PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
Facultad de Ingeniería
IIC2613 Inteligencia Artificial
2024-2

Tarea 3
Problemas de Búsqueda

Fecha de entrega: miércoles 16 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. El lugar de entrega es en el repositorio de la tarea, en la branch por defecto, hasta el miércoles 16 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. Teoría (1.75 pts.)

El algoritmo *Early-A** es una variante de A*. Su pseudocódigo se presenta a continuación:

Algoritmo Early-A*

Input: Un problema de búsqueda (S, A, s_0, G)
Output: Un nodo objetivo

1. **for each** $s \in S$ **do** $g(s) \leftarrow \infty$
2. $g(s_0) \leftarrow 0$; $f(s_0) \leftarrow h(s_0)$;
3. $U \leftarrow \infty$;
4. Insertar s_0 a *Open*
5. **while** *Open* $\neq \emptyset$ **do**
6. Extrae un u desde Open con menor valor-f
7. **if** $U \leq f(u)$ **return** *sol*
8. **for each** $v \in Succ(u)$ **do**
9. $cost_v \leftarrow g(u) + c(u, v)$
10. **if** $cost_v \geq g(v)$ **then continue**
11. **if** v es objetivo *and* $g(v) < U$ **then**
12. $sol \leftarrow v$
13. $U \leftarrow g(v)$
14. $parent(v) \leftarrow u$
15. $g(v) \leftarrow cost_v$
16. $f(v) \leftarrow g(v) + h(v)$
17. **if** $f(v) \leq U$ **then**
18. **if** $v \notin$ *Open* **then**
19. Insertar v en *Open*
20. **else**
21. Ajustar *Open*

1.1. Preguntas

Actividad 1 (0.75 ptos.):

Haz un argumento a favor de usar Early-A* en vez de la implementación típica de A*. Tu argumento debe aludir a la ventaja práctica de usarlo en vez de A*.

Actividad 2 (1 pto.):

Demuestra que cuando h es admisible, entonces Early-A* encuentra una solución óptima. Si quieres usar, como parte de tu demostración, este video (click acá) no hay problema, mientras entiendas lo que estás haciendo.
Entrega tu solución a ambas actividades en un archivo llamado `respuesta_p1.pdf` en tu repositorio personal.

2. DCCarrera (2.25 pts.)

2.1. Contexto: La Carrera de los Autos Inteligentes

En un futuro cercano, un grupo de autos autónomos equipados con inteligencia artificial compite para demostrar cuál es la mejor estrategia para optimizar sus rutas. Los desarrolladores aún no están seguros de cuál heurística es más eficiente para navegar terrenos y curvas que afectan el tiempo y el consumo de combustible. En el circuito de prueba, los autos se enfrentan a diversos terrenos, desde caminos rápidos de hielo hasta otros caminos de arena que ralentizan. Además, las curvas pronunciadas aumentan el consumo si se derrapan, pero pueden ahorrar tiempo si se hacen bien. Cada auto debe decidir si prioriza rutas más largas y eficientes en combustible o caminos cortos y costosos. ¿Quién logrará el equilibrio perfecto entre velocidad, combustible y eficiencia?

Evaluaremos distintos modelos de conectividad del automóvil con respecto al terreno:

- Conectividad 4: El auto se moverá entre estados (x, y) que representan su posición el terreno en las cuatro direcciones principales Norte, Sur, Este y Oeste. Cada uno de los movimientos tendrá costo 1.
- Conectividad 8: El auto se moverá entre estados (x, y) que representan su posición el terreno en los 8 cuadrados adyacentes. Los movimientos en la cuatro direcciones principales Norte, Sur, Este y Oeste tienen costo 1 y las diagonales tienen costo $\sqrt{2}$.
- *State lattices*: El auto se moverá entre *state lattices*, las cuales se representan como una tupla (x, y, p) donde (x, y) son la posición en el terreno y p es una tupla (p_x, p_y) que denotara la pose hacia donde esta mirando el agente. Para hacer más realista el movimiento del automóvil, éste podrá ejecutar acciones que respetan restricciones de movimiento (usualmente llamadas *cinemáticas*) a las que vamos a llamar *primitivas*. Cada primitiva m es una tupla:

  $$(p_m^s, p_m^e, x_m, y_m, l_m, C_m)$$

  Donde:

  1. p_m^s: Pose inicial del movimiento
  2. p_m^e: Pose final del movimiento
  3. x_m: Movimiento relativo a la posición de partida en el eje x
  4. y_m: Movimiento relativo a la posición de partida en el eje y
  5. l_m: Longitud (costo) del movimiento.
  6. C_m: Lista de posiciones $(r_m^1, s_m^1), \ldots, (r_m^k, s_m^k)$, relativas a la posición de partida, que deben estar libres para poder realizar el movimiento.

  Una primitiva $P = (p_m^s, p_m^e, x_m, y_m, l_m, C_m)$ se puede ejecutar en una estado (x_s, y_s, p_s) si solo si:

  1. para cada $(r, s) \in C_m$, la celda $(x_s + r, y_s + s)$ están libres
  2. $p_s = p_m^s$

  Al ser aplicada, el estado resultante es $(x_s + x_m, y_s + y_m, p_m^e)$.

  Esta forma de definir espacios de estados es muy expresiva. Para simplificar nuestra tarea, solo utilizaremos las siguientes poses: $(1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1)$.

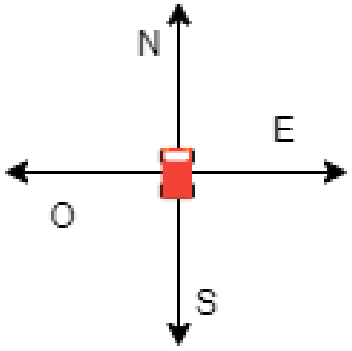


Figura 1: Conectividad 4

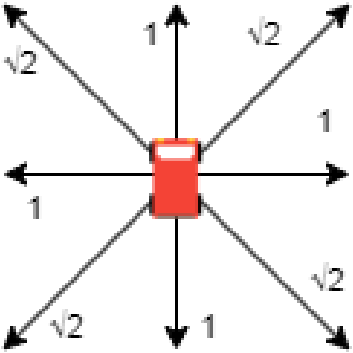


Figura 2: Conectividad 8

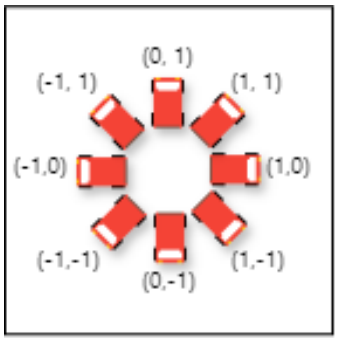


Figura 3: Posibles poses

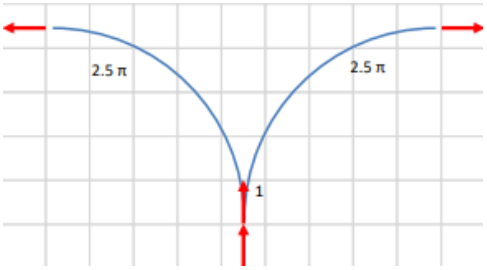


Figura 4: Ejemplo tres de las primitivas

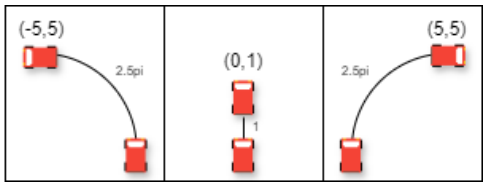


Figura 5: Posición y longitudes del movimiento state lattices

Se te entregará implementada *Conectividad 4* y parte de *State lattices* como base. Como parte del desarrollo de la tarea se te pedirá que implementes *Conectividad 8* y la parte faltante de *State lattices*.

Se te entrega una proyecto con una implementación de A^* (`astar.py`). El objetivo será ejecutar el comando **`python test_map.py`** en tu terminal, donde se realizará la búsqueda y recibirás un archivo tipo `.json` (por defecto se guarda como `output.json`) donde podrás utilizar una página para visualizar (`Visualizador/main.html`). Con el visualizador podras selecionar un mapa (`.map`) y una ruta (`.json`) e ir probando la velocidad de animación del auto.

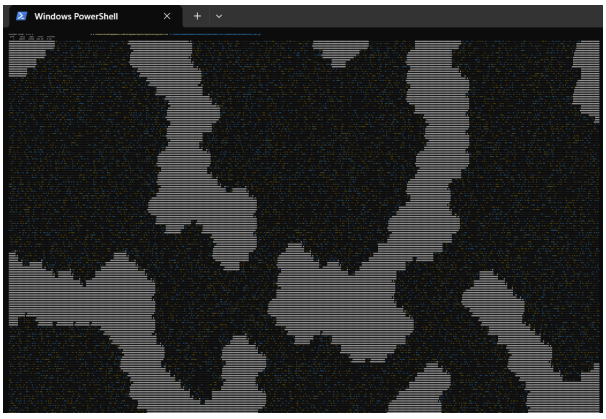


Figura 6: Ejemplo test starcraftObstacles.map ejecutando test_map.py

Figura 7: Ejemplo test starcraftObstacles.map con la página main.html

Explicación Código

A continuación se entrega una explicación de los archivos y códigos que se te entregaran:

- **map.py:** Este archivo contiene gran parte de la implementación del problema. Dentro de el se encuentra la función `set_connectivity` la cual deberás modificar para implementar *Conectivdad 8* y *State Lattices*. A su vez, se encuentran las heurísticas que debes completar. Estas corresponden a:
  - **octile:** Calcula la distancia mínima entre dos puntos en una cuadrícula, considerando movimientos en las ocho direcciones (horizontales, verticales y diagonales). **Modificar**.
  - **manhattan:** Calcula la distancia mínima entre dos puntos moviéndose solo en direcciones ortogonales (horizontales y verticales), sin considerar diagonales. **Modificar**.
  - **euclidian:** Calcula la distancia recta entre dos puntos en el espacio utilizando la fórmula de la distancia euclidiana. **Modificar**.
- **cell.py:** Archivo que define a una celda/casilla del mapa para Conectividad 4 y Conectividad 8. **No modificar**.
- **lcell.py:** Archivo que define a una celda/casilla del mapa para *State Lattices*. **Modificar**.
- **node.py:** Archivo que define a un nodo. **No modificar**.
- **astar.py:** Contiene la implementación del algoritmo A^*, donde retorna si llego a una solución (un binary heap) o no (retorno nulo). **No Modificar**.
- **early_astar.py:** Contiene la implementación incompleta del algoritmo *Early-A** mostrado en la pregunta anterior. Este retorna la solución si es que llegó a ella (un binary heap) o no (retorno nulo). **Modificar**.
- **binary_heap.py:** Archivo de la estructura de datos a utilizar. **No modificar**.
- **json_output.py:** Archivo que da estructura a los archivos **json** a generar. **No modificar**.
- **test_map.py:** Es el archivo es el que debes ejecutar para probar tu programa. El terminal visualiza el mapa y la ruta de solución de la implementación elegida. Además al ejecutar el archivo se va generar un **json** que codifica la ruta que genero la implementación elegida. Tienes las siguientes variables que puedes cambiar para testear tu código:
  - **epsilon**: Debes analizarla y explicarla en la actividad 6.
  - **m**: Elección de archivo **.map** para testear. Ejemplo: `Maps/starcraft.map`.
  - **connectivity**: Define las conectividades. Puede ser: `int(4)`, `int(8)`, `"primitiveSimpleRight"`, `"primitiveSimpleDiagonal"`, `"primitiveFull"`. Valor por defecto: `"primitiveFull"`.
  - **primitivasActivated**: Booleano que indica si se está utilizando alguna primitiva. Debe ser `True` para: `"primitiveSimpleRight"`, `"primitiveSimpleDiagonal"`, `"primitiveFull"`.
  - **heuristic**: Heurística a testear. Ejemplo: `euclidian_heuristic`.
  - **weight**: Peso que se usa en la heurística de A*.
  - **visualize**: Booleano que indica si se debe visualizar o no la solución en el terminal.
  - **num_probs**: Número de problemas que se ejecutan.
- **Visualizador/:** La carpeta contiene los archivos para implementar la animación. Se puede seleccionar un mapa (`.map`) y varias rutas de los autos (archivos `.json`), y se veran autos de distintos colores para comparar las distintas rutas. El archivo `main.html` es la página principal de este visualizador. **No modificar**
- **Maps/:** La carpeta contiene los archivos de varios mapas (`.map`) que puedes probar. **No modificar**

2.2. Preguntas

Actividad 1: Implementación de las heurísticas (0.25 ptos.)

Para poder ejecutar `A*` lo primero que debemos hacer es completar las heurísticas `manhattan`, `euclidian` y `octile` del archivo `map.py`.
Luego de implementarlas, haz una comparación de las tres heurísticas, junto con usar la heurística `zero_heuristic`. Haz esto eligiendo 5 mapas de la carpeta `Maps/` según estimes convenientes. Para esto utiliza el archivo `test_map.py` y para cada mapa ejecuta 10 problemas, lo que puedes hacer cambiando la variable `num_probs`. Para cada una de las heurísticas reporta el número de expansiones (nodos abiertos y nodos generados), el tiempo de computo y el costo del camino obtenido. Para esto presenta una tabla que contenga las ejecuciones con los resultados.
Por último, indica cual es la heurística que mejor funciona y explica a que puede deberse aquello. Recuerda mencionar porque funciona mejor que las otras.

Actividad 2: Implementación *Early-A** y comparación con *A** (0.5 ptos.)

Ahora deberás completar el archivo `early_astar.py` para poder ejecutar dicho algoritmo. En dicho archivo deberás completar el método `search` de la clase `EarlyAstar` siguiendo el pseudocódigo de la pregunta anterior (1). Para eso se te entrega la estructura de datos `binary_heap.py` junto a `node.py` para que puedas realizar una implementación óptima. Además, verás que se dejaron los atributos `star_time` y `end_time` para que computen el tiempo de búsqueda.
Cuando hayas implementado el algoritmo, elige nuevamente 5 mapas de la carpeta `Maps/` según estimes convenientes para comparar *Early-A** con *A** y utiliza la mejor heurística obtenida en la actividad anterior para ambos algoritmos. Para esto utiliza el archivo `test_map.py` y para cada mapa ejecuta 10 problemas, lo que puedes hacer cambiando la variable `num_probs`. Reporta el número de expansiones (nodos abiertos y nodos generados), el tiempo de computo y el costo del camino obtenido por los algoritmos en los mapas. Para esto presenta una tabla que contenga las ejecuciones.
Por último comenta sobre cual de los dos algoritmos es mejor y a que se puede deber esto. Puedes utilizar los argumentos utilizados en la pregunta 1 si estimas conveniente.

Actividad 3: Implementación de Conectividad 8 (0.25 ptos.)

A continuación deberás implementar Conectividad 8. Para esto en el archivo `map.py` deberás modificar el método `set_connectivity` de la clase `Map`.
Ahora debes comprar Conectividad 4 y Conectividad 8. Para esto puedes usar tanto *Early-A** como *A**, y utiliza la mejor heurística obtenida anteriormente. Luego escoge según tu criterio 5 mapas de la carpeta `Maps/` y ejecuta el archivo `test_map.py`. Para cada mapa ejecuta 10 problemas, lo que puedes hacer cambiando la variable `num_probs`. Reporta una tabla con el tiempo, costo y nodos generados.
Por último comenta sobre los resultados obtenidos. ¿Cómo afecta agregar los nuevos movimientos de Conectividad 8 a los caminos obtenidos? ¿A qué se debe esto? Puedes utilizar imágenes o lo que estimes conveniente.

Actividad 4: Implementación de las primitivas de *State Lattices* (0.5 ptos.)

Es momento de probar las famosas `state lattices` junto a sus primitivas. Se te otorgan 3 conjuntos distintos de primitivas:

1. `primitiveSimpleRight`: Las primitivas de ángulos rectos presentadas en la Figura 1 en cambios de pose de $\pi/2$ junto al movimiento hacia adelante. El costo del giro es 2.5π y el movimiento recto 1.

2. `primitiveSimpleDiagonal`: Contiene el movimiento hacia adelante y un giro en $\pi/4$ que nos cambia de pose a las diagonales. El costo de giro es de aproximadamente 4.888, el movimiento recto es de coste 1 y el diagonal $\sqrt{2}$

3. `primitiveFull`: Contiene todos los movimientos de `primitiveSimpleRight`, así como los movimientos de `primitiveSimpleDiagonal`.

Para que esto funcione, debes implementar las siguientes primitivas:

1. `primitiveStraight`: Esto permite que se hagan movimientos rectos.
2. `primitiveDiagonal`: Esto permite que se hagan movimientos en diagonal.

Luego responde a las siguientes preguntas:
¿Cuantos vecinos tiene que revisar `successors` para cada conjunto? ¿Por qué los costos de los movimientos de giro cuestan 2.5π y 4.888?.

Actividad 5: Completar implementación de *State Lattices* en `lcell.py` (0.5 ptos.)

A continuación deberás implementar la función `succesors` del archivo `lcell.py` que corresponde a la última parte de la implementación de los *State Lattices*.
En el se presenta una clase `LCell` que recibe el estado de la celda y una clase `Map` que contiene toda la información necesaria de los vecinos. Debes retornar una lista de los sucesores válidos, por lo que tienes que revisar que el movimiento se pueda realizar con las funciones integradas de `Map`, es decir, `inside` y `line_of_sight`, que revisan si el movimiento esta dentro de los límites válidos y si las celdas que estamos apuntado están disponibles. Además debes buscar si el movimiento se realiza a algún área de hielo o arena, y de ser así suma el coste `sandCost` o `iceCost` al calculo de costo.
Recuerda que para la implementación en `lcell.py` debes respetar las poses, verificar que toda la caminata está disponible y evaluar el terreno para ver si hay hielo o arena. (Por si no quedo claro, por cada casilla de arena o hielo simplemente sumar una vez el coste, ejemplo en el camino hay 2 hielos y una arena, entonces al costo debes sumarle 2 veces el costo de hielo y 1 vez el de arena).
Ahora debes comprar *State Lattices* con Conectividad 4 y Conectividad 8. Para esto ejecuta el mismo algoritmo (*Early-A** o *A**) con la misma heurística en los mismos mapas que en la actividad 3 y utiliza el archivo `test_map.py`, para cada uno de los conjuntos de primitivas de *State Lattices*. Para cada mapa ejecuta 5 problemas, lo que puedes hacer cambiando la variable `num_probs`. Si tuviste problemas con algún mapa repórtalo y explica cual podría ser el problema. Agrega a la tabla anterior los resultados obtenidos con cada conjunto de primitivas de *State Lattices*.
Comenta sobre los resultados obtenidos. ¿Cómo afecta agregar los nuevos movimientos de *State Lattices* a los caminos obtenidos? ¿Cómo afectan los terrenos diferentes terrenos a la selección de caminos? Puedes utilizar imágenes o lo que estimes conveniente.

Actividad 6: Variable `epsilon` (0.25 ptos.)

Te podrás haber fijado que hay una variable `epsilon` en el archivo `map.py`. Explica que es lo que hace dicho parámetro y cual es el efecto que puede tener en las rutas. Para esto, prueba variando `epsilon` en mapas en que funcionaron los conjuntos de primitivas y explica cual es el comportamiento que empieza a tener el auto. También se recomienda utilizar el visualizador simple desde la terminal para visualizar los nodos expandidos y lograr entender el comportamientos de las curvas (el visualizador oficial lo que hace es saltar de nodo en nodo por lo que no se aprecian bien las curvas).

3. DCChain Reaction (2 pts.)

3.1. Introducción

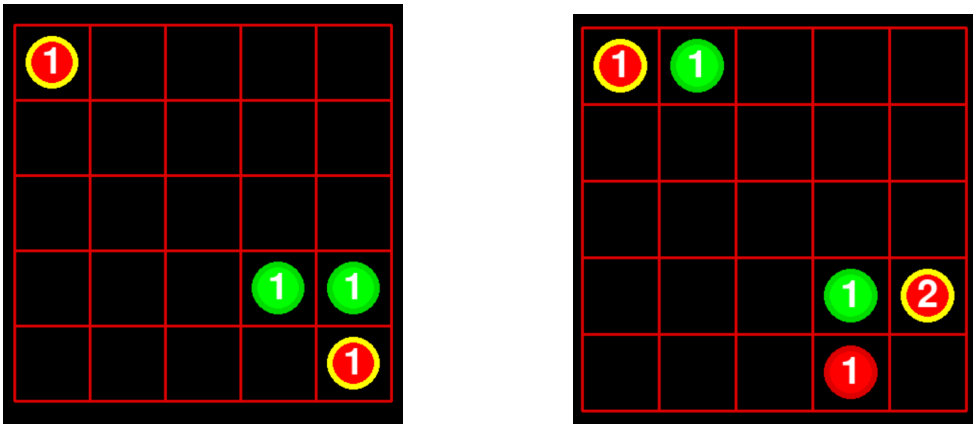
En el año 2087, el mundo enfrenta la devastación provocada por el virus Baier, una pandemia que ha diezmado la población global y que sigue extendiéndose sin control. Como último recurso, tú, un renombrado químico y una de las pocas grandes mentes sobrevivientes, has descubierto una posible cura. Sin embargo, para que esta cura sea efectiva, debe ser distribuida rápidamente a través de todas las células infectadas antes de que el virus mute y se vuelva incontrolable.

El método para administrar la cura está inspirado en un tablero de nanotecnología del juego DCChain Reaction. Debes colocar pequeñas cápsulas con anticuerpos en cada celda del tablero. Cada celda tiene un límite de capacidad, y si se sobrepasa, las cápsulas explotan, expandiendo la cura a las celdas cercanas. Tu objetivo es desencadenar explosiones estratégicas para maximizar la difusión de la cura y propagarla lo más rápido posible por todo el tablero.

Mientras tanto, el virus se adapta a cada movimiento, creando obstáculos que dificultan la propagación de la cura. Un error podría provocar una explosión en el momento equivocado, acelerando la mutación del virus. Deberás planificar cuidadosamente cada jugada para erradicar el virus antes de que sea demasiado tarde.

3.1.1. Explicación Juego

DCChain Reaction[1] consta de un tablero con $N \times M$ celdas, donde los jugadores colocan círculos en las celdas vacías. Cada jugador puede colocar una circulo de su color en un celda, y cuando el número de círculos en una celda excede su capacidad, los círculos explotan y se distribuyen en las celdas vecinas. Si una celda adyacente contiene a un circulo enemigo, al explotar, el circulo enemigo, con su valor, pasarán a ser del jugador (Ver Figuras 8a y 8b).

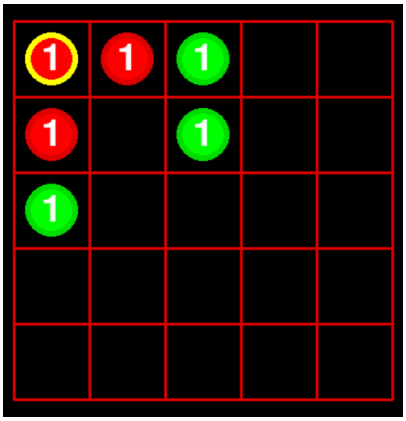


(a) (b)

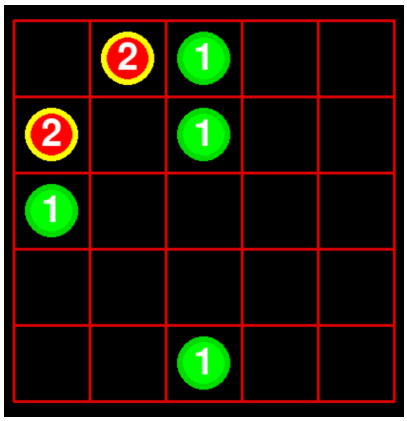
Figura 8: Ejemplo de Explosión en donde el jugador rojo ubica una círculo en la esquina inferior derecha

Esta explosión puede desencadenar una reacción en cadena si las celdas adyacentes también alcanzan su capacidad máxima. Esto se puede observar en las Figuras 9a y 9b, donde al poner un circulo rojo en la esquina superior, este circulo explota, aumentando el valor de los círculos vecinos.

[1] Video ejemplo del juego: https://youtu.be/DqFlEBObg1Q



(a) Circulo rojo por explotar



(b) Circulo rojo explotado

Figura 9

El objetivo del juego es lograr que todas las celdas del tablero estén controladas por los círculos de un solo jugador, eliminando los círculos de los oponentes mediante explosiones estratégicas.
El máximo de capacidad de cada celda esta determinado por la cantidad de celdas adyacentes. Por ejemplo, una celda en una esquina, al tener dos celdas adyacentes, el circulo explota cuando tiene valor 2. En la Figura 10 se pueden observar círculos en capacidad máxima (aquellos que tienen borde amarillo).

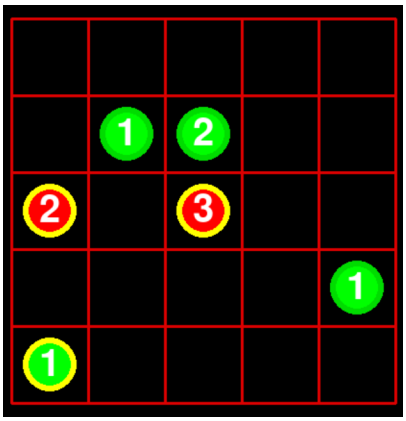


Figura 10: Círculos a punto de explotar

3.1.2. Explicación Código

A continuación se entrega una explicación de los archivos y códigos que se te entregaran:

- **Game/Front.py:** Contiene las funciones para implementar la animación. **No modificar**.
- **Game/Board.py:** Contiene funciones para manejar y realizar cambios en el tablero. **No modificar**.
- **Game/Circle.py:** Contiene la clase Circle, la cual controla y modela los atributos de los circulos en el tablero. **No modificar**.
- **Game/Player.py:** Contiene la clase Player, la cual contiene los atributos del jugador. **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:
  - ANIMACION: Si su valor es True, se mostrará el visualizador en pantalla. Este por defecto será True si el jugador es HUMANO.
  - DEPTH_J1 y DEPTH_J2: Determinan la profundidad de búsqueda de cada jugador. Estas variables son utilizadas al instanciar a los jugadores.
  - ALPHABETA: Si su valor es True, se implementa la poda Alpha-Beta en el Minimax.
  - JUGADOR_1 y JUGADOR_2: Instancias de la clase Player. Contiene los siguientes atributos:
    - Tipo de Jugador: Los jugadores pueden ser HUMANO, MINIMAX o RANDOM. **Modificar** para poder evaluar diferentes situaciones.
    - Color: El color de las fichas del jugador. No es necesario modificarlo.
    - ID: El ID del jugador. Siempre tienen que ser 0 y 1. **No modificar**
    - DEPTH: Determina la profundidad de búsqueda del jugador para Minimax. **Modificar** para evaluar diferentes profundidades.
    - Función de Evaluación: Determina la función de evaluación a utilizar por el jugador en Minimax. **Modificar** para utilizar diferentes funciones de evaluación.
- **minimax.py:** Contiene una implementación del algoritmo Minimax. **Modificar**.
- **score.py:** En este archivo se definirán las funciones de evaluación que utilizarán tus implementaciones de Minimax. **Modificar**. Este archivo contiene las siguientes funciones de evaluación:
  - **score_circle_amount:** Premia la cantidad de círculos del jugador en el tablero
  - **score_circle_values:** Premia los valores de cada circulo del jugador en el tablero.

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

3.2. Preguntas

Actividad 1: 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 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. Por ultimo 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 usar tableros no muy grandes (5x5 esta bien), para que no se tarde tanto el código. Además, pueden desactivar el visualizador con el parámetro `ANIMACION = FALSE`.

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 **score_circle_values** (es el parámetro `eval_func` de la clase **Player**).
Configuraciones:
Se recomienda usar un tablero de 5x5 y tenga en cuenta que el agente Minimax Rojo juega primero

- Agente Minimax **Rojo** con profundidad 1, contra Agente Minimax **Verde** con profundidad 1.
- Agente Minimax **Rojo** con profundidad 1, contra Agente Minimax **Verde** con profundidad 2.
- Agente Minimax **Rojo** con profundidad 2, contra Agente Minimax **Verde** con profundidad 1.
- Agente Minimax **Rojo** con profundidad 1, contra Agente Minimax **Verde** con profundidad 3.
- Agente Minimax **Rojo** con profundidad 3, contra Agente Minimax **Verde** 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 funciones de valor entregadas (0.5 ptos.)

En esta actividad debes comparar las dos funciones de evaluación entregadas en el código base, es decir, *score_circle_amount* y *score_circle_values*. Para esto, prueba las siguientes configuraciones:
Se recomienda usar un tablero de 5x5 y tenga en cuenta que el agente Minimax Rojo juega primero

- Agente Minimax **Rojo** con *score_circle_amount* con profundidad 1, contra Agente Minimax **Verde** con *score_circle_values* con profundidad 1.
- Agente Minimax **Rojo** con *score_circle_values* con profundidad 1, contra Agente Minimax **Verde** con *score_circle_amount* con profundidad 1.
- Agente Minimax **Rojo** con *score_circle_amount* con profundidad 1, contra Agente Minimax **Verde** con *score_circle_values* con profundidad 3.
- Agente Minimax **Rojo** con *score_circle_amount* con profundidad 3, contra Agente Minimax **Verde** con *score_circle_values* con profundidad 1.

Deberás reportar el ganador y el tiempo promedio para una toma de decisión de cada profundidad. Indica cual de las funciones de evaluación es mejor según tus resultados, y explica a que se puede deber esto.

Actividad 4: Función de valor (0.5 ptos.)

Deberás implementar una función de evaluación en el archivo *score.py*, las cuales reciben un parámetro *circles*, el cual representa al tablero, siendo una lista de listas de objetos de la clase *Circle*. Cada objeto tiene los atributos de posición (x, y), el valor que posee (*number*), la capacidad máxima, y el *id* del jugador al que pertenecen. También reciben un parámetro *JUGADOR*, el cual es un objeto de la clase *Player*, 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 *score_circle_amount*, 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 Bonus (0.5 ptos.)

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 la función de evaluación *score_circle_amount*, con un nivel menos de profundidad (por ejemplo, la nueva función de evaluación con profundidad 1 y *score_circle_amount* 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 el agente con *score_circle_amount*. Este porcentaje de victoria debe ser al menos sobre 10 partidas. Se revisará que esto se cumpla.