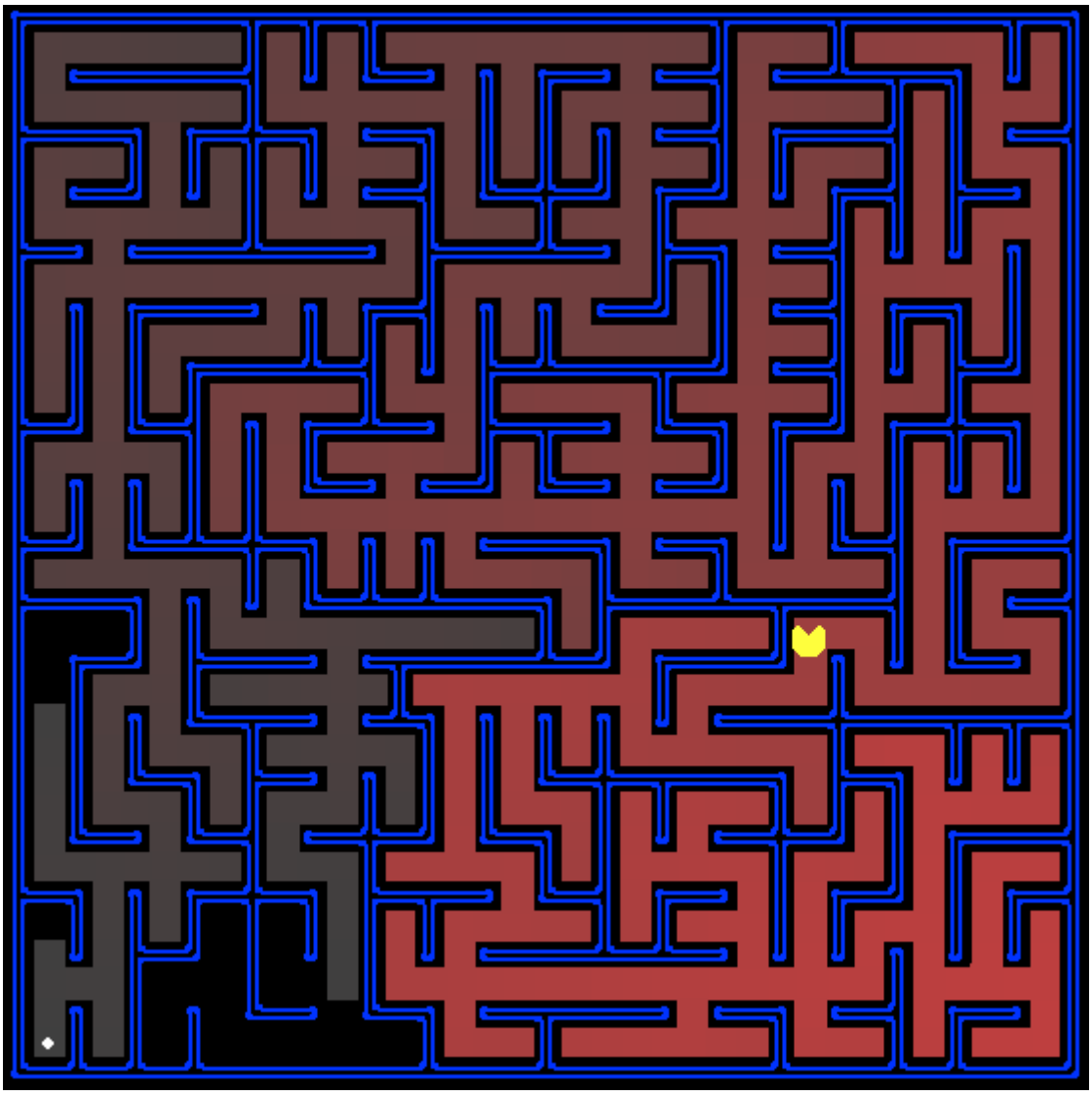
PL1b: Búsqueda (Pacman) 2

Tabla de contenido



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. En este proyecto usaremos Python 3. **Copiar vuestro fichero search.py implementado del PL1a en este proyecto.**
Podemos descargar todo el código y los archivos con el nombre **PL1b.zip**:

Fichero 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 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 4 (3 puntos): Búsqueda A*

Debemos implementar la búsqueda en grafo A* en la función (que está vacía) aStarSearch en search.py. A* toma una función heurística como argumento. La heurística toma dos argumentos:
un estado en el problema de búsqueda (el argumento principal) y el problema en sí (para información de referencia). La función heurística a utilizar es **nullHeuristic**.

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

Si queremos, para ver la diferencia, podemos probar nuestra implementación A* en el problema original de encontrar una ruta a través de un laberinto a una posición fija utilizando la heurística de distancia de Manhattan (implementada ya como manhattanHeuristic en searchAgents.py). Tened en cuenta que el autograder solo puntuará con **nullHeuristic**.

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

Deberíamos ver que A* encuentra la solución óptima ligeramente antes que UCS (alrededor de 549 vs. 620 nodos de búsqueda expandidos en nuestra implementación, aunque empates en prioridad, podrían variar esos números ligeramente). ¿Qué pasa en openMaze para las distintas estrategias de búsqueda?

Pregunta 5 (3 puntos): Buscando todas las esquinas (all the corners)

El poder real de A* solo será evidente con un problema de búsqueda más desafiante. Es hora de formular un nuevo problema y diseñar un heurístico para él.

En el “laberinto de esquinas” (four corners), hay cuatro puntos, uno en cada esquina. Nuestro nuevo problema de búsqueda es encontrar el camino más corto a través del laberinto que toque las cuatro esquinas (ya sea que el laberinto tenga comida allí o no). Debemos tener en cuenta que para algunos laberintos como tinyCorners, ¡el camino más corto no siempre llega primero a la comida más cercana! Sugerencia: el camino más corto a través de tinyCorners toma 28 pasos.

Nota: Debemos asegurarnos de completar la pregunta 2 antes de trabajar en la pregunta 5, porque la pregunta 5 se basa en nuestra respuesta de la pregunta 2.

Debemos implementar el problema de búsqueda CornersProblem en searchAgents.py. Deberemos elegir una representación de estado que codifique toda la información necesaria para detectar si se han alcanzado las cuatro esquinas. Ahora, nuestro 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
```

Para recibir toda la puntuación, debemos definir una representación abstracta de estados que no codifique información irrelevante (como la posición de los fantasmas, dónde está el alimento

adicional, etc.). En particular, no usaremos un Pacman GameState como estado de búsqueda. Nuestro código sería muy, muy lento si lo hacemos (y también incorrecto).

Sugerencia: Las únicas partes del estado del juego a las que debemos hacer referencia en nuestra implementación son una tupla de **la posición inicial de Pacman y la ubicación/representación de las cuatro esquinas**.
Nuestra implementación de breadthFirstSearch expande menos de 2000 nodos de búsqueda en mediumCorners. Sin embargo, el heurístico (utilizada con la búsqueda A * en la siguiente pregunta) puede reducir la cantidad de búsqueda requerida.

Pregunta 6 (3 puntos): Problema de las cuatro esquinas: Heurístico

Nota: Debemos asegurarnos de completar la pregunta 4 antes de empezar con la pregunta 6, porque la pregunta 6 se construye sobre nuestra respuesta de la pregunta 4.

Debemos implementar un heurístico no trivial y admisible para el CornersProblem en cornersHeuristic.

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

Nota: AStarCornersAgent es una abreviatura para:

-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic

Admisibilidad vs. Consistencia: Debemos recordar que los heurísticos son solo funciones que toman estados de búsqueda y devuelven números que estiman el coste al objetivo más cercano. Un heurístico más efectivo devolverá valores más cercanos a los costes reales al objetivo. Para ser admisible, los valores del heurísticos deben ser inferiores o igual que el coste real de la ruta más corta al objetivo más cercano (y no negativos). Para ser consistente, además debe suceder que si una acción ha costado *c*, entonces tomar esa acción solo puede causar una caída en el heurístico de a lo sumo *c*.
Debemos recordar que la admisibilidad no es suficiente para garantizar la corrección en la búsqueda en grafo: necesita una condición más sólida de consistencia. Sin embargo, los heurísticos admisibles suelen ser también consistentes, especialmente si se derivan de relajaciones de problemas. Por lo tanto, generalmente es más fácil comenzar haciendo una lluvia de ideas de heurísticos admisibles. Una vez que tengamos un heurístico admisible que funcione bien, podemos verificar si también es consistente. La única forma de garantizar la consistencia es con una prueba. Sin embargo, la inconsistencia a menudo se puede detectar verificando que para cada nodo que expandamos, sus nodos sucesores tengan un valor *f* igual o mayor. Además, si UCS y A* alguna vez devuelven rutas de diferentes longitudes, nuestro heurístico no es inconsistente.
¡Esto es complicado!

Heurístico no trivial: los heurísticos triviales son los que devuelven cero en todas partes (UCS) y el heurístico que calcula el verdadero coste de finalización. La primera no nos salvará en ningún momento, mientras que la última expirará el autograder (timeout). Deseamos un heurístico que reduzca el tiempo total de cómputo, aunque para esta asignación el autograder solo verificará los recuentos de nodos (además de imponer un límite de tiempo razonable).

Puntuación: Nuestro heurístico debe ser un heurístico no trivial admisible para recibir puntuación. Debemos asegurar que el heurístico devuelve 0 en cada estado objetivo, y nunca devuelva un valor negativo. Dependiendo en cuántos nodos expanda nuestro heurístico para bigCorners, recibiremos esta puntuación:

Número de nodos expandidos	Puntuación
más de 2000	0/3
como mucho 2000	1/3
como mucho 1600	2/3
como mucho 1200	3/3

Recordad: Si nuestro heurístico es inconsistente, no recibiremos ninguna puntuación, por lo que debemos ser cuidadosos.

Pregunta 7 (4 puntos): Comiendo todos los puntos (Eating All The Dots)

Ahora resolveremos un problema de búsqueda difícil (hard): comer toda la comida de Pacman en el menor número de pasos posible. Para esto, necesitaremos una nueva definición del problema de búsqueda que formalice el problema de eliminación de alimentos: FoodSearchProblem en searchAgents.py (está implementado). Una solución se define como un camino que recolecta toda la comida en el mundo de Pacman. Para el presente proyecto, las soluciones no tienen en cuenta los fantasmas o las pastillas de energía; Las soluciones solo dependen de la colocación de paredes, comida y Pacman. (¡Por supuesto, los fantasmas pueden arruinar la ejecución de una solución! Llegaremos a eso en el próximo proyecto). Si hemos escrito correctamente nuestros métodos de búsqueda generales, A* con un heurístico nulo (equivalente a la búsqueda de coste uniforme) debería rápidamente encontrar una solución óptima para testSearch sin cambiar el código (coste total de 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

Nota: AStarFoodSearchAgent es una abreviatura para

-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

Deberíamos ver que UCS comienza a ralentizarse incluso para el aparentemente simple tinySearch. Como referencia, nuestra implementación tarda 2.5 segundos en encontrar una ruta de longitud 27 después de expandir 5057 nodos de búsqueda.

Nota: Asegúrese de completar la pregunta 4 antes de trabajar en la pregunta 7, porque la pregunta 7 se basa en nuestra respuesta a la pregunta 4. Debemos completar foodHeuristic en searchAgents.py con un heurístico consistente para el FoodSearchProblem. Probaremos nuestro agente en el tablero trickySearch:

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Nuestro agente UCS encuentra la solución óptima en aproximadamente 13 segundos, explorando más de 16,000 nodos.

Cualquier heurístico consistente no negativo y no trivial recibirá 1 punto. Debemos asegurar que nuestro heurístico devuelve 0 en cada estado objetivo y nunca devuelve un valor negativo. Dependiendo de los nodos que expanda nuestro heurístico, obtendremos puntos adicionales:

Número de nodos expandidos	Puntuación
más de 15000	1/4
como mucho 15000	2/4
como mucho 12000	3/4
como mucho 9000	4/4 (puntuación total; medio)

Recordad: Si nuestro heurístico no es admisible, no recibiremos ninguna puntuación, por lo que debemos ser cuidadosos. ¿Podemos resolver mediumSearch en poco tiempo? En caso de que sí, o bien estamos muy impresionados, o el heurístico es inadmisible (inconsistente).

Pregunta 8 (3 puntos): Búsqueda subóptima (Suboptimal Search)

A veces, incluso con A* y un buen heurístico, es difícil encontrar la ruta óptima a través de todos los puntos. En estos casos, aún nos gustaría encontrar un camino razonablemente bueno, rápidamente. En esta sección, escribiremos un agente que siempre come con voracidad el punto más cercano. ClosestDotSearchAgent se ha implementado parte del programa en searchAgents.py, pero le falta una función clave que encuentre una ruta al punto más cercano.

Debemos implementar la función findPathToClosestDot en searchAgents.py. Nuestro agente resuelve este laberinto (¡subóptimamente!) en menos de un segundo con un coste de ruta de 350:

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Sugerencia: La forma más rápida de completar findPathToClosestDot es completar el AnyFoodSearchProblem, al que le falta la prueba de objetivo. Luego, debemos resolver ese problema con una función de búsqueda adecuada. ¡La solución debe ser muy corta!

Nuestro ClosestDotSearchAgent no siempre encontrará la ruta más corta posible a través del laberinto. Debemos asegurarnos de entender por qué y tratar de encontrar un pequeño ejemplo en el que ir repetidamente al punto más cercano no resulte en encontrar el camino más corto para comer todos los puntos.

Entrega

Para presentar el proyecto, se debe entregar:

- Fichero **search.py** y **searchAgents.py**
- **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 (cual ha sido vuestra representación de las 4 esquinas y por qué, por qué vuestra elección del heurístico en cada ejercicio etc etc)! Si se explica el código tendrá una penalización!**