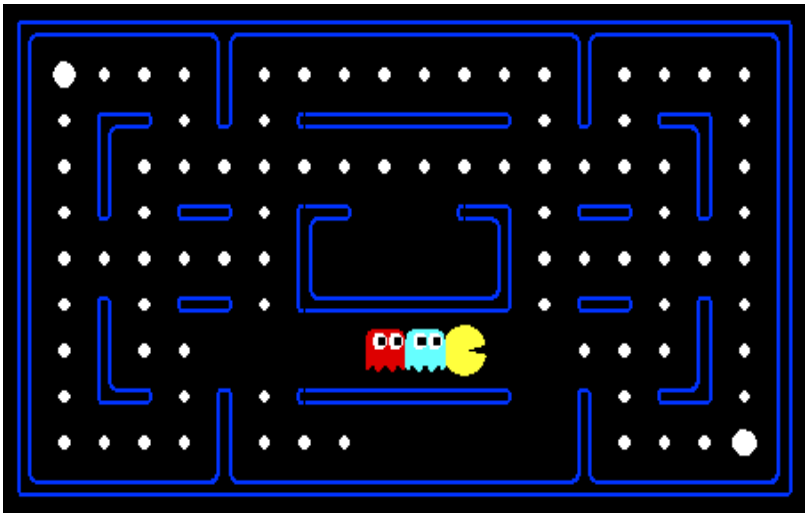
PL1c: Búsqueda Multi-Agente

Introducción

En este proyecto, diseñarás agentes para la versión clásica de Pacman, incluidos los fantasmas. En el camino, implementarás la búsqueda minimax y expectimax y probarás el diseño de la función de evaluación.

La base del código no ha cambiado mucho desde el proyecto anterior, pero hay que comenzar con un proyecto nuevo, en lugar de mezclar archivos de los proyectos anteriores.



Como en el proyecto anterior, este proyecto incluye un autograder para que puedas calificar tus respuestas en tu máquina. Se puede ejecutar con el comando:

python autograder.py

Nota: En este proyecto usaremos Python 3.

Se puede ejecutar para una pregunta concreta como para la pregunta 2 (q2), usando:

python autograder.py -q q2

Se puede ejecutar para un test concreto:

python autograder.py -t test_cases/q2/0-small-tree

Por defecto, el autograder muestra gráficos con la opción -t. Se puede forzar que no aparezcan gráficos con el flag --no-graphics.

El código para este proyecto tiene los siguientes ficheros, disponibles en un archivo zip:

Fichero que hay que editar:

multiAgents.py	Donde se encuentran todos los agentes de búsqueda multi-agente.

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 multiAgents.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 Multi-Agente

Para empezar, puedes jugar al Pacman clásico, usando las flechas para moverte:

python pacman.py

Ahora, ejecuta el ReflexAgent proporcionado en multiAgents.py:

python pacman.py -p ReflexAgent

Notarás que juega de manera muy pobre incluso en tableros sencillos:

python pacman.py -p ReflexAgent -l testClassic

Inspecciona su código (en multiAgents.py) y asegúrate de entender lo que está haciendo.

Pregunta 1 (4 puntos): Agente Reflex

Mejora el ReflexAgent en multiAgents.py para jugar respetablemente. El código del agente reflex proporciona algunos ejemplos útiles de métodos que consultan el GameState para obtener información. Un agente Reflex debe ser capaz de considerar tanto las ubicaciones de alimentos como las ubicaciones de los fantasmas para tener un buen rendimiento. Tu agente debe solucionar de manera fácil y confiable el diseño testClassic:

python pacman.py -p ReflexAgent -l testClassic

Prueba el agente ReflexAgent en el tablero mediumClassic con un fantasma o dos:

python pacman.py -p ReflexAgent -k 1
python pacman.py -p ReflexAgent -k 2

Para acelerar el proceso, añade **--frameTime 0**

python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2

¿Cómo se comporta tu agente? Posiblemente morirá a menudo con 2 fantasmas, a menos que tu función de evaluación sea bastante buena.
Nota*:* Recuerda que newFood tiene la función asList(), devolviendo las coordenadas de los alimentos en tuplas dentro de una lista.

Nota: Como características, prueba el valor recíproco de valores importantes (como la distancia a los alimentos) en lugar de solo los valores en sí.

Nota: La función de evaluación que estás escribiendo evalúa pares de estado-acción; en partes posteriores del proyecto, estará evaluando estados.

Nota: puede resultar útil ver el contenido interno de varios objetos para la depuración. Puedes hacerlo imprimiendo las representaciones de cadena de los objetos. Por ejemplo, puedes imprimir newGhostStates con print(newGhostStates).

Opciones: los fantasmas predeterminados son aleatorios. También puedes jugar por diversión con fantasmas direccionales ligeramente más inteligentes usando -g DirectionalGhost. Si la aleatoriedad te impide saber si tu agente está mejorando, puedes usar -f para ejecutar con una semilla aleatoria fija (las mismas opciones aleatorias en cada juego). También puedes jugar varios juegos seguidos con -n. Apaga los gráficos con -q para ejecutar muchos juegos rápidamente.

Calificación:

Ejecutaremos tu agente en el diseño openClassic 10 veces. Recibirás 0 puntos si tu agente agota el tiempo de espera o nunca gana. Recibirás 1 punto si tu agente gana al menos 5 veces, o 2 puntos si tu agente gana los 10 juegos. Recibirás 1 punto adicional si el puntaje promedio de tu agente es mayor a 500, o 2 puntos si es mayor a 1000. Puedes probar tu agente bajo estas condiciones con:

python autograder.py -q q1

Para ejecutarlo sin gráficos, usa: python autograder.py -q q1 --no-graphics

Conviene no pasar demasiado tiempo en esta pregunta, porque la miga del proyecto se encuentra más adelante.

Pregunta 2 (5 puntos): Minimax

Ahora escribirás un agente de búsqueda adversarial en el código de la clase MinimaxAgent proporcionado en multiAgents.py. Tu agente de minimax debería funcionar con cualquier número de fantasmas, por lo que tendrás que escribir un algoritmo que sea un poco más general. En particular, tu árbol minimax tendrá múltiples capas mínimas (una para cada fantasma) para cada capa máxima.

Tu código también debe expandir el árbol del juego a una profundidad arbitraria. Califica las hojas de tu árbol minimax con la función de autoevaluación suministrada, que por defecto es scoreEvaluationFunction. MinimaxAgent extiende MultiAgentSearchAgent, que da acceso a self.depth y self.evaluationFunction. Asegúrate de que tu código minimax haga referencia a estas dos variables cuando corresponda, ya que estas variables se completan en respuesta a las opciones de la línea de comandos.

Importante: una sola capa de búsqueda se considera un movimiento de Pacman y todas las respuestas de los fantasmas, por lo que la búsqueda de profundidad 2 implicará que Pacman y cada fantasma se muevan dos veces.

Calificación:

Comprobaremos tu código para determinar si explora el número correcto de estados del juego. Esta es la única forma confiable de detectar algunos errores muy sutiles en las implementaciones de minimax. Como resultado, el autograder será muy exigente acerca de cuántas veces se llama a GameState.generateSuccessor. Si lo llamas más o menos de lo necesario, el autograder se quejará.

Para probar y depurar tu código, ejecuta

python autograder.py -q q2

Esto te mostrará qué hace tu algoritmo en algunos árboles pequeños, así como en el juego pacman. Para ejecutarlo sin gráficos, usa:

python autograder.py -q q2 –no-graphics

Ideas y observaciones:

- Idea: implementa el algoritmo de manera recursiva usando funcion(es) auxiliar(es).

- La implementación correcta de minimax llevará a Pacman a perder el juego en varios tests. Esto no es un problema: al ser un comportamiento correcto, pasará los tests.

- La función de evaluación para el test de Pacman en esta parte ya está escrita (self.evaluationFunction). No deberías cambiarla, pero date cuenta de que estamos evaluando estados en vez de acciones, como se hacía en el agente reflex. Los agentes look-ahead evalúan estados futuros mientras que los agentes reflex evalúan acciones desde el estado actual.

- Los valores minimax del estado inicial en el tablero minimaxClassic son 9, 8, 7, -492 para profundidades 1, 2, 3 y 4 respectivamente. Nota que tu agente minimax ganará muchas veces (665/1000 juegos en nuestro caso), a pesar de la terrible predicción del minimax de profundidad 4.

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

- Pacman es siempre el agente 0, y los agentes se mueven en orden ascendente del índice del agente.

- Todos los estados en minimax deberían ser GameStates, o bien pasados en getAction o generados vía GameState.generateSuccessor. En este proyecto no se abstraerá a estados simplificados.

- En tableros más grandes como openClassic y mediumClassic (el predeterminado), encontrarás que Pacman es bueno para no morir, pero bastante malo para ganar. A menudo se revolverá sin progresar. Incluso podría moverse justo al lado de un punto sin comerlo porque no sabe a dónde iría después de comer ese punto. No te preocupes si ves este comportamiento, la pregunta 5 arreglará todos estos problemas.

- Cuando Pacman cree que su muerte es inevitable, intentará terminar el juego lo antes posible debido a la penalización constante por vivir. A veces, esto es incorrecto con los fantasmas aleatorios, pero los agentes minimax siempre asumen lo peor:

python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

Asegúrate de que entiendes por qué pacman se abalanza sobre el fantasma más cercano en este caso.

Pregunta 3 (5 puntos): Podado Alpha-Beta

Crea un nuevo agente que utilice la poda alfa-beta para explorar de manera más eficiente el árbol minimax, en AlphaBetaAgent. Nuevamente, tu algoritmo será un poco más general, por lo que parte del desafío es extender la lógica de poda alfa-beta de manera apropiada a múltiples agentes minimizadores.

Deberías ver una aceleración (quizás la profundidad 3 alfa-beta se ejecutará tan rápido como la profundidad 2 minimax). Idealmente, la profundidad 3 en smallClassic debería ejecutarse más rápido.

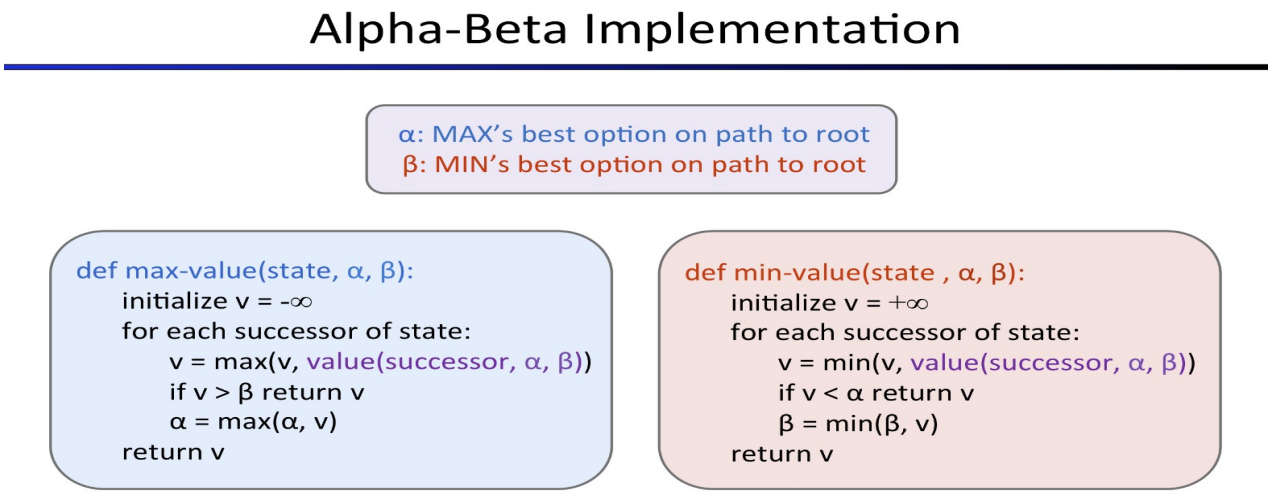
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

Los valores minimax de AlphaBetaAgent deben ser idénticos a los valores minimax de MinimaxAgent, aunque las acciones que selecciona pueden variar debido a diferentes comportamientos de desempate. Nuevamente, los valores minimax del estado inicial en el diseño minimaxClassic son 9, 8, 7 y -492 para las profundidades 1, 2, 3 y 4 respectivamente.

Calificación:

Debido a que verificamos tu código para determinar si explora el número correcto de estados, es importante que realices la poda alfa-beta sin reordenar los hijos. En otras palabras, los estados sucesores siempre deben procesarse en el orden devuelto por GameState.getLegalActions. Nuevamente, no llames a GameState.generateSuccessor más de lo necesario.
No debes podar en caso de igualdad para que coincida con el conjunto de estados explorados por nuestro autograder. (De hecho, alternativamente, pero incompatible con nuestro autograder, se podría permitir también la poda en igualdad e invocar alfa-beta una vez en cada hijo del nodo raíz, pero esto no coincidirá con el autograder).

El pseudocódigo siguiente representa el algoritmo que debes implementar para esta pregunta:



Para probar y depurar tu programa, ejecuta:

python autograder.py -q q3

Esto te mostrará qué hace tu algoritmo en varios árboles pequeños, y también en un juego pacman. Para ejecutarlo sin gráficos, usa:

python autograder.py -q q3 --no-graphics

La implementación correcta del podado alfa-beta llevará a pacman a perder en algunos tests. Esto no es un problema: al ser un comportamiento correcto, pasará los tests.

Pregunta 4 (5 puntos): Expectimax

Minimax y alpha-beta son geniales, pero ambos asumen que estás jugando contra un adversario que toma decisiones óptimas. Como puede decir cualquiera que haya ganado al tic-tac-toe, este no es siempre el caso. En esta pregunta implementarás el ExpectimaxAgent, que es útil para modelar el comportamiento probabilístico de los agentes que pueden tomar decisiones subóptimas.

Al igual que con los problemas de búsqueda hechos hasta ahora en esta clase, la belleza de estos algoritmos es su aplicabilidad general. Para acelerar tu propio desarrollo, proporcionamos algunos casos de prueba basados en árboles genéricos. Puedes depurar tu implementación en pequeños árboles de juego usando el comando:

python autograder.py -q q4

Se recomienda la depuración de estos casos de prueba pequeños y manejables y te ayudará a encontrar errores rápidamente.

Una vez que tu algoritmo funciona en árboles pequeños, puedes observar su éxito en Pacman. Los fantasmas aleatorios, por supuesto, no son agentes minimax óptimos, por lo que modelarlos con la búsqueda minimax puede no ser apropiado. ExpectimaxAgent, ya no tomará el mínimo sobre todas las acciones de fantasmas, sino la expectativa según el modelo de tu agente de cómo actúan los fantasmas. Para simplificar tu código, supón que solo te enfrentarás a un adversario que elija entre sus getLegalActions de manera uniforme al azar.

Para ver cómo se comporta el ExpectimaxAgent en Pacman, ejecuta:

python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3

Ahora deberías observar un enfoque más razonable en lugares cercanos con fantasmas. En particular, si Pacman percibe que podría estar atrapado pero podría escapar para agarrar algunas piezas más de comida, al menos lo intentará. Investiga los resultados de estos dos escenarios:

python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10

Deberías ver que tu agente ExpectimaxAgent gana alrededor de la mitad de las veces, mientras que AlphaBetaAgent siempre pierde. Asegúrate de entender por qué el comportamiento difiere del caso de minimax.
La implementación correcta de expectimax llevará a Pacman a perder algunos de los tests. Esto no es un problema: al ser un comportamiento correcto, pasará los tests.

Pregunta 5 (6 puntos): Función de evaluación

Escribe una mejor función de evaluación para pacman en la función proporcionada betterEvaluationFunction. La función de evaluación debe evaluar estados, en lugar de acciones como lo hizo la función de evaluación de tu agente reflex. Puedes utilizar cualquier herramienta a tu disposición para la evaluación, incluido tu código de búsqueda del anterior proyecto. Con la búsqueda de profundidad 2, tu función de evaluación debe superar el diseño smallClassic con un fantasma aleatorio más de la mitad de las veces y aún así ejecutarse a un ritmo razonable (para

obtener el crédito completo, Pacman debería promediar alrededor de 1000 puntos cuando está ganando).

Calificación:

El autograder ejecutará tu agente en el diseño smallClassic 10 veces. Asignaremos puntos a tu función de evaluación de la siguiente manera:

- Si ganas al menos una vez sin agotar el tiempo del autograder, recibirás 1 punto. Cualquier agente que no cumpla con estos criterios recibirá 0 puntos.
- +1 por ganar al menos 5 veces, +2 por ganar las 10 veces
- +1 para un puntaje promedio de al menos 500, +2 para un puntaje promedio de al menos 1000 (incluyendo puntajes en juegos perdidos)
- +1 si tus juegos tardan en promedio menos de 30 segundos en el autograder, cuando se ejecutan con --no-graphics. Tu computadora personal podría ser mucho menos eficiente (netbooks) o mucho más eficiente (equipos de juego).
- Los puntos adicionales para el puntaje promedio y el tiempo de cálculo solo se otorgarán si gana al menos 5 veces.

Puedes probar tu agente bajo estas condiciones con

python autograder.py -q q5

Para ejecutarlo sin gráficos, usa:

python autograder.py -q q5 --no-graphics

Entrega

Para presentar el proyecto, se debe entregar:

- Fichero **multiAgents.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! Por ejemplo, por qué vuestra elección del heurístico en la pregunta 5, etc etc