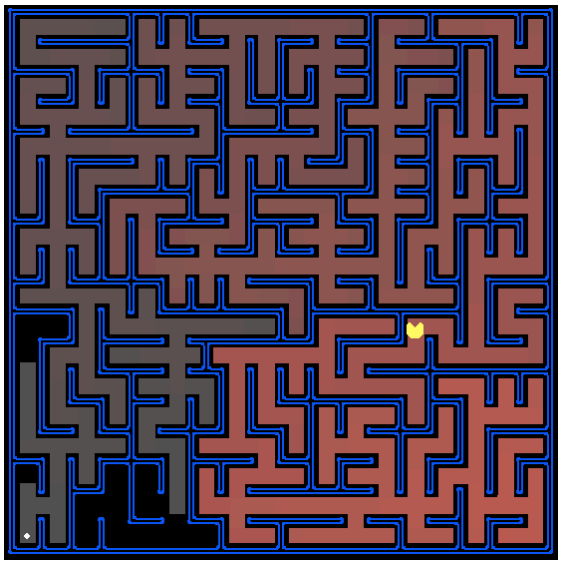
Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados



Introducción

En este proyecto utilizaremos un simulador del juego Pac-Man™ como plataforma para desarrollar nuestra propia representación de un espacio de estados e implementar algoritmos de búsqueda. En nuestro caso utilizaremos una versión simplificada del problema, en el que sólo queremos que Pacman encuentre el camino en el laberinto hacia uno o más elementos, de forma eficiente u óptima en algunos casos.

En las próximas secciones introduciremos el mundo de Pac-Man, sus archivos, qué cosas hay que tocar y qué cosas no.
Se pide entregar el código implementando las soluciones pedidas, así como un breve informe detallando los resultados obtenidos, analizando las soluciones propuestas y/o explorando distintas soluciones a lo pedido.

El proyecto está fuertemente basado en el "Berkeley Pac-Man Project" originalmente propuestos por John DeNero y Dan Klein para la Berkeley University. Procure no utilizar soluciones provenientes de internet, ya que no se tolerarán plagios.

Bienvenidos a Pac-Man

El entorno que utilizaremos corre en el intérprete de Python3, el mismo se puede descargar desde https://www.python.org/downloads/. Cualquier version superior a Python3.8 debería funcionar (versión por defecto en Ubuntu 20.04). Además puede descargar un IDE (integrated development environment) para Python3 si así lo desea:

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

- IDLE (desarrollado por la fundación Python)
- Visual Studio Code (o su versión abierta VSCodium)
- PyCharm
- etc…

Después de descargar el código (search.zip), puede descomprimirlo con cualquier herramienta que trabaje con .zip, y obtendrá un directorio llamado *search*, debería poder jugar a una versión simple de Pac-Man ejecutando el siguiente comando en dicho directorio:

```
python3 pacman.py
```

Como verá, Pacman vive en un mundo azul brillante formado por pasillos y pastillas. Recorrer este mundo eficientemente será el primer paso de Pacman.

Agentes

El objetivo de Pacman en primera instancia es poder recorrer los laberintos. Para ésto utilizaremos distintos tipos de agentes. El agente más simple que se encuentra implementado en *searchAgents.py* es el llamado GoWestAgent, el cual siempre se mueve hacia el Oeste (un agente trivial). Este agente es útil en laberintos sencillos, por ejemplo:

```
python3 pacman.py --layout testMaze --pacman GoWestAgent
```

pero no nos es útil si en el laberinto se requiere doblar:

```
python3 pacman.py --layout tinyMaze --pacman GoWestAgent
```

> **Nota:** si se quiere salir del juego se puede presionar el boton de cerrar ventana, o a veces se requiere realizar el comando Ctrl+C en la terminal en la que se estaba ejecutando.

Nuestros agentes no sólo podrán resolver el laberinto tinyMaze, sino que deberían poder resolver cualquier laberinto que se les presente (que tenga una solución posible).
Es importante notar que *pacman.py* soporta algunas opciones que pueden ser comandadas en formato largo (e.g. –layout) así como en formato corto (e.g. -l) las cuales se pueden ver imprimiendo la ayuda en pantalla con el comando:

```
python3 pacman.py -h
```

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

También proveemos todos los comandos que aparecen en este proyecto, así como otros comandos útiles, en *commands.txt* para facilitar el copiado y pegado.

Archivos

Se pedirá editar y entregar únicamente los siguientes archivos:

- *search.py* : aquí implementará los algoritmos de búsqueda que requiera para que pacman pueda recorrer los laberintos,
- *searchAgents.py* : la información que reciben y transmiten los agentes debe estar definida aquí, así como sus formas de interaccionar con el entorno.

Les será útil revisar los siguientes archivos para comprender el funcionamiento del entorno de juego de Pac-Man:

- *pacman.py* : es el archivo principal, que interpreta los comandos y corre el entorno visual. En el mismo se describe un estado de tipo GameState que utilizará durante el proyecto para comprender el estado del juego,
- *game.py* : en este archivo se describe la lógica tras el funcionamiento de Pacman. Les será útil leer las definiciones de AgentState, Agent, Direction y Grid, en que se describe el comportamiento y los gráficos de dichos elementos.
- *util.py* : este archivo contiene estructuras de datos útiles para implementar los algoritmos de búsqueda, así como también otras funciones predefinidas. Es muy recomendable darle un vistazo antes de comenzar a trabajar.

Los siguientes archivos pueden ser ignorados para este trabajo:

- *graphicsDisplay.py* : gráficos para Pac-Man,
- *graphicsUtils.py* : funciones útiles para mostrar gráficos,
- *textDisplay.py* : gráficos ASCII para Pac-Man.,
- *ghostAgents.py* : agentes para controlar los fantasmas,
- *keyboardAgents.py* : agente mediante interfaz de teclado,
- *layout.py* : primitivas para leer y almacenar los laberintos de pacman

¿Qué enviar? Se deberán enviar únicamente los archivos *search.py* y *searchAgents.py* que contienen las implementaciones realizadas. Además de el archivo de informe que, en lo posible, debe ser en formato PDF. Por favor no cambiar los nombres de las funciones provistas o cualquiera de las clases del código suministrado. Puede realizar copias o implementaciones por fuera de las mismas.

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

Encontrando comida en lugares fijos utilizando algoritmos de búsqueda

En *searchAgents.py* encontrarán una implementación de un SearchAgent, el cual planificará el camino de Pacman a través de los laberintos, y luego los ejecutará paso a paso. Los algoritmos de búsqueda para ejecutar un plan no están implementados (este es su trabajo).
Ante todo es importante verificar que SearchAgent está trabajando correctamente, esto lo podemos hacer ejecutando:

```
python3 pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

El commando anterior le dice al SearchAgent que use tiniMazeSearch, el cual está implementado en *search.py*, como su algoritmo de búsqueda para el laberinto tinyMaze. Pacman debería poder navegar el laberinto exitosamente.
Ahora hay que escribir las funciones de búsqueda genéricas para ayudar a Pacman a planificar sus rutas. El pseudocódigo de los algoritmos de búsqueda los pueden encontrar en las transparencias y los libros recomendados por la cátedra. Recuerden que un nodo de búsqueda debe contener no solo un estado sino también la información necesaria para reconstruir el camino que lo llevó al estado dado.

> **Nota Importante:** todas las funciones de búsqueda deben retornar una lista de **acciones** que llevan al agente desde el inicio hasta el objetivo. Estas acciones deben ser movimientos válidos (Pacman no se puede mover a través de las paredes del laberinto o salirse del mismo).
>
> **Nota:** Asegúrese de chequear los tipos Stack, Queue y PriorityQueue que están dados en util.py

Ejercicio 1: Implemente el algoritmo de búsqueda DFS (Depth-First Search) en la función llamada depthFirstSearch del archivo *search.py* (recuerde que la versión completa de DFS evita expandir estados ya visitados).

El código debería rápidamente encontrar una solución para los siguientes casos:

```
python3 pacman.py -l tinyMaze -p SearchAgent
```

```
python3 pacman.py -l mediumMaze -p SearchAgent
```

```
python3 pacman.py -l bigMaze -z .5 -p SearchAgent
```

El tablero de Pac-Man puede mostrar además una capa de color con los estados explorados y el orden en el que fueron explorados (cuanto más rojo, más temprano fue explorado). Pregúntese lo siguiente: ¿es el orden

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

de exploración el esperado? ¿Tiene Pacman realmente que ir por todas las casillas exploradas en su camino a la meta?

Ejercicio 2: Implemente el algoritmo BFS (Breadth-First Search) en la función llamada breadthFirstSearch en el archivo *search.py*. Testee su código de la misma forma que lo hizo para el DFS.

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

> **Nota:** si Pacman se mueve muy despacio puede usar la opción –frameTime 0

Variando la function costo

Mientras que BFS encontrará el camino con la menor cantidad de acciones que nos lleven al objetivo, a veces necesitamos encontrar caminos “mejores” en otros sentidos. Por ejemplo analice el problema de correr BFS en los mapas mediumDottedMaze y mediumScaryMaze:

```
python3 pacman.py -l mediumDottedMaze [... opciones]
```

```
python3 pacman.py -l mediumScaryMaze [... opciones]
```

Si cambiamos la función de costo podemos llevar a Pacman por distintos caminos. Por ejemplo podríamos decir que sean mos costosos los caminos en áreas de fantasmas, o más baratos en áreas ricas en comida, y el agente Pacman debería ajustar su comportamiento respondiendo a estos cambios.

Ejercicio 3: Implemente el algoritmo de búsqueda UCS (Uniform Cost Search) en la función uniformCostSearch del archivo *search.py*. Una vez implementado observe la conducta del agente Pacman en los tres laberintos que se presentan a continuación, donde se modifica la función de costo que utiliza el agente (los agentes y las funciones de costo deben ser implementadas también):

```
python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python3 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python3 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

Nota: debería ver caminos de costos muy altos y muy bajos para el StayEastSearchAgent y StayWestSearchAgent respectivamente, debido a su función de costo exponencial (leer *searchAgents.py* para más detalles)

Búsqueda A*

Ejercicio 4: Implemente la búsqueda mediante el algoritmo A* en la función aStarSearch del archivo *search.py*. Tenga en cuenta que A* toma una función heurística como argumento. Las heurísticas también deben ser definidas, para ésto deben tomar dos argumentos: un estado en el problema de búsqueda (el argumento principal), y el problema en sí mismo (para poder extraer información de referencia). La función heurística nullHeuristic que se encuentra en *search.py* es un ejemplo trivial de la misma.

Una vez implementado A* puede verificar su implementación en el problema original de encontrar un camino a través del laberinto hasta una posición fija dada usando la heurística de distancia de Manhattan, que ya está implementada como manhattanHeuristic en *searchAgents.py*:

```
python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

Deberían ver que A* encuentra la solución óptima un poco más rápido que UCS (una diferencia de ~100 nodos expandidos para una implementación usual, puede diferir ligeramente según las implementaciones de ambos). ¿Qué sucede con el laberinto “openMaze” para distintas estrategias de búsqueda? Analize y detalle en el informe.

Encontrando todas las esquinas

El poder real de A* solo se verá con un problema de búsqueda más desafiante. Ahora es tiempo de formular un nuevo problema y diseñar una heurística para él.

En este caso plantearemos el problema en que Pacman comienza en una posición arbitraria del laberinto, con 4 pastillas, una en cada esquina. Nuestro objetivo es que Pacman encuentre el camino más corto a través del laberinto que toque las cuatro esquinas, y coma dichas pastillas. Noten que en algunos laberintos (como tinyCorners) el camino más corto no siempre es el que pasa por la comida mas cercana primero, por ejemplo en el caso de tinyCorners el camino más corto toma 28 pasos.

Ejercicio 5: Implemente el problema de búsqueda CornersProblem en el archivo *searchAgents.py*. Necesitará elegir una representación de estados que codifique toda la información necesaria para detectar cuál de las 4

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

esquinas ya ha sido alcanzada. Desarrolle heurísticas válidas para el problema y explique la admisibilidad de las mismas.
Debería poder correr y resolver los siguientes problemas:

```
python3 pacman.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

```
python3 pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

Considere definir una representación de estado abstracto que abstraiga o no codifique información irrelevante (por ejemplo la posición de los fantasmas, la ubicación de comida extra, etc.). En particular, no utilize como estado el GameState completo, el código será prohibitibamente lento si lo hace.

Nuestra implementación con breadthFirstSearch expande menos de 2000 nodos de búsqueda en el laberinto de mediumCorners. Como siempre, las heurísticas (usadas con A*) pueden reducir la cantidad de estados buscados requeridos.

Ejercicio 6: Implemente una heurística no trivial consistente para el CornersProblems en cornersHeuristic. Se puede construir una heurística consistente que expanda menos de 800 nodos. Desarrolle las heurísticas que probó, y cómo llegó o por qué la elegida es consistente.

```
python3 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

> **Nota:** AStarCornersAgent is un atajo para:
> `-p SearchAgent -a fn=aStarSearch prob=CornersProblem heuristic=cornersHeuristic`

Recuerde que la admisibilidad no es suficiente para garantizar la correctitud en el grafo de búsqueda, necesita una condición más fuerte para la consistencia. Como siempre, las heurísticas admisibles son usualmente consistentes especialmente si son derivadas de problemas de relajación. Una vez que se tiene una heurística admisible que trabaja bien, es momento de chequear la consistencia. La única forma de garantizar la consistencia es con una demostración.
La inconsistencia podría ser detectada verificando que cada nodo que fue expandido sus nodos sucesores son mayor o iguales en el valor de f.

Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

Comiendo todas las pastillas

Ahora nos enfocaremos en resolver un problema de búsqueda con mayor dificultad: comer toda la comida de Pacman en el menor número de pasos posible. Para esto, necesitaremos una nueva definición de problema de búsqueda que formalice el problema de comer alimentos, i.e., definir una solución capaz de recolectar toda la comida en el mundo de Pacman.

Para el presente proyecto, las soluciones no tienen en cuenta fantasmas o píldoras mágicas; las soluciones solo dependen de la colocación de paredes, comida y Pacman. Si el desarrollo de los métodos de búsqueda han sido de forma general, A * con una heurística nula (equivalente a la búsqueda de costo uniforme) debe encontrar rápidamente una solución óptima para testSearch sin cambiar el código (costo total de 7):

```
python3 pacman.py -l testSearch -p AStarFoodSearchAgent
```

> **Nota:** AStarFoodSearchAgent es un atajo para:
> `-p SearchAgent -a fn=astar prob=FoodSearchProblem heuristic=foodHeuristic.`

Deberían notar que la búsqueda UCS se hace más lento, incluso para el layout más simple tinySearch. Como referencia, nuestra implementación demora 2.5 segundos para encontrar una ruta de longitud 27 luego de expandir 5057 nodos de búsqueda.

Ejercicio 7: Implementar en foodHeuristic, en el archivo *searchAgents.py*, una heurística para el FoodSearchProblem capaz de comer toda la comida en el menor tiempo posible. Explique y desarrolle el porqué de elegir esta heurística.
Pruebe su agente sobre el layout trickySearch:

```
python3 pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Como referencia, nuestro agente de UCS encuentra la solución óptima en aproximadamente 13 segundos, explorando más de 16,000 nodos. Recuerde que en este caso su heurística puede ser inconsistente, pero hay puntos extra para aquellos que encuentren una consistente y lo expliquen.