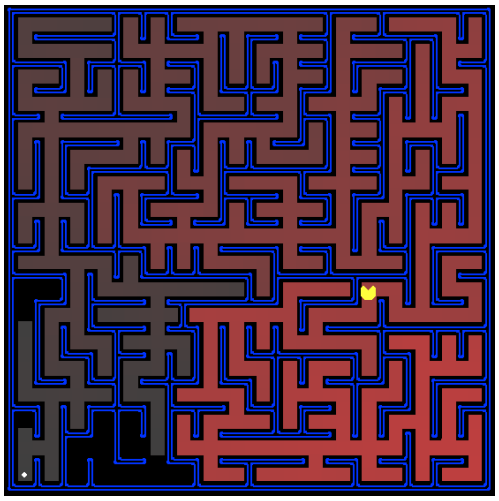
Proyecto 1: Búsqueda en Pacman



Introducción

En este proyecto, hay que permitir que el agente Pacman encuentre el camino a través del laberinto para alcanzar una posición particular o recolectar la comida eficientemente. El objetivo es construir algoritmos de búsquedas generales y aplicarlos a los escenarios de Pacman dados.

El código consistirá de varios archivos Python, algunos de los cuales se necesitarán leer y entender para poder cumplir con el objetivo.

Archivos para editar:

search.py	Los algoritmos de búsquedas. Deben estar escritos aquí.
searchAgents.py	La información de los agentes debe estar aquí.

Archivos que hay que ver:

pacman.py	El archivo principal que corre el juego Pacman. Este archivo describe un estado de tipo GameState, el cual hay que usar en este proyecto.
game.py	La lógica detrás de cómo el Pacman trabaja. Se describen varios tipos que son utilizados para modelar el comportamiento y los gráficos como AgentState, Agent, Direction, y Grid.
util.py	Estructuras de datos útiles para implementar los algoritmos de búsqueda.

Archivos que se pueden ignorar:

graphicsDisplay.py	Gráficos para Pacman
graphicsUtils.py	Utilidades para gráficos
textDisplay.py	Gráficos ASCII para Pacman
ghostAgents.py	Agentes para controlar los fantasmas
keyboardAgents.py	Interfaz del teclado para controlar el Pacman
layout.py	Código para leer archivos y almacenar su contenido.

Qué enviar: Se deberá enviar sólo los archivos `search.py` y `searchAgents.py`, junto con un archivo `grupo.txt`.

Observación: Por favor, no cambie los nombres de las funciones provistas o cualquiera de las clases del código suministrado.

Bienvenido a Pacman

Ante todo se debe descargar `python` de: https://www.python.org/downloads/ allí hay dos versiones, pueden descargar la 2.7.6 con la cual funciona el código proporcionado. Además, puede descargar un IDLE (integrated development environment) para Python, en algunas distribuciones viene por defecto con los instaladores mientras que en otras es opcional.

Después de descargar el código (search.zip), descomprimirlo y yendo al directorio *search*, se tendría que poder jugar al Pacman tipeando la línea de comando:

```
python pacman.py
```

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

El agente más simple en searchAgents.py es llamado `GoWestAgent`, el cual siempre va al Oeste (un agente trivial). Este agente puede ganar:

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

pero no funciona cuando se requiere doblar:

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

Si se quiere salir del juego, se puede presionar CTRL-c en la terminal.

Nuestros agentes no solo resolverán `tinyMaze`, sino cualquier laberinto que se necesite.

Es importante notar que `pacman.py` soporta algunas opciones que pueden ser expresadas, cada una de ellas, en un formato largo (e.g., `--layout`) o en un formato corto (e.g., `-l`). Se pueden ver la lista de todas las opciones y sus valores por defecto con:

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

También, todos los comandos que aparecen en este proyecto también aparecen en commands.txt, para facilitar el copiar y pegar.

Encontrando comida en lugares fijos utilizando algoritmos de búsqueda

En `searchAgents.py`, encontrarás una implementación de un `SeachAgent`, el cual planificará nuestro camino a través del mundo de Pacman y luego, ejecutara el camino paso a paso. Los algoritmos de búsqueda para formular un plan no están implementados (este es tu trabajo). Como surgirán preguntas durante el desarrollo, es conveniente ir al glosario de objetos en el código que está al final de este documento.
Ante todo, es importante verificar que el `SearchAgent` está trabajando correctamente, esto lo podemos hacer ejecutando:

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

El commando anterior le dice al `SearchAgent` que use `tinyMazeSearch` como su algoritmo de búqueda, el cual está implementado en `search.py`. Pacman debería navegar el laberinto exitosamente.

Ahora hay que escribir las funciones de búsqueda genérica para ayudar al Pacman a planificar sus rutas. El pseudocódigo de los algoritmos de búsqueda lo pueden encontrar en las transparencias y los libros recomendados por la cátedra. Recuerden que un nodo búsqueda debe contener no solo un estado sino también la información necesaria para reconstruir el path que lo llevó al estado dado.

Nota Importante: Todas las funciones de búsqueda debe retorna una lista de *acciones* que llevaran al agente del comienzo al objetivo. Estas acciones deben ser movimientos válidos (no se puede mover a través de las paredes del laberinto).

Pista: Cada algoritmo es muy similar. Algoritmos para DFS, BFS, UCS, and A* difieren solo en los detalles de como la frontera es manipulada. Por lo tanto, es importante concentrarse en generar el DFS correctamente y el resto debería ser relativamente sencillo. Por ejemplo, una posible implementación requiere únicamente un único método de búsqueda el cual es configurado con un algoritmo que especifica la estrategia.

Pista: Asegúrese de chequear los tipos `Stack, Queue` y `PriorityQueue` que son provistos en `util.py`

Cuestión 1 Implemente el algoritmo de búsqueda DFS en la función `depthFirstSearch` en `search.py`.

El código debería rápidamente encontrar 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 del Pacman mostrará un overlay de los estados explorados, y el orden en el cual fueron explorados (rojo más brillante significa una exploración más temprana). ¿Es el orden de exploración esperado? ¿Tiene Pacman realmente que ir por todos las casillas exploradas en su camino a la meta?

Pista: Si usas un `Stack` como tu estructura de datos, la solución encontrada por el algoritmo DFS que desarrollaste para `mediumMaze` debería tener una longitud de 130 (siempre y cuando pongan los sucesores en el orden provisto por getSuccessors; deberías tener 244 si los pusiste en el orden inverso).

Cuestión 2 Implemente el alogirtmo breadth-first search (BFS) en la function `breadthFirstSearch` en `search.py`. Otra vez, escriba un algoritmo de búsqueda que permita expandir cualquier estado que fue visitado. Testee su código de la misma forma que lo hizo para el DFS.

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

Pista: Si Pacman se mueve muy despacio, puede usar la opción `--frameTime 0`.

Nota: Si el código que escribió fue escrito genéricamente, el código deberá funcionar igual de bien para resolver el problema del 8-puzzle.

```
python eightpuzzle.py
```

Variando la function Costo

Mientras BFS encontrará el camino con la menor cantidad de acciones que nos lleven al objetivo, nosotros necesitamos encontrar los caminos que sean "mejores" en otros sentidos. Analice los problemas `mediumDottedMaze` y `mediumScaryMaze`.

Cambiando la función costo, podemos llevar a Pacman a encontrar diferentes caminos. Por ejemplo, podemos poner que sea más costosos los pasos en áreas de fantasmas o más baratos en áreas ricas en comida y el agente Pacman debería ajustar su comportamiento respondiendo a estos cambios.

Cuestión 3 Implemente el algoritmo de búsqueda de costo uniforme (UCS) en la función `uniformCostSearch` que se encuentra en `search.py`. Recomendamos que vea el archivo `util.py` para algunas estructuras de datos que serán útiles a la implementación. Debería ahora observar la conducta del agente en los tres laberintos que están a continuación donde los agentes son todos agentes UCS que difieren solo en la función costo que usan (los agentes y funciones de costo deben ser escritas):

```
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: Se deberían ver caminos de costos muy altos y muy bajos para el `StayEastSearchAgent` y `StayWestSearchAgent` respectivamente debido a su función de costo exponencial (vea `searchAgents.py` para más detalles).

A* search

Cuestion 4 Implemente una búsqueda A* en la función `aStarSearch` en `search.py`. A* toma una función heurística como argumento. Las heurísticas tienen dos argumentos: un estado en el problema de búsqueda (el argumento principal), y el problema en sí mismo (para información de referencia). La función heurística `nullHeuristic` que se encuentra en `search.py` es un ejemplo trivial.

Se puede verificar la implementación de A* en el problema original de encontrar un camino a través del laberinto a una posición fija usando la heurística de la distancia de Manhattan (ya está implementada como `manhattanHeuristic` en `searchAgents.py`).

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

Deberían ver que A* encontró la solución óptima un poco más rápido que la búsqueda de costo uniforme (cerca de 549 vs. 620 nodos de búsqueda expandidos en nuestra implementación, pero la prioridad puede hacer que sus cifras difieran ligeramente). Qué pasara en `openMaze` para varias estrategias de búsqueda?

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 *corner mazes*, hay 4 pastillas, una en cada esquina. Nuestro nuevo problema será encontrar el camino más corto a través del laberinto que toque las cuatro esquinas (más allá de que el laberinto tenga comida allí o no). Note que algunos laberintos como tinyCorners, el camino más corto no siempre va a la comida más cercana primero. *Hint*: el camino más corto a través de `tinyCorners` toma 28 pasos.

Cuestión 5 Implemente el problema de búsqueda `CornersProblem` en `searchAgents.py`. Necesitará elegir una representación de estados que codifique toda la información necesaria para detectar cuál de las 4 esquinas han sido alcanzadas. Ahora, tu agente de búsqueda debería resolver:

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

Considere definir una representación de estado abstracto que *no* codifique información irrelevante (como la posición de los fantasmas, donde hay comida extra, etc.). En particular, no use un estado de búsqueda como `GameState`. El código será muy, muy lento si lo hace.

Pista: Las únicas partes del estado del juego que necesita saber en tu implementación son la posición inicial del Pacman y la ubicación de las 4 esquinas.

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

Cuestión 6 Implemente una heurística no trivial consistente para el `CornersProblem` en `cornersHeuristic`. Se puede construir una heurística consiste que expanda menos de 800 nodos.

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

Nota: `AStarCornersAgent` is un shortcht 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 es mayor o igual en el valor de f.

Glosario

Aquí está el glosario de los objetos claves del código base relacionados a problemas de búsqueda:

`SearchProblem (search.py)`
: Un problema de Búsqueda es un objeto abstracto que representa el espacio de estados, la function successor, costos y el estado objetivo del problema. Se podrá interactuar con cualquier SearchProblem solo a través de los métodos definidos al comienzo de `search.py`

`PositionSearchProblem (searchAgents.py)`
: Un tipo específico de Problema de Búsqueda con el cual se tendrá que trabajar. Esto corresponde a una búsqueda con una única píldora como objetivo en el laberinto.

`CornersProblem (searchAgents.py)`
: A specific type of SearchProblem that you will define --- it corresponds to searching for a path through all four corners of a maze.

Search Function
: Una función de búsqueda es una función que toma una instancia de SearchProblem como parámetro, ejecuta algún algoritmo y retorna la secuencia de acciones que llevan al objetivo. Ejemplos de funciones de búsqueda son `depthFirstSearch` y `breadthFirstSearch`, las cuales tienes que escribir. La función de búsqueda `tinyMazeSearch` ya está declarada, aunque no es buena y sólo trabaja correctamente en `tinyMaze`

`SearchAgent`
: `SearchAgent` es una clase que implementa un Agente (un objeto que interactúa con el mundo) y hace su planificación a través de la función de búsqueda. El `SearchAgent` primero usa la función de búsqueda provista para hacer un plan de acciones y tratar de alcanzar el estado objetivo, luego ejecuta las acciones una por vez.