Trabajo Práctico 1

Unidad 1 - Introducción a la IA
Unidad 2 - Resolucón de problemas

Problema 1 - Grados de separacion

Objetivo

Escriba un programa que determine cuántos "grados de separación" hay entre dos actores.

Según el juego Grados de separación de Kevin Bacon, cualquier persona en la industria cinematográfica de Hollywood puede conectarse con Kevin Bacon en seis pasos, donde cada paso consiste en encontrar una película protagonizada por dos actores.

En este problema, nos interesa encontrar el camino más corto entre dos actores cualesquiera eligiendo una secuencia de películas que los conecte. Por ejemplo, el camino más corto entre Jennifer Lawrence y Tom Hanks es 2: Jennifer Lawrence está conectada con Kevin Bacon porque ambos protagonizan *X-Men: First Class*, y Kevin Bacon está conectado con Tom Hanks porque ambos protagonizan *Apollo 13*

Podemos enmarcar esto como un problema de búsqueda: nuestros estados son personas. Nuestras acciones son películas, que nos llevan de un actor a otro (es cierto que una película podría llevarnos a varios actores diferentes, pero eso está bien para este problema). Nuestro estado inicial y nuestro estado objetivo están definidos por las dos personas que intentamos conectar. Al utilizar la búsqueda en amplitud, podemos encontrar el camino más corto de un actor a otro.

Se adjunta un comprimido con el codigo inicial para trabajar `problema1-gds.zip`. Ejemplo de funcionalidad:

```
$ python degrees.py large
Loading data...
Data loaded.
Name: Emma Watson
Name: Jennifer Lawrence
3 degrees of separation.
1: Emma Watson and Brendan Gleeson starred in Harry Potter and
the Order of the Phoenix
2: Brendan Gleeson and Michael Fassbender starred in Trespass Against Us
3: Michael Fassbender and Jennifer Lawrence starred in X-Men: First Class
```

Explicación del problema

El código contiene dos conjuntos de archivos de datos CSV: uno en el directorio `large` y otro en el directorio `small`. Cada uno contiene archivos con los mismos nombres y la misma estructura, pero es un conjunto de datos mucho más pequeño para facilitar las pruebas y la experimentación.

Cada conjunto de datos consta de tres archivos **CSV**. Un archivo CSV (**C**omma **S**eparated **V**alues), si no está familiarizado, es solo una forma de organizar datos en un formato basado en texto: cada fila corresponde a una entrada de datos, con comas en la fila que separan los valores de esa entrada.

Abra `small/people.csv`. Podra ver que cada persona tiene una identificación única, que se corresponde con su identificación en la base de datos de IMDb. También tienen un nombre y un su año de nacimiento.

A continuación, abra `small/movies.csv`. Verá aquí que cada película también tiene una identificación única, además de un título y el año en que se estrenó la película.

Ahora, abre `small/stars.csv`. Este archivo establece una relación entre las personas en `people.csv` y las películas en movies.csv. Cada fila es un par de un valor *person_id* y un valor *movie_id*. La primera fila (ignorando el encabezado), por ejemplo, indica que la persona con ID 102 protagonizó la película con ID 104257. Al comparar eso con `people.csv` y `movies.csv`, encontrarás que esta línea dice que Kevin Bacon Protagonizó la película .[A] few good man".

A continuación, revise `degrees.py`. En la parte superior, se definen varias estructuras de datos para almacenar información de los archivos CSV. El diccionario de nombres es una forma de buscar una persona por su nombre: asigna nombres a un conjunto de identificadores correspondientes (porque es posible que varios actores tengan el mismo nombre). El diccionario de personas asigna la identificación de cada persona a otro diccionario con valores para el nombre de la persona, el año de nacimiento y el conjunto de todas las películas que han protagonizado. Y el diccionario de películas asigna la identificación de cada película a otro diccionario con valores para el título de esa película, año de estreno y el conjunto de todas las estrellas de la película. La función **load_data** carga datos de los archivos CSV en estas estructuras de datos.

La función principal de este programa primero carga datos en la memoria (el directorio desde el que se cargan los datos se puede especificar mediante un argumento de línea de comando). Luego, la función solicita al usuario que escriba dos nombres. La función **person_id_for_name** recupera la identificación de cualquier persona (y se encarga de solicitar al usuario que aclare, en caso de que varias personas tengan el mismo nombre). Luego, la función llama a la función **short_path** para calcular el camino más corto entre las dos personas e imprime el camino.

Sin embargo, la función **short_path** no esta implementada, y es lo que debemos resolver.

Especificación

Complete la implementación de la función **short_path** de modo que devuelva la ruta más corta desde la persona con la identificación de origen hasta la persona con la identificación de destino.

- Suponiendo que hay una ruta desde el origen hasta el destino, su función debería devolver una lista, donde cada elemento de la lista es el siguiente par *(movie_id, person_id)* en la ruta desde el origen hasta el destino. Cada par debe ser una tupla de dos cadenas.
    - Por ejemplo, si el valor de retorno de **short_path** fuera `[(1, 2), (3, 4)]`, eso significaría que la fuente protagonizó la película 1 con la persona 2, la persona 2 protagonizó la película 3 con la persona 4 y la persona 4 es el objetivo.
- Si hay varias rutas de longitud mínima desde el origen al destino, su función puede devolver cualquiera de ellas.
- Si no hay una ruta posible entre dos actores, su función debería devolver None.
- Puede llamar a la función **neighbours_for_person**, que acepta la identificación de una persona como entrada y devuelve un conjunto de pares *(movie_id, person_id)* para todas las personas que protagonizaron una película con una persona determinada.

No debe modificar nada más en el archivo que no sea la función short_path, aunque puede escribir funciones adicionales y/o importar otros módulos de la biblioteca estándar de Python.

Consejos

- Si bien la implementación de la búsqueda en las clases busca un objetivo cuando un nodo sale de la frontera, usted puede mejorar la eficiencia de su búsqueda verificando un objetivo a medida que se agregan nodos a la frontera: si detecta un nodo objetivo, no es necesario agregarlo a la frontera, simplemente puede devolver la solución inmediatamente.
- Le invitamos a "tomar prestado" y adaptar cualquier código de los ejemplos de clase y el repositorio del libro AI-A Modern Approach. Ya le proporcionamos un archivo `util.py` que contiene las implementaciones vistas en clase de **Node**, **StackFrontier** y **QueueFrontier**, que puede usar (y modificar si lo desea).

Problema 2 - Ta-te-ti

Objetivo

Programar una IA que juegue al Ta-Te-Ti con nosotros.

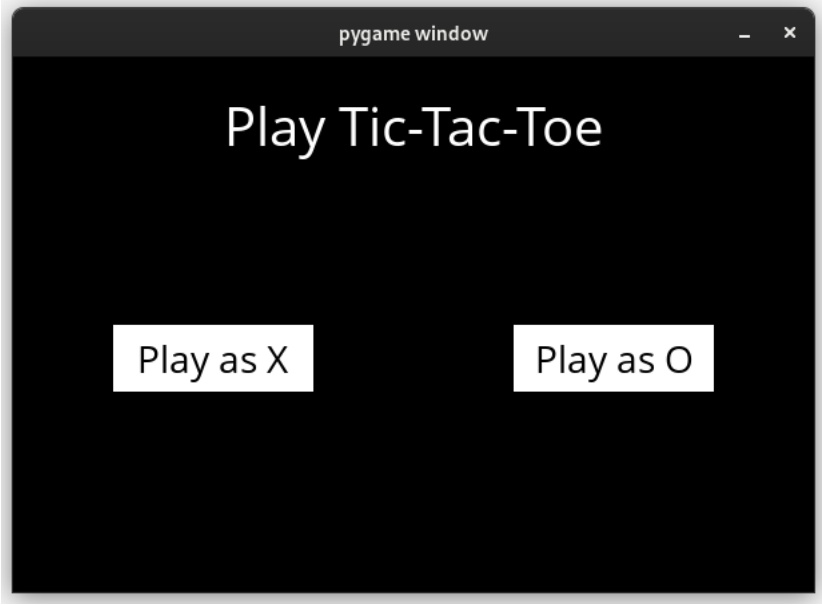

Figura 1: Juego del tateti al correr python runner.py

- Descargue el código en la carpeta tictactoe.zip y descomprímalo.
- Una vez en el directorio del proyecto, ejecute pip3 install -r requirements.txt para instalar el paquete Python requerido (pygame) para este proyecto.

Explicación

Hay dos archivos principales en este proyecto: `runner.py` y `tictactoe.py`. `tictactoe.py` contiene toda la lógica para jugar y para realizar movimientos óptimos. `runner.py` se ha implementado para ti y contiene todo el código para ejecutar la interfaz gráfica del juego. Una vez que hayas completado todas las funciones requeridas en `tictactoe.py`, deberías poder ejecutar python `runner.py` para jugar contra tu IA.

Abramos `tictactoe.py` para comprender lo que se proporciona. Primero, definimos tres variables: **X**, **O** y **EMPTY**, para representar posibles estados de los casilleros del tablero.

La función **initial_state** devuelve el estado inicial del tablero. Para este problema, hemos elegido representar el tablero como una lista de tres listas (que representan las tres filas del tablero), donde cada lista interna contiene tres valores que son **X**, **O** o **EMPTY**. Lo que sigue son funciones que deben implementar ustedes.

Especificación

Complete las implementaciones de **player**, **actions**, **result**, **winner**, **terminal**, **utility**, y **minimax**.

- La función **player** debe tomar el estado del tablero como entrada y devolver qué jugador tiene el turno siguiente (ya sea **X** u **O**).
  - En el estado inicial del juego, X hace el primer movimiento. Posteriormente, el jugador alterna con cada movimiento adicional.
  - Cualquier valor de retorno es aceptable si se proporciona una tablero terminal como entrada (es decir, el juego ya ha terminado).
- La función de **actions** debe devolver un conjunto de todas las acciones posibles que se pueden realizar en un tablero determinado.
  - Cada acción debe representarse como una tupla `(i, j)` donde i corresponde a la fila del movimiento (0, 1 o 2) y j corresponde a qué celda de la fila (columna) corresponde al movimiento (también 0, 1 o 2).
  - Los posibles movimientos son cualquier celda del tablero que aún no tenga una **X** o un **O**.
  - Cualquier valor de retorno es aceptable si se proporciona tablero terminal como entrada.
- La función de **result** toma un tablero y una acción como entrada, y debe devolver un nuevo estado del tablero, sin modificar el tablero original.
  - Si la acción no es una acción válida para el tablero, su programa debería generar una excepción.
  - El estado del tablero devuelto debe ser el tablero que resultaría de tomar el tablero de entrada original y dejar que el jugador a quien le toca haga su movimiento en la celda indicada por la acción de entrada.
  - Es importante destacar que el tablero original no debe modificarse: dado que Minimax, en última instancia, requerirá considerar muchos estados diferentes del tablero durante su cálculo. Esto significa que simplemente actualizar una celda en el tablero no es una implementación correcta de la función de resultado. Es probable que primero desee hacer una copia profunda del tablero antes de realizar cualquier cambio.
- La función **winner** debe aceptar un tablero como entrada y devolver el ganador del tablero, si lo hay.
  - Si el jugador X ganó el juego, su función debería devolver X. Si el jugador O ganó el juego, su función debería devolver O.
  - Uno puede ganar el juego con tres de sus movimientos (marcas) seguidos en horizontal, vertical o diagonal.

  - Puedes suponer que habrá como máximo un ganador (es decir, ningún tablero tendrá a ambos jugadores con tres en fila, ya que ese sería un estado del tablero no válido).
  - Si no hay un ganador del juego (ya sea porque el juego está en progreso o porque terminó en empate), la función debería devolver Ninguno.

- La función **terminal** debe aceptar un tablero como entrada y devolver un valor booleano que indique si el juego ha terminado.
  - Si el juego termina, ya sea porque alguien ganó o porque todas las celdas se llenaron sin que nadie ganara, la función debería devolver Verdadero.
  - De lo contrario, la función debería devolver False si el juego aún está en progreso.

- La función **utility** debe aceptar un tablero terminal como entrada y calcular la utilidad del tablero
  - Si X ganó el juego, la utilidad es 1. Si O ganó el juego, la utilidad es -1. Si el juego ha terminado en empate, la utilidad es 0.
  - Puede asumir que la utilidad solo se llamará en un tablero si terminal(tablero) es True.

- La función **minimax** debe tomar un tablero como entrada y devolver el movimiento óptimo para que el jugador se mueva en ese tablero.
  - El movimiento devuelto debe ser la acción óptima (i, j), que es una de las acciones permitidas en el tablero. Si varios movimientos son igualmente óptimos, cualquiera de ellos es aceptable.
  - Si la placa es una placa de terminales, la función minimax debería devolver Ninguno.

Para todas las funciones que aceptan un tablero como entrada, puede asumir que es un tablero válido (es decir, que es una lista que contiene tres filas, cada una con tres valores de X, O o EMPTY). No debe modificar las declaraciones de funciones (el orden o el número de argumentos de cada función) proporcionadas.

Una vez que todas las funciones estén implementadas correctamente, deberías poder ejecutar python runner.py y jugar contra tu IA. Y, dado que Ta Te Ti termina en un empate si ambos lados juegan de manera óptima, nunca deberías poder vencer a la IA (aunque si no juegas de manera óptima también, ¡puede que ella te gane!)

Consejos

- Si desea probar sus funciones en un archivo Python diferente, puede importarlas con from tictactoe import initial_state.
- Le invitamos a agregar funciones auxiliares adicionales a tictactoe.py, siempre que sus nombres no se superpongan con los nombres de funciones o variables que ya están en el módulo.
- El algoritmo alfa-beta prunning es opcional, ¡pero puede hacer que tu IA funcione de manera más eficiente!

Problema 3 - Pacman

Objetivo

Resuelva 3 de las distintas preguntas en las que ayudara a un agente Pacman a navegar por el laberinto y cumplir diferentes objetivos!

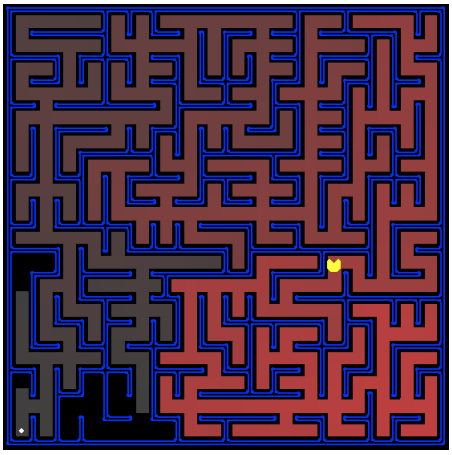

Figura 2: Juego del Pacman - Mapa de calor

Introducción

En este proyecto, tu agente de Pacman encontrará caminos a través de su mundo laberíntico, tanto para llegar a un lugar concreto como para recolectar comida de forma eficiente. Construirás algoritmos de búsqueda generales y los aplicarás a escenarios de Pacman.

Este proyecto incluye un autocalificador para que usted califique sus respuestas en su máquina. Esto se puede ejecutar con el comando:

```
$ python autograder.py
```

El código de este proyecto consta de varios archivos Python, algunos de los cuales deberá leer y comprender para completar la tarea, y otros puede ignorarlos. Puede descargar todo el código y los archivos en el drive de la materia, `problema2-pacman.zip`.

- **Archivos que editaras:**
  - `search.py` - Donde residirán todos sus algoritmos de búsqueda.
  - `searchAgents.py` - Donde residirán todos sus agentes de búsqueda
- **Archivos que quizás quieras/necesites entrar a ver:**
  - `pacman.py` - El archivo principal que ejecuta los juegos de Pacman. Este archivo describe un tipo Pacman GameState, que se utiliza 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.

- **Archivos de soporte que puedes 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` - Autocalificador de proyectos
  - `testParser.py` - Analiza los archivos de solución y prueba del autograduador
  - `testClasses.py` - Clases de prueba de autocalificación general
  - `test_cases/` - Directorio que contiene los casos de prueba para cada pregunta.
  - `searchTestClasses.py` - Clases de prueba de autocalificación específicas del Proyecto 1

Bienvenido a Pacman

Después de descargar el código, descomprimirlo y cambiar al directorio, deberías poder jugar un juego de Pacman escribiendo lo siguiente en la línea de comando:

```
$ python pacman.py
```

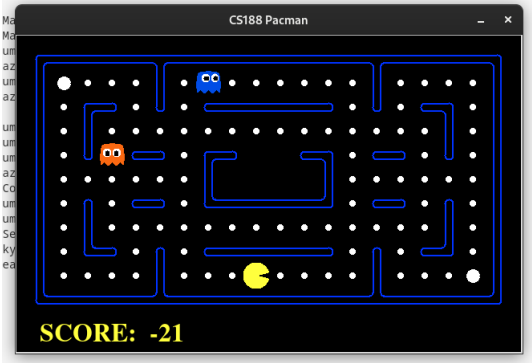

Figura 3: Juego base implementado en el código: pacman.py

Pacman vive en un mundo azul brillante de pasillos sinuosos y deliciosas delicias redondas. Navegar por este mundo de manera eficiente será el primer paso de Pacman para dominar su dominio.

El agente más simple `searchAgents.py` se llama **GoWestAgent**, que siempre va hacia el Oeste (un agente reflejo trivial). Este agente ocasionalmente puede ganar:

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

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

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

Si Pacman se atasca, puedes salir del juego con las teclas CTRL-c en tu terminal.

Pronto, tu agente resolverá no solo **tinyMaze**, sino cualquier laberinto que desees.

Tenga en cuenta que `pacman.py` admite una serie de opciones, cada una de las cuales puede expresarse de forma larga (p. ej., `--layout`) o corta (p. ej., `-l`). Puede 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 poder copiarlos y pegarlos fácilmente. En UNIX/Mac OS X, incluso puedes ejecutar todos estos comandos en orden con bash `commands.txt`.

Nueva sintaxis

Es posible que no hayas visto esta sintaxis antes:

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

```

Esto está anotando el tipo de argumentos que Python debería esperar para esta función. En el siguiente ejemplo, **a** debería ser un **int** entero, **b** debería ser **tuple** de 2 **ints**, **c** debería ser **List** de **Lists**, cualquier cosa; por lo tanto, una matriz 2D de cualquier cosa, **d** es esencialmente lo mismo que no anotado y puede ser de cualquier cosa, y **e** debería ser un **float**. **e** también se inicializa en el valor 1.0 si no se le pasa nada, es decir:

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

```

La llamada anterior se ajusta a las anotaciones de tipo y no pasa nada para e. Las anotaciones de tipo están destinadas a ser una adición a las cadenas de documentación para ayudarle a saber con qué funcionan las funciones. Python en sí no los aplica. Al escribir sus propias funciones, depende de usted si desea anotar sus tipos; pueden ser útiles para mantenerse organizado o no ser algo en lo que desee dedicar tiempo.

Preguntas

P1 : Encontrar un punto de comida fijo mediante la búsqueda en profundidad

En `searchAgents.py`, encontrarás un agente de busqueda completamente implementado, **SearchAgent**, que planifica un camino a través del mundo de Pacman y luego ejecuta ese camino paso a paso. Los algoritmos de búsqueda para formular un plan no están implementados, ese es su trabajo.

Primero, pruebe que **SearchAgent** esté funcionando correctamente ejecutando:

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

```

El comando anterior le indica a **SearchAgent** que utilice **tinyMazeSearch** como algoritmo de búsqueda, que se implementa en `search.py`. Pacman debería recorrer el laberinto con éxito.

¡Ahora es el momento de escribir funciones de búsqueda genéricas y completas para ayudar a Pacman a planificar rutas! El pseudocódigo de los algoritmos de búsqueda que escribirá se puede encontrar en la bibliografía compartida en drive. Recuerde que un nodo de búsqueda debe contener no sólo un estado sino también la información necesaria para reconstruir el camino (plan) que llega a ese estado.

Nota importante: todas sus funciones de búsqueda deben devolver una lista de acciones que llevarán al agente desde el principio hasta la meta. Todas estas acciones tienen que ser movimientos legales (direcciones válidas, no atravesar paredes).

Nota importante : asegúrese de utilizar las estructuras de datos **Stack**, **Queue** y **PriorityQueue** que se le proporcionan en `util.py`! Estas implementaciones de estructuras de datos tienen propiedades particulares que son necesarias para la compatibilidad con el autocalificador.

Sugerencia: cada algoritmo es muy similar. Los algoritmos para DFS (busque en profundidad), BFS (busqueda a lo ancho), UCS (Búsqueda heurística costo uniforme) y A* difieren sólo en los detalles de cómo se gestiona la frontera. Por lo tanto, concéntrese en lograr que DFS sea correcto y el resto debería ser relativamente sencillo. De hecho, una posible implementación requiere sólo un único método de búsqueda genérico que esté configurado con una estrategia de cola específica del algoritmo. (No es necesario que su implementación sea de esta forma para recibir el crédito completo).

Implemente el algoritmo de búsqueda en profundidad (DFS) en la función **depthFirstSearch** en `search.py`. Para completar su algoritmo, escriba la versión de búsqueda gráfica de DFS, que evita expandir los estados ya visitados.
Su 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
```

El tablero de Pacman mostrará una superposición de los estados explorados y el orden en que fueron explorados (el rojo más brillante significa exploración más temprana). ¿El orden de exploración es el que esperaba? ¿Pacman realmente recorre todas las casillas exploradas en su camino hacia la meta?

Sugerencia: si utiliza **Stack** como estructura de datos, la solución encontrada por su algoritmo DFS **mediumMaze** debe tener una longitud de 130 (siempre que coloque los sucesores en el margen en el orden proporcionado por **getSuccessors**; podría obtener 246 si los coloca en el orden inverso). ¿Es esta una solución de menor costo? Si no es así, piense en qué está haciendo mal la búsqueda en profundidad.

Calificación : ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P2 : Búsqueda en amplitud

Implemente el algoritmo de búsqueda en amplitud (BFS) en la función **breadthFirstSearch** en `search.py`. Nuevamente, escriba un algoritmo de búsqueda de gráficos que evite expandir los estados ya visitados. Pruebe su código de la misma manera que lo hizo para la búsqueda en profundidad.

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

¿BFS encuentra una solución de menor costo? Si no, verifique su implementación.

Sugerencia: si Pacman se mueve demasiado lento para usted, pruebe la opción –frameTime 0.

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P3 : Variación de la función de costos

Si bien BFS encontrará un camino con menos acciones hacia la meta, es posible que deseemos encontrar caminos que sean "mejores.[en] otros sentidos. Considere **mediumDot-**

tedMaze y **mediumScaryMaze**.

Al cambiar la función de costo, podemos alentar a Pacman a encontrar caminos diferentes. Por ejemplo, podemos cobrar más por 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.

Implemente el algoritmo de búsqueda de gráficos de costo uniforme en la función **uniformCostSearch** en `search.py`. Le recomendamos que busque `util.py` algunas estructuras de datos que puedan resultar útiles en su implementación. Ahora debería observar un comportamiento exitoso en los tres diseños siguientes, donde los agentes a continuación son todos agentes UCS que difieren solo en la función de costos que utilizan (los agentes y las funciones de costos están escritos para usted):

```
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: Debería obtener costos de ruta muy bajos y muy altos para **StayEastSearchAgent** y **StayWestSearchAgent** respectivamente, debido a sus funciones de costo exponencial (consulte `searchAgents.py` para obtener más detalles).

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P4 : Búsqueda A*

Implemente la búsqueda gráfica 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 (para información de referencia). La función heurística **nullHeuristic** en `search.py` es un ejemplo trivial.

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

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

Debería ver que A* encuentra la solución óptima un poco más rápido que la búsqueda de costo uniforme (aproximadamente 549 frente a 620 nodos de búsqueda expandidos en

nuestra implementación, pero los empates en la prioridad pueden hacer que sus números difieran ligeramente). ¿Qué sucede **openMaze** con las distintas estrategias de búsqueda?

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P5: Encontrar todas las esquinas

El verdadero poder de A* sólo 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 los laberintos de las esquinas, 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 realmente tenga comida allí o no). Tenga en cuenta que en algunos laberintos como **tinyCorners**, el camino más corto no siempre va primero a la comida más cercana. Pista: el camino más corto **tinyCorners** toma 28 pasos.

Nota: asegúrese de completar la Pregunta 2 antes de trabajar en la Pregunta 5, porque la Pregunta 5 se basa en su respuesta a la Pregunta 2.

Implemente el problema de búsqueda CornersProblem en `searchAgents.py`. Deberá elegir una representación de estado que codifique toda la información necesaria para detectar si se han alcanzado las cuatro esquinas. 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
```

Para recibir el crédito completo, debe definir una representación de estado abstracta que no codifique información irrelevante (como la posición de los fantasmas, dónde hay comida extra, etc.). En particular, no utilice un Pacman **GameState** como estado de búsqueda. Su código será muy, muy lento si lo hace (y también incorrecto).

Una instancia de la clase **CornersProblem** representa un problema de búsqueda completo, no un estado particular. Las funciones que usted escribe devuelven estados particulares, y sus funciones devuelven una estructura de datos de su elección (por ejemplo, tupla, set, etc.) que representa un estado.

Además, mientras se ejecuta un programa, recuerde que existen muchos estados simultáneamente, todos en la cola del algoritmo de búsqueda, y deben ser independientes entre sí. En otras palabras, no debería haber un solo estado para todo el objeto **CornersProblem**; su clase debería poder generar muchos estados diferentes para proporcionarlos

al algoritmo de búsqueda.

Consejo 1: Las únicas partes del estado del juego a las que debes hacer referencia en tu implementación son la posición inicial de Pacman y la ubicación de las cuatro esquinas.

Consejo 2: Al codificar **getSuccessors**, asegúrese de agregar niños a su lista de sucesores con un costo de 1.

Nuestra implementación de **breadthFirstSearch** se expande a 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.

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P6: Problema de esquinas: Enfoque heurístico

Nota: asegúrese de completar la Pregunta 4 antes de trabajar en la Pregunta 6, porque la Pregunta 6 se basa en su respuesta a la Pregunta 4.

Implemente una heurística consistente y no trivial para **CornersProblem** en **cornersHeuristic**.

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

Nota: AStarCornersAgent nos ahorra tipear
`-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic`
Admisibilidad versus consistencia: recuerde, las heurísticas son solo funciones que toman estados de búsqueda y devuelven números que estiman el costo hasta el objetivo más cercano. Unas heurísticas más eficaces arrojarán valores más cercanos a los costes objetivos reales. Para ser admisibles , los valores heurísticos deben ser límites inferiores del costo real del camino más corto hacia el objetivo más cercano (y no negativos). Para ser coherente , debe sostenerse además que si una acción ha costado **c**, entonces realizar esa acción sólo puede provocar una caída en la heurística de como máximo **c**.

Recuerde que la admisibilidad no es suficiente para garantizar la corrección en la búsqueda de gráficos: necesita la condición más estricta de coherencia. Sin embargo, las heurísticas admisibles suelen ser también consistentes, especialmente si se derivan de relajar el requerimiento de los problemas. Por lo tanto, suele ser más fácil comenzar con una lluvia de ideas sobre heurísticas admisibles. Una vez que tenga una heurística admisible que funcione bien, podrá comprobar si también es coherente. La única forma de garantizar la coherencia es con

una prueba. Sin embargo, la inconsistencia a menudo se puede detectar verificando que para cada nodo que expanda, sus nodos sucesores sean iguales o superiores en valor f. Además, si UCS y A* alguna vez devuelven caminos de diferentes longitudes, su heurística es inconsistente. ¡Esto es complicado!

Heurísticas no triviales: las heurísticas triviales son las que devuelven cero en todas partes (UCS) y la heurística que calcula el costo real de finalización. El primero no le ahorrará tiempo, mientras que el segundo limitará el tiempo de espera del autocalificador. Desea una heurística que reduzca el tiempo total de cálculo, aunque para esta tarea el autocalificador solo verificará el recuento de nodos (aparte de imponer un límite de tiempo razonable).

Metrica de eficiencia su heurística debe ser una heurística consistente, no trivial, no negativa. Asegúrese de que su heurística devuelva 0 en cada estado objetivo y nunca devuelva un valor negativo. Dependiendo de cuántos nodos expanda su heurística, el autocalificador evalua la calidad de la heuristica usada:

Número de nodos ampliados	**Calificación**
más de 2000	0/3
como mucho 2000	1/3
como mucho 1600	2/3
como mucho 1200	3/3

Necesita por lo menos 1/3 para considerar esta pregunta resuelta.

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

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

P7: Comer todos los puntos

Ahora resolveremos un difícil problema de búsqueda: 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 limpieza de alimentos: **FoodSearchProblem** en `searchAgents.py` (implementado para usted). Se define una solución como un camino que recoge toda la comida del mundo Pacman. Para el presente proyecto, las soluciones no tienen en cuenta fantasmas ni bolitas de energía; Las soluciones sólo dependen de la ubicación de las paredes, la comida habitual y Pacman. (¡Por supuesto que los fantasmas pueden arruinar la ejecución de una solución! Llegaremos a eso en el próximo proyecto). Si ha escrito correctamente sus métodos de búsqueda generales, A* con una heurística nula (equivalente a una búsqueda de costo uniforme) debería rápidamente encuentre una solución óptima **testSearch** sin cambiar el código de su parte (coste total de 7).

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

Nota: AStarFoodSearchAgentes un atajo para

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

Deberías encontrar que UCS comienza a disminuir incluso en lo que parece 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 su respuesta a la Pregunta 4.

Complete **foodHeuristic** en `searchAgents.py` con una heurística consistente para **FoodSearchProblem**. Pruebe su 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 trivial y no negativa recibirá 1 punto. Asegúrese de que su heurística devuelva 0 en cada estado objetivo y nunca devuelva un valor negativo. Dependiendo de cuántos nodos se expanda tu heurística, se obtienen puntos adicionales:

Número de nodos ampliados	**Calificación**
más de 15000	1/4
como mucho 15000	2/4
como mucho 12000	3/4
como máximo 9000	4/4
como máximo 7000	5/4 - Bonus!

Para considerar esta pregunta resuelta, debe sacar 2 puntos.

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

```
python autograder.py -q q7
```

P8: Búsqueda subóptima

A veces, incluso con A* y una buena heurística, es difícil encontrar el camino óptimo a través de todos los puntos. En estos casos, todavía nos gustaría encontrar un camino razonablemente bueno y rápidamente. En esta sección, escribirás un agente que siempre se come

con avidez el punto más cercano. **ClosestDotSearchAgent** está implementado para usted en `searchAgents.py`, pero le falta una función clave que encuentra una ruta al punto más cercano.

Implementar la función **findPathToClosestDot** en `searchAgents.py`. Nuestro agente resuelve este laberinto (¡de forma subóptima!) 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 la prueba **AnyFoodSearchProblem**, a la que le falta la prueba objetivo. Luego, resuelva ese problema con una función de búsqueda adecuada. ¡La solución debería ser muy breve!

ClosestDotSearchAgent no siempre encontrarás el camino más corto posible a través del laberinto . Asegúrate de entender por qué e intenta encontrar un pequeño ejemplo en el que ir repetidamente al punto más cercano no resulte en encontrar el camino más corto para comerse todos los puntos.

Calificación: ejecute el siguiente comando para ver si su implementación pasa todos los casos de prueba del autocalificador.

```
python autograder.py -q q8
```