PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2025-2

Tarea 3
Problemas de Búsqueda

Fecha de entrega: lunes 20 de octubre 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 según se detalla en cada sección. El lugar de entrega es en el repositorio de la tarea, en la branch por defecto, hasta el lunes 20 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 **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 Brazo (3 puntos)

1.1. Introducción

Un día en medio de una semana de pruebas te encuentras estudiando en el DCC y de pronto comienzas a sentir mucho sueño por el cansancio de la semana. Este sueño hace que te distraigas de tu estudio y repentinamente empiezas a fantasear con tomarte un merecido café. De tanto pensar en esto te quedas dormido profundamente en la sala de estudios y empiezas a soñar con una cafetería del futuro!!! Esta cafetería es algo totalmente nuevo para ti, llena de tecnología muy avanzada. Uno de los aspectos que más llama tu atención de esta cafetería es que cuenta con un brazo robótico que ayuda a los baristas a preparar los más diversos bebestibles con una velocidad nunca antes vista. Te quedas un largo tiempo mirándolo e intentando descifrar como funciona, pero no logras descubrirlo. De pronto, alguien en la sala de estudios estornuda y tu te despiertas, dejándote para siempre con la incógnita del funcionamiento de este avanzado brazo.

Al día siguiente vas a tus clases del curso *IIC2613 Inteligencia Artificial*. Mientras sigues pensando en como implementar el brazo robótico, el profesor Jorge Baier comienza a hablar sobre búsqueda y sobre como esta área tiene como objetivo encontrar soluciones a problemas de lo más variado. En ese instante todo te hace sentido y resuelves tu incógnita! Puedes implementar el movimiento del brazo utilizando búsqueda y decides poner manos (o brazos) a la obra!

Figura 1: Ejemplo de robot cafetero

1.2. Explicación Problema

El objetivo de este problema (simplificado, pues solo pensaremos en movimientos en 2 dimensiones), es llevar el extremo de un brazo robótico a un punto específico del mapa, evitando chocar con los obstáculos. Para llegar a este punto final, hay que pasar primero por otro punto intermedio, así cada búsqueda tiene dos objetivos consecutivos. Este brazo cuenta con dos grados de libertad, es decir, dos articulaciones. A continuación un ejemplo de como se ve el brazo:

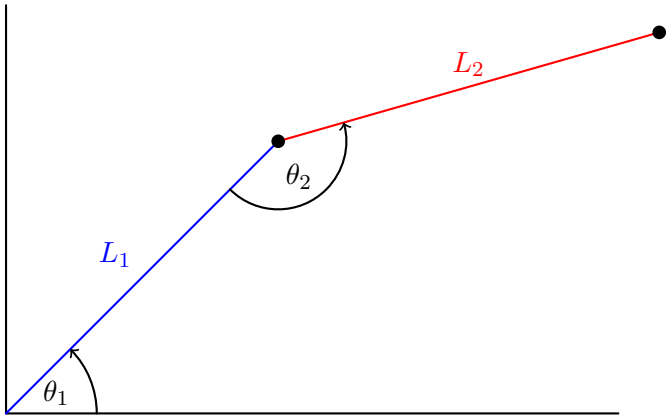

Figura 2: Componentes del brazo

Como podrás notar, el brazo tiene dos secciones llamadas `L1` y `L2`, donde `L2` comienza donde `L1` termina. Notemos que asociamos al brazo dos ángulos llamados θ_1 y θ_2 (nos referiremos a ellos como `theta1` y `theta2`), en donde `theta1` es el ángulo que se forma entre el eje `X` y `L1`, mientras que `theta2` es el ángulo que se forma entre `L1` y `L2`. Estos serán nuestros dos grados de libertad o articulaciones, y dado que `L1` y `L2` no cambian durante la ejecución, determinan por completo la posición del brazo. Para mayor claridad, cuando `theta1` es igual a 0 radianes, `L1` estará recostado sobre el eje. Si `theta2` es igual a 0 radianes, entonces `L2` estará doblado hacia dentro y se posará sobre `L1`, mientras que si `theta2` es igual a π radianes, entonces estará completamente extendido.

Las acciones posibles en cada instante de tiempo corresponden a modificar `theta1` y/o `theta2` en un Δ definido (más detalles de esto los puedes encontrar más adelante). En la implementación usarás radianes como unidad de medida de los ángulos, y Δ se definirá como **0.1 radianes**.

A continuación te mostramos una imagen que ilustra la posición inicial del brazo robótico `theta1 = 0`, `theta2 =` π. Esta imagen fue extraída del visualizador que podrás utilizar más adelante:

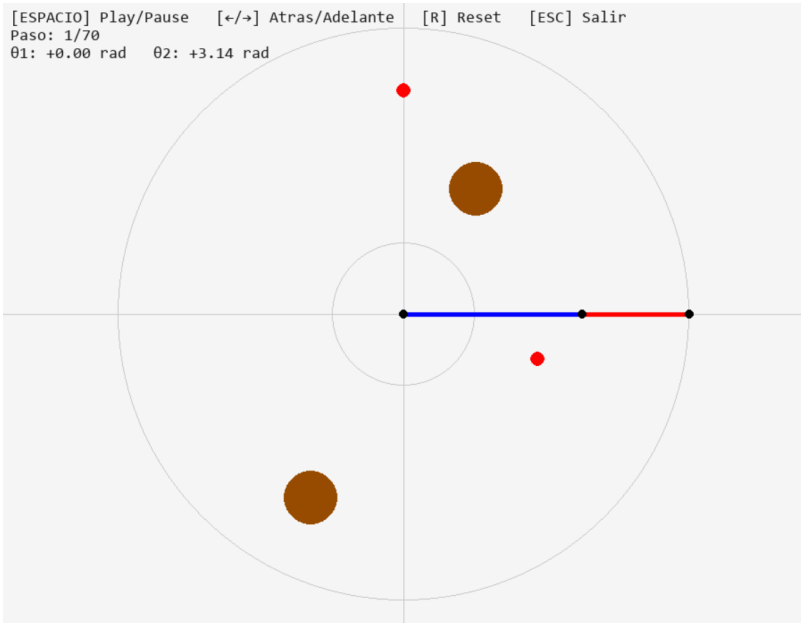

Figura 3: Posición Inicial del brazo `theta1 = 0,` π `radianes`

En la imagen puedes ver que, además del brazo, hay dos tipos de círculos. Los círculos cafés corresponden a los objetivos a los cuales el extremo del brazo robótico (es decir, el final de `L2`) debe llegar. Por otro lado, los círculos rojos corresponden a obstáculos en el mapa. El brazo no puede pasar sobre ellos.

1.3. Archivos

- `problems`: Carpeta que contiene archivos `.txt` con configuraciones para distintos problemas. Los problemas que terminan en `_obs` son aquellos que contienen obstáculos. **No modificar.**
- `algorithms/ara.py`: Contiene una implementación del algoritmo `ARA*`. **Se debe completar en la Actividad 4.**
- `algorithms/astar.py`: Contiene una implementación del algoritmo `A*`. **No modificar.**
- `algorithms/bfs.py`: Contiene una implementación del algoritmo `BFS`. **No modificar.**
- `algorithms/binary_heap.py`: Contiene la clase `BinaryHeap` que corresponde a una implementación de un Heap binario en donde cada elemento del heap tiene la propiedad `heap_index` que corresponde a su posición en el heap. **No modificar.**
- `algorithms/node.py`: Contiene la clase `Node` que implementa un nodo en un árbol de búsqueda. **No modificar.**
- `arm_viewer.py`: Contiene la clase `ArmViewer` la cual nos permite obtener una visualización del brazo robótico. **No modificar.**
- `consts.py`: Contiene constantes que necesitarás durante la actividad. Por ejemplo, las longitudes de los segmentos del brazo, el radio de algunos objetos, las dimensiones de la pantalla, entre otras. **No modificar.**
- `main.py`: Contiene el flujo principal. **No modificar.** Al ejecutar este archivo se abrirá el visualizador. Si deseas que no se vea el visualizador, puedes comentar las siguientes lineas (50 y 51):

```
arm_viewer = ArmViewer(robot, search_1_path, search_2_path, GOAL_1, GOAL_2,
    R_MESA, obs_info)

arm_viewer.run()
```

- `obstacles.py`: Contiene la clase `Obstacle` y la función `create_obstacles`. **No modificar.**
- `read_problems.py`: Contiene las funciones `read_problem.py` y `read_tuple.py`. **No modificar.**
- `robot.py`: contiene la clase `RobotState` la cuál contiene métodos para el correcto funcionamiento del brazo robótico, estas son:
  - `self(L1, L2, OBSTACLES)`: Es el constructor de la clase, recibe los largos de los segmentos del brazo, y una lista con los obstáculos. **No modificar.**
  - `succ(state)`: Dado un estado, devuelve todos sus sucesores. Considera que el cambio (Δ) para cada ángulo puede ser de 0.1 radianes. A modo de ejemplo, si tengo una configuración `(theta1, theta2)` sus sucesores son: `(theta1` $\pm$ Δ `*` y_1`, theta2` $\pm$ Δ_2 `*` y_2`)`, donde y_1, $y_2 \in \{0,1\}$, exceptuando el caso en que $y_1 = 0$ y $y_2 = 0$. **No modificar.**
  - `is_goal(state, goal)`: Revisa si el estado `state` se encuentra en la coordenada objetivo. **No modificar.**

- `check_collisions(state, next_state)`: Revisa si al pasar de un estado a otro el brazo choca con algún obstáculo. **No modificar.**
- `interpolate_states` y `check_collision_segment`: Funciones que se usan dentro de `check_collisions`. **No modificar.**

- `trig.py`: Contiene funciones que usan trigonometría para funcionalidades del brazo, estas son:
  - `ang_to_cart(theta1, theta2, L1, L2)`: Función que dado los ángulos `theta1, theta2` y los largos `L1, L2` de los segmentos del brazo, retorna la posición del extremo de cada segmento. **Se debe completar en la Actividad 1.**
  - `heuristic_trig(state, goal, L1, L2)`: Función heurística que recibe el estado (los ángulos del brazo), el objetivo (las coordenadas a las que queremos que llegue el final del brazo) y los largos de los segmentos del brazo. **Se debe completar en la Actividad 1.**
  - `normalize_angle(angle)`: Asegura que el ángulo resultante quede dentro del rango $[0, 2\pi)$. **No modificar.**
  - `angle_dist(a, b)`: Calcula la distancia mínima entre dos ángulos, considerando que el brazo puede girar tanto en sentido horario como antihorario. **No modificar.**
- `tests_ang_to_cart.py`: Contiene 3 tests para evaluar si la función `ang_to_cart(theta1, theta2, L1, L2)` ha sido bien implementada. **No modificar.**
- `tests_heuristic.py`: Contiene 3 tests para evaluar si la función `heuristic_trig(state, goal, L1, L2)` ha sido bien implementada. **No modificar.**

1.4. Preguntas

Actividad 1: Ecuaciones y heurística (0.75 puntos)

Uno de los principales desafíos de este problema es plantear correctamente todas las ecuaciones que guiarán el movimiento de nuestro brazo. Debido a eso, en esta parte de la tarea deberás completar las funciones `ang_to_cart(theta1, theta2, L1, L2)` y `heuristic_trig(state, goal, L1, L2)` que se encuentran en el módulo `trig.py`. Esto incluye su desarrollo matemático, por lo que te pediremos que incluyas dichos procedimientos en un archivo `respuesta.pdf` en el directorio `DCCBrazo`.

- La función `ang_to_cart` recibe como entrada los ángulos `theta1, theta2` y las longitudes `L1, L2` de cada segmento del brazo. Como salida, debe devolver las coordenadas de los extremos de los segmentos en el plano cartesiano, en el formato `((x1, y1), (x2, y2))`. En esta notación la primera tupla corresponde al **extremo del primer segmento** (P_1 en la imagen), mientras que la segunda tupla corresponde al **extremo del segundo segmento** (P_2 en la imagen).

  Como ayuda puede ser útil imaginar el brazo en un plano cartesiano, como se puede ver en la siguiente imagen:

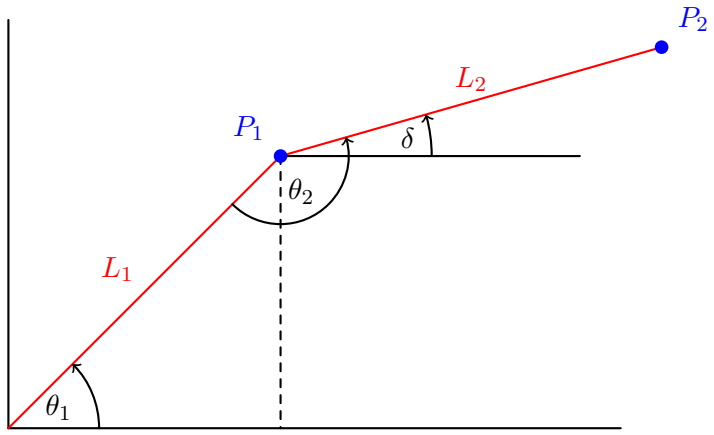

Figura 4: Brazo en el plano cartesiano

Si necesitas refrescar algún conocimiento de trigonometría, te podría ser útil revisar la siguiente página web[1]. Para calcular la posición final del segundo segmento conviene calcular primero el ángulo δ de la figura. Afortunadamente δ se puede calcular a partir de θ_1 y de θ_2 usando las propiedades básicas de la geometría euclidiana. Es tu labor dar ese paso!

- La función `heuristic_trig` recibe como entrada el estado para el cual queremos calcular la heurística, las coordenadas del objetivo y las longitudes de cada segmento del brazo. Como salida, devuelve la heurística correspondiente a ese estado.

  Para entender bien por qué te pedimos lo que hay que hacer, recordemos un poco la materia. Vamos a llegar a usar A*, que requiere de un valor heurístico $h(s)$. Nuestro estado s está determinado por una configuración de (θ_1, θ_2). $h(s)$ debe devolver una estimación de costo de llegar hasta un cierto punto objetivo dado por una posición en el plano cartesiano (x, y). Entonces lo que queremos hacer es determinar cuánto se debe mover el brazo, es decir, cuánto deben cambiar los ángulos θ_1 y θ_2 para alcanzar la posición (x, y) del plano cartesiano.

  Para calcular la heurística podemos volver a imaginarnos el brazo en un plano cartesiano. Para llegar a un punto objetivo, el brazo puede adoptar dos posiciones diferentes: una con el codo hacia arriba (color rojo en la imagen) y otra con el codo hacia abajo (color naranjo en la imagen). En la siguiente imagen te mostramos ambas configuraciones para mayor claridad:

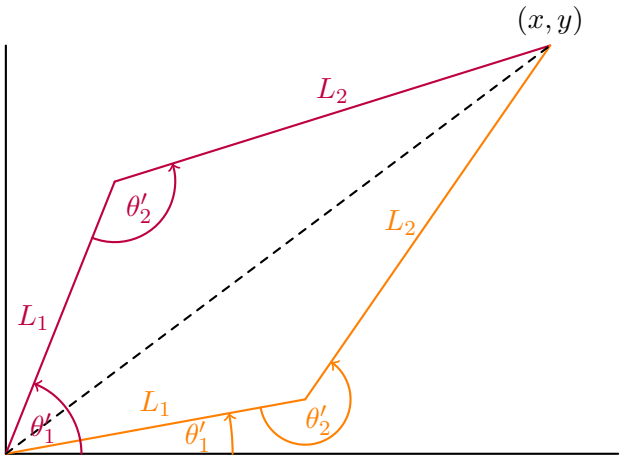

Figura 5: Brazo en el plano cartesiano

[1] https://es.wikipedia.org/wiki/Función_trigonométrica

Ciertos valores para los ángulos de cada brazo nos permiten llegar al punto objetivo. Como ayuda para calcular los ángulos puedes volver a usar la página web que usaste para completar `ang_to_cart` y también puedes revisar el Teorema del Coseno[2].

Una vez que tienes los ángulos para ambas configuraciones, la heurística se define como la cantidad de pasos necesarios para pasar de una configuración a otra. Para ello te puedes apoyar de las funciones `normalize_angle` y `angle_dist`.

Recuerda que en cada movimiento cada ángulo puede variar 0.1, -0.1 o 0 radianes, y que ambos ángulos `theta1, theta2` pueden cambiar de forma simultánea. Para ver un ejemplo concreto de cómo se generan los sucesores, revisa el método `succ` de la clase `RobotState` en `robot.py`.

Ten presente además que cada configuración puede requerir una cantidad distinta de pasos y nosotros **queremos obtener el camino más corto**.

Finalmente, la función `heuristic_trig` debe devolver un número entero; para esto te recomendamos usar la función `round`.

Para verificar que tus funciones `ang_to_cart` y `heuristic_trig` estén correctamente implementadas, te entregamos los módulos `tests_ang_to_cart.py` y `tests_heuristic.py`. Cada módulo incluye 3 pruebas. Para ejecutarlas, basta con correr el archivo correspondiente en la consola de la siguiente manera:

```
python <nombre_tests>
```

Aquí, `nombre_tests` puede ser `tests_ang_to_cart.py` o `tests_heuristic.py`. Si las pruebas se ejecutan correctamente, se mostrará `PASS`; en caso contrario, aparecerá `FAIL` junto con una comparación entre tu salida y la respuesta esperada.

Actividad 2: Problemas sin obstáculos (0.5 puntos)

Ya con las funciones `ang_to_cart` y `heuristic_trig` correctamente implementadas, es hora de utilizarlas! Para aquello deberás realizar los siguientes experimentos:
Ejecutar los 5 problemas **sin** obstáculos disponibles en el directorio `problems`. Para cada uno debes realizar la búsqueda con los siguientes algoritmos:

1. `BFS`.
2. `A*` con la heurística `zero`
3. `A*` con la heurística `heuristic_trig`

Para cada experimento debes registrar la siguiente información:

- Número de expansiones
- Tiempo (en segundos) que tomó la búsqueda
- Largo de la solución obtenida

Para poder ejecutar el juego sin obstáculos debes escribir lo siguiente en la consola:

```
python main.py <n_problema> <algoritmo> False
```

Donde `n_problema` corresponde al número del problema que deseas ejecutar y `algoritmo` es el nombre del algoritmo que quieres utilizar. Por ejemplo, si quieres ejecutar el segundo problema usando BFS debes escribir:

[2] https://es.wikipedia.org/wiki/Teorema_del_coseno

```
python main.py 2 bfs False
```

IMPORTANTE: si deseas correr A* con la heurística `zero`, debes usar `zero` como nombre de algoritmo. Para más detalles, puedes revisar la implementación en `main.py`.

En tu archivo de respuestas debes incluir una tabla que contenga esta información para cada problema y para cada algoritmo. Luego debes realizar un análisis sobre los resultados. Para esto considera las siguientes preguntas: ¿Qué algoritmo/configuración fue más eficiente? ¿Dónde se observa aquello? ¿Cómo se comparan las soluciones que entrega cada algoritmo/configuración? ¿Hay diferencias entre usar `A*` con la heurística `zero` versus la heurística `heuristic_trig`? ¿En qué se expresa y a qué se debe?, ¿Cómo se comparan los tres algoritmos? etc. (Se recomienda fuertemente incluir más aspectos en la respuesta).

Actividad 3: Problemas con obstáculos (0.5 puntos)

Luego de analizar los casos sin obstáculos, te preguntas si es que al agregar obstáculos habrán cambios en los resultados. Debido a esto, tu misión ahora es ejecutar los 5 problemas **con** obstáculos disponibles en el directorio `problems`. Para cada uno debes realizar la búsqueda con los siguientes algoritmos:

1. `BFS`.
2. `A*` con la heurística `zero`
3. `A*` con la heurística `heuristic_trig`

Para cada experimento debes registrar la siguiente información:

- Número de expansiones
- Tiempo (en segundos) que tomó la búsqueda
- Largo de la solución obtenida

Para poder ejecutar el juego con obstáculos debes escribir lo siguiente en la consola:

```
python main.py <n_problema> <algoritmo> True
```

En tu archivo de respuestas debes incluir una tabla que contenga esta información para cada problema y para cada algoritmo.

Luego debes realizar nuevamente un análisis sobre los resultados. Para esto considera las preguntas respondidas en la sección anterior además de responder a ¿Se mantienen las diferencias entre usar `A*` con la heurística `zero` versus la heurística `heuristic_trig`? ¿Estas se acrecientan o disminuyen? ¿A qué se debe? ¿Qué diferencia notas entre el rendimiento de los algoritmos con obstáculos versus sin obstáculos? ¿A qué se debe esta diferencia? ¿Qué podemos concluir sobre las heurísticas `zero` y `heuristic_trig`?, etc. (Se recomienda fuertemente incluir más aspectos en la respuesta).

Actividad 4: `Weighted A*` (0.5 puntos)

Como hemos visto en clases, en el algoritmo `A*` es posible agregar un peso w al cálculo de f, lo cual sacrifica la optimalidad de la solución, pero entrega soluciones más rápidas. Este algoritmo es conocido como `Weighted A*`. Como podrás notar, en la implementación entregada del algoritmo `A*` hay un atributo `self.weight` que hace referencia a este peso w. Esta versión del algoritmo despierta enormemente tu curiosidad, por lo que decides probarlo para este problema y evaluar su efecto en las soluciones.

Ahora tu tarea será ejecutar los 5 problemas **sin y con** obstáculos disponibles en el directorio `problems`. Para cada uno debes realizar la búsqueda con las siguientes configuraciones del algoritmo `Weighted A*`:

1. `A*` con la heurística `heuristic_trig` y $w = 2$
2. `A*` con la heurística `heuristic_trig` y $w = 3$
3. `A*` con la heurística `heuristic_trig` y elegir un valor de w que te resulte interesante

Para cada experimento debes registrar la siguiente información:

- Número de expansiones
- Tiempo (en segundos) que tomó la búsqueda
- Largo de la solución obtenida

Para ejecutar con weighted A* puedes hacerlo de la misma forma que en la actividad anterior pero debes editar el atributo `weight`, para ello debes editar la línea 26 de `main.py` de la forma

```
alg = AStar(robot, weight=2)
```

En tu archivo de respuestas debes incluir una tabla que contenga esta información para cada problema y para cada algoritmo. Incluye en esta tabla los resultados de `A*` con la heurística `heuristic_trig` y $w = 1$ para los mismos problemas, los cuales ya fueron obtenidos en las actividades 2 y 3.

Luego deberás realizar un análisis sobre los resultados. Para esto considera las preguntas respondidas en la sección anterior además de responder a: ¿Cómo influye el valor de w en los resultados de las búsquedas? ¿Fueron todos los valores de w igual de eficientes? ¿Qué efectos se ven al aumentar el valor de w? ¿Cambia el largo del camino encontrado al cambiar el valor de w? ¿Cuál crees que fue la mejor configuración de w? ¿Qué nos dice esto sobre la informatividad de la heurística `heuristic_trig` al tener obstáculos? (Se recomienda fuertemente incluir más aspectos en la respuesta).

Actividad 5: Búsqueda *Anytime* (0.75 puntos)

Con lo que has hecho hasta el momento esperamos que sientas que has aprendido dos cosas: (1) usar una heurística más informativa reduce el esfuerzo para encontrar una solución óptima y (2) usar un peso en la heurística afecta enormemente el tiempo de búsqueda, sacrificando la optimalidad en forma controlada (dependiente del valor de w).

Pero en la práctica muchas veces ocurre que tenemos un **tiempo fijo y conocido** para encontrar una solución y quisiéramos usar todo ese tiempo para **encontrar la mejor solución posible**. Para estos casos, se han creado algoritmos que hacen lo que se conoce como *anytime heuristic search*.

En esta parte deberás implementar una algoritmo de búsqueda *anytime*. El algoritmo está basado en A* y funciona de esta forma:

1. Para buscar, funciona como `Weighted A*`, recibiendo un estado inicial, un peso, y un *límite de expansiones* como parámetro. Mantiene Open ordenada por $f = g + wh$
2. La búsqueda siempre es interrumpida cuando el límite de expansiones ha sido alcanzado, en cuyo caso se retorna la última solución encontrada.
3. Cuando una solución es encontrada, ésta 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 $w = c_{ult} / \min_{s \in Open} g(s) + h(s)$, donde c_{ult} es el costo de la última solución encontrada.

   b) Cambia el valor f de cada nodo en la Open de acuerdo a este nuevo peso w. Acá, puedes usar el método `reorder` de la clase `BinaryHeap`. Lo único que debe hacer es cambiar la clave de todos los nodos en open (notar que `BinaryHeap` es iterable) y luego llamar a `self.open.reorder()`.

5. Una vez que una solución con costo c_{ult} es conocida este algoritmo **no expande ni agrega a open** a un nodo s si $g(s) + h(s) \geq c_{ult}$. Esto último se conoce como **poda**.

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

Para implementar este algoritmo modifica la clase `Ara` en el archivo `ara.py`.

Luego, para evaluar su rendimiento, deberás ejecutar los 5 problemas **sin y con** obstáculos disponibles en el directorio `problems`. Para cada uno debes realizar la búsqueda con `ARA*` usando la heurística `heuristic_trig`.

Para este experimento debes registrar la siguiente información:

- Número de expansiones
- Tiempo (en segundos) que tomó la búsqueda
- Largo de la solución obtenida

En tu archivo de respuestas debes incluir una tabla que contenga esta información para cada problema y para cada algoritmo. Incluye también los resultados para los mismos problemas pero al utilizar `weighted A*` y `A*` (ambos con la heurística `heuristic_trig`) obtenidos anteriormente.

Por último, debes analizar los resultados. Para esto considera las preguntas respondidas en la sección anterior además de responder a ¿Cómo se comparan las respuestas encontradas por `ARA*` versus `A*`? ¿Y en comparación a `weighted A*`? ¿Cuál algoritmo es más eficiente? ¿Por qué consideras aquello? ¿Crees que en este problema vale la pena sacrificar optimalidad por velocidad? ¿Cambia el comportamiento de `ARA*` al enfrentar problemas con obstáculos versus sin obstáculos? (Se recomienda fuertemente incluir más aspectos en la respuesta).

Qué entregar en esta parte

En un subdirectorio `DCCBrazo/`:

- 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)

En esta sección te pediremos que entiendas el porqué del paso de cambio de clave (paso 4.a) del algoritmo ARA*, que vimos en la sección anterior.

Como vimos en clases, el algoritmo Weighted A*, base de cada búsqueda de ARA*, entrega una importante garantía cuando es ejecutado con una heurística admisible h y un peso $w \geq 1$: si retorna una solución de costo c, entonces c es menor o igual a $w \cdot c^*$, donde c^* es el costo de una solución óptima. La demostración de este teorema está en esta cápsula. La consecuencia de este teorema es agradable, dado que el algoritmo entrega una garantía de suboptimalidad.

Sin embargo, si tenemos acceso a los estados en Open justo al momento de retornar, entonces generalmente podemos obtener una mejor cota, es decir, una mejor idea de cuán subóptima es la solución que retornamos.

Concretamente, demuestra el siguiente teorema. En una ejecución de Weighted A*, denotemos por O al contenido de la Open *justo antes* que extraemos el nodo solución que reportamos al usuario. Sea c el costo de la solución que encontraremos. Entonces c es menor o igual a $w' \cdot c^*$, donde c^* es el costo de una solución óptima y w' está dado por:

$$w' = \frac{c}{\min_{s \in O} g(s) + h(s)} \tag{1}$$

La demostración debe ser completa, es decir, incluir todos los detalles. Te aconsejamos fuertemente que utilices, como parte de la demostración, el lema de la cápsula, pero en tu respuesta **debe estar demostrado (por escrito) el lema** si lo utilizas.

Reflexión: Este resultado teórico es interesante, pues aunque hayamos utilizado $w = 2$, podríamos obtener que $w' = 1.2$; esto significaría que para buscar una nueva solución, mejor que la anterior, debieramos apuntar a buscar una solución que no exceda en 20 % al costo óptimo.

Qué entregar en esta parte

En un subdirectorio `teoria/`:

- Un archivo `respuesta.pdf` con tu demostración.

3. DCC Quoridor (2.5 puntos)

3.1. Introducción

Como estudiante universitario es común tener ventanas entre clases, sin mucho que hacer. Para aprovechar el tiempo, decides juntarte con tus amigos y uno de ellos aparece con un juego de mesa llamado Quoridor. El tablero incluye dos peones y un conjunto de pequeñas paredes, y el objetivo es claro: llegar primero al extremo opuesto mientras entorpeces el avance de tu rival.

Después de varias rondas, descubres estrategias que te aseguran la victoria con demasiada facilidad (¿o será que tus amigos no son tan buenos jugadores?). Empiezas entonces a preguntarte cómo hacer que este juego represente un verdadero reto. En ese instante recuerdas tus clases del curso *IIC2613 Inteligencia Artificial*, donde aprendiste sobre el famoso algoritmo Minimax, y se te ocurre la idea perfecta: adaptar Quoridor para competir contra una IA que siga este algoritmo.

Tu misión en esta tarea será implementar DCC Quoridor utilizando Minimax, con el fin de enfrentarte a un desafío real y, de paso, darle un buen uso a esas ventanas entre clases.

3.2. Explicación juego Quoridor

El juego se desarrolla en un tablero de 9×9 casillas y cada jugador dispone de un peón y de diez paredes. El objetivo consiste en llevar el peón propio desde la fila inicial hasta alcanzar cualquiera de las casillas de la fila opuesta, siendo ganador el primer jugador en lograrlo.

En cada turno, un jugador debe elegir entre mover su peón o colocar una pared. El peón puede avanzar una casilla en dirección vertical u horizontal, siempre que no haya una pared bloqueando el paso. También es posible saltar al peón rival si se encuentra justo enfrente y no hay una pared entre ambos; si el salto está bloqueado por una pared, entonces el movimiento puede hacerse en diagonal hacia la izquierda o hacia la derecha.

Las paredes se colocan de forma horizontal o vertical entre dos casillas y sirven para obstaculizar el avance del adversario. No obstante, está prohibido colocar paredes que bloqueen completamente todos los caminos hacia la meta; siempre debe quedar al menos una ruta disponible.

El juego termina en el momento en que uno de los jugadores alcanza con su peón la fila opuesta a la que comenzó. Ese jugador es declarado ganador.

Reglas

1. Los jugadores juegan por turnos.
2. En su turno, un jugador debe elegir entre:
   - **Mover su peón:** una casilla en dirección vertical u horizontal, siempre que no haya una pared bloqueando.
   - **Colocar una pared:** en orientación horizontal o vertical, bloqueando el paso entre dos casillas.
3. Está prohibido colocar una pared que bloquee completamente el acceso del oponente a su fila objetivo; siempre debe quedar al menos un camino disponible.
4. **Saltos:**
   - Si el peón de un jugador está frente al peón del rival y no hay una pared en medio, puede saltar sobre él.
   - Si detrás del rival hay una pared, el jugador puede moverse en diagonal hacia la izquierda o derecha.

Recomendamos probar el juego en https://es.igre.games/quoridor-online/play/.
A continuación se puede ver una imagen del tablero con ambos peones y paredes:

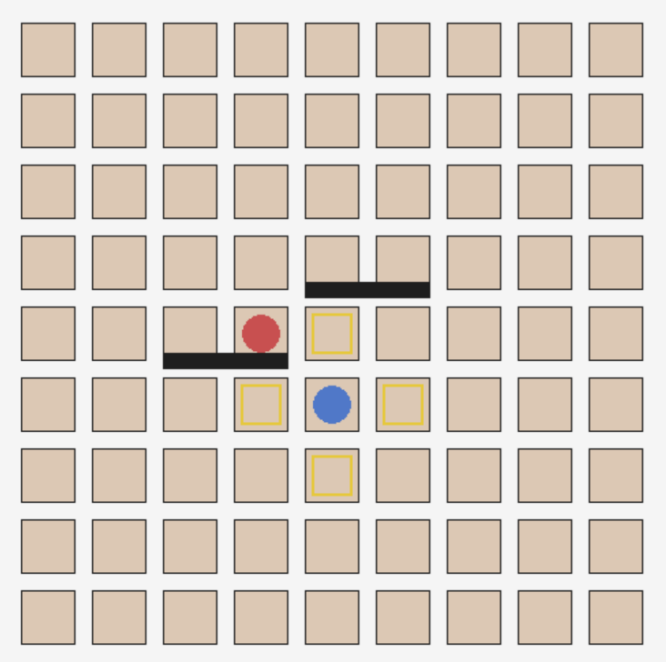

Figura 6: Tablero

Para probar el juego puedes utilizar el modo *visualizador* y en `main.py` seleccionar que uno de los *players* sea un jugador humano. Para seleccionar la acción que quieres realizar hay 3 botones distintos:

- **M**: Al apretar la tecla **M** entrarás al modo movimiento, ahí se indicarán hacia que casillas el jugador puede moverse. Basta con seleccionar en el *visualizador* la casilla a la que quieres mover el peón.
- **H**: Al apretar la tecla **H** entrarás al modo de colocación de paredes horizontales, en el *visualizador* se indicará todos los posibles lugares donde es posible colocar una pared, basta con seleccionar un casillero posible para que la acción se ejecute.
- **V**: Al apretar la tecla **V** entrarás al modo de colocación de paredes verticales, que funciona del mismo modo que con las paredes horizontales.

3.3. Archivos

A continuación se entrega una explicación de los archivos y códigos disponibles en la carpeta **DCCQuoridor**:

- `board.py`: Archivo que contiene las clases de peón y su posición, del tablero, y sus atributos. **No modificar**.
- `game.py`: Archivo que contiene la clase Game que determina la lógica y funcionalidad del juego. **No modificar**.
- `ui.py`: Archivo que contiene la clase UI que determina la vista del juego. **No modificar**.
- `player.py`: Archivo que contiene las clases HumanPlayer y MinimaxPlayer que modela a un jugador. **No modificar**.

- `minimax.py`: Archivo que contiene una implementación del algoritmo Minimax. **Modificar para implementar la poda Alpha-Beta**.
- `algorithms.py`: Archivo que contiene algoritmos utilizados para encontrar un camino hacia la meta. **No modificar**.
- `score.py`: Archivo que contiene las funciones de evaluación que utilizará el algoritmo minimax. **Modificar para implementar las funciones de evaluación**. Este archivo contiene las siguientes funciones de evaluación:
  - `evaluate()`: Función básica que pondera distancia hacia la meta de los agentes y las paredes restantes
  - `chat_gpt_eval()`: Pondera las distancias hasta la meta, las paredes restantes y premia el manejo del centro del tablero.
- `main.py`: Archivo que contiene el flujo principal del juego, que controla como se inicializan las clases y se inicia el juego. Este es el archivo que se debe ejecutar para correr el juego. **Modificar**. Adicionalmente, tiene las siguientes variables que se pueden cambiar para testear el código:
  - `main`: Instancia de main al final del archivo donde se ingresa las dimensiones del tablero, las cuales se pueden modificar si es que es necesario.
  - `player1` y `player2`: Instancias de la clase `Player`. Estos pueden ser `HumanPlayer` y `MinimaxPlayer`. Contienen los siguientes atributos:
    - `name`: El nombre del jugador. **No modificar**.
    - `depth` (solo para `MinimaxPlayer`): Determina la profundidad de búsqueda del jugador para el algoritmo Minimax. **Modificar para evaluar diferentes profundidades.**
    - `eval_fun` (solo para `MinimaxPlayer`): Determina la función de evaluación a utilizar por el jugador en Minimax. **Modificar para utilizar diferentes funciones de evaluación.**
    - `use_alphabeta` (solo para `MinimaxPlayer`): Booleano que determina si se va a utilizar la poda Alpha-Beta. `True` indica que si va a utilizarse. **Modificar para evaluar con la poda, una vez implementada.**
  - `ui`: Instancia de la clase `UI`. Contiene los siguientes atributos:
    - `player1`, `player2`: Se entregan las instancias de ambos jugadores. **No modificar**
    - `headless`: Booleano que determina si se quiere ver la visualización del juego. `True` indica que **no** se muestra la visualización. Si alguno de los jugadores es humano, **siempre** se mostrará el juego. **Modificar si se necesita.**

Para ejecutar el juego solo basta con correr el archivo `main.py`.

3.4. Preguntas

Actividad 1: Poda Alpha-Beta (0.5 puntos)

Investiga sobre el uso de la llamada "poda alfa-beta"[3] para el algoritmo Minimax e implémentala en tu código, para que se utilice cuando el argumento de la función `alphabeta` sea `True`. Luego deberás comparar el desempeño de utilizar la poda alfa-beta verus no hacerlo. Para esto deberás hacer lo siguiente:

[3]Puedes apoyarte en el libro de Russell y Norvig (detalles en el programa) o las diapos de clases.

- Simular cinco enfrentamientos de dos Algoritmos Minimax **sin** poda con profundidad 1 y reportar el tiempo promedio de toma de decisión de cada jugador.
- Simular cinco enfrentamientos de dos Algoritmos Minimax **con** poda con profundidad 1 y reportar el tiempo promedio de toma de decisión de cada jugador.

Para los enfrentamientos debes utilizar la misma función de evaluación. Para eso elige una de las presentes en el archivo `score.py` y declararlo en tu respuesta. Considera un tablero de 9×9 para estos experimentos.

Luego repite este proceso para profundidad 2 y 3, con al menos 5 repeticiones. 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.
Por último, ejecuta una vez el algoritmo con profundidad 4 con y sin poda y reporta el tiempo promedio de ejecución (**Aviso:** puede tomar mucho tiempo en que termine la partida, en ese caso puedes reducir el tamaño del mapa a 7x7 e indicar aquello en tu respuesta). Luego comenta sobre este resultado.

Para comparar el desempeño de las IAs, se recomienda desactivar el visualizador utilizando

```
ui = UI(player1, player2, board_size=board_size, headless=True)
```

IMPORTANTE: En las siguientes preguntas utiliza la poda alfa-beta para todos los agentes. Si consideras que los agentes demoran mucho en jugar puedes reducir el tamaño del tablero en `main.py`, pero debes indicarlo en tu informe. Para la corrección se considerará como tiempo máximo de ejecución 10 minutos.

Actividad 2: Comparación profundidades (0.5 puntos)

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 `evaluate()`.

Ten en cuenta que el agente Minimax player1 juega primero

Configuraciones:

- Agente Minimax **player1** con profundidad 1, contra Agente Minimax **player2** con profundidad 1.
- Agente Minimax **player1** con profundidad 1, contra Agente Minimax **player2** con profundidad 2.
- Agente Minimax **player1** con profundidad 2, contra Agente Minimax **player2** con profundidad 1.
- Agente Minimax **player1** con profundidad 1, contra Agente Minimax **player2** con profundidad 3.
- Agente Minimax **player1** con profundidad 3, contra Agente Minimax **player2** con profundidad 1.

Nuevamente, para los enfrentamientos debes utilizar la misma función de evaluación. Para eso elige una de las presentes en el archivo `score.py` y declararlo en tu respuesta.

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 en la forma de juego y argumenta por qué, haciendo referencia al funcionamiento de Minimax.

Actividad 3: Comparación de funciones de evaluación (0.5 puntos)
En esta actividad, deberás comparar las dos funciones de evaluación que se te entregaron en el archivo `score.py`: `evaluate()` y `chat_gpt_eval()`. Para ello, se te pide que pruebes las siguientes configuraciones, nuevamente con 10 repeticiones cada una:

- Agente Minimax **player1** con `evaluate()` y profundidad 1, contra Agente Minimax **player2** con `chat_gpt_eval()` y profundidad 1.
- Agente Minimax **player1** con `chat_gpt_eval()` y profundidad 2, contra Agente Minimax **player2** con `evaluate()` y profundidad 2.
- Agente Minimax **player1** con `evaluate()` y profundidad 1, contra Agente Minimax **player2** con `chat_gpt_eval()` y profundidad 3.
- Agente Minimax **player1** con `evaluate()` y profundidad 3, contra Agente Minimax **player2** 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 (0.5 puntos)
Deberás implementar una función de evaluación en el archivo `score.py`, las cuales reciben un parámetro `game`, el cual representa el juego de la clase `Game`, y el parámetro `player_idx`, el cual representa el indice del jugador actual. Los indices impares corresponden al `player1` y los pares al `player2`.

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 50 % de las partidas a la función `chat_gpt_eval()`, con un mismo nivel de profundidad. Este porcentaje de victoria debe ser al menos sobre 10 partidas. Puedes seleccionar el orden en que comienza cada jugador como estimes conveniente, pero debes mencionarlo en tu respuesta. Reporta esto como una tabla en la que se incluyan las 10 partidas con sus resultados.

Qué entregar en esta parte
En un subdirectorio `DCCQuoridor/`:

- 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.