



## 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 21 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. Esto quiere decir que si ocupas dos cupones debes entregar la tarea el domingo 23 a las 23:59 hrs. y no el martes 25 a las 23:59 hrs.

#### 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

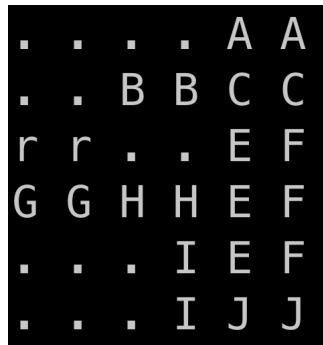
El objetivo de esta tarea es que puedan utilizar algoritmos de búsqueda con y sin adversario, como A\* y MiniMax, aplicándolos en problemas donde pueden ser de gran utilidad. Es fundamental que pongan énfasis en las justificaciones de sus respuestas, cuidando la redacción, ortografía; manteniendo el 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. DCCarros (3 pts.)

Es el 19 de septiembre y te encuentras en un taco devolviéndote de la playa. El taco es bastante poco convencional ya que hay autos posicionados paralelos a ti y otros posicionados perpendiculares a ti. Parece imposible encontrar la manera en que puedas salir de esta pero por suerte recordaste lo que aprendiste en el curso de Inteligencia Artificial sobre algoritmos de búsqueda. Es por esto que decides programar estos algoritmos para salir de la situación.

### Funcionamiento del juego

El juego consiste en mover los autos del tablero para despejar el paso para que el auto rojo (tu auto) pueda salir por el lado derecho de este. Un ejemplo de tablero sin resolver es el siguiente:



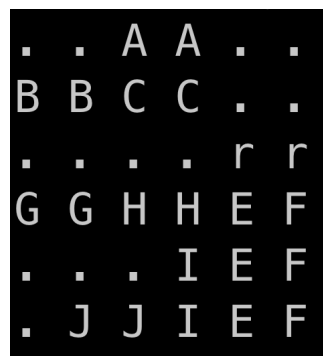
A 6x6 grid representing the initial game state. The grid contains letters representing cars and dots representing empty spaces. The cars are positioned as follows: Row 1: A, A; Row 2: B, B, C, C; Row 3: r, r, E, F; Row 4: G, G, H, H, E, F; Row 5: I, E, F; Row 6: I, J, J. The cars are oriented horizontally or vertically based on their position.

.	.	.	.	A	A
.	.	B	B	C	C
r	r	.	.	E	F
G	G	H	H	E	F
.	.	.	I	E	F
.	.	.	I	J	J

Figura 1: Ejemplo de tablero inicial

Como se puede observar, los autos E y F bloquean el paso del auto rojo (r). Los autos solo pueden moverse en la dirección en la que están sobre el tablero. En este ejemplo, A,B,C,G,H y J están posicionados de manera horizontal mientras que E, F e I están posicionados de manera vertical. r siempre está posicionado de manera horizontal.

Un ejemplo de tablero resuelto es el siguiente:



A 6x6 grid representing the solved game state. The grid contains letters representing cars and dots representing empty spaces. The cars are positioned as follows: Row 1: A, A; Row 2: B, B, C, C; Row 3: r, r; Row 4: G, G, H, H, E, F; Row 5: I, E, F; Row 6: J, J, I, E, F. The cars are oriented horizontally or vertically based on their position.

.	.	A	A	.	.
B	B	C	C	.	.
.	.	.	.	r	r
G	G	H	H	E	F
.	.	.	I	E	F
.	J	J	I	E	F

Figura 2: Ejemplo de tablero resuelto

La idea es encontrar la secuencia de movidas que nos permitan hacer que el auto rojo llegue al lado derecho del tablero. Si no estás familiarizado con el juego te recomiendo jugarlo en línea<sup>1</sup> para entender su funcionamiento.

<sup>1</sup>Puedes jugarlo aquí: <https://www.playit-online.com/puzzle-onlinegames/rush-hour/>

## Archivos del Repositorio

En la carpeta DCCarros podrás encontrar dos carpetas `/rushhour` y `/tests`. En la primera 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. No es necesario que lo modifiques.
- `solver.py`: Este archivo contiene el algoritmo A\*. Es la misma implementación vista en clases pero adaptada al problema de DCCarros. No es necesario modificarlo.
- `solver2.py`: Este archivo contiene la implementación de otro algoritmo que se utilizará más adelante en la tarea. No es necesario modificarlo.
- `run.py`: Este archivo permite correr los algoritmos implementados.

En la `/tests` encontrarás tests para probar el funcionamiento de A\* y el otro algoritmo. Para correr A\* debes ubicarte en `/DCCarros` y correr el siguiente comando:

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

Por ejemplo:

```
python rushhour/run.py astar tests/test7.txt blockingcars 1.5
```

Las heurísticas que puedes implementar son *blockingcars* y *zero*. Ahora, para correr el segundo algoritmo necesitas utilizar un estado inicial y el estado objetivo. En este caso el estado objetivo de cada test se encuentra en `/tests/solutions`. El comando para correr el segundo algoritmo es:

```
python rushhour/run.py solver2 <board> <final.board>
```

Por ejemplo:

```
python rushhour/run.py solver2 tests/test1.txt tests/solutions/test1.txt
```

### 1.1. Actividad 1: Anytime A\* (1 pts)

De lo visto en clases, sabemos que **Weighted A\*** obtiene soluciones en menos tiempo sacrificando la optimalidad de estas. Asimismo, muchas veces para resolver un problema tratamos con tiempo acotado para obtener la solución. Para este tipo de problemas se han desarrollado algoritmos que se conocen como *anytime heuristic search*.

En esta actividad deberás implementar un algoritmo de búsqueda *anytime* basado en A\* el cual funciona de la siguiente forma:

1. Para buscar, funciona como Weighted A\*, recibiendo un estado inicial, un peso y un límite de expansiones. Mantiene Open ordenada por  $f = g + wh$
2. la búsqueda siempre es interrumpida cuando el límite de de expansiones ha sido alcanzado, en cuyo caso se retorna la última solución encontrada.

3. Cuando una solución es encontrada, esta se retorna. En tu implementación Python **debes utilizar el keyword yield** para retornar. De esta manera se puede retomar la búsqueda si el usuario así lo desea.
4. Si el usuario desea una nueva solución, el algoritmo retoma la búsqueda, pero antes
  - a) Ajusta el peso a  $w = c_{\text{ult}} / \min_{s \in \text{Open}} \{g(s) + h(s)\}$ , donde  $c_{\text{ult}}$  es el costo de la última solución encontrada.
  - b) Cambia el valor  $f$  de todos los nodos en la Open de acuerdo a este nuevo peso  $w$ . Acá puedes utilizar el método `reorder` de la clase `BinaryHeap`. Lo único que debes hacer es cambiar la clave para todos los nodos en Open y luego llamar a `self.open.reorder()`.
5. Una vez que la solución con costo  $c_{\text{ult}}$  es conocida este algoritmo **no expande ni agrega a Open** a un nodo  $s$  si  $g(s) + h(s) \geq c_{\text{ult}}$ . Esto último se conoce como **poda**.

Este algoritmo es una variante de *Anytime Restarting A\** (ARA\*), pero funciona muchas veces mejor que ARA\*.

Para esta actividad se te entregará una implementación del algoritmo A\* en el archivo `solver.py`.

Para esta actividad deberás crear la clase `Anytime` en el archivo `anytime_solver.py` que implemente el algoritmo descrito anteriormente. Para esto deberás implementar el método `search` de esta clase. Se recomienda utilizar los siguientes métodos y/o atributos

- `BinaryHeap.get_nodes()`: Retorna una lista con los nodos que están dentro del `BinaryHeap`.
- `BinaryHeap.reorder()`: Reordena todos los nodos dentro del `BinaryHeap`
- `Node.g`: Valor  $g$  del nodo, es decir, el costo para llegar al nodo.
- `Node.h`: Valor  $h$  del nodo, es decir, el valor de la heurística.
- `Node.key`: Valor de  $f = g + w * h$  de la heurística

Recuerda comentar tu código con las partes nuevas que programes. También debes explicar la forma en la que se debe correr tu programa en el PDF asociado a esta pregunta.

## 1.2. Actividad 2: Admisibilidad de la heurística (0.35 pts)

Demuestra la admisibilidad o no admisibilidad de la heurística `BlockingCarsHeuristic` que cuenta la cantidad de autos bloqueando al auto rojo.

## 1.3. Actividad 3: Comparación de heurísticas (0.35 pts)

Haz una comparación empírica entre las heurísticas (`ZeroHeuristic` y `BlockingCarsHeuristic`) comparando la cantidad de nodos expandidos y el tiempo de ejecución. Para esto utiliza los tableros de ejemplo.

## 1.4. Actividad 4: Creación de heurística (0.3 pts)

Piensa en otra posible heurística admisible y demuestra su admisibilidad.

### 1.5. Actividad 5: Bidirectional Search (0.5 pts)

Investiga sobre Bidirectional Search (Búsqueda Bidireccional) y responde las siguientes preguntas. Recuerda citar tus fuentes.

- a) Explica en tus propias palabras qué es Bidirectional Search.
- b) Si observas con detenimiento, el algoritmo implementado en `solver2.py` se parece mucho a BFS. Explica en qué se diferencia con BFS y qué es lo que lo hace un algoritmo de búsqueda bidireccional. Para hacer esto, menciona líneas de código donde se observe lo pedido.

### 1.6. Actividad 6: Bidirectional Search y A\* (0.5 pts)

Con respecto a los dos algoritmos utilizados para resolver el problema de los autos, responde las siguientes preguntas:

- a) ¿Es posible incluir la idea de heurísticas en algoritmos de búsqueda bidireccional? De ser posible, explica cómo lo harías en el código de `solver2.py` (No es necesario programar nada, basta con que lo expliques en palabras). En esta pregunta se espera demuestres tu manejo con heurísticas y el entendimiento del algoritmo con el que se está trabajando.
- b) Compara el algoritmo bidireccional entregado con la implementación de A\* y explica sus principales diferencias, ventajas y desventajas. Para esto considera datos como la cantidad de nodos expandidos, la cantidad de pasos de la solución, el tiempo de ejecución, etc.

## 2. DCCanicas (3 pts.)

Luego de luchar contra los tacos por varias horas, finalmente lograste devolvete de la playa. Para relajarte después de tanto manejar, decides reunirte con tus amigos a jugar juegos de mesa. Increíblemente, perdiste todas las partidas en todos los juegos.

Para revalidar tu honra, decides buscar un juego que nadie conozca para así poder ganar. Te pones a buscar y encuentras en un rincón del DCC el fabuloso juego DCCanicas...luego de perder en ese también, decides hacer un algoritmo para resolverlo y nunca más perder.



Figura 3: El fabuloso DCCanicas

### Juego

El juego se trata de un tablero formado por varias *Piezas* donde puedes colocar *Canicas* de diferentes colores.

Los elementos son :

#### Piezas (tiles):

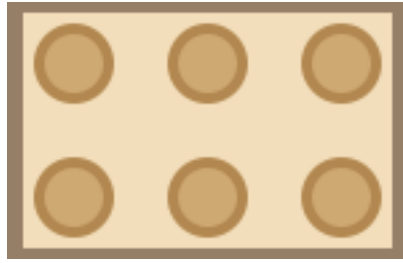
Son los espacios donde se pueden colocar canicas. Estas tienen dimensiones arbitrarias siendo siempre menores a las dimensiones del tablero.



Pieza de 1x3



Pieza de 2x2



Pieza de 2x3

### Canicas:

Cada jugador utiliza un color de canicas. El que comienza juega con las canicas rojas.



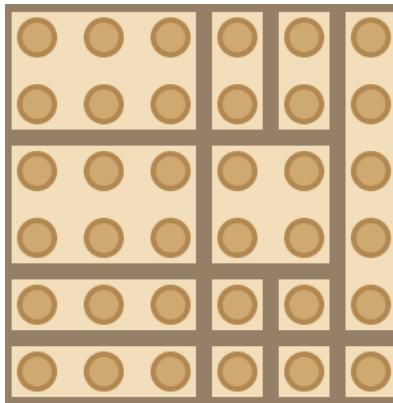
Figura 4: Canica roja.



Figura 5: Canica negra.

### Tablero:

El tablero del juego está formado por varias piezas.



### Reglas

Cada jugador dispone de una reserva de canicas de su color y, en cada turno, el jugador deberá colocar una canica en el tablero, cumpliendo:

- El espacio a ocupar tiene que encontrarse en la misma fila o columna que el espacio ocupado por el jugador rival en su turno anterior. Esta regla no aplica para la primera jugada. Así, la canica roja se puede jugar en cualquier posición

- No se puede colocar la canica en las piezas donde se colocaron las canicas de los dos turnos anteriores (el turno del rival y el del jugador activo).

En el ejemplo, vemos que la primera jugada fue rojo, luego negro y le toca a rojo de nuevo:



El jugador rojo está limitado a colocar su canica en una posición horizontal o vertical a la ultima puesta por el oponente (la negra).

El jugador rojo no puede colocar su canica en la pieza donde éste colocó su **ultima** canica, ni en la pieza donde negro colocó su **última** canica.

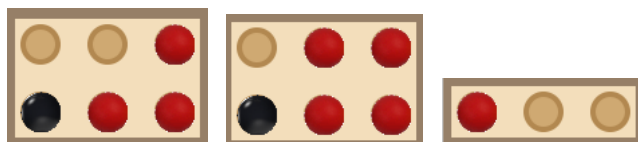
## Término del juego

El juego se termina cuando en un turno, ningún jugador tiene algún lugar válido donde colocar su canica. En ese momento se decide el puntaje.

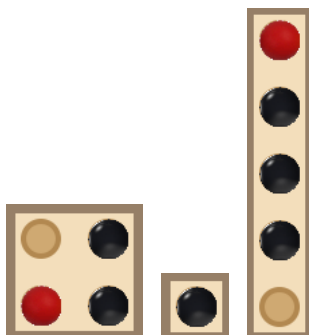
## Puntaje

El puntaje de una pieza corresponde a los espacios totales que tiene la pieza para canicas. Un jugador obtiene el puntaje de la pieza, si la mayoría de las canicas en la pieza son suyas.

En este caso Rojo suma 15 puntos:

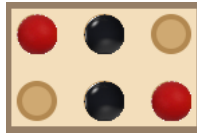


Negro suma 10 puntos:





Dado que esta pieza tiene la misma cantidad de canicas de cada color, ningun jugador recibe sus 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`
- `settings.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:
  - `Settings.intelligence_robot_color`: color puede ser `red` o `black` dependiendo de las canicas con las que juegue el robot. Este atributo es un string que puede ser `'Random'` si el robot realiza movimientos aleatorios o `'Minmax'` si el robot usa el algoritmo Minimax para hacer sus jugadas.
  - `Settings.robot_color_IQ`: color puede ser `red` o `black` dependiendo de las canicas 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.
- `status.py`: archivo que contiene la clase `gameStatus`, que contiene información importante sobre el estado del juego. En particular, los atributos que son de interés para ustedes son los siguientes:
  - `gameStatus.turn`: guarda el turno del jugador actual. Vale 0 si juega rojo y 1 si juega negro.
  - `gameStatus.available`: una lista de listas que representa las celdas del juego. Hay un 0 en la posición `[x][y]` si dicha celda está vacía, y un 1 en caso contrario.
  - `gameStatus.occupy`: similar a la lista anterior, pero esta guarda **quien** ocupa cada celda. 0 si está ocupada por rojo, 1 si está ocupada por negro, y -1 si está desocupada.

- `gameStatus.last_opponent`: guarda el último movimiento realizado como una tupla (x,y).
- **minimax.py**: archivos donde se debe implementar el algoritmo minimax. Contiene las siguientes funciones:
  - `get_potential_positions(gameStatus, board)`: entrega una lista de tuplas con los movimientos posibles dado el estado actual del juego y el tablero. No deben modificarla
  - `minimax(gameStatus, board, depth, score_function, 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.
- **score.py**: archivo que contiene las funciones de puntaje para el juego. Solo se debe modificar si quieren acceder al bonus. Contiene lo siguiente:
  - `get_score(gameStatus, board, player)`: A partir del estado del juego, el tablero y un jugador objetivo, retorna (ptje jugador objetivo - ptje. oponente). El robot la usa como función de evaluación por defecto para minimax. Ya viene implementada y no es necesario modificarla.
  - `bonus_score(gameStatus, board, player)`: solo deben implementarla aquellos que quieran participar del bonus

## 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.

- **image**: carpeta que contiene las imágenes e íconos del juego. No se debe modificar.
- **assets.py**: archivo que contiene clases para que funcionen los botones y canicas dentro del juego. Por ejemplo, está la clase `StartButton` que representa al botón para iniciar el juego. No se debe modificar.
- **board\_functions.py**: archivo que contiene funciones que relacionan el tablero y el estado del juego. Por ejemplo, contiene la función `check_available`, que verifica si un espacio está disponible para jugar, según las reglas. No se debe modificar.
- **board.py**: archivo que contiene la clase `Board` que genera el tablero de juego. No se debe modificar.
- **screen\_handling.py**: archivo que contiene clases y funciones para chequear clickeos sobre los botones en las pantallas iniciales y finales del juego. No se debe modificar.
- **robot.py**: archivo que contiene la clase `Robot` para controlar al contrincante del 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` (y `score.py` si desean acceder al bonus).

## ¿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 (1.5 pts)

En esta primera actividad, deberás completar la implementación del algoritmo Minimax que se encuentra en el archivo `minimax.py` <sup>2</sup>. Para ello, lee bien los comentarios incluidos 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 'Minimax' y asignarle una profundidad de búsqueda. Para ello, modifica los atributos correspondientes en `settings.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 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 50 juegos en el tablero de 7x7 utilizando Random como jugador negro y Minimax con profundidad 5 como jugador rojo.
2. Los resultados de 50 juegos en el tablero de 7x7 utilizando Minimax en ambos robots, con la misma profundidad.
3. Los resultados de 50 juegos en el tablero de 7x7 utilizando Minimax en ambos robots, pero con profundidades diferentes (profundidad 3 en robot rojo y profundidad 5 en robot negro, por ejemplo).

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 <sup>3</sup> (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:

---

<sup>2</sup>Debes implementar tu código en las secciones marcadas como TODO.

<sup>3</sup>Con recurso nos referimos a cualquier figura o tabla que ayude a la visualización de los datos.

- Es Minimax mejor que Random, ¿por cuánto?
- 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.

IMPORTANTE: Para esta parte de la tarea, al momento de correr el juego con `main.py`, pueden agregar en consola los siguientes parámetros que les pueden ser de utilidad:

- `-noplayer` o `-np` que automáticamente cambia al jugador humano por un robot.
- `-nomarbles` o `-nm` que provoca que no se muestren las canicas en el tablero.
- `-noscreens` o `-ns` que provoca que no se muestren las pantallas de inicio y finales del juego. Así no tienen que apretar `start` o `restart` para volver a jugar.
- `-sizeN` donde `N` es un entero, automáticamente cambia el tamaño del tablero a uno de `NxN`.
- `-ngamesN` donde `N` es un entero, se usa en conjunto con los otros parámetros para simular varios juegos de forma automática.
- `-write` genera un archivo `simulacion.json` con los datos de la simulacion.

Así por ejemplo, si ejecutan `python main.py -np -ns -size8 -ngames10 -write` entonces se jugarán 10 partidas entre robots sin mostrar las pantallas de inicio ni final y se escribirán los resultados al archivo `simulacion.py`<sup>4</sup>.

## 2.4. Bonus 2-6 décimas

Se hará una lucha de robots para ver cual es el mejor. Se pondrán a competir los algoritmos de los alumnos entre ellos y se definirá cual es el mejor.

Para obtener el bonus deberás crear una nueva función: `bonus_score(gameStatus, board, player)`<sup>5</sup>. (Si se creó el min-max con podas alpha beta esta es la forma más simple de obtener mejores resultados diferentes).

Para esta competencia se utilizará un tablero de 9x9 y la profundidad de exploración de Minmax será de 3 niveles. Por lo que deberán pensar en algún método de mejorar este juego específico. El desafío está en que deben buscar algún método en que su puntaje pueda tomar mejores decisiones sin aumentar la profundidad de min\_max.

Para que este bonus sea válido deben hacer una breve explicación sobre que cambio que aplicaron a la obtención de puntaje y por qué creen que debería mejorar con este en una sección que se llame "Bonus-MinMax" (Debido a que los tableros son aleatorios, puede pasar que un cambio beneficioso no sea evidente en los resultados).

Para mostrar los resultados se espera una tabla o gráfico en su informe donde se comparen:

- 50 partidas entre los robots Minmax y Minmax-Bonus, con Minmax como primer jugador.
- 50 partidas entre los robots Minmax y Minmax-Bonus, con Minmax-Bonus como primer jugador.

<sup>4</sup>El archivo `simulacion.py` contiene además un respaldo de todos los settings.

<sup>5</sup>Para crear esta nueva función de score, se recomienda revisen bien la función de score original en `score.py`. Pueden, al igual que en esa función, utilizar el atributo `board.tiles`, que contiene información sobre las piezas del tablero, en una lista de tuplas, donde cada tupla tiene la forma `(x,y, largo_x, largo_y)`, donde `x`, `y` determinan la posición de la esquina arriba-izquierda de la pieza y `largo_x`, `largo_y` las dimensiones de la pieza.

## **Puntaje**

Se obtendrán 2 décimas por implementar una nueva función `bonus_score` junto con una explicación, gráficos de resultados y justificación de por qué creen que debería mejorar. Si no mejora y la explicación es considerada válida por el ayudante, recibirán el puntaje.

El algoritmo ganador recibirá 4 décimas extra como bonus.