*. Se usará como parte de una heurística inadmisible y su uso se detallará más adelante.
- `display.py`: Archivo que contiene el procedimiento encargado de visualizar el tablero y mostrar paso a paso las soluciones encontradas por los algoritmos. Se requiere de la librería `tabulate`.
- `litemodel.py`: Archivo que permite cargar y ejecutar inferencias usando el modelo de red neuronal de forma eficiente. Se requiere de la librería `tensorflow`.
- `multi_binary_heap.py`: Implementación de un heap binario con múltiples prioridades, utilizado internamente por algoritmos como *Focal Search*.
- `multi_node.py`: Define la estructura de los nodos utilizados en la búsqueda. Cada nodo almacena su estado, costos acumulados, heurísticas, acción tomada y referencia al nodo padre.
- `puzzle.py`: Contiene la lógica del puzzle, incluyendo las clases `Puzzle15State` y `Puzzle15`, que modelan los estados y permiten ejecutar los movimientos.

Archivos que <u>DEBES</u> editar:

- `algorithms.py`: Aquí se encuentran los algoritmos que viste en clases que usarás para resolver el *Puzzle15*, deberás completar `BFS` y `AStar`. Es importante que cada método `solve()` retorne una instancia del elemento `SearchResult`, este se define como `SearchResult(path, expansions, elapsed_ms)`, donde:
  - `path`: Lista que contiene cada paso de la solución del problema.
  - `expansions`: Número de expansiones que le tomó al algoritmo encontrar la solución.
  - `elapsed_ms`: Tiempo (en segundos) que le tomó al algoritmo encontrar la solución.
- `focal_search.py`: Contiene las implementaciones de *Focal Search* y su variante *Focal Discrepancy Search*, algoritmos basados en $A*$ que permiten utilizar heurísticas no admisibles. Aquí deberás modificar el método `solve()` para especificar cuál de los algoritmos deseas ejecutar (será explicado con mayor profundidad más adelante) y completar el método `heuristic_search()` que contendría la implementación de *Focal Search* (también lo explicaremos en profundidad más adelante).
  IMPORTANTE: aquí el método `solve()` también debe retornar una instancia de `SearchResult`.
- `heuristics.py`: Este archivo contiene funciones con distintas heurísticas. Deberás completar la función `euclidian()`, implementando la distancia euclidiana como heurística para el *Puzzle15*. Además, el archivo incluye la función `get_heuristics()`, que recibe un `str` con el nombre de una heurística y retorna la función correspondiente. Deberás crear una heurística propia e integrarla en `get_heuristics()` para que pueda ser utilizada al ser solicitada.
- `main.py`: Contiene el flujo principal. Puedes editarlo en el paso 5 para escoger cuáles algoritmos vas a ejecutar y en qué orden.

Esta sección de la tarea se divide en tres partes, cada una con un enfoque distinto sobre los diferentes algoritmos de búsqueda que vieron en el curso (y algunos nuevos!!). Para cada parte deberás modificar ciertos archivos, ejecutar pruebas en distintas instancias del puzzle, registrar los resultados y analizarlos.

Para facilitar el desarrollo de la tarea, puedes ejecutar en la terminal un comando con el siguiente formato:

```
python main.py <dificultad> <num_problema> <heu_inadmisible> <heu_admisible> [--show]
```

Donde:

- `dificultad`: Dificultad del problema a resolver. Debe ser `easy`, `medium` o `hard`.
- `num_problema`: Número del problema dentro de la carpeta seleccionada
- `heu_inadmisible`: Nombre de la heurística no admisible a utilizar. Debe estar definida en `heuristics.py` o ser `zero`.
- `heu_admisible`: Nombre de la heurística admisible a utilizar. También definida en `heuristics.py` o ser `zero`.
- `--show` **(opcional)**: Permite visualizar, paso a paso, la solución encontrada por el **último algoritmo ejecutado**, solo si este encuentra una solución válida.

Ejemplos de uso:

```
python main.py easy 1 zero manhattan --show    # Muestra solución
python main.py easy 1 zero manhattan           # Solo ejecuta sin mostrar
```

En cada parte de esta pregunta deberás ejecutar diferentes algoritmos, algunos requieren heurísticas admisibles, otros no admisibles, algunos necesitan de las dos y otros ninguna. Cuando un algoritmo no necesita una heurística específica, puedes usar el placeholder `zero`, ya definido en `heuristics.py` como una heurística constante nula.

En el archivo `main.py` encontrarás una lista llamada `algorithms`, que controla cuáles algoritmos se ejecutan en una misma corrida. Puedes combinar libremente los algoritmos que desees, pero para evitar confusiones y resultados innecesarios, te recomendamos seguir la pauta indicada en cada parte de la tarea. Inicialmente, la lista contiene solo BFS.

Parte 1: Búsquedas no informadas (0.5 pts)

En esta parte deberás completar la implementación del algoritmo `BFS` en el archivo `algorithms.py`, utilizando como referencia el algoritmo `DFS`, el cual ya se encuentra implementado. No es necesario que ejecutes `DFS`, ni que utilices los archivos `multi_binary_heap.py` ni `multi_node.py` para completar el código.

Como `BFS` no requiere heurísticas, puedes ejecutar las pruebas con la heurística `zero`, que ya se encuentra definida en `heuristics.py`. Un ejemplo de ejecución sería:

```
python main.py easy 1 zero zero --show    # Muestra solución
```

Una vez implementado el algoritmo, responde las siguientes preguntas:

a) (0.3 pts.) Ejecuta tu implementación de `BFS` sobre los distintos tests de la carpeta `easy`. Registra el tiempo de ejecución, el número de nodos expandidos y el largo del camino solución. Puedes presentar tus resultados mediante una tabla u otro método visual que facilite su análisis. ¿Cómo describirías el comportamiento de `BFS` en este tipo de problemas? ¿Qué pasaría si lo ejecutaras sobre tests de mayor dificultad?.

b) (0.2 pts.) Explica por qué `DFS` podría no terminar una ejecución en este contexto. Puedes usar una ejecución real como referencia si lo deseas. Considera cómo se comporta DFS respecto al orden de expansión y a la existencia de ciclos o caminos muy profundos.

Parte 2: A* con heurísticas admisibles (1 pt)

En esta parte deberás completar la implementación del algoritmo `A*` en el archivo `algorithms.py`, y compararlo utilizando distintas heurísticas admisibles.

El archivo ya contiene una plantilla básica con el nodo inicial configurado y estructuras listas para usar (`MultiNode`, `MultiBinaryHeap`). Si lo deseas, puedes mantener esta estructura y completar únicamente el bucle principal. Sin embargo, si prefieres usar otras estructuras de datos (como listas, sets, etc.), puedes modificar el algoritmo según lo estimes conveniente, siempre y cuando se respete el formato de retorno con `SearchResult`.

a) (0.5 pts) Completa el método `solve()` de la clase `AStar` en `algorithms.py`. Puedes continuar utilizando la estructura de datos provista o reemplazarla por otra si lo prefieres. El algoritmo debe retornar un objeto `SearchResult(path, expansions, elapsed)` al finalizar, donde `path` es la secuencia de nodos desde el estado inicial hasta la meta.

b) (0.2 pts) Asegúrate de usar la heurística `manhattan`, ya implementada en `heuristics.py`, y completa la función `euclidian`, que corresponde a la distancia euclidiana desde el estado actual a la meta. Luego, crea una heurística propia adicional, que también sea admisible e idealmente mejore los resultados en comparación a las otras dos. Esta heurística debe estar definida en `heuristics.py`, y debe tener un nombre único. En tu informe describe que es lo que hace esta heurística y demuestra su admisibilidad. Esta demostración no tiene que ser matemáticamente rigurosa pero debe quedar clara la idea.

c) (0.1 pts) Ejecuta el algoritmo `A*` sobre distintos problemas de dificultad `easy` y `medium`, utilizando las tres heurísticas: `manhattan`, `euclidian` y tu heurística personalizada.

   Recuerda que, como A* no utiliza heurísticas inadmisibles, puedes pasar `zero` en ese campo. Por ejemplo:

```
python main.py medium 2 zero manhattan
python main.py medium 2 zero euclidian
python main.py medium 2 zero custom # custom sería el nombre de tu heurística
```

d) (0.2 pts) Registra la cantidad de nodos expandidos, el tiempo de ejecución y el largo del camino de la solución de los test `medium`.

   Presenta los resultados en una tabla comparativa u otro recurso visual que facilite el análisis. Luego responde: ¿Cuál heurística produjo soluciones más eficientes? ¿Qué relación observas entre el diseño de la heurística y el número de nodos expandidos?

Parte 3: Focal Discrepancy Search y red neuronal (1 pto.)

En esta parte utilizarás dos variantes del algoritmo `A*` llamadas `Focal Search` y `Focal Discrepancy Search`. Ambas permiten utilizar heurísticas no admisibles, es decir, funciones que pueden sobreestimar el costo real hacia la meta. Además, trabajarás con una red neuronal entrenada para aprender una política de resolución del *Puzzle 15*, y observarás cómo se comporta esta política al ser integrada en una estrategia de búsqueda.

¿Qué es Focal Search?

`Focal Search` es una variante de `A*` que trabaja con dos colas de prioridad. Por un lado, mantiene una **open list** ordenada por valores admisibles $f(n) = g(n) + h(n)$, donde $h(n)$ debe ser una heurística admisible. Por otro lado, mantiene una **focal list**, que corresponde a un subconjunto de la `open list` con todos los nodos que cumplen:

$$f(n) \leq w \cdot f_{\text{mín}}$$

Donde $w \geq 1$ es un parámetro de peso. La `focal list` se ordena por una heurística alternativa, que puede ser no admisible. Este diseño permite balancear optimalidad y eficiencia, explorando nodos con bajo costo estimado que además son prometedores según una segunda heurística.

Un pseudocódigo para *Focal Search* sería el siguiente [1]:

Algorithm 1: FOCAL SEARCH

Input: A search task $P = (G, s_{start}, s_{goal})$, an admissible heuristic h, a suboptimality bound w, a function h_{Focal}
Output: A goal node reachable from n_{start}

```
foreach s ∈ S do
    g(n) ← ∞
g(s_start) ← 0
parent(s_start) ← null
f(s_start) ← h(s_start)
Insert n_start to Open and Focal
while Focal is not empty do
    f_min ← f-value of node at the top of Open
    Extract s from Focal which maximizes h_Focal
    Remove s from Open
    if n = n_goal then
        return n
    foreach t ∈ Succ(s) do
        cost_t ← g(s) + c(s, t)
        if cost_t < g(t) then
            parent(t) ← n
            g(t) ← cost_t
            f(t) ← g(t) + h(t)
            Insert t into Open
            if f(t) ≤ w f_min then
                Insert t into Focal
    top ← state at the top of Open
    if f_min < f(top) then
        foreach s ∈ Open do
            if f_min < f(s) ≤ f(top) then
                Insert s into Focal
return "no solution found"
```

¿Qué es Focal Discrepancy Search (FDS)?

`Focal Discrepancy Search` es una versión especializada de Focal Search en la que la heurística inadmisible no es una medida de costo, sino que usa un concepto llamado **discrepancia**.

La red neuronal entregada predice un vector de 4 probabilidades (una por cada dirección: izquierda, abajo, derecha, arriba). La red induce una función que, dado un estado, entrega la acción de mayor probabilidad asociada. Este tipo de funciones, que reciben un estado y retornan una acción, se conocen en inteligencia artificial como **políticas**.

Denotamos la discrepancia asociada a un nodo v por $discrepancia(v)$. Al generar el nodo v a partir de u, calculamos la discrepancia de v de la siguiente forma:

[1] Obtenido de https://arxiv.org/pdf/2104.10535

- Sea a acción que genera a v desde u.
- Definimos d como 0 si a coincide con la acción que la política retorna para el estado asociado a u. En caso contrario, definimos d como 1.
- Hacemos $discrepancia(v) = discrepancia(u) + d$.

Finalmente, la discrepancia del nodo asociado al estado inicial se define como 0.

De esta manera, FDS prioriza caminos que siguen de cerca las decisiones recomendadas por la red neuronal, pero sin seguirlas ciegamente, permitiendo desviaciones cuando sea necesario.

¿Por qué usamos heurísticas no admisibles aquí? Ambos algoritmos permiten usar heurísticas no admisibles para guiar la exploración. La restricción $f(n) \leq w \cdot f_{\text{mín}}$ garantiza que no se exploren caminos completamente arbitrarios, manteniendo un límite sobre la suboptimalidad. Esto permite incorporar conocimiento aprendido o estrategias heurísticas más agresivas sin comprometer por completo la eficiencia.

A continuación, se detallan las actividades de esta parte:

a) (0.7 pts.) **Focal Search con heurística no admisible**

Implementa Focal Search en el archivo `focal_search.py`. Para implementar la lista ponderada descrita en el algoritmo, puedes utilizar la clase `MultiBinaryHeap` que implementa la estructura de datos `Heap`. Luego, crea una heurística no admisible, defínela en `heuristics.py` y describe brevemente en tu informe que es lo que hace. Demuestra que tu heurística es no admisible. Esta demostración no tiene que ser matemáticamente rigurosa pero debe quedar clara la idea.

Asegúrate de que el método `solve()` en `focal_search.py` utilice la función `heuristic_search(w)`. Luego, ejecuta `A*` y `Focal Search` sobre al menos tres problemas de dificultad `easy` y `medium`, utilizando una heurística admisible a elección y pesos en el rango $w \in [1.5,\ 2]$ para `easy` y en el rango $w \in [1.2,\ 2]$ para `medium`.

Como el algoritmo requiere de dos heurísticas, las ejecuciones deben seguir el siguiente formato:

```
python main.py medium 3 custom <heu_admisible> --show
# custom sería la heurística no admisible creada por ustedes
```

Luego de ejecutar ambos algoritmos con diferentes pesos, presenta tus resultados en una tabla comparativa u otro recurso visual que facilite el análisis. Finalmente, responde:

- ¿Cómo varía el número de expansiones y el tiempo al cambiar el valor de w?
- ¿Qué impacto tiene usar una heurística no admisible en este contexto?
- ¿Qué ventajas observas en Focal Search respecto a A*?

b) (0.3 pts.) **Focal Discrepancy Search con la política de la red neuronal**

No es necesario que implementes Focal Discrepancy Search, ya se encuentra dentro de `focal_search.py`. Utiliza la heurística `nn`, ya definida en `heuristics.py`, la cual ejecuta una red neuronal que actúa como política. Esta política se utilizará para medir las discrepancias en el algoritmo `Focal Discrepancy Search`.

Modifica el método `solve()` en `focal_search.py` para que ejecute `discrepancy_search(w)`. Luego, ejecuta `A*` y `FDS` sobre todos los problemas de dificultad `hard`, utilizando una heurística admisible a elección y pesos en el rango $w \in [1.2,\ 2]$. Ten en cuenta que estos problemas pueden tardar más en resolverse.

Para correr la red neuronal, debes ejecutar el siguiente código:

```
python main.py hard 2 nn <heu_admisible> --show
```

Una vez obtenidos los resultados, compáralos y responde:

- ¿La política aprendida fue una buena guía en estos casos?
- ¿Qué diferencias observas en rendimiento y calidad de solución entre A* y FDS?
- ¿Cómo influye el valor de w en la exploración y los resultados?
- ¿Se cumple que FDS sacrifica optimalidad por eficiencia? ¿En qué medida?

Presenta los resultados en una tabla comparativa o gráfico, si lo consideras útil para el análisis.

Qué entregar en esta parte

En un subdirectorio `DCCpuzzle/`:

- Un archivo `respuesta.pdf` con tus respuestas ordenadas. Incluye aquí las tablas y desarrollos de las preguntas.
- Los archivos de código modificados según se explicó anteriormente.

2. Teoría (1 punto)

1. (0.5 ptos.) Demuestra que Focal Search retorna una solución w-optima (es decir, que no excede en w veces el costo de una solución óptima), independientemente de cuál sea la función h_{Focal}. Para esta demostración debes usar e incluir la demostración del Lema del siguiente video (click acá [2]).

2. (0.5 ptos.) Si c es el costo de la solución retornada, ¿qué es $\frac{c}{f_{min}}$, donde f_{min} es el valor de la variable de Focal Search al terminar la búsqueda?

Qué entregar en esta parte

En un subdirectorio `teoria/`:

- Un archivo `respuesta.pdf` con tus respuestas ordenadas. Incluye aquí las tablas y desarrollos de las preguntas.

[2] Link del video: https://www.youtube.com/watch?v=_41v4I5GTNc

3. DCConecta4 Spin (2.5 pts.)

3.1. Introducción

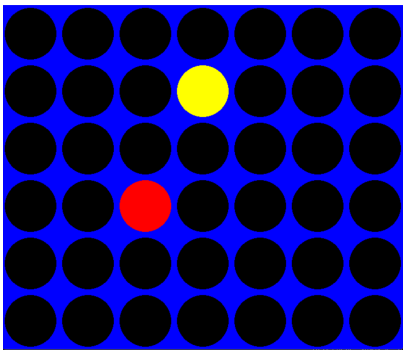
Un día estas en clases de tu electivo favorito *IIC2612.5 Inteligencia Natural* y todo va de lo más bien. El profesor está explicando como los humanos hacemos representaciones lógicas del mundo y luego como logramos obtener caminos óptimos cuando queremos ir de un punto a otro. En un momento decides mirar tu celular para ver la hora, pero te topas con un anuncio sobre una nueva versión del clásico juego Connecta4! Esta versión permite hacer giros en las columnas del tablero y elimina la gravedad!!! Ilusionado por este nuevo juego, le escribes a tu compañero Jorge Baier para que se descargue el juego y jueguen juntos, pero cuando lo van a abrir se dan cuenta que todo era una estafa. La publicidad era falsa y el juego que viste en ella no existe. Cansado de que te pase esto con todos los anuncios, decides programar tu propia versión de este juego y decides llamarla DCConecta4 Spin!! No contento con esto, recuerdas el famoso algoritmo Minimax de tus clases de *IIC2613 Inteligencia Artificial* y decides agregarlo para hacerlo más competitivo!
Por lo tanto, tu misión en esta tarea será implementar DCConecta4 Spin con Minimax.

3.1.1. Explicación juego

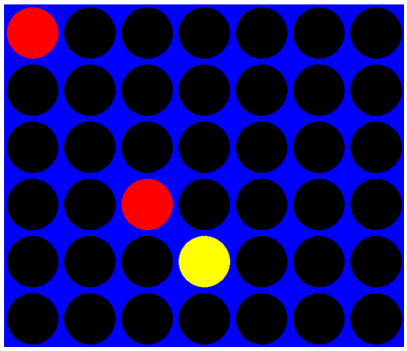
En DCConecta 4 Spin, el objetivo es jugar tus fichas alternadamente contra otro jugador e intentar que formen una línea de 4 fichas consecutivas. Estas pueden estar horizontal, vertical o diagonalmente conectadas, pero no debe haber ninguna ficha rival en el camino. Se juega en un tablero de N $\times$ M celdas, y puedes colocar una ficha en cualquiera de ellas.

Esta variante, sin embargo, cuenta con un mecanismo extra al Conecta 4 tradicional. En primer lugar, las fichas al colocarlas, no caen hasta la parte inferior del tablero, sino que permanece en la posición colocada. Cada vez que alguien juegue una pieza en la posición (i, j), la j-ésima columna rotará, haciendo que las fichas en la parte superior bajen, y la inferiores suban. Como cada vez que se realice una jugada esto ocurrirá, los jugadores deben pensar bien dónde colocar su pieza, cuidando de no beneficiar al rival.

A continuación se muestra un ejemplo de la rotación:



(a) Tablero antes de rotar la columna 4



(b) Tablero tras rotar la columna 4 y que el adversario haya jugado

Figura 3

Como se puede observar en las figuras, en (a) se agrega la ficha amarilla en la columna 4 fila 2. Con el spin, la ficha puesta se invierte, pasando a estar en la columna 4 fila 5, como se puede ver en (b)

3.1.2. Explicación código

A continuación se entrega una explicación de los archivos y códigos en la carpeta DCConecta4:

- **view/view.py:** Clase abstracta View para la vista del tablero. **No modificar**.

- **view/console_view.py:** Contiene la clase ConsoleView para observar el tablero en la consola. Hereda de view. **No modificar**.
- **view/graphic_view.py:** Contiene la clase GraphicView para observar la visualización del juego. Hereda de view. **No modificar**.
- **view/no_view.py:** Contiene la clase NoView para no generar visualización del juego. Hereda de view. **No modificar**.
- **board/board.py:** Contiene la clase Board, la cual modela el tablero. **No modificar**.
- **board/board_helper.py:** Contiene la clase BoardHelper, la cual contiene métodos para gestionar el tablero y el juego. **No modificar**.
- **players/player.py:** Contiene la clase abstracta Player, la cual modela a un jugador. **No modificar**.
- **players/human_player.py:** Contiene la clase HumanPlayer, la cual permite que puedas jugar. Hereda de Player **No modificar**.
- **players/minimax_player.py:** Contiene la clase MinimaxPlayer, la cual utiliza el algoritmo Minimax para elegir el siguiente movimiento. Hereda de Player **No modificar**.
- **players/max3_player.py:** Contiene la clase Max3Player, la cual utiliza el algoritmo Minimax para elegir el siguiente movimiento, pero permitiendo jugar con más de un adversario. Hereda de Player **No modificar**.
- **players/random_player.py:** Contiene la clase RandomPlayer, la cual elige movimientos al azar. Hereda de Player **No modificar**.
- **game_controller.py:** Contiene la clase GameController, y contiene los métodos para poder realizar el juego. **No modificar**.
- **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:
  - view: Se debe inicializar una instancia de la clase View. Estas pueden ser GraphicView, ConsoleView, NoView.
  - player1 y player2: Instancias de la clase Player. Estos pueden ser MinimaxPlayer, RandomPlayer, HumanPlayer. Contiene los siguientes atributos:
    - ID: El ID del jugador. Siempre tienen que ser 1 y 2 **No modificar**
    - view: Se le debe entregar la view elegida. **No modificar**
    - view: Se le debe entregar el tablero. **No modificar**
    - eval_function (solo para MinimaxPlayer): Determina la función de evaluación a utilizar por el jugador en Minimax. **Modificar** para utilizar diferentes funciones de evaluación.
    - depth (solo para MinimaxPlayer): Determina la profundidad de búsqueda del jugador para Minimax. **Modificar** para evaluar diferentes profundidades.
    - use_alphabeta (solo para MinimaxPlayer): Booleano que determina si se va a utilizar la poda Alpha-Beta. True es que si va a utilizarse. **Modificar** para evaluar con la poda, una vez implementada.
- **main_3_players.py:** Contiene la implementación del juego con 3 jugadores. Funciona de forma similar a **main.py**, pero se deben instanciar los jugadores Max3Player.

- **algorithms/minimax.py:** Contiene una implementación del algoritmo Minimax. **Modificar** para implementar la poda Alpha-Beta.
- **algorithms/max_n.py:** En este archivo deberás implementar el algoritmo Max N, para 3 o más jugadores. **Modificar**
- **evaluation/score.py: Modificar.** En este archivo se definirán las funciones de evaluación que utilizarán tus implementaciones de Minimax. Este archivo contiene las siguientes funciones de evaluación ya implementadas:
  - **defensive_simple_score:** Castiga que el oponente tenga fichas consecutivas en el tablero.
  - **chat_gpt_eval:** Función de evaluación entregada por ChatGPT.

  Luego tu tendrás que añadir tu función de valor según se detalla más abajo en las actividades.
- **evaluation/score_3.py: Modificar.** En este archivo se definirán las funciones de evaluación que utilizarán tus implementaciones de Minimax con 3 jugadores. Este archivo contiene las siguientes funciones de evaluación:
  - **fight_against_third:** Utiliza la función *evaluate_line* para recompensar al tercer jugador y castigar a los oponentes si el jugador 3 contiene piezas consecutivas, y recompensar a los oponentes por bloquear al tercer jugador.
  - **chat_gpt_eval_3:** Función de evaluación entregada por ChatGPT.

  Luego tu tendrás que añadir tu función de valor según se detalla más abajo en las actividades.

Se les recomienda que prueben el juego por su cuenta agregando un HumanPlayer.

3.2. Preguntas

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

Investiga sobre el uso de la llamada 'poda alfa-beta' para el algoritmo Minimax e implémentala en tu código cuando el argumento de la función `alphabeta` es `True`. Luego deberás comparar el desempeño de utilizar la poda alfa-beta. Para esto deberás hacer lo siguiente:

- Simular un enfrentamiento de dos Algoritmos MINIMAX **sin** poda con profundidad 1 y reportar el tiempo promedio de toma de decisión de cada jugador.
- Simular un enfrentamiento de dos Algoritmos MINIMAX **con** poda con profundidad 1 y reportar el tiempo promedio de toma de decisión de cada jugador.

Luego repite este proceso para profundidad 2, 3 y 4, con al menos 5 repeticiones. Por último elabora un tabla con tus resultados y responde a la pregunta ¿en que contribuye la implementación de la poda Alfa-Beta para el desempeño/eficiencia del algoritmo? Argumenta teóricamente a que se puede deber esto.

Se recomienda desactivar el visualizador, utilizando $view = NoView(Board)$.

IMPORTANTE: En las siguientes preguntas utiliza la poda alfa-beta para todos los agentes

Actividad 2: Comparación de rendimiento (0.5 ptos.)

Ahora, deberás comparar el desempeño de agentes que utilizan Minimax pero con profundidades distintas. Para esto debes hacer 10 experimentos para cada una de las siguientes configuraciones (Es muy recomendado modificar el código para que corra 10 juegos seguidos y reporte los resultados al final). Debes reportar el ganador y tiempo promedio de toma de decisión para cada profundidad. Utiliza la función de evaluación **defensive_simple_score** (es el parámetro `eval_function` de la clase **MinimaxPlayer**).
Configuraciones:
Tenga en cuenta que el agente Minimax Rojo juega primero

- Agente Minimax **id=1** con profundidad 1, contra Agente Minimax **id=2** con profundidad 1.
- Agente Minimax **id=1** con profundidad 1, contra Agente Minimax **id=2** con profundidad 2.
- Agente Minimax **id=1** con profundidad 2, contra Agente Minimax **id=2** con profundidad 1.
- Agente Minimax **id=1** con profundidad 1, contra Agente Minimax **id=2** con profundidad 3.
- Agente Minimax **id=1** con profundidad 3, contra Agente Minimax **id=2** con profundidad 1.

Deberás reportar el ganador y el tiempo promedio para una toma de decisión de cada profundidad. Indica (tanto teórica como experimentalmente) en que se traduce el cambio de profundidad del algoritmo en la forma de juego y argumenta por qué haciendo referencia al funcionamiento de Minimax.

Actividad 3: Comparación de funciones de evaluación (0.5 ptos.)

En esta actividad, deberás comparar las dos funciones de evaluación que se te entregaron: `defensive_simple_score()` y `chat_gpt_eval()`.
Para ello, se te pide que pruebes las siguientes configuraciones, nuevamente con 10 iteraciones cada una:

- Agente Minimax de **id=1** con `defensive_simple_score()` y profundidad 1, contrar Agente Minimax de **id=2** con `chat_gpt_eval()` y profundidad 1.
- Agente Minimax de **id=1** con `chat_gpt_eval()` y profundidad 2, contrar Agente Minimax de **id=2** con `defensive_simple_score()` y profundidad 2.
- Agente Minimax **id=1** con `defensive_simple_score()` y profundidad 1, contrar Agente Minimax **id=2** con `chat_gpt_eval()` y profundidad 3.
- Agente Minimax **id=1** con `defensive_simple_score()` y profundidad 3, contrar Agente Minimax **id=2** con `chat_gpt_eval()` y profundidad 1.

Para todas las pruebas deberás reportar el ganador y el tiempo promedio de decisión de cada jugada, para las distintas funciones y profundidades. Además, indica cuál de las funciones de evaluación es mejor según tus resultados, y analiza a qué puede deberse aquello.

Actividad 4: Nueva función de evaluación (0.6 ptos.)

Deberás implementar una función de evaluación en el archivo *score.py*, las cuales reciben un parámetro *board*, el cual representa al tablero de la clase *Board*. El tablero esta compuesto de una matriz, donde el valor 0 indica que no hay una pieza, y los valores 1 y 2 para indicar las fichas puestas por los jugadores, representados por su ID. También reciben un parámetro *player_id*, el corresponde al ID del jugador, siendo este el jugador al que se está evaluando.

Explica que hace la función creada y cual es la idea detrás de ella, es decir, porque crees que esa función de evaluación es buena para este problema.

Esta función tiene que ganarle como mínimo en un 70 % de las a la función $defensive\_simple\_score$, con un mismo nivel de profundidad. Este porcentaje de victoria debe ser al menos sobre 10 partidas. Puedes seleccionar el orden en que comienza como lo desees, pero debes mencionarlo. Reporta esto como una tabla en la que se incluyan las 10 partidas con sus resultados.

Actividad 5: Extender el juego a 3 jugadores (0.4 ptos.)

Luego de toda esta implementación y contento con el resultado, en vez de ir a tu clase de *IIC2612.5 Inteligencia Natural* te quedas jugando con tu compañero Jorge Baier en el patio de ingeniería. En eso, llega tu compañero Hans Löbel y les dice que quiere sumarse al juego, y que si no lo dejan va a ir a avisarle al profesor de *IIC2612.5 Inteligencia Natural* sobre la situación. El problema es que el juego no se puede jugar de a 3, por lo que están en un verdadero aprieto. Para salvar la situación decides investigar y encuentras un algoritmo llamado $\mathbf{Max}^N$ que permite extender el juego a N jugadores!
En esta sección tu misión será implementar este algoritmo.

IMPORTANTE: A partir de ahora, se recomienda modificar el tablero para las dimensiones 5×6, para evitar tiempos de búsqueda muy altos.

Algoritmo $\mathbf{Max}^N$

Para extender un juego como este a N jugadores, es posible utilizar una variante de Minimax llamada $\mathbf{Max}^N$ [3]. A diferencia del primero, este algoritmo no se basa en dos jugadores adversarios, en el que uno minimiza y el otro maximiza, sino que considera a los N participantes como agentes que buscan maximizar su propia utilidad.

Por esto, para cada iteración del algoritmo es necesario conocer las N utilidades que cada estado le otorga a los jugadores, a partir de lo cual cada jugador escogerá la que maximice la suya.

A continuación, se presenta un pseudocódigo del algoritmo:

max_n(S, i)
Input: Un estado S para el jugador i
Output: Vector $v \in \mathbb{R}^N$ con la evaluación de S

1. **if** S es terminal **or** se alcanzó la profundidad máxima
2. **return** funcion_evaluacion(S)
3. $v \leftarrow [-\infty, \ldots, -\infty]$
4. $j \leftarrow$siguiente_jugador(i)
5. **for each** $S' \in$ hijos(S) **do**
6. $v' \leftarrow$ max_n(S', j)
7. **if** $v'[i] > v[i]$
8. $v \leftarrow v'$
9. **return** v

Funciones auxiliares:

- funcion_evaluacion(S): para este algoritmo debe entregar un vector en $\mathbb{R}^N$, que representa la utilidad esperada para cada uno de los N jugadores.

[3] Más información: https://cdn.aaai.org/AAAI/1986/AAAI86-025.pdf

- siguiente_jugador(i): debe entregar el índice del siguiente jugador, pudiendo ciclar al primero si se llega al último.

A continuación deberás implementar este algoritmo en el archivo **algorithms/max_n.py**. El algoritmo debe considerar la extensión a N jugadores.
Hint: Para que el algoritmo funcione correctamente con el código proporcionado, debes retornar el mejor movimiento además del vector v (sigue la estructura de return del archivo **algorithms/minimax.py**).

Luego, para probar que el algoritmo es válido, debes ejecutar el archivo **main_3_players.py** que hace una simulación del juego con 3 jugadores.

3.2.1. Bonus (0.3 ptos.)

Para el caso de 3 jugadores, investiga e implementa una nueva función de evaluación, tal que haciendo un estudio comparativo, sea capaz de ganarle al menos el 60 % de las veces a 2 agentes Max^N con la función de evaluación `chat_gpt_eval_3()`, con un nivel menos de profundidad (por ejemplo, la nueva función de evaluación con profundidad 1 y `chat_gpt_eval_3()` con profundidad 2). Puedes considerar cualquier orden de jugadas entre los agentes, es decir, puede comenzar el agente con tu nueva función de evaluación o cualquiera de los otros dos agentes con `chat_gpt_eval_3()`, pero se debe especificar. Este porcentaje de victoria debe ser al menos sobre 10 partidas. Se revisará que esto se cumpla.

Qué entregar en esta parte

En un subdirectorio `DCConecta4/`:

- Un archivo `respuesta.pdf` con tus respuestas ordenadas. Incluye aquí las tablas y desarrollos de las preguntas.
- Los archivos de código modificados según se explicó anteriormente.