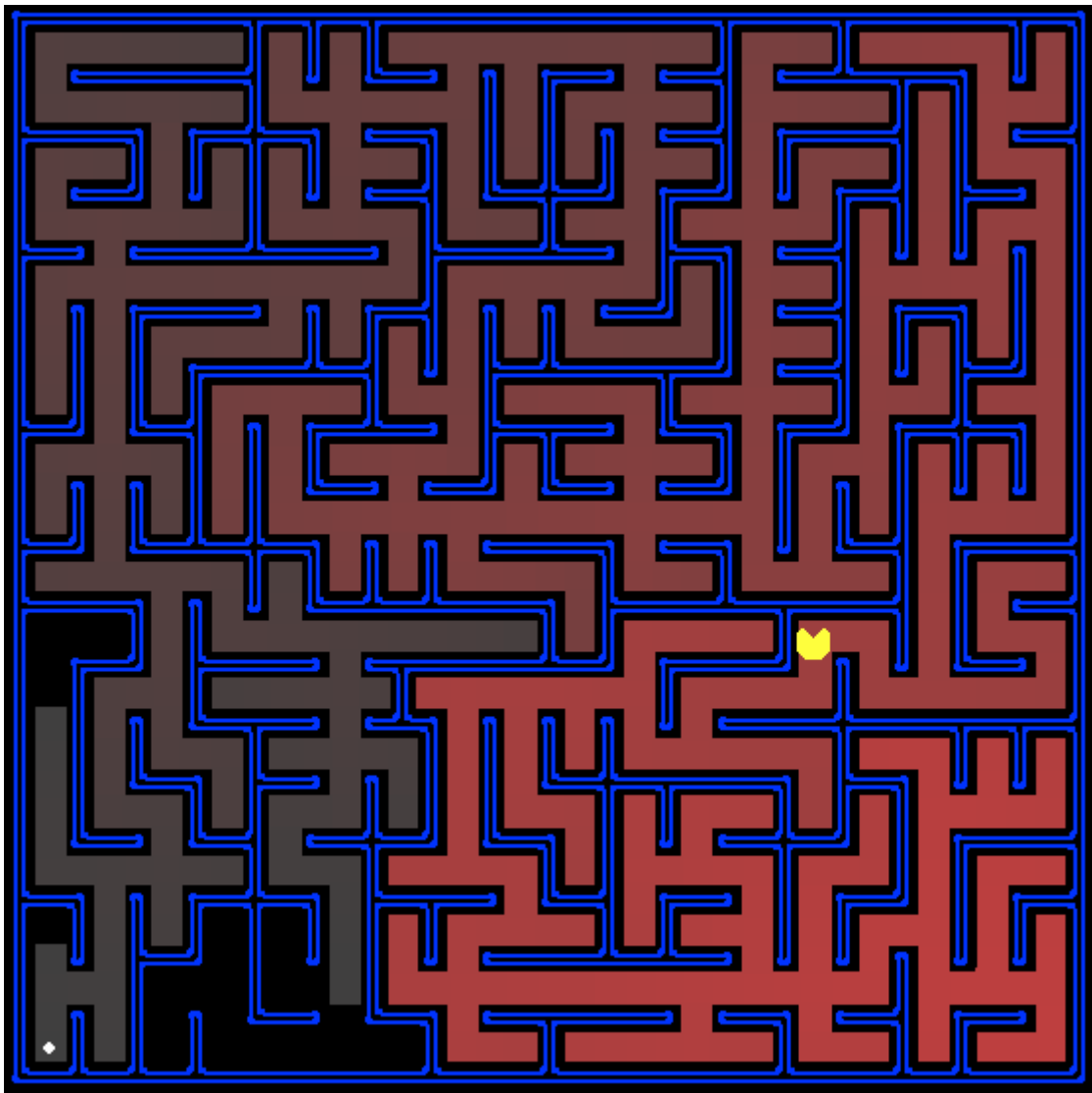


PL1a: Búsqueda (Pacman) 1

Tabla de contenido

- Introducción
- Bienvenidos
- Q1: Depth First Search
- Q2: Breadth First Search
- Q3: Uniform Cost Search



Introducción

En este proyecto, nuestro agente de Pacman encontrará caminos a través de su mundo de laberintos, tanto para llegar a un lugar en particular como para recolectar alimentos de manera eficiente. Construiremos algoritmos de búsqueda generales y los aplicaremos a los escenarios de Pacman.

Como en el proyecto 0, este proyecto incluye un programa de autoevaluación que podrás ejecutar en tu máquina de la siguiente manera:

python autograder.py

El código para este proyecto consta de varios ficheros de Python, algunos de los cuales se deberán leer y comprender para completar la tarea, y algunos de los cuales podemos ignorar.

Podemos descargar todo el código y los archivos con el nombre **busqueda_pacman.zip**:

Ficheros que hay que editar:

search.py	Donde estarán los algoritmos de búsqueda.
searchAgents.py	Donde estarán los agentes basados en búsqueda.

Ficheros que puede interesar mirar:

pacman.py	El fichero principal que ejecuta los juegos Pacman. Este fichero define un tipo Pacman GameState, que podemos usar en este proyecto.
game.py	La lógica detrás de la que funciona el mundo Pacman. Este fichero describe tipos de apoyo como AgentState, Agent, Direction, y Grid.
util.py	Estructuras de datos útiles para implementar algoritmos de búsqueda.

Ficheros de apoyo que se pueden ignorar:

graphicsDisplay.py	Gráficos para Pacman
graphicsUtils.py	Soporte para gráficos de Pacman
textDisplay.py	Gráficos ASCII para Pacman
ghostAgents.py	Agentes para controlar los fantasmas
keyboardAgents.py	Interfaces de teclado para controlar Pacman
layout.py	Código para leer ficheros de configuración y almacenar sus contenidos
autograder.py	Autocalificador
testParser.py	Ejecuta un test de autograder y ficheros de solución
testClasses.py	Clases de test generales de autograding
test_cases/	Directorio que contiene los casos de test para

	cada pregunta
searchTestClasses.py	Clases de test específicas de autograder del proyecto 1

Ficheros a editar y entregar:

Tendremos que completar partes de `search.py` y `searchAgents.py`. No hay que cambiar los demás ficheros.

Evaluación:

No se pueden cambiar los nombres de las funciones o clases proporcionadas dentro del código, o causará conflictos en el autograder.

Ayuda:

¡No estas solo/a! Si os encontráis atascados en algo, debéis poneros en contacto con los profesores del curso para obtener ayuda. Las tutorías y el foro de discusión están a vuestra disposición; por favor usadlos. Se quiere que estos proyectos sean gratificantes e instructivos, no frustrantes y desmoralizadores. Pero no sabemos cuándo o cómo ayudar a menos que nos lo solicitéis.

Bienvenidos a Pacman

Después de descargar el código (`busqueda_pacman.zip`), descomprimirlo, y movernos al directorio, deberíamos poder jugar al Pacman tecleando esta línea:

```
python pacman.py
```

Pacman vive en un mundo azul brillante de corredores retorcidos y deliciosas golosinas redondas. Navegar por este mundo de manera eficiente será el primer paso de Pacman para dominar su mundo. El agente más simple en `searchAgents.py` se llama `GoWestAgent`, que siempre va al Oeste (un agente reflex trivial). Este agente puede ganar ocasionalmente:

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

Pero las cosas se ponen feas cuando hay que girar:

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

Si Pacman se atasca, podemos salir del juego tecleando CTRL - C en nuestro terminal. Más adelante, nuestro agente no solucionará únicamente el laberinto `tinyMaze`, sino cualquier laberinto que queramos.

`Pacman.py` soporta un número de opciones que pueden expresarse de diferente manera (p.e., `--layout`) o de manera abreviada (p.e., `-l`). Todos los laberintos se encuentran dentro de la carpeta `layouts`. Se puede ver la lista de todas las opciones y sus valores por defecto con:

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

Además, todos los comandos que aparecen en este proyecto también se encuentran en `commands.txt`, para que podamos copiar y pegar con facilidad. En UNIX/Mac OS X, también se pueden ejecutar estos comandos con `bash commands.txt`.

Pregunta 1 (3 puntos): Buscando un punto de comida fijo usando Depth First Search

En `searchAgents.py`, encontraremos un `SearchAgent` completamente implementado, que planifica un camino a través del mundo de Pacman y luego ejecuta ese camino paso a paso. Los algoritmos de búsqueda para formular un plan no están implementados, ese es vuestro trabajo.

Primero, comprobaremos que `SearchAgent` funciona correctamente ejecutando:

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

El comando anterior le dice a `SearchAgent` que use `tinyMazeSearch` como algoritmo de búsqueda, que está implementado en `search.py`. Pacman debería navegar por el laberinto sin problema.

¡Ahora es el momento de escribir funciones de búsqueda genéricas completas para ayudar a Pacman a planificar rutas! El pseudocódigo para los algoritmos de búsqueda que escribiremos se puede encontrar en el material de clase. Debemos recordar que un nodo de búsqueda debe contener no solo un estado sino también la información necesaria para reconstruir la ruta (plan) que llega a ese estado.

Nota importante

Todas las funciones de búsqueda deben devolver una lista de acciones que guiarán al agente desde el principio hasta el objetivo. Todas estas acciones tienen que ser movimientos legales (instrucciones válidas, no moverse a través de las paredes).

Cuidado!!!!

¡Tenemos que asegurarnos de utilizar las estructuras de datos `Stack`, `Queue` y `PriorityQueue` que se proporcionan en `util.py`! Estas implementaciones de estructuras de datos tienen propiedades particulares que son necesarias para la compatibilidad con el autograder.

Pista

Cada algoritmo es muy similar. Los algoritmos para DFS, BFS, UCS y A* solo difieren en los detalles de cómo se gestiona el borde (fringe). Por lo tanto, debemos concentrarnos en obtener DFS correctamente y el resto debería ser relativamente sencillo. De hecho, una posible implementación requiere solo un único método de búsqueda genérico que se configura con una estrategia de almacenamiento específica de algoritmo. (Nuestra implementación no necesita ser de esta forma para recibir toda la puntuación).

Se debe implementar el algoritmo de búsqueda en profundidad (DFS) en la función `depthFirstSearch` en `search.py`. Para que el algoritmo sea completo, debemos usar la versión de búsqueda en grafo de DFS, que evita expandir los estados ya visitados.

Nuestro código debe encontrar rápidamente una solución para:

```
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 (un rojo más brillante significa una exploración más temprana). ¿Es el orden de exploración lo que esperábamos? ¿Pacman realmente va a todos los cuadrados explorados en su camino hacia el objetivo?

Sugerencia: si utilizamos una pila como estructura de datos, la solución encontrada por nuestro algoritmo DFS para `mediumMaze` debe tener una longitud de 130 (siempre que se añadan los sucesores al borde (fringe) en el orden proporcionado por `getSuccessors`; podemos obtener 246 si se añaden en el orden inverso). ¿Es esta una solución de menor coste? Si no es así, pensemos qué se está haciendo mal.

Pregunta 2 (3 puntos): Breadth First Search

Debemos implementar el algoritmo de búsqueda en anchura (BFS) en la función `breadthFirstSearch` en `search.py`. Nuevamente, escribiremos un algoritmo de búsqueda en grafos que evite expandir los estados ya visitados. Probaremos nuestro código de la misma manera que lo hicimos para 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
```

¿Encuentra BFS una solución de coste mínimo? En caso de que no, deberemos examinar nuestra implementación.

Sugerencia: si Pacman se mueve demasiado despacio, podemos probar la opción:

```
--frameTime 0.
```

Nota: si hemos escrito el código de manera genérica, el código debería funcionar igualmente bien para el problema de búsqueda del puzzle de 8 piezas sin ningún cambio.

```
python eightpuzzle.py
```

Pregunta 3 (3 puntos): Cambiando la función de coste

Si bien el algoritmo BFS encontrará la ruta de menos acciones hacia el objetivo, es posible que deseemos encontrar rutas que sean "mejores" en otros sentidos. Consideremos `mediumDottedMaze` y `mediumScaryMaze`.

Al cambiar la función de coste, podemos alentar a Pacman a encontrar diferentes caminos. Por ejemplo, podemos cobrar más por pasos peligrosos en áreas llenas de fantasmas o menos por pasos en áreas ricas en alimentos, y un agente de Pacman racional debería ajustar su comportamiento en respuesta a esas situaciones.

Debemos implementar el algoritmo de búsqueda en grafo de coste uniforme (UCS) en la función `uniformCostSearch` en `search.py`. Recomendamos consultar `util.py` para ver algunas estructuras de datos que pueden ser útiles en nuestra implementación. Ahora deberemos observar un comportamiento exitoso en los tres diseños siguientes, donde los agentes son todos agentes UCS que difieren solo en la función de coste que usan (los agentes y las funciones de coste están escritos):

```
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: Deberíamos obtener costes muy bajos y muy altos para `StayEastSearchAgent` y `StayWestSearchAgent` respectivamente, dadas sus funciones de coste exponenciales (ver los detalles en `searchAgents.py`).

Entrega

Para presentar el proyecto, se debe entregar:

- Documentación en la que se presentan los problemas abordados con una explicación razonada de la solución empleada (estructuras de datos, aspectos reseñables, ...). No hay que explicar el código, si no, el por qué!