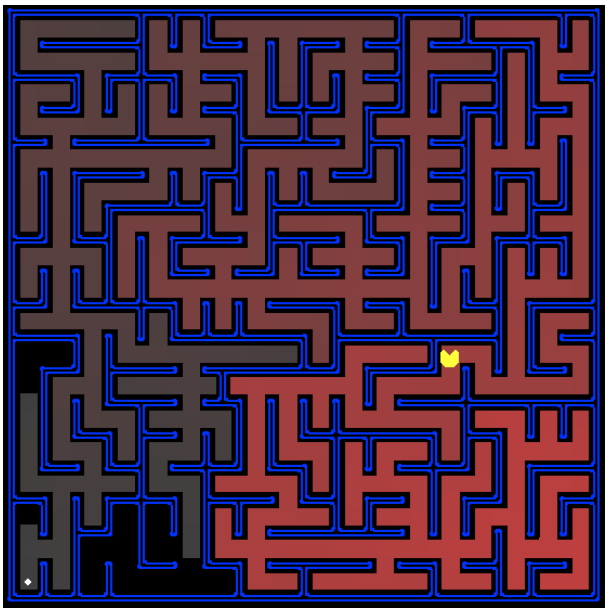
Universidad
Rey Juan Carlos

Práctica 1: Resolución de problemas por búsqueda[1]

Inteligencia Artificial



Introducción

En esta práctica, el agente Pacman encontrará caminos a través del mundo del laberinto, alcanzando una posición concreta y recogiendo comida de forma óptima. Se construirán algoritmos de búsqueda generales y se aplicarán en los escenarios de Pacman.

La práctica incluye un evaluador automático con el que se pueden ir comprobando las respuestas mientras se desarrolla (*autograder.py*). Para ejecutar el evaluador:

```
python3 autograder.py
```

El código de la práctica está compuesto por una serie de archivos Python, algunos de los cuales será necesario leer y entender para completar el trabajo. Todo este código está disponible en el Aula Virtual, desde donde se puede descargar.

[1] Esta práctica es una adaptación de una práctica de la Asignatura de Inteligencia Artificial de la Universidad de Berkeley, disponible aquí.

Archivos a editar	
`search.py`	Donde se implementarán los algoritmos de búsqueda.
`searchAgents.py`	Donde se implementarán los agentes basados en búsqueda.
Archivos que deberían consultarse	
`pacman.py`	El archivo principal para ejecutar juegos de Pacman. En este archivo se describe un tipo Pacman `GameState`, que se usará en el proyecto.
`game.py`	La lógica detrás de cómo funciona el mundo Pacman. Aquí se describen varios tipos de soporte como `AgentState`, `Agent`, `Direction` y `Grid`.
`utils.py`	Estructuras de datos útiles para la implementación de algoritmos de búsqueda.
Archivos que pueden ignorarse	
`graphicsDisplay.py`	Parte gráfica de Pacman.
`graphicsUtils.py`	Soporte para la parte gráfica de Pacman.
`textDisplay.py`	Gráficos ASCII para Pacman.
`ghostAgents.py`	Agente para controlar a los fantasmas.
`keyboardAgents.py`	Interfaces de teclado para controlar el Pacman.
`layout.py`	Código para leer archivos de mapas y guardar su contenido.
`autograder.py`	Autoevaluador.
`testParser.py`	Código para leer archivos de mapas y guardar su contenido.
`testClasses.py`	Clases de test para autoevaluación.
`test_cases/`	Carpeta que contiene casos de test para cada pregunta.
`searchTestClasses.py`	Clases de autoevaluación

Bienvenido a Pacman

Después de descargar el código, descomprimirlo y cambiar el directorio de trabajo a la carpeta del proyecto, se puede jugar a Pacman ejecutando el siguiente comando:

```
python3 pacman.py
```

Pacman vive en un mundo azul lleno de pasillos y golosinas redondas. El primer paso que debe completar Pacman dentro del mundo será navegar de forma eficiente. Dentro de `searhAgents.py`, el agente más simple se llama `GoWestAgent`, que siempre se dirige a la izquierda (un agente por reflejos). Este agente puede incluso ganar en alguna ocasión. Para ejecutarlo:

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

En cuanto girar se hace necesario, este agente empieza a tener problemas:

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

Si Pacman se queda atrapado, se puede salir del juego utilizando *CTRL-C* en la terminal.

Hay que tener en cuenta que `pacman.py` soporta una serie de opciones que pueden ser expresadas tanto en formato largo (`--layout`) como en corto (`-l`). Para ver la lista de opciones disponibles:

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

Todos los comandos se pueden encontrar en `commands.txt` para que el *copy/pasting* sea más sencillo.

Nueva sintaxis

Puede que no hayas visto esta sintaxis anteriormente:

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

Se trata de anotar el tipo que los argumentos en Python deberían esperar para la función. En el ejemplo que aparece debajo, `a` debería ser de tipo `int` (integer), `b` debería ser una `tuple` de 2 `int`s, `c` debería ser una `List` de `List`s de cualquier tipo (un array 2D de cualquier tipo), `d` es como si no estuviera anotado, ya que puede ser de cualquier tipo y `e` debería ser de tipo `float`. `e` incluso está predefinido como 1.0 si no recibe nada:

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

La llamada que aparece encima es acorde con la anotación de tipos, no pasándole nada para el parámetro e. Las anotaciones de tipos deberían ser una adición al docstring para ayudar al programador a conocer las funciones con las que trabaja. No es necesario hacer esto en Python, pero puede ser útil para mantener el código organizado.

Pseudocódigo para la búsqueda en grafo

Para la implementación de los algoritmos de búsqueda en las P1-4 se implementará el siguiente pseudocódigo para la búsqueda en grafo:

```
# Algoritmo: Búsqueda en grafo

frontier = {startNode}
expanded = {}
while frontier is not empty:
    node = frontier.pop()
    if isGoal(node):
        return path_to_node
    if node not in expanded:
        expanded.add(node)
        for each child of node's children:
            frontier.push(child)
return failed
```

Pregunta 1 (3 puntos): encontrar un punto de comida fijo utilizando búsqueda en profundidad

En `searchAgents.py` hay un `SearchAgent` implementado que planea todo el camino a través del mundo Pacman y va ejecutándolo paso a paso. Los algoritmos de búsqueda no están implementados y deberán ser implementados por el alumno.

Primero se puede ejecutar el siguiente comando para comprobar que `SearchAgent` funciona correctamente.

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

El comando le dice a `SearchAgent` que utilice `tinyMazeSearch` como algoritmo de búsqueda. El agente está implementado en `search.py`. Pacman debería navegar el laberinto correctamente.

Ahora se debería implementar el código genérico de las funciones de búsqueda para ayudar a Pacman a planear las rutas.

Nota: Todas las funciones de búsqueda deben devolver una lista de acciones que deben llevar al agente del punto de inicio al punto objetivo. Estas acciones deben ser movimientos legales.

Nota: Se tienen que utilizar las estructuras de datos `Stack`, `Queue` y `PriorityQueue` que aparecen en `util.py`. Estas implementaciones tienen propiedades particulares que son requeridas por el autoevaluador.

Pista: Los algoritmos son bastante parecidos, solo difiriendo en cómo se maneja la frontera. Consiguiendo este primer algoritmo, los siguientes serán más sencillos de conseguir.

Hay que implementar el algoritmo de búsqueda en profundidad (DFS) en la función `depthFirstSearch` dentro de `search.py`. Para hacer que el algoritmo sea completo, hay que escribir la versión en grafo de la búsqueda en profundidad evitando visitar nodos que ya se han visitado con anterioridad.

El código debería funcionar en los siguientes escenarios:

```
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 Pacman mostrará una superposición de los estados explorados y el orden en que fueron explorado (rojo más brillante para los explorados más temprano).

Pista: Si se utiliza `Stack` como estructura de datos, la solución para el algoritmo de búsqueda en profundidad en `mediumMaze` debería tener una longitud de 130 (teniendo en cuenta que se introduzcan los sucesores a la frontera en el orden que te da getSucessors; la longitud podría ser 246 si se hace en orden inverso). ¿La solución es la de menor coste? Si no es así, ¿qué está haciendo mal la búsqueda en profundidad?

Para comprobar el código, ejecuta los tests de autoevaluación con

```
python3 autograder.py -q q1
```

Para lanzar solamente uno de los tests:

```
python3 autograder.py -t test_cases/q1/graph_backtrack
```

Pregunta 2 (3 puntos): búsqueda en anchura

Implementar el algoritmo de búsqueda en anchura (BFS) en la función `breadthFirstSearch` en `search.py`. De nuevo, hay que escribir un algoritmo de búsqueda de grafo que no expanda nodos de estados previamente visitados.

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

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

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

Nota: si se ha escrito un algoritmo de búsqueda genérica, el algoritmo debería funcionar de igual forma para el problema del puzzle de 8 sin cambios.

```
python3 eightpuzzle.py
```

Para comprobar el código:

```
python3 autograder.py -q q2
```

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

Mientras que la búsqueda en anchura encuentra el camino con el menor número de acciones hasta el objetivo, puede que queramos encontrar el mejor camino aplicando otro criterio. Podemos considerar `mediumDottedMaze` y `mediumScaryMaze`.

Haciendo cambios en la función de coste, animamos a Pacman a encontrar otros caminos. Por ejemplo, podemos dar más peso a pasos peligrosos en zonas con muchos fantastams y menos para pasos en zonas con mucha comida. Un agente Pacman racional debería ajustar su comportamiento teniendo en cuenta estos hechos.

Implementar el algoritmo de búsqueda en grafo de coste uniforme en la función `uniformCostSearch` en `search.py`. Se puede revisar el archivo `util.py` para encontrar estructuras de datos que pueden ser útiles para esta implementación. Se debería observar ahora un comportamiento satisfactorio en los siguientes tres entornos, donde los agentes son todos UCS con el único cambio en la función de coste que utilizan. Los agentes y las funciones de coste serían:

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

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

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

Nota: se deberían obtener costes bajos para `StayEastSearchAgent` y altos para `StayWestSearchAgent`, debido a su función de coste exponencial (revisar `searchAgents.py` para entender los detalles).

Para comprobar el código, de nuevo:

```
python3 autograder.py -q q3
```

Pregunta 4 (3 puntos): búsqueda A*

Implementar la búsqueda A* en la función `aStarSearch` en `search.py`. A* recibe una función heurística como argumento. Las heurísticas reciben 2 argumentos: el estado en el problema de búsqueda (argumento principal) y el problema como tal (para información de referencia). La función heurística `nullHeuristic` en `search.py` es un ejemplo trivial.

Se puede probar la implementación A* en el problema original de encontrar un camino a través del laberinto hasta una posición fijada utilizando la heurística de la distancia Manhattan (implementada en la función `manhattanHeuristic` en `searchAgents.py`).

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

La búsqueda A* debería encontrar la solución óptima un poco más rápido que la búsqueda en anchura para este caso. ¿Qué pasa con `openMaze` para las diferentes estrategías de búsqueda?

Para comprobar el código, de nuevo:

```
python3 autograder.py -q q4
```

Pregunta 5 (3 puntos): encontrar todas las esquinas

El verdadero poder de A* solo se apreciará con un problema de búsqueda más complejo. En esta pregunta se pide formular un nuevo problema y diseñar la heurística para dicho problema.

En el nuevo problema, hay cuatro puntos, uno en cada esquina. Nuestro nuevo problema de búsqueda es encontrar el camino más corto a través del laberinto que toque los cuatro puntos (sin importar si hay comida ahí o no). Hay que tener en cuenta que para algunos laberintos como `tinyCorners`, el camino más corto no siempre va por la comida más cercana en primer lugar.

Pista: el camino más corto a través de `tinyCorners` tiene 28 pasos.

Nota: hay que completar primero la *Pregunta 2* antes de esta, ya que es una extensión.

Implementar el problema de búsqueda `CornersProblem` en `searchAgent.py`. Se tendrá que elegir una forma de representación de los estados que codifique toda la información necesaria para detectar si todas las esquinas se han alcanzado. Ahora el algoritmo de búsqueda debería resolver:

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

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

Para conseguir la puntuación completa se deberá definir una representación de los estados que no codifique información irrelevante como la posición de los fantasmas, donde está la comida, etc. En particular, no hay que usar el `GameState` de Pacman como un estado de búsqueda.

Pista: las únicas partes del estado del juego que se deben referenciar son la posición de inicio y la posición de las cuatro esquinas.

Pista: al implementar `getSuccessors`, hay que asegurarse de añadir cada nodo hijo a la lista de hijos con coste 1.

Pista: se deberían guardar los estados en formato tupla (x,y), __). Se deberá decidir qué información se debe guardar en el espacio en blanco.

Para comprobar la solución:

```
python3 autograder.py -q q5
```

Pregunta 6 (3 puntos): heurística del problema de las esquinas

Implementar una heurística no trivial y consistente para el `CornersProblem` en `cornersHeuristic`.

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

El `AStarCornersAgent` es una abreviación de

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

Admisibilidad vs. Consistencia: las heurísticas son funciones que toman estados de búsqueda y devuelven números que estiman el coste al objetivo más próximo. Las heurísticas más eficientes devolverán valores cercanos al coste del objetivo actual. Para que sean admisibles, los valores de las heurísticas deben ser límites inferiores en el coste del camino más corto al objetivo más cercano (y no negativos). Para ser consistentes, tiene que sostener

que si una acción tiene coste *c*, entonces tomar esa acción solo puede causar una caída en la heurística de como máximo *c*.

La admisibilidad no es suficiente para garantizar que sea correcto en la búsqueda en grafo. Sin embargo, las heurísticas admisibles suelen ser además consistentes, especialmente si son derivadas de relajaciones del problema. Por lo tanto, suele ser más fácil empezar con una serie de heurísticas admisibles. Una vez se tiene una admisible que funciona correctamente, se puede comprobar si es además consistente. La única forma de garantizar la consistencia es con una prueba. Se puede verificar para una heurística comprobando que para cada nodo que se expande, sus nodos hijos son iguales o inferiores en su valor f. Si esta conducción no se cumple para todos los nodos, entonces la heurística es inconsistente. Además, si UCS (A* con la heurística 0) y A* devuelven en algún momento caminos de diferente longitud, la heurística será inconsistente.

Heurísticas no triviales: La heurísticas triviales son las que devuelven valores de cero siempre (UCS) y la heurística que calcula el coste real de terminación. La primera no ahorrará tiempo mientras que la segunda hará que el autoevaluador devuelva un *timeout*. Se buscará entonces una heurística que reduzca el tiempo total de computo, aunque para esta práctica solo se verificará el recuento de nodos, además de cumplir un límite de tiempo razonable.

Calificación: La heurística debe ser no trivial, no negativa y consistente para recibir puntuación. Se debe comprobar que devuelve 0 en cada estado objetivo y que nunca devuelve un valor negativo. Dependiendo de la cantidad de nodos que se expanda, se calificará de la siguiente forma:

Número de nodos expandidos	**Calificación**
> 2000	0/3
Como mucho 2000	1/3
Como mucho 1600	2/3
Como mucho 1200	3/3

Para comprobar la solución:

```
python3 autograder.py -q q6
```

Pregunta 7 (4 puntos): comer todos los puntos

Ahora se resolverá un problema de búsqueda complejo: comer toda la comida en el menor número de pasos posible. Para esto, se deberá implementar una nueva definición de un problema de búsqueda que formalice el problema de comer toda la comida en `FoodSearchProblem` en `searchAgents.py`. Una solución se define como el camino que recoja

toda la comida en el mundo Pacman. Para el proyecto presente, las soluciones no tienen en cuenta ningún fantasma ni las cápsulas de poder. La solución solo depende del emplazamiento de las paredes, la comida y el Pacman. Si se ha escrito el método de búsqueda general correctamente, A* con una heurística nula debería encontrar una solución óptima rápidamente para el caso de `testSearch` sin cambios en el código.

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

El `AStarFoodSearchAgent` es una abreviación de

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

Se debería ver que UCS empieza a ralentizarse incluso con el simple `tinySerch`.

Hay que completar `foodHeuristic` en `searchAgents.py` con una heurística consistente para el `FoodSearchProblem`. Se puede probar el agente en el tablero `trickySearch` con:

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

Una heurística no trivial, no negativa y consistente recibirá 1 punto de calificación. Se debe comprobar que devuelve 0 en cada estado objetivo y que nunca devuelve un valor negativo. Dependiendo de la cantidad de nodos que se expanda, se calificará de la siguiente forma:

Número de nodos expandidos	**Calificación**
> 15000	1/4
Como mucho 15000	2/4
Como mucho 12000	3/4
Como mucho 9000	4/4
Como mucho 7000	5/4

Para comprobar la solución:

```
python3 autograder.py -q q7
```

Pregunta 8 (3 puntos): búsqueda subóptima

En determinados casos, incluso con A* y una heurística buena, encontrar el camino óptimo a través de todos los puntos es complejo. En estos casos, se tratará de encontrar un camino que sea razonablemente bueno rápidamente. En esta sección se buscará implementar un agente que siempre coma el punto más cercano de forma codiciosa. `ClosestDotSearchAgent` está

implementado en `searchAgent.py` pero le falta la función clave que encuentra el punto más cercano.

Hay que implementar la función `findPathToClosestDot` en `searchAgents.py`. El agente puede resolver este laberinto en menos de 1 segundo con un coste de 350. Para ejecutar:

```
python3 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Pista: la forma más rápida de completar `findPathToClosestDot` es completando `AnyFoodSearchProblem`, al que le falta su objetivo de prueba, así que completando las dos funciones se tendría la solución.
El `findPathToClosestDot` no siempre encontrará el camino más corto a través del laberinto.

Para comprobar la solución:

```
python3 autograder.py -q q8
```

Entregables

- Archivos de código Python `search.py` y `searchAgents.py`.
- El código tiene que ir obligatoriamente comentado explicando su funcionalidad. Debe ser legible y estar debidamente tabulado.
- Se utilizarán sistemas anticopia y se podrá requerir explicación individual de la práctica en caso de duda.
- Esta entrega se realizará en formato zip (incluyendo los archivos .py) vía Aula Virtual.
- Fecha de entrega: la entrega de la práctica se realizará **el 14 de octubre** al finalizar la clase de prácticas (11:00-13:00). Durante la sesión, se propondrá una pequeña modificación a la práctica que deberá ser también entregada por los alumnos, por lo que la asistencia ese día será obligatoria.