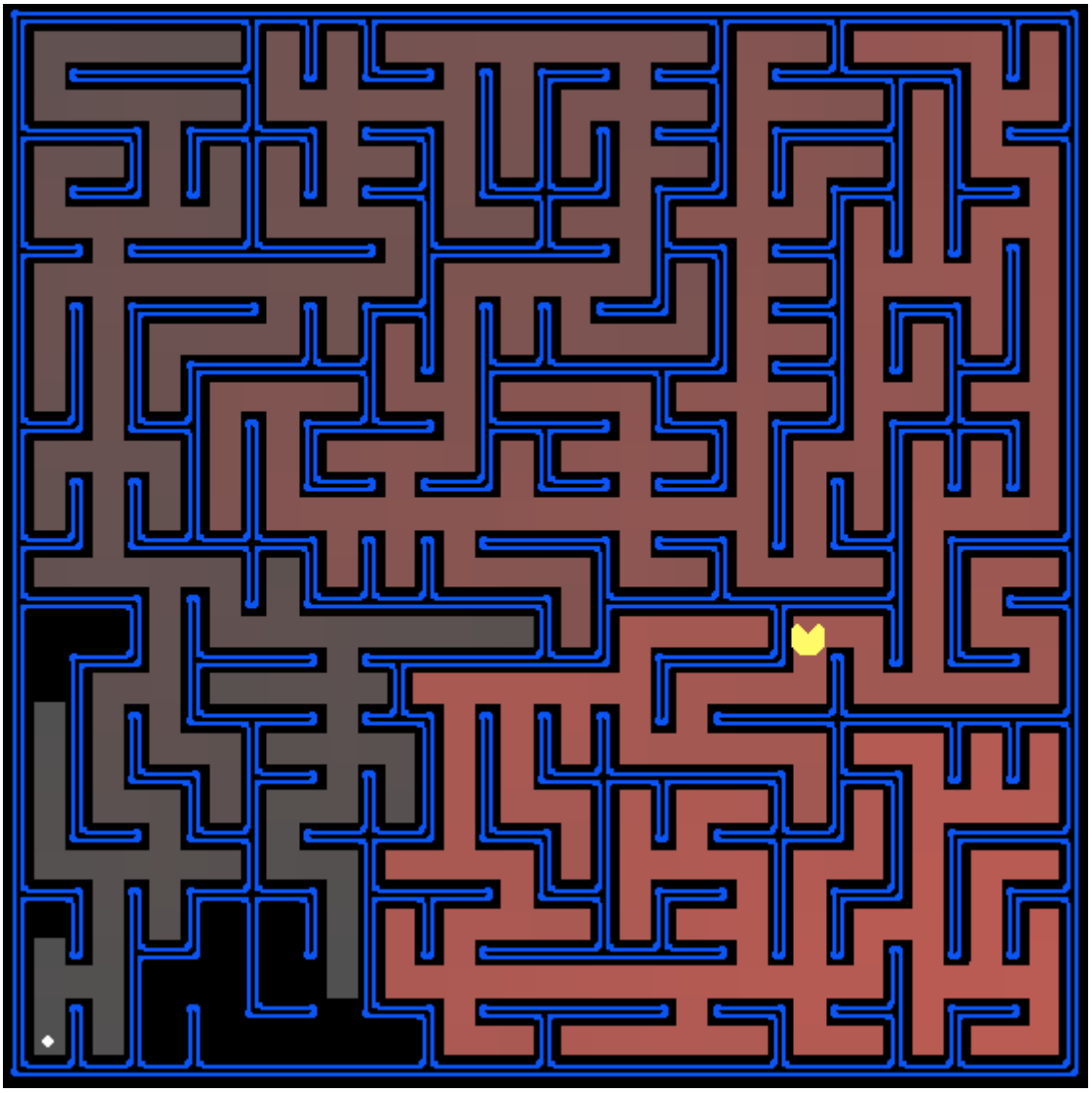
Práctica 1: Búsqueda

Versión 3.00. Última actualización: 2023-09-19

Fecha de publicación: 2023-09-19
Fecha de entrega: miércoles, 2023-10-11 23:59



Todas esas paredes de colores,
Los laberintos ponen mohíno a Pacman,
¡Enséñale a buscar!

Importante

Antiplagio, copia y publicación de resultados

Como dice el descargo de responsabilidad de Berkeley, puede utilizarse la infraestructura y los proyectos de Pac-Man para cualquier uso tanto académico como personal. Sin embargo, **está estrictamente prohibido distribuir o publicar soluciones de cualquiera de los proyectos**. Asimismo, en cumplimiento con el REGLAMENTO DE EVALUACIÓN ACADÉMICA DE LA ESCUELA POLITÉCNICA SUPERIOR DE LA UNIVERSIDAD DE AUTÓNOMA DE MADRID, artículo 14; "En el caso de **copia**, la asignatura se puntuará en la convocatoria donde se produjo la copia con cero puntos. Como medida adicional, el profesor puede iniciar un expediente informativo, de acuerdo con el Reglamento de Evaluación de la UAM".

Método de entrega

Las prácticas se harán por parejas.

Cada envío debe constar de un único archivo zip llamado **p1_[gggg]_[mm]_[apellido1]_[apellido2].zip**, donde gggg es el número de vuestro grupo de laboratorio y mm el identificador asignado a cada persona/pareja en este grupo. Este archivo zip debe enviarse a través de Moodle.

Es importante, siempre, poner los nombres de los autores y el número del grupo de prácticas en todos los archivos que envíes: memoria y código.

Material a entregar

El fichero zip a entregar debe contener **una única carpeta** con el mismo nombre que el zip pero sin la extensión. Esta carpeta contendrá:

- El *código* de la práctica, incluyendo **únicamente** los ficheros modificados con vuestro código que se indican más abajo. En esta carpeta deben aparecer completadas las secciones de código que en la versión entregada al alumno se indican como "***YOUR CODE HERE /TU CÓDIGO AQUÍ***". No incluyáis ningún otro fichero de código.
- Un fichero *autograder_results.txt* con el resultado de ejecutar el autocalificador
- Una memoria de la práctica (en formato PDF) que incluya: i) un resumen de los

conceptos vistos en la práctica; ii) para cada ejercicio/sección a completar en la práctica:

1. un comentario personal sobre cómo se decidió abordar el problema planteado y por qué, en concreto, una descripción de la estructura de datos utilizada para representar el nodo de búsqueda y si ha sido necesario modificarla a medida que se realizaba la práctica para facilitar la implementación de alguno de los algoritmos de búsqueda junto con una explicación de las alternativas que se han barajado y las razones por las cuales se ha elegido dicha representación;
2. lista y explicación de las funciones del framework que se han utilizado;
3. el código introducido por el alumno comentando su funcionalidad;
4. capturas de pantalla de las pruebas realizadas para verificar su correcto funcionamiento;
5. explicación de las conclusiones del comportamiento observado, centrándose en la eficiencia del algoritmo correspondiente (número de nodos que expande, si llega a la solución óptima, si es óptimo, etc.);
6. respuestas a las preguntas de cada sección.

Se recomienda que los resultados o comparaciones incluidas en la memoria se muestren **en forma de tablas**. Igualmente, es aconsejable que los razonamientos sobre complejidad no se hagan en base a un solo mapa, sino, en la medida de lo posible, sean *razonamientos generales*, bien aplicando aspectos teóricos o habiendo revisado varios ejemplos de mapas.

Calificación

La calificación se dividirá entre la memoria (40%) y el código en sí (60%). Ambos elementos se evaluarán dentro de un rango de 0-10.

Descripción de las tareas a realizar

La tarea a realizar en esta práctica está relacionada con la parte del curso dedicada a la búsqueda. Para ello utilizaremos una versión modificada del software proporcionado por la Universidad de Berkeley. El software específico para realizar nuestra práctica está en Moodle. El lenguaje de programación es Python.

Esta práctica consta de 6 ejercicios.

Empecemos...

Introducción

En este proyecto, tu agente Pacman encontrará caminos a través de su mundo laberíntico, tanto para llegar a un lugar en particular como para recolectar comida de manera eficiente. Construirá algoritmos de búsqueda generales y los aplicará a escenarios de Pacman. Este proyecto incluye un autocalificador (autograder) para que califique tus respuestas en la máquina. Este se puede ejecutar con el comando:

```
python autograder.py
```

El tutorial de Python enlazado arriba contiene también información sobre el uso del autograder. El código de este proyecto consta de varios archivos de Python, algunos de los cuales deberéis leer y comprender para completar las tareas, y algunos de los cuales podéis ignorar. Podéis descargar todo el código y los archivos de apoyo como archivo zip desde Moodle.

Ficheros a editar	
`search.py`	Dónde residirán todos vuestros algoritmos de búsqueda.
`searchAgents.py`	Dónde residirán todos vuestros agentes basados en búsquedas.
Archivos que "quizás" queráis ver:	
`pacman.py`	El archivo principal que ejecuta los juegos de Pacman. Este archivo describe un tipo GameState para Pacman, que se usa en este proyecto.
`game.py`	La lógica detrás de cómo funciona el mundo Pacman. Este archivo describe varios tipos de soporte como AgentState, Agent, Direction y Grid
`util.py`	Estructuras de datos útiles para implementar algoritmos de búsqueda.

Ficheros a editar	
Archivos de apoyo que podéis ignorar:	
`graphicsDisplay.py`	Gráficos para Pacman.
`graphicsUtils.py`	Soporte para gráficos Pacman.
`textDisplay.py`	Gráficos ASCII para Pacman.
`ghostAgents.py`	Agentes para controlar fantasmas.
`keyboardAgents.py`	Interfaces de teclado para controlar Pacman.
`layout.py`	Código para leer archivos de diseño y almacenar su contenido.
`autograder.py`	El calificador automático.
`testParser.py`	Analiza los archivos de prueba y solución del autograder.
`testClasses.py`	Clases generales de prueba de calificación automática.
`test_cases /`	Directorio que contiene los casos de prueba para cada pregunta.
`searchTestClasses.py`	Clases de prueba de autocalificación específicas para esta práctica.

Archivos para editar y enviar: Deberéis completar partes de `search.py` y `searchAgents.py` durante la tarea. Una vez completada la tarea, incluid estos ficheros junto con el resto para subirlos a Moodle como se describe más arriba. *No* cambiéis los otros archivos de esta distribución.

Evaluación del código: Vuestro código se calificará automáticamente para su corrección técnica. Por favor *no* cambiéis los nombres de las funciones o clases provistas dentro del código, o causará estragos en el autograder. Sin embargo, la **exactitud de su implementación**, no los resultados del autocalificador, serán lo que defina vuestra calificación. Revisaremos y calificaremos las tareas individualmente para asegurarnos de que reciben el debido crédito por su trabajo.

Deshonestidad académica: Comprobaremos vuestro código con otras entregas en la clase para verificar la redundancia lógica. Si copiáis el código de otra persona y lo enviáis con cambios menores, lo sabremos. Estos detectores de trampas son bastante difíciles de engañar, así que no lo intentéis. Confiamos en que todos vosotros enviareis únicamente vuestro propio trabajo; *por favor* no nos defraudéis. Si lo hacéis, utilizaremos los recursos disponibles más severos.

Obtener ayuda: ¡No estás solo! Si os encontráis atascados en algo, comunicádnoslo a los profesores para poder obtener ayuda. Las tutorías y el foro de discusión están disponibles para vuestro apoyo; por favor utilizadlos. Queremos que estos proyectos sean gratificantes e instructivos, no frustrantes ni desmoralizadores. Pero no sabemos cuándo ni cómo ayudar a menos que nos lo pidáis!

OJO: Tened cuidado de no publicar spoilers en foros, etc. * * *

Bienvenidos a Pacman

Después de descargar el código (fichero search.zip), descomprimirlo y cambiar al directorio, deberíais poder jugar un juego de Pacman escribiendo lo siguiente en la línea de comandos:

```
python pacman.py
```

Pacman vive en un mundo azul brillante de pasillos retorcidos y deliciosos bocadillos redondos. Navegar por este mundo de manera eficiente será el primer paso de Pacman para afianzar su dominio. El agente más simple en `searchAgents.py` se llama `GoWestAgent` , que siempre va hacia el oeste (un agente reflejo trivial). Este agente puede ganar ocasionalmente:

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

Pero las cosas se ponen feas para este agente cuando se requiere girar:

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

Si Pacman se atasca, podéis salir del juego escribiendo CTRL-c en el terminal. Pronto, vuestro agente resolverá no solo “tinyMaze”, sino cualquier laberinto que desee. Tened en cuenta que `pacman.py` admite una serie de opciones que pueden expresarse de forma larga (por ejemplo, `--layout`) o corta (por ejemplo, `-l`). Podéis ver la lista de todas las

opciones y sus valores predeterminados a través de:

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

Además, todos los comandos que aparecen en este proyecto también aparecen en `commands.txt`, para copiar y pegar fácilmente. En UNIX / Mac OS X, incluso podéis ejecutar todos estos comandos en orden con `bash commands.txt`.

Una función útil que puede interesaros es la opción frameTime (en segundos), que controla la velocidad de Pacman. Modificarlo puede ayudaros a depurar o simplemente a ver mejor el juego. Por ejemplo:

```
python pacman.py --frameTime 0.5
```

Sección 1: Encontrar un punto de comida fijo usando la búsqueda primero en profundidad

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

Primero, probad que el `SearchAgent` esté funcionando correctamente ejecutando:

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

El comando anterior le dice al `SearchAgent` que use `tinyMazeSearch` como su algoritmo de búsqueda, que está implementado en `search.py`. Pacman debería navegar por el laberinto con éxito. ¡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 escribiréis se puede encontrar en las diapositivas de las clases teóricas.

Nota importante: No es necesario construir el árbol de búsqueda de manera explícita. Además del estado del sistema que corresponde a una etapa en la búsqueda (**estado de búsqueda**), el **nodo de búsqueda** debe incluir información adicional. En concreto, debería ser posible tener acceso a la trayectoria desde el inicio hasta el estado actual a partir de esta información.

Nota importante: Las funciones de búsqueda implementadas deben devolver una lista de

acciones que llevan al agente desde el inicio hasta la meta. Estas acciones deben ser movimientos legales (direcciones válidas, no moverse a través de las paredes). En el caso de que la búsqueda falle (el algoritmo no encuentra solución), se debe devolver `None`.

Sugerencia: Todos los algoritmos son muy parecidos. Las implementaciones para búsqueda en profundidad (BP), en anchura (BA), de coste uniforme (BCU) y A* difieren solo en los detalles de cómo se gestionan los nodos a expandir. Por lo tanto, concentraos en hacer bien DFS y el resto debería ser relativamente sencillo. Hay que diseñar un único método de búsqueda genérico, el cual es configurado con una estrategia de encolado específica para cada algoritmo.

Nota importante: ¡Aseguraos de utilizar las estructuras de datos `Stack`, `Queue` y `PriorityQueue` definidas en `util.py`! Estas implementaciones de las estructuras de datos correspondientes están diseñadas para que funcione el corrector automático y, además, facilitan el diseño del código.

Nota importante: Para garantizar que la búsqueda es óptima, se debe comprobar si los estados han sido visitados con anterioridad **en el momento en el que son considerados para ser expandidos**, no cuando son generados.

Implementad el algoritmo de búsqueda en profundidad (BP) en la función `depthFirstSearch` en `search.py`. Para que vuestro algoritmo sea *completo*, escribid la versión de búsqueda en profundidad en grafo, que evita expandir los estados ya visitados. Vuestro código debería 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
```

*__Recordad__ imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

El tablero de Pacman mostrará una superposición de los estados explorados y el orden en el que fueron explorados (un rojo más brillante significa que se han explorado antes).
Pregunta 1.1: ¿El orden de exploración es el que esperabais? ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?

Sugerencia: Si usáis un `Stack` como su estructura de datos, la solución encontrada por

vuestro algoritmo BP para `mediumMaze` debería tener una longitud de 130 (siempre que añadáis los sucesores en el orden proporcionado por getSuccessors; podría obtener 246 si los añadís en el orden inverso).
Pregunta 1.2: ¿Es esta una solución de menor coste? Si no es así, pensad qué está haciendo mal la búsqueda en profundidad.

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba del corrector automático.

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

Sección 2: Búsqueda en anchura

Implementad el algoritmo de búsqueda primero en anchura (BA) en la función `breadthFirstSearch` en `search.py` . Nuevamente, escribid un algoritmo de búsqueda en grafo que evite expandir los estados ya visitados. Probad vuestro código de la misma manera que lo hicisteis para la búsqueda en profundidad.

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

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

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

*__Recordad__ incluir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

Pregunta 2.1: ¿BA encuentra una solución de menor coste? Si no es así, verificad vuestra implementación.

Pista: Si Pacman se mueve demasiado lento para ti, prueba la opción `--frameTime 0` .

Nota: Si habéis escrito el código de búsqueda de forma genérica, el código debería funcionar igualmente bien para el problema de búsqueda del ocho puzzle sin ningún cambio.

```
python eightpuzzle.py
```

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba de autograder.

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

Sección 3: Variar la función de coste

Si bien BA encontrará un camino con el menor número de acciones hacia la meta, es posible que deseemos encontrar caminos que sean "mejores" en otros sentidos. Considerad `mediumDottedMaze` y `mediumScaryMaze`. Al cambiar la función de coste, podemos alentar a Pacman a encontrar diferentes caminos. Por ejemplo, podemos considerar más costosos pasos peligrosos en áreas plagadas de fantasmas o menos por pasos en áreas ricas en alimentos, y un agente Pacman racional debería ajustar su comportamiento en respuesta.

Implementad el algoritmo de búsqueda en grafo de coste uniforme (BCU) en la función `uniformCostSearch` en `search.py`. Os recomendamos que busquéis en `util.py` algunas estructuras de datos que puedan ser útiles en su implementación. Deberíais observar un comportamiento exitoso en los tres siguientes diseños, donde los agentes utilizados son todos agentes BCU que difieren solo en la función de coste que utilizan (los agentes y las funciones de coste ya se incluyen el el código proporcionado):

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

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

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

*__Recordad__ imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

Nota: Deberíais obtener costes de ruta muy bajos y muy altos para `StayEastSearchAgent` y `StayWestSearchAgent`, respectivamente, debido a sus funciones de coste exponencial (consultad `searchAgents.py` para más detalles).

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba del corrector automático.

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

Sección 4: Búsqueda A*

Implementad una búsqueda de grafo A* en la función 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 (como información de referencia). La función heurística `nullHeuristic` en `search.py` es un ejemplo trivial.

Podéis probar vuestra implementación de A* en el problema original de encontrar un camino a través de un laberinto hasta una posición fija usando la heurística de distancia de Manhattan (implementada ya como `manhattanHeuristic` en `searchAgents.py`).

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

*__Recordad__ imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

Deberíais ver que A* encuentra la solución óptima un poco más rápido que la búsqueda de coste uniforme (aproximadamente 549 frente a 620 nodos de búsqueda expandidos en nuestra implementación, pero los empates en la prioridad pueden hacer que los números difieran ligeramente).
Pregunta 4.1: ¿Qué sucede en `openMaze` para las diversas estrategias de búsqueda?

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba del corrector automático.

```
python autograder.py -q q4
```

Sección 5: Encontrar todas las esquinas

El verdadero poder de A* solo será evidente con un problema de búsqueda más desafiante. Ahora es el momento de formular un nuevo problema y diseñar una heurística para él. En *corner mazes*, hay cuatro puntos, uno en cada esquina. Nuestro nuevo problema de búsqueda es encontrar el camino más corto a través del laberinto que toca las cuatro

esquinas (ya sea si el laberinto tiene comida o no). Tened en cuenta que para algunos laberintos como `tinyCorners`, ¡el camino más corto no siempre va primero a la comida más cercana! *Pista*: el camino más corto a través de `tinyCorners` toma 28 pasos.

Nota: Aseguraos de completar la sección 2 antes de trabajar en la sección 5, porque la sección 5 se basa en vuestra respuesta a la sección 2.

Implementad el problema de búsqueda `CornersProblem` en `searchAgents.py`. debéis elegir (**y justificar en la memoria**, incluyendo cualquier alternativa que se hubiera descartado) una representación del estado del sistema (no confundir con el *nodo de búsqueda*) que codifique toda la información necesaria para detectar si se han alcanzado las cuatro esquinas. Ahora, vuestro 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=CornersProbler
```

*__Recordad__ imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

Para recibir la puntuación máxima, debéis definir una representación de estados abstracta que *no* codifique información irrelevante (como la posición de los fantasmas, dónde está la comida extra, etc.). En particular, no utilicéis un Pacman `GameState` como estado de búsqueda. Vuestro código será muy, muy lento si lo hacéis (y también incorrecto).

Pista 1: Las únicas partes del estado del juego que debéis hacer referencia en vuestra implementación son la posición inicial de Pacman y la ubicación de las cuatro esquinas.

Pista 2: Cuando codifiquéis `getSuccessors`, aseguraos de agregar hijos a la lista de sucesores con un coste de 1. Nuestra implementación de `breadthFirstSearch` expande poco menos de 2000 nodos de búsqueda en `mediumCorners`. Sin embargo, la heurística (utilizada con la búsqueda A*) puede reducir la cantidad de búsqueda requerida.

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba del corrector automático.

```
python autograder.py -q q5
```

Sección 6: Problema de las esquinas: heurística

Nota: Aseguraos de completar la sección 4 antes de trabajar en la sección 6, porque la sección 6 se basa en vuestra respuesta a la sección 4. Implementad una heurística consistente y no trivial para el `CornersProblem` en `cornersHeuristic`.

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

Nota: `AStarCornersAgent` es un atajo para

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

*__Recordad__ imprimir capturas de pantalla de estos tests y explicar lo que habéis observado para la memoria de la práctica.

Admisibilidad frente a consistencia: Recordad que una heurística es simplemente una función que toma como argumento un estado de búsqueda y devuelve una estimación el coste de la trayectoria óptima (de menor coste) desde ese estado hasta una meta. Las heurísticas más eficaces devolverán valores más cercanos a los costes reales del objetivo. Para ser *admisible*, las estimaciones proporcionadas por la heurística deben ser cotas inferiores del coste de la trayectoria óptima (de menor coste) hasta una meta. Para ser *consistente*, además debe cumplir que, si una acción tiene un coste *c*, tomar esa acción solo puede causar una reducción del coste estimado por la heurística de, como máximo, *c*. Recordad que la admisibilidad no es suficiente para garantizar optimalidad que la búsqueda en grafo; es necesario una condición más restrictiva: que la heurística sear consistente (monótona). Afortunadamente, **las heurísticas que se derivan de relajaciones del problema no solo son admisibles, sino que también son consistentes**.

Nota importante: La heurística debe ser consistente para cualquier configuración del laberinto, no solo para los laberintos concretos considerados.

Nota importante: Es posible diseñar heurísticas inconsistentes para las cuales el número de nodos expandidos es inferior al obtenido con heurísticas consistentes. Puede que estas respuestas superen los tests del corrector automático. Sin embargo, no son correctas, y recibirán una puntuación inferior a la máxima.

Heurísticas no triviales: Las heurísticas triviales son las que devuelven cero para todos los estados (BCU) y la heurística que computa el coste optimo para cada uno los estados de búsqueda. La primera no reduce el coste computacional del algoritmo. Para calcular la segunda, es necesario resolver tantos problemas de búsqueda como estados de búsqueda haya. En general, el corrector automático se detendrá por tener tal cálculo un coste

computacionalexcesivo. El objetivo es diseñar una heurística que reduzca el tiempo de cómputo total, que incluye no solo el coste de la búsqueda, sino también el coste del cálculo de la heurística. Busca relajaciones del problema en las cuales los costes de las trayectorias óptimas puedan ser computados de manera eficiente. En esta práctica, el corrector automático solo comprueba el número de nodos (aparte de imponer un límite en el tiempo de cómputo, de forma que este se mantenga dentro de lo razonable).

Pregunta 6.1: Describe el proceso que se ha seguido para diseñar la heurística y explica la lógica de la misma.

Calificación: Vuestra heurística debe ser una heurística consistente, no trivial y no negativa, para recibir puntos. Aseguraos de que vuestra heurística devuelva 0 en cada estado objetivo y nunca devuelva un valor negativo. Dependiendo de la cantidad de nodos que expanda vuestra heurística, se os calificará de acuerdo con el siguiente baremo:

Número de nodos expandidos	Nota
más de 2000	0/3
como máximo 2000	1/3
como máximo 1600	2/3
como máximo 1200	3/3

Recordad: Si vuestra heurística es inconsistente, vuestra puntuación en esta pregunta será 0, ¡así que tened cuidado!

Corrección: Ejecutad el siguiente comando para ver si vuestra implementación pasa todos los casos de prueba del corrector automático.

```
python autograder.py -q q6
```