Laboratorio 1: Búsqueda no informada

Introducción

En este proyecto, tu agente Pacman deberá encontrar rutas a través de su laberinto, ya sea para llegar a una ubicación específica o para recolectar comida de manera eficiente. Construirás algoritmos de búsqueda genéricos y los aplicarás a diversos escenarios de Pacman.

Al igual que en el Laboratorio 0, este proyecto incluye un evaluador automático que te permitirá calificar tus respuestas directamente en tu máquina. Puedes ejecutarlo con el siguiente comando:

```
python autograder.py
```

El código de este proyecto consta de varios archivos de Python. Algunos deberás leerlos y comprenderlos para completar la tarea, mientras que otros puedes ignorarlos. Puedes descargar todo el código y los archivos de soporte como un archivo `search.zip`.

Archivos que debes editar

- `search.py`: En este archivo residirán todos tus algoritmos de búsqueda.

Archivos que podrías consultar

- `searchAgents.py`: Aquí se ubicarán todos tus agentes basados en búsqueda.
- `pacman.py`: El archivo principal que ejecuta los juegos de Pacman. Este archivo describe el tipo `Pacman GameState`, que utilizarás en este proyecto.
- `game.py`: Contiene la lógica detrás de cómo funciona el mundo de Pacman. Este archivo describe varios tipos de soporte como `AgentState`, `Agent`, `Direction` y `Grid`.
- `util.py`: Contiene estructuras de datos útiles para la implementación de algoritmos de búsqueda.

Archivos de soporte que puedes ignorar

- `graphicsDisplay.py`: Gráficos para Pacman.
- `graphicsUtils.py`: Soporte para los gráficos de Pacman.
- `textDisplay.py`: Gráficos ASCII para Pacman.
- `ghostAgents.py`: Agentes para controlar a los fantasmas.
- `keyboardAgents.py`: Interfaces de teclado para controlar a Pacman.
- `layout.py`: Código para leer archivos de diseño y almacenar su contenido.
- `autograder.py`: Autocalificador del proyecto.
- `testParser.py`: Analiza los archivos de prueba y solución del autocalificador.
- `testClasses.py`: Clase de prueba general del autocalificador.

- `test_cases/`: Directorio que contiene los casos de prueba para cada pregunta.
  `searchTestClasses.py`: Clases de prueba específicas del autocalificador para el Proyecto 1.

Archivos para editar y enviar

Durante el desarrollo de la tarea, completarás las secciones correspondientes en el archivo **search.py**.

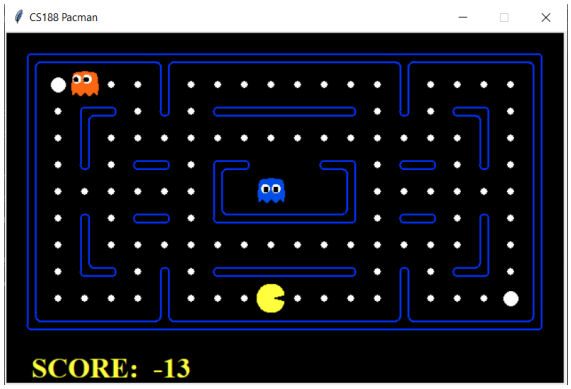
Una vez que hayas completado el laboratorio, deberá subir capturas de pantalla de cada una de las tres preguntas de este con el resultado del script de autoevaluación y el código fuente con las soluciones con el nombre de archivo searchnombreestudiante.zip, por ejemplo: ***searchArlesRodriguez.zip***

¡Bienvenido a Pacman!

Después de descargar el código, descomprimirlo y acceder al directorio, podrás jugar una partida de Pacman escribiendo lo siguiente en la línea de comandos:

```
python pacman.py
```

Pacman habita un brillante mundo azul de corredores sinuosos y deliciosos bocadillos redondos. Navegar por este mundo de manera eficiente será el primer paso de Pacman para dominar su entorno.



El agente más sencillo en **searchAgents.py** se llama `GoWestAgent`, y siempre se mueve hacia el Oeste (un agente de reflejo trivial). Este agente a veces puede ganar:

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

Pero las cosas se complican para este agente cuando es necesario girar:

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

Si Pacman se queda atascado, puedes salir del juego escribiendo `CTRL-c` en tu terminal.

Pronto, tu agente resolverá no solo `tinyMaze`, sino cualquier laberinto que desees.

Ten en cuenta que pacman.py admite varias opciones que pueden expresarse de forma larga (por ejemplo, `--layout`) o corta (por ejemplo, `-l`). Puedes ver la lista de todas las opciones y sus valores predeterminados con el siguiente comando:

```
python pacman.py -h
```

Anotaciones de tipo

Es posible que no hayas visto esta sintaxis antes:

```
def my_function(a: int, b: Tuple[int, int], c: List[List], d: Any, e:
float=1.0):
```

Esto se conoce como **anotaciones de tipo**, y se utiliza para indicar el tipo de datos que Python debe esperar para los argumentos de una función. En el ejemplo anterior:

- **a** debe ser un `int` (entero).
- **b** debe ser una `tuple` (tupla) de dos enteros.
- **c** debe ser una `List` (lista) de listas, lo que la convierte en un arreglo bidimensional de cualquier cosa.
- **d** puede ser de cualquier tipo, lo que es similar a no tener anotación.
- **e** debe ser un `float` (número decimal) y, si no se le pasa ningún valor, su valor predeterminado será **1.0**.

Por ejemplo, la siguiente llamada cumple con las anotaciones de tipo y no pasa un valor para `e`:

```
my_function(1, (2, 3), [['a', 'b'], [None, my_class], [[]]], ('h', 1))
```

Las anotaciones de tipo son un complemento a la documentación (docstrings) de tu código. Ayudan a entender con qué tipos de datos está trabajando una función, pero Python no las exige. Al escribir tus propias funciones, es tu decisión si quieres usarlas para mantener tu código organizado o si prefieres no invertir tiempo en ellas.

Procedimiento

A continuación deberás resolver tres desafíos que corresponden al tema de búsqueda no informada. Complementario a la presentación de clase, se encuentran las siguientes lecturas que pueden ser de utilidad:

- 1.2 State Spaces and Search Problems
- 1.3 Uninformed Search

P1 (3 pts): Encontrar un punto de comida fijo usando Búsqueda en Profundidad

En el archivo **searchAgents.py**, encontrarás un agente completamente implementado, el `SearchAgent`, que planea una ruta a través del mundo de Pacman y luego la ejecuta paso a paso. Los algoritmos de búsqueda para formular ese plan aún no están implementados: ¡ese es tu trabajo!

Primero, comprueba que el `SearchAgent` funciona correctamente ejecutando el siguiente comando:

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

El comando anterior le indica al `SearchAgent` que utilice **`tinyMazeSearch`** como su algoritmo de búsqueda, el cual ya está implementado en **search.py**. Pacman debería poder navegar por el laberinto sin problemas.

¡Ahora es momento de que escribas funciones de búsqueda genéricas completas para ayudar a Pacman a planificar sus rutas! Recuerda que un nodo de búsqueda debe contener no solo un estado, sino también la información necesaria para reconstruir el camino (plan) que conduce a ese estado.

Notas importantes

- Todas tus funciones de búsqueda deben **devolver una lista de acciones** que guíen al agente desde el punto de inicio hasta el objetivo. Todas estas acciones deben ser movimientos válidos (direcciones legales, sin atravesar paredes).
- Asegúrate de utilizar las estructuras de datos `Stack, Queue y PriorityQueue` que se te proporcionan en **util.py**. Estas implementaciones tienen propiedades particulares que son necesarias para la compatibilidad con el autocalificador.

Sugerencias

Cada algoritmo es muy similar. Los algoritmos para DFS, BFS y UCS solo se diferencian en los detalles de cómo se gestiona la la frontera o lista de nodos a explorar. Por lo tanto, concéntrate en implementar correctamente DFS, y el resto debería ser relativamente sencillo. De hecho, una posible implementación podría requerir una única función de búsqueda genérica configurada con una estrategia de encolamiento específica para cada algoritmo (tu implementación no necesita ser de esta forma para obtener la máxima calificación).

Implementación y pruebas

Implementa el algoritmo de búsqueda en profundidad (DFS) en la función `depthFirstSearch` en el archivo **search.py**. Para que tu algoritmo sea completo, escribe una versión de **DFS** que evite expandir cualquier estado que ya haya sido visitado.

Tu código debería encontrar rápidamente una solución para los siguientes comandos:

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

python pacman.py -l mediumMaze -p SearchAgent

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

El tablero de Pacman mostrará una superposición de los estados explorados y el orden en que fueron explorados (el rojo más brillante indica una exploración más temprana). ¿El orden de exploración es lo que esperabas? ¿Pacman realmente visita todos los cuadros explorados en su camino hacia el objetivo?

Calificación

Ejecuta el siguiente comando para ver si tu implementación pasa todos los casos de prueba del autocalificador y guarda el pantallazo:

```
python autograder.py -q q1
```

P2 (3 pts): Búsqueda en Amplitud (BFS)

Implementa el algoritmo de **búsqueda en amplitud (BFS)** en la función `breadthFirstSearch` dentro de **search.py**. De nuevo, escribe un algoritmo de búsqueda en grafo que evite expandir los estados ya visitados. Prueba tu código de la misma manera que lo hiciste con la búsqueda en profundidad.

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

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

¿El algoritmo BFS encuentra la solución de menor costo? Si no, revisa tu implementación.

Sugerencias y notas adicionales

- **Velocidad de juego:** Si Pacman se mueve demasiado lento, puedes usar la opción `--frameTime 0` para acelerar el juego.
- **Generalidad del código:** Si has escrito tu código de búsqueda de forma genérica, debería funcionar igual de bien para el problema de búsqueda del ocho-rompecabezas sin necesidad de hacer cambios. Puedes probarlo con el siguiente comando:

```
python eightpuzzle.py
```

Calificación

Ejecuta el siguiente comando para ver si tu implementación pasa todos los casos de prueba del autocalificador y guarda el pantallazo:

```
python autograder.py -q q2
```

P3 (3 pts): Variación de la función de costo

Aunque la búsqueda en amplitud (BFS) encuentra el camino con el menor número de acciones, podríamos querer encontrar rutas que sean "mejores" en otros sentidos. Considera los laberintos `mediumDottedMaze` y `mediumScaryMaze`.

Al modificar la función de costo, podemos incentivar a Pacman a encontrar caminos distintos. Por ejemplo, podemos penalizar los movimientos en áreas peligrosas infestadas de fantasmas o, por el contrario, hacer más atractivos los movimientos en zonas con mucha comida. Un agente Pacman racional adaptará su comportamiento en consecuencia.

Implementa el algoritmo de **búsqueda de costo uniforme (UCS)** en la función `uniformCostSearch` en **search.py**. Te recomendamos revisar **util.py** en busca de algunas estructuras de datos que podrían ser útiles para tu implementación. Ahora deberías observar un comportamiento exitoso en los tres laberintos siguientes. Los agentes mencionados son todos agentes UCS que solo se diferencian en la función de costo que utilizan (los agentes y las funciones de costo ya están escritos para ti):

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

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

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

Nota: Deberías obtener costos de ruta muy bajos y muy altos para los agentes StayEastSearchAgent y StayWestSearchAgent respectivamente, debido a sus funciones de costo exponenciales (para más detalles, revisa searchAgents.py).

Calificación

Ejecuta el siguiente comando para ver si tu implementación pasa todos los casos de prueba del autocalificador y guarda el pantallazo:

```
python autograder.py -q q3
```

Envío

Una vez que hayas completado el laboratorio, deberás subir capturas de pantalla de cada una de las tres preguntas de este con el resultado del script de autoevaluación

```
python autograder.py -q q1

python autograder.py -q q2

python autograder.py -q q3
```

Y el código fuente con las soluciones con el nombre de archivo searchnombreestudiante.zip, por ejemplo:

searchArlesRodriguez.zip

Referencias:

Project 1. (s. f.). CS 188 Fall – University of California, Berkeley 2025. https://inst.eecs.berkeley.edu/~cs188/fa25/projects/proj1/