Inteligencia Artificial 2024

Segundo Proyecto

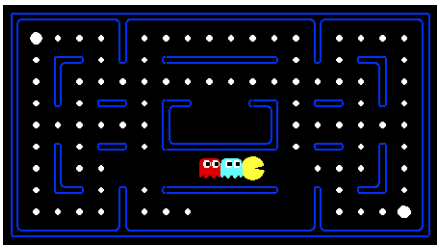
18.marzo.2024

Antes de comenzar el desarrollo del proyecto conviene recordar la documentación general del proyecto, así como la guía y recomendaciones: `http://ai.berkeley.edu/project_overview.html`.
`http://ai.berkeley.edu/project_instructions.html`.

Segundo Proyecto: Adversarial Search.
En este proyecto, diseñaremos agentes para la versión clásica de Pac-Man, incluidos fantasmas. En el camino, se implementará la búsqueda minimax y expectimax y se probará el diseño de funciones de evaluación.
La base del código no ha cambiado mucho con respecto al proyecto anterior, pero conviene comenzar una instalación nueva, en lugar de mezclar archivos del proyecto 1.
Al igual que en el proyecto 1, este proyecto incluye un *autograder* para que usted pueda calificar sus respuestas en su máquina. No se usará este autocalificador para efectos de evaluación (úselo únicamente a modo de test y verificación de sus algoritmos).
Consulte el tutorial del autocalificador en el Proyecto 0 para obtener más información sobre el uso del autocalificador.

Archivos a editar:

- `multiAgents.py` Donde residirán todos sus agentes de búsqueda multiagente.
- `pacman.py` El archivo principal que ejecuta los juegos de Pacman. Este archivo también describe un tipo Pac-Man **GameState**, que utilizará ampliamente en este proyecto.
- `game.py` La lógica detrás de cómo funciona el mundo Pac-Man. Este archivo describe varios tipos de soporte como **AgentState**, **Agent**, **Direction** y **Grid**.
- `util.py` Estructuras de datos útiles para implementar algoritmos de búsqueda.



Ingresar al sitio del proyecto Search, `https://inst.eecs.berkeley.edu/~cs188/sp21/project2/`.
Asegurarse de leer las especificaciones del proyecto.

1. **Problema 1**: Implementar Algoritmos de búsqueda.

   En el archivo `multiAgents.py`, ya se encuentra una clase RelflexAgent completamente implementada, con métodos que consultan GameState para obtener información. Sin embargo esta clase se desempeña de forma muy limitada (Hacer ejemplos con el escenario testClassic para verificar el funcionamiento)

   ```
   python pacman.py -p ReflexAgent -l testClassic
   ```

   Entender el funcionamiento de la clase ReflexAgent en `multiAgents.py`, y mejorar su desempeño para que el Pac-Man juegue de manera respetable. Su agente reflejo deberá que considerar tanto las ubicaciones de comida como las de los fantasmas para tener un buen desempeño.

Testar su función con 1 ó 2 fantasmas en el escenario mediumClassic:

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

Indicar cómo se desempeña el Pac-Man con 1 fantasmal (con 2 fantasmas en el escenario default usualmente muere en poco tiempo, a menos que su función de evaluación sea realmente muy buena). (Pregunta 1 del proyecto de Berkeley).

2. **Problema 2**: Minimax.

Ahora diseñaremos un agente de búsqueda adversario en el código auxiliar de clase MinimaxAgent proporcionado en el archivo `multiAgents.py`. Su agente minimax debe funcionar con cualquier cantidad de fantasmas, por lo que tendrá que escribir un algoritmo que sea un poco más general que lo visto en clase. En particular, su árbol minimax tendrá múltiples capas mínimas (una para cada fantasma) para cada capa máxima.

Los fantasmas reales que operan en el entorno pueden actuar parcialmente de forma aleatoria, pero el algoritmo minimax para Pac-Man supone siempre el peor escenario:

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

Asegúrate de entender por qué Pac-Man corre hacia el fantasma más cercano en este caso.

Su código también debería expandir el árbol del juego a una profundidad arbitraria (Para simplificar el procesamiento, usted puede utilizar un enfoque como *Iterative Deepening*, o fijar un parámetro Max Depth). Marque las hojas de su árbol minimax con la función self.evaluationFunction, que por defecto es scoreEvaluationFunction.

Asegurarse que su código hace referencia a las funciones self.profundidad y self.evaluationFunction cuando corresponda, ya que estas variables se completan en respuesta a las opciones de la línea de comando. (Pregunta 2 del proyecto de Berkeley).

3. **Problema 3**: α-β *Pruning*.

Crear un nuevo agente que utilice el esquema de poda α-β para explorar de manera más eficiente el árbol minimax en AlphaBetaAgent. Nuevamente, su algoritmo será un poco más general que el pseudocódigo visto en clase, por lo que parte del desafío es extender la lógica de poda alfa-beta de manera adecuada a múltiples agentes minimizadores.

Se debe observar una aceleración (quizás la profundidad 3 en alfa-beta se ejecuta tan rápido como la profundidad 2 minimax). Idealmente, la profundidad 3 en el escenario SmallClassic debería ejecutarse en solo unos segundos por movimiento o 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. (Pregunta 3 del proyecto de Berkeley).

4. **Problema 4**: Expectimax.

Minimax y α-β son geniales, pero ambos asumen que se juega contra un adversario que toma decisiones óptimas. Este no es siempre el caso. En este ejercicio, implementaremos ExpectimaxAgent, que es útil para modelar el comportamiento probabilístico de agentes que pueden tomar decisiones subóptimas.

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

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

Se recomienda depurar estos casos de prueba pequeños y manejables, lo que le ayudará a encontrar errores rápidamente.

Una vez que su algoritmo esté funcionando en árboles pequeños, podrá observar su éxito en Pac-Man. Los fantasmas aleatorios, por supuesto, no son agentes Minimax óptimos, por lo que modelarlos con búsqueda Minimax puede no ser apropiado. En lugar de tomar el control mínimo de todas las acciones de los fantasmas, ExpectimaxAgent tomará las expectativas de acuerdo con el modelo de su agente sobre cómo actúan los fantasmas. Para simplificar su código, suponga que solo se enfrentará a un adversario que elige entre sus getLegalActions de manera uniforme y aleatoria.

Para ver cómo se comporta ExpectimaxAgent en Pac-Man, ejecute:

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

Ahora debe observar un enfoque más fino en espacios reducidos con fantasmas. En particular, si Pac-Man percibe que podría quedar atrapado pero podría escapar para agarrar algunos trozos más de comida, al menos lo intentará. Investigue 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
```

Debería descubrir que su ExpectimaxAgent gana aproximadamente la mitad de las veces, mientras que su AlphaBetaAgent siempre pierde. Asegúrese de comprender por qué el comportamiento aquí difiere del caso minimax. (Pregunta 4 del proyecto de Berkeley)

Evaluación : 5 puntos cada uno de los cuatro problemas, para un total de 20.
Fecha de Entrega: lunes 22 de abril.