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 una heurística 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 para 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 de estado abstracta 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 la posición inicial de Pacman y la ubicación de las cuatro esquinas.

Nuestra implementación de breadthFirstSearch expande menos de 2000 nodos de búsqueda en mediumCorners. Sin embargo, la heurística (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 para la pregunta 4.

Debemos implementar un heurístico no trivial y consistente 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 las heurísticas son solo funciones que toman estados de búsqueda y devuelven números que estiman el coste al objetivo más cercano. Una heurística más efectiva devolverá valores más cercanos a los costes reales del objetivo. Para ser admisible, los valores heurísticos deben ser límites inferiores en 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 la heurística 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, las heurísticas 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 heurísticas admisibles. Una vez que tengamos una heurística 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, nuestra heurística es inconsistente. ¡Esto es complicado!

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

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

Número de nodos expandos	**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 de problema de búsqueda que formalice el problema de eliminación de alimentos: FoodSearchProblem en searchAgents.py (implementado para nosotros). 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 regular 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 una heurística nula (equivalente a la búsqueda de coste uniforme) debería rápidamente encontrar una solución óptima para testSearch sin cambio de código por nuestra parte (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: Asegurarse 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 una heurística 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 heurística consistente no negativa no trivial recibirá 1 punto. Debemos asegurar que nuestra heurística devuelve 0 en cada estado objetivo y nunca devuelve un valor negativo. Dependiendo de los pocos nodos que expanda nuestra heurística, obtendremos puntos adicionales:

Número de nodos expandos	**Puntuación**
Más de 15000	1/4
Como mucho 15000	2/4
Como mucho 12000	3/4
Como mucho 9000	4/4 (puntuación total; medio)
Como mucho 7000	5/4 (puntuación extra opcional; difícil)

Recordad: Si nuestro heurístico es inconsistente, 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 inconsistente.

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

A veces, incluso con A* y una buena heurística, 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 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:

- Documentación en la que se presentan los problemas abordados y su solución, con una explicación razonada de la solución empleada (estructuras de datos, aspectos reseñables, ...)

- Resultados para los casos de prueba dados

- Resultados del autograder