PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2024-1

# Tarea 3
# Problemas de Búsqueda

Fecha de entrega: lunes 20 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 lunes 20 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. Teoría (1.5 pts.)

El algoritmo *Rushed-A** es una variabte de A*. Su pseudocodigo contiene una línea adicional a A* y se presenta a continuación, donde la función Insertar está implementada de la misma manera que el tradicional A*.

**Algoritmo Rushed-A***

**Input:** Un problema de busqueda $(S, A, s_0, G)$
**Output:** Un nodo objetivo

**for each** $s \in S$ **do** $g(s) \leftarrow \infty$
$g(s_0) \leftarrow 0$; $f(s_0) \leftarrow h(s_0)$;
Open $\leftarrow \{s_0\}$
**while** Open $\neq \emptyset$
    Extrae un $u$ desde Open con menor valor-$f$
    **if** $u$ es objetivo **return** $u$ *// retorno tradicional*
    **for each** $v \in$ Succ$(u)$ **do**
        **if** $v$ es objetivo y $f(v) = f(u)$ **return** $v$ *// retorno adicional*
        Insertar $v$

# Actividad

1.1. (0.75 ptos.) Haz un argumento a favor de usar Rushed-A* en vez de la implementación típica de A*. Tu argumento debe aludir a la ventaja práctica de usarlo en vez de A*. Trata de que tu análisis sea lo más formal posible, refiriéndote a una situación hipotética en donde Rushed-A* podría generar una gran ventaja en cómputo respecto de A*.

1.2. (0.75 ptos.) Encuentra y describe una condición no trivial de $h$ que garantiza que Rushed-A* solo retorna soluciones óptimas. Presenta una completa demostración de que Rushed-A* es óptimo en estas condiciones. Si quieres usar, como parte de tu demostración, la transcripción de la demostración de este video (click acá) no hay problema, mientras entiendas lo que estás haciendo.

## 2. DCComilon (2.5 pts.)

Un día estas jugando Pacman en el Arcade de tu ciudad, y después de muchas horas evitando que los fantasmas te atrapen e intentando obtener la mayor cantidad de puntos, te acuerdas de la materia vista en las clases de Inteligencia Artificial. Eureka! Pacman no es mas que **un laberinto que puede ser navegado usando algoritmos de búsqueda**! Te das cuenta que con eso puedes ayudar a Pacman a llegar a de una posición a otra en el menor tiempo posible. Es por esto que decides probar algunos algoritmos de búsqueda para ver cual de ellos funciona mejor en ayudar a Pacman a llegar más rápido de un punto a otro (premiándolo con una gloriosa hamburguesa).

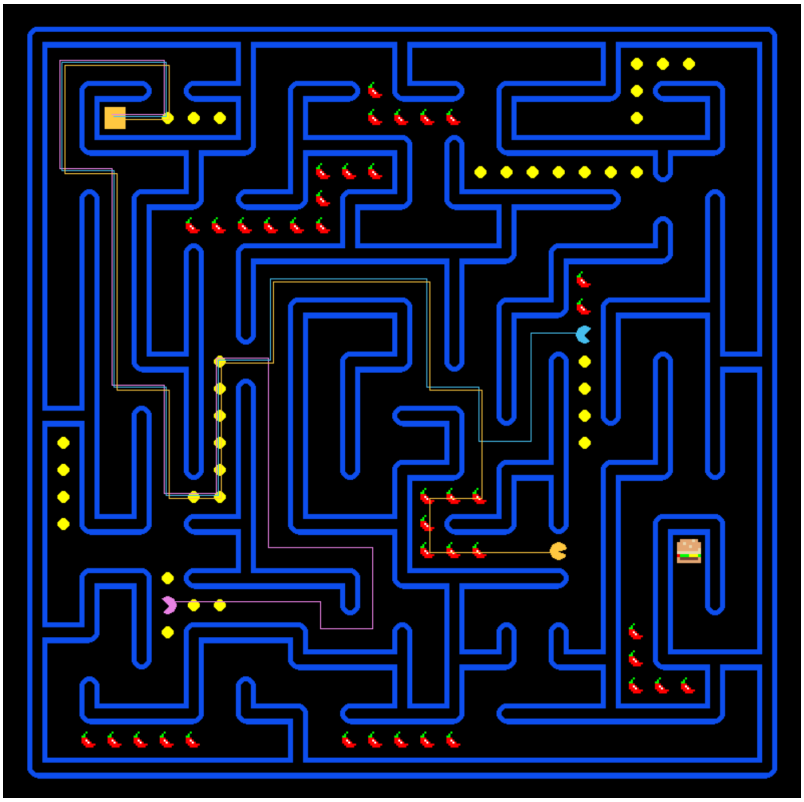

Figura 1: Imagen referencial del juego

En este problema deberás lograr que Pacman recorra un laberinto sin fantasmas **desde un comienzo designado hasta una meta establecida**, el laberinto cuenta con celdas especiales tales como la de ají o miel, que afectan positiva y negativamente al movimiento. Para ello, deberás **implementar algunos algoritmos de búsqueda** que permitan resolver el problema de qué recorrido seguir, y compararlos según a los resultados obtenidos y otras estadísticas como tiempo de ejecución o número de estados expandidos.

Con respecto a los tipos de celdas posibles, estos corresponden a:

- **Pasillo (P):** Celdas por las cuales Pacman si puede pasar (con costo 1).
- **Barrera (B):** Celdas por las cuales Pacman no puede pasar.
- **Ají (A):** Celdas que contienen ají, lo que hace que Pacman avance el doble de rápido al pasar por ellas en comparación a una celda normal (tienen costo 0.5).
- **Miel (M):** Celdas que tienen miel, lo que hace que Pacman se demore el doble en pasar por ellas en comparación a una celda normal (tienen costo 2).

## Archivos entregados

- `main.py`: Este archivo contiene la **lógica del laberinto**. Acá debes instanciar los algoritmos de búsqueda para que resuelvan el problema (ver código). Deberás ejecutarlo para poder visualizar el correcto funcionamiento de estos (especificando un mapa de búsqueda en la variable `MAP_PATH`).
- `search.py`: En este archivos **debes implementar los distintos algoritmos** que son solicitados a lo largo de esta pregunta.
- `tests/test_X.py`: Contiene un **mapa de juego** leido por el archivo `main.py`.
- `visualization.py`: Este archivos contienen los elementos para visualizar la navegación.
- `vis_utils.py`: En este archivos se contienen funciones auxiliares para visualizar la navegación.

Además se te entregaran 8 mapas en la carpeta `tests/`, los cuales tiene las siguientes características:

- Solo los mapas con número par cuentan con celdas de tipo `A` y `M`.
- Cada mapa cuenta con una sola celda de inicio y otra de comienzo

Para facilitar el desarrollo de tu tarea, se entrega una implementación del algoritmo DFS ya funcional dentro del archivo `search.py`, úsala como referencia para la implementación del resto de los algoritmos, reportando tiempo de ejecución, numero de expansiones, largo de camino encontrado y costo total de camino (ojo que retorna dos variables: el nodo objetivo y el camino para llegar a este **en reversa**, es decir, desde `G` hasta `S`).

## Actividades

### Actividad 1: Implementación BFS y DFS invertido (0.25 ptos.)

Para resolver esta actividad debes completar las funciones del archivo `search.py`, que implementan el algoritmo BFS estudiado en clases y un nuevo algoritmo al que llamaremos `inverted_DFS`, el cual se comporta de forma idéntica a DFS pero añade los sucesores (`succ`) en sentido inverso, es decir, los elementos en `succ` se encuentran en orden invertido (para llevar esto a cabo se recomienda utilizar la funcion `reversed` de Python).

Una vez implementados ambos algoritmos debes reportar en una tabla (para cada algoritmo, incluyendo DFS) el tiempo de búsqueda de este, el número de expansiones realizadas, el largo del camino encontrado y el costo del camino encontrado, para cada uno de los mapas entregados.

### Actividad 2.1: Implementación de A* (0.5 ptos.)

Para resolver esta actividad debes completar la función que implementa del algoritmo A* (como fue estudiado en clases) en el archivo `search.py`, puedes usar la estructura de datos que desees para representar a *Open*, pero recuerda que esta debe estar ordenada según el valor-$f$ de los nodos.

### Actividad 2.2: Implementación de heurísticas (0.5 ptos.)

Para resolver esta actividad debes completar el archivo `search.py`, con la implementación de las siguientes dos heurísticas:

- Distancia Manhattan dividida en 2 (la mitad de la distancia Manhattan tradicional)
- Distancia Euclidiana dividida en 2 (la mitad de la distancia Euclidiana tradicional)

Luego debes reportar en una tabla para cada heurística el tiempo de búsqueda de esta, el número de expansiones realizadas, el largo del camino encontrado y el costo del camino encontrado, para cada uno de los mapas entregados. Finalmente, argumenta, ¿por qué utilizamos la mitad del valor original de estas heurísticas? ¿Qué ocurriría para un algoritmo como A* si utilizamos el valor completo?

### Actividad 3: Implementación de Recursive Best-First Search (RBFS) (0.75 ptos.)

Para resolver esta actividad debes completar el archivo `search.py`, con la implementación del algoritmo RBFS, cuyo pseudocódigo se presenta a continuación:

RBFS($n$, $B$)
1. if $n$ is a goal
2. $\quad solution \leftarrow n$; exit()
3. $C \leftarrow expand(n)$
4. if $C$ is empty, return $\infty$
5. for each child $n_i$ in $C$
6. $\quad$ if $f(n) < F(n)$ then $F(n_i) \leftarrow max(F(n), f(n_i))$
7. $\quad$ else $F(n_i) \leftarrow f(n_i)$
8. $(n_1, n_2) \leftarrow best_{\mathrm{F}}(C)$
9. while ($F(n_1) \leq B$ and $F(n_1) < \infty$)
10. $\quad F(n_1) \leftarrow$ RBFS($n_1, min(B, F(n_2))$)
11. $\quad (n_1, n_2) \leftarrow best_{\mathrm{F}}(C)$
12. return $F(n_1)$

Figure 1: Pseudo-code for RBFS.

Para mas detalles al respecto al funcionamiento del algoritmo se recomienda fuertemente leer la publicación en cual fue incluido (https://www.cs.unh.edu/~ruml/papers/rbfscr-aaai-15.pdf). Te recomendamos que leas cuidadosamente la explicación de este algoritmo en el mismo paper, pues te ayudará a programarlo correctamente; el pseudocódigo no es enteramente claro si no lees la explicación.

Debes reportar en una tabla el tiempo de búsqueda de este algoritmo, el número de expansiones realizadas, el largo del camino encontrado y el costo del camino encontrado para cada uno de los mapas entregados. Finalmente, recuerda que el formato de salida de esta función debe ser idéntico al de las otras funciones de búsqueda, retornando dos elementos: el nodo objetivo y el *traceback* (camino encontrado) desde este al punto de inicio.

### Actividad 4: Comparación (0.5 ptos.)

Debes reportar en una tabla todos los algoritmos implementados (para A* deben estar sus dos heurísticas) junto con sus tiempos de búsqueda, los número de expansiones realizadas, el largo del camino encontrado y el costo del camino encontrado para cada uno de los mapas entregados.

Luego debes decidir cual de los algoritmos es mejor para este caso y porque, haciendo un análisis teórico y cualitativo en base a su comportamiento dentro del visualizador.

***Nota:*** *La corrección de los algoritmos sera llevada a cabo mediante test-cases, por lo que asegúrate que el retorno de las funciones siga el mismo formato que el algoritmo DFS entregado de referencia. Si la búsqueda puede ser visualizada correctamente, el formato de salida es correcto.*

## 3. DCCheckers (2 pts.)

En la Europa Medieval de la peste negra, un caballero sueco llamado Antonius Block, debe enfrentar en una partida de damas a la Muerte, quien ha venido a tomar su alma. Pero este no es un juego común; la Muerte, dotada de una inteligencia artificial temible, está decidida a llevarse a Antonius al reino de los perdidos.

El tablero está listo, con piezas de obsidiana y marfil brillando bajo la luz mortecina. Antonius, confiado en su habilidad en las damas, acepta el desafío, pero pronto descubre que la dama de la Muerte es una oponente formidable. Con cada movimiento, la IA de la Muerte calcula estrategias más allá de la comprensión humana, desafiando las leyes mismas del juego.

Antonius, sudando bajo su yelmo, lucha valientemente contra la inteligencia artificial y mientras la partida progresa, se da cuenta de que la Muerte no hace más que implementar el algoritmo minimax que él alguna vez vio en clases de IA por allá en el año 1350 y se dispone a recordar lo que sabe del algoritmo para posponer su muerte y realizar algún acto cuya ejecución le de sentido a su vida.

Figura 2: La Muerte y Antonius Block. El Séptimo Sello, película de 1957 (Ingmar Bergman).

## ¡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`, explicitando que tipo de jugador/algoritmo usara cada lado de juego en las variables `white_algorithm` y `black_algorithm`.

## Archivos entregados

- `main.py`: Este archivo inicia el juego, aquí puedes determinar si quieres que los jugadores sean controlados por ti o por la IA.
- `minimax/algorithm.py`: En este archivo deberás implementar el algoritmo minimax.
- La carpeta checkers contiene toda la lógica del juego y por ende no debes hacer cambio en ninguno de sus archivos.

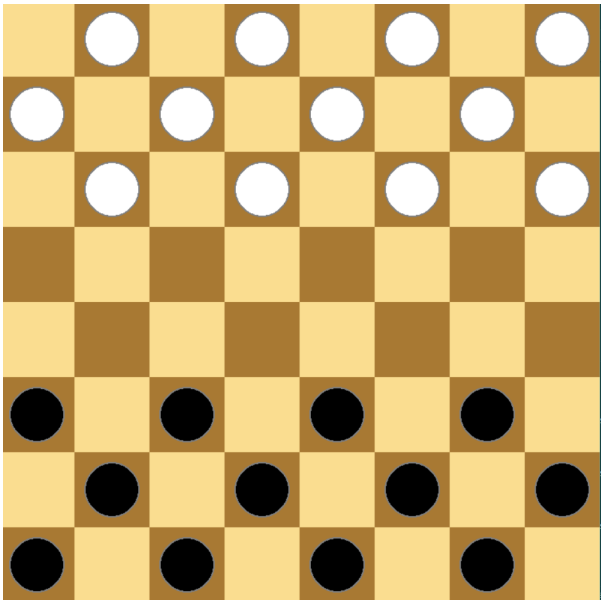

Figura 3: Imagen referencial del juego

## Actividades

### Actividad 1: Minimax (1 ptos)

En esta primera actividad, deberás completar la implementación del algoritmo Minimax que se encuentra en el archivo `algorithm.py`. Para ello, lee bien los comentarios incluídos en el código y sigue 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.

### Actividad 2: Monte Carlo Tree Search (0.25 ptos.)

En esta segunda actividad, deberás comparar el algoritmo de Minimax con otro algoritmo de búsqueda adversaria, llamado Monte Carlo Tree Search (MCTS), el cual se te entrega ya implementado en los archivos base. Deberás comparar el desempeño de MiniMax utilizando 3 profundidades distintas (se recomienda usar 1, 2 y 3) contra MCTS en 10 partidas distintas para cada profundidad, indicando el algoritmo ganador para cada una de ellas y presentando una tabla con los resultados promedio para cada profundidad.

Discute brevemente sobre como el cambio de profundidad se traduce en el desempeño del algoritmo y argumenta teóricamente a que se puede deber esto.

### Actividad 3: Profundidades distintas (0.25 ptos.)

Ahora, deberás comparar el desempeño de agentes que utilizan Minimax entre si en 3 partidas distintas, utilizando profundidades de búsqueda de 1, 3 y 5 (1v3, 3v5 y 5v1). Además, 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 brevemente por qué.

### Actividad 4: Poda Alfa-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, repite el experimento de la Actividad 3 pero utilizando la poda; ¿en que contribuye su implementación para el desempeño/eficiencia del algoritmo?