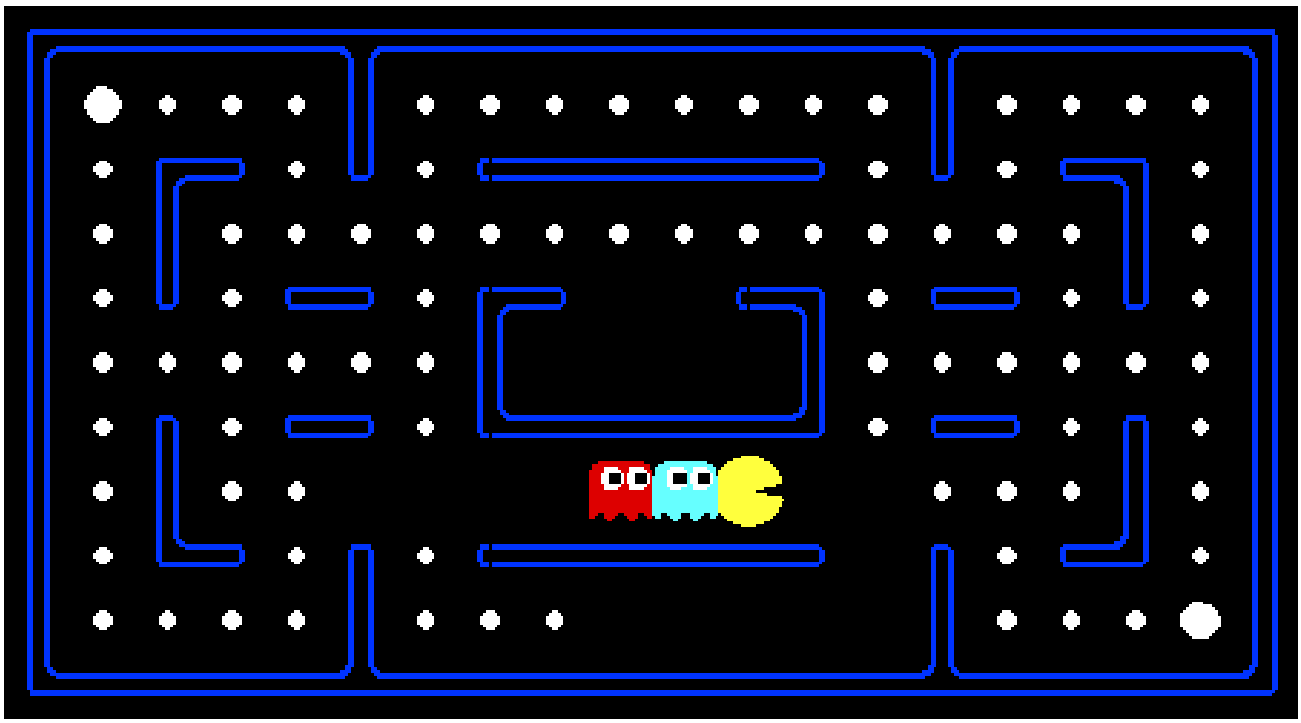
Técnicas de Inteligencia Artificial

Búsqueda Multi-Agente



Realizado por:

Cuesta Alario, David

Fernández Andrés, Cristian

Índice

En este proyecto nuestro agente de Pacman deberá comer todos los puntos del laberinto sin ser alcanzado por los Fantasmas.

Aspectos a tener en cuenta:

- En esta ocasión no se resolverá completamente el laberinto sino que se devolverá la acción a realizar en cada paso
- Todas estas acciones tienen que ser movimientos legales (no moverse a través de las paredes)
- Los fantasmas se moverán de forma aleatoria

Agente Réflex

Vamos a mejorar el comportamiento del nuestro Pacman mediante un heurístico que le permita decidor el mejor movimiento posible en cada paso en función de:

- La posición de los fantasmas
- La posición de las comidas
- El comportamiento de los fantasmas (si están asustados)

Utilizaremos la distancia Manhattan para calcular la distancia a las comidas y a los fantasmas

```
from util import manhattanDistance
from game import Directions
import random, util

from game import Agent

class ReflexAgent(Agent):

    def getAction(self, gameState):
        legalMoves = gameState.getLegalActions()
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        print("PUNTUACIONES A ELEGIR = ",scores)
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        return legalMoves[chosenIndex]
```

Por cada movimiento legal en el estado actual del juego comprobaremos la puntuación que le asigna nuestra función de evaluación y elegiremos la mayor de todas.

```
    def evaluationFunction(self, currentGameState, action):
        successorGameState = currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()

        newGhostStates = successorGameState.getGhostStates()
        newGhostPosition = [ghostState.getPosition() for ghostState in newGhostStates]
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        puntuacion = successorGameState.getScore()
        margenSeguro = 2
        velocidad = 1000

        for gost in newGhostPosition:
            distRiesgo = manhattanDistance(newPos,gost)
            if ( newScaredTimes[0] > 0 ):
                print("Persiguiendo fantasmas")
                puntuacion = puntuacion + newScaredTimes[0]*velocidad
            else:
                if ( distRiesgo <= margenSeguro ):
                    print("Riesgo de fantasma a", distRiesgo)
                    puntuacion = puntuacion - velocidad*margenSeguro

        for food in newFood.asList():
            distComida = manhattanDistance(newPos,food)
            puntuacion = puntuacion + 1/distComida

        return puntuacion
```

Obtenemos el siguiente estado del juego y del sacamos la nueva posición del Pacman, de las comidas y los fantasmas así como el tiempo que les queda estando asustados.

Definimos e inicializamos las siguientes variables:

- **Puntuación**: contendrá la puntuación que le vamos a asignar al movimiento legal que estamos analizando. Se inicializa como la puntación que establece el estado de la partida.
- **distRiesgo**: es la distancia real a la que se encuentra cada fantasma de nuestro Pacman
- **distComida**: es la distancia real a la que se encuentra cada comida de nuestro Pacman

Definimos las siguientes constantes:

- **margenSeguro**: es la distancia a partir de la cual comenzaremos a huir de los fantasmas
- **velocidad**: es el factor por el que multiplicamos la puntuación que sustraeremos al Pacman por elegir un movimiento que nos acerque a una distancia menor que el margen seguro o le incrementaremos por elegir un movimiento que nos acerca a un fantasma asustado.

Comprobamos si los fantasmas están asustados y si lo están alentamos al Pacman a que se acerquen a él tanto más cuando más asustados estén.

Si los fantasmas no están asustados comprobamos la distancia de cada uno de los fantasmas a nuestro Pacman y si se supera la distancia del margen seguro penalizamos al Pacman a acercarse a ellos tanto más cuando más cerca estén

Calculamos la distancia a cada comida del mapa y sumamos al Pacman una puntuación por cada comida que tenga cerca, cuando más cerca este la comida mayor puntuación le añadiremos.

- Obsérvese que dos comidas a una distancia de 2 valen lo mismo que una comida a una distancia de 1.

Ejecuciones

Para que el Pacman juegue automáticamente contra uno o dos fantasmas ejecutaremos los siguientes comandos:

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

Conclusiones

El valor asignado a las constantes ha sido establecido en base a numerosas pruebas de las que hemos obtenido las siguientes conclusiones:

- **margenSeguro**:
  - Un valor demasiado bajo hace que el Pacman no detecte a los fantasmas hasta que es demasiado tarde
  - Un valor demasiado alto hace que el Pacman se quede bloqueado, incluso cuando el fantasma se acerca
- **velocidad**:
  - Un valor demasiado bajo provoca que el Pacman cometa errores cuando la partida está muy avanzada debido a que la puntuación obtenida del estado del juego va creciendo conforme pasa el tiempo
  - Un valor demasiado alto provoca que el Pacman cometa errores al principio de la partida

Mini Max

Se trata de un algoritmo recursivo que genera un método de decisión para minimizar la pérdida máxima esperada en juegos con adversario y con información perfecta.

El funcionamiento de mini Max consiste en elegir el mejor movimiento para ti mismo suponiendo que tu contrincante escogerá el peor para ti.

Este algoritmo es óptimo siempre y cuando juegue contra un oponente óptimo

La complejidad computacional de esta solución es:

- **Tiempo** Exponencial b^m
- **Espacio** Lineal $b * m$
  - Factor de ramificación **b**: Número medio de arcos que parten de cada nodo.
  - Profundidad máxima considerada$m$

Vamos a realizar un algoritmo genérico que permita:

- Funcionar con cualquier número de fantasmas
- Establecer un límite a la profundidad del problema

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):
        accion = self.MinMaxUtiliti( gameState, 1 , 0 )
        return accion

    def MinMaxUtiliti(self, gameState, depth, agent):

        # Variables
        max_depth = self.depth
        next_depth = depth

        num_agents = gameState.getNumAgents()
        next_agent = agent + 1

        legalMoves = [action for action in gameState.getLegalActions(agent) if action!='Stop']

        puntuacion = 0
        accion = None

        # Caso Basico
        if( gameState.isLose() or gameState.isWin() or depth > max_depth ):
            return self.evaluationFunction(gameState)

        # Restablecer los indices
        if ( next_agent >= num_agents ):
            next_agent = 0
            next_depth = next_depth + 1

        # acciones posibles
        scores = [self.MinMaxUtiliti( gameState.generateSuccessor(agent, action) , next_depth, next_agent) for action in legalMoves]

        # Primer Movimiento --> Seleccionamos la accion final de Pacman
        if agent == 0 and depth == 1:
            bestMove = max(scores)
            bestIndices = [index for index in range(len(scores)) if scores[index] == bestMove]
            chosenIndex = random.choice(bestIndices) # Pick randomly among the best
            accion = legalMoves[chosenIndex]
            return accion

        # Movimientos siguientes
        else:
            # Pacman Maximiza
            if ( agent == 0 ):
                puntuacion = max(scores)
                print("PUNTUACIONES A ELEGIR = ",scores," Maximo -->", puntuacion)
            # Fantasmas Minimizan
            else:
                puntuacion = min(scores)
                print(" PUNTUACIONES A ELEGIR = ",scores," Minimo -->", puntuacion)
            return puntuacion
```

La función **getAction(gameState)** inicia el problema llamando a la función recursiva **MinMaxUtiliti** estableciendo el valor inicial de las variables de recursión:

- **depth:** Es la profundidad actual a la que estamos evaluando los movimientos que deberían efectuar los agentes.
  Se inicializa en uno debido a que la recursividad comienza desde el primer movimiento y finaliza al alcanzar la profundidad máxima especificada por el juego
- **agent:** Es la el agente que está actualmente analizando cuál es su mejor movimiento posible
  - Pacman es el agente número Cero
  - Los fantasmas son los agentes comprendidos del uno al cuatro

  Se inicializa en cero porque el primer movimiento siempre es de Pacman

La función **MinMaxUtiliti (gameState , deph , agent)** va a calcular para cada profundidad y cada agente el mejor movimiento posible de dicho agente.

Para efectuar los cálculos necesitaremos definir las siguientes variables locales

- **max_depth :** Es la profundidad máxima a la que vamos a evaluar.
  Viene especificada por el juego: $self.deph$
- **next_depth :** Es la siguiente profundidad que vamos a evaluar el problema.
  La inicializamos al valor de la profundidad actual a la que estamos evaluando
- **num_agents :** Es el número de agentes que están participando en el problema
  Viene especificada por el estado actual del juego: $self.gameState.getNumAgents()$
- **next_agent :** Es el siguiente agente al que le tocara analizar su mejor movimiento posible
  Lo inicializamos al valor del agente actual más uno

Lo primero que deberemos hacer es actualizar los índices:

- Si el índice del siguiente agente al que le toca jugar es superior al número de agentes que está jugando es porque ya han jugado todos los agentes que tenían que hacerlo por lo que:
  - Definimos que el siguiente agente será otra vez Pacman (el primer agente)
  - Consideramos que ya hemos explorado todas las opciones de esta profundidad y pasamos a la siguiente actualizando **next_depth**

Esta función está dividida en cuatro pasos:

- **Caso recursivo básico**: devuelve la solución de la hoja del árbol.

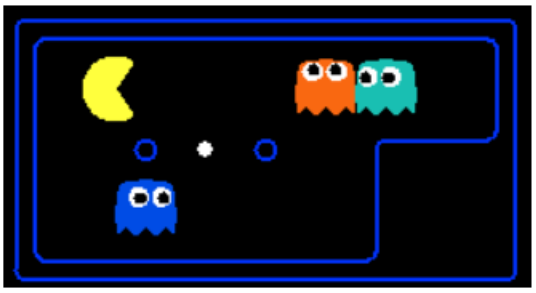
  Cuando nuestro algoritmo explore todas las profundidades previstas, o bien el juego haya finalizado porque Pacman ha perdido o ganado. Consideraremos que hemos llegado al último nodo explorable del árbol y devolveremos el peso que le asigna nuestra función de evaluación a dicho estado de juego.
  - Nuestra función de evaluación por defecto será **evaluationFunction(gameState)**
- **Expansión del árbol:** Si el nodo que estamos evaluando no es una hoja del árbol deberemos expandir los siguientes nodos.

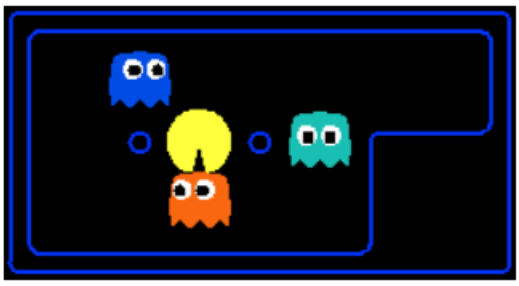
  Hasta que no lleguemos a una hoja del árbol no podremos saber que puntuaciones están asignando los agentes en cada una de las ramas, por ello debemos ir bajando hasta que termine la recursión.
  Para efectuar el proceso recursivo llamamos a nuestra función recursiva con los siguientes parámetros:
  - **gameState** deberá ser el nuevo estado de juego generado al moverse el agente que está siendo evaluado en cada una de las direcciones posibles en las que puede hacerlo
    $gameState.generateSucessors(agent , action) for\ action\ in\ legalMoves$
    - $legalMoves$ son todos los estados de juego validos excepto el de Stop debido a que generaba algunos problemas $gameState.getLegalActions(agent)$
  - **deph** deberá ser siguiente profundidad a la que tenemos que evaluar que ya habíamos definido en **next_depth**
  - **agent** deberá ser el nuevo agente al que le toca jugar a continuación que ya habíamos definido en **next_agent**
- **Primer movimiento**: La primera recursión es distinta de las demás debido a que Pacman debe decidir cuál será la acción que va a realizar en función de los pesos que reciba de cada una de las ramas que llegan hasta él.
  Para identificar la primera recursión definimos que debemos estar en la deph uno y el agente que debe estar jugando es Pacman
  Para identificar la acción que devuelve el mejor coste he utilizado las mismas funciones que ya estaban en el primer ejercicio
- **Movimientos recursivos:** Una vez que alcancemos las hojas del árbol comenzaremos a devolver valores hacia la copa en función del agente que esté jugando:
  - Pacman maximiza su movimiento por lo que cuando el agente actual sea el cero devolveremos a la función padre el máximo de los pesos recogidos por los nodos anteriores
  - Los fantasmas minimizan nuestro movimiento por lo que cuando el agente actual no sea el cero devolveremos a la función padre el mínimo de los pesos recogidos por los nodos anteriores

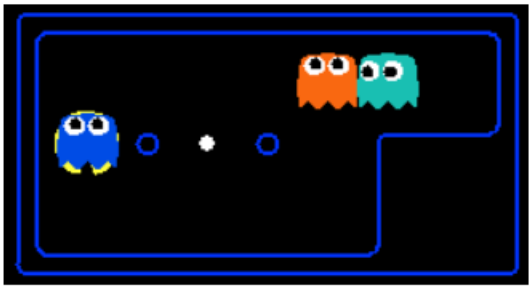
Ejecuciones

Para que el Pacman juegue automáticamente contra uno o dos fantasmas ejecutaremos los siguientes comandos:

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



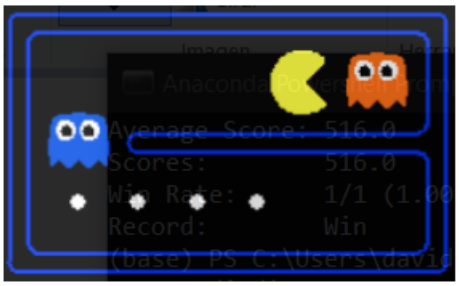


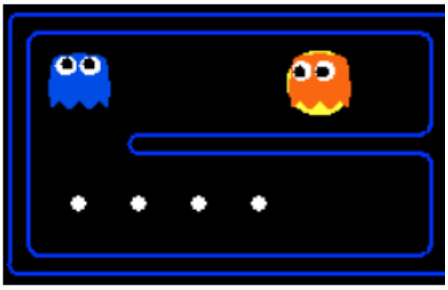


```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
Pacman died! Score: -492
Average Score: -492.0
Scores:        -492.0
Win Rate:      0/1 (0.00)
Record:        Loss
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores:        516.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=3
Pacman emerges victorious! Score: 515
Average Score: 515.0
Scores:        515.0
Win Rate:      1/1 (1.00)
Record:        Win
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=2
Pacman emerges victorious! Score: 515
Average Score: 515.0
Scores:        515.0
Win Rate:      1/1 (1.00)
Record:        Win
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=1
Pacman emerges victorious! Score: 510
Average Score: 510.0
Scores:        510.0
Win Rate:      1/1 (1.00)
Record:        Win
```

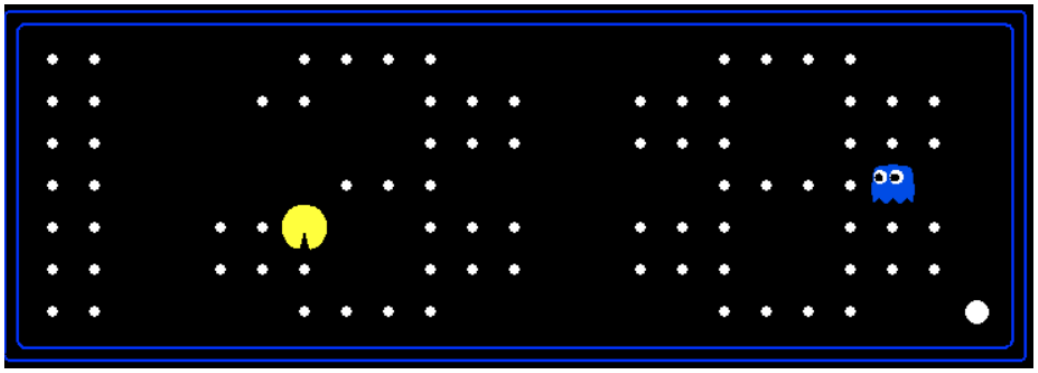
python pacman.py −p MinimaxAgent −l trappedClassic −a depth = 3

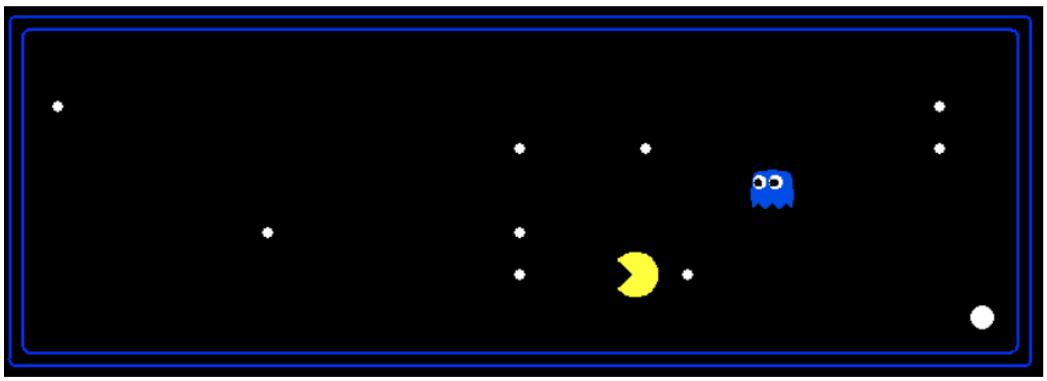




```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
 python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

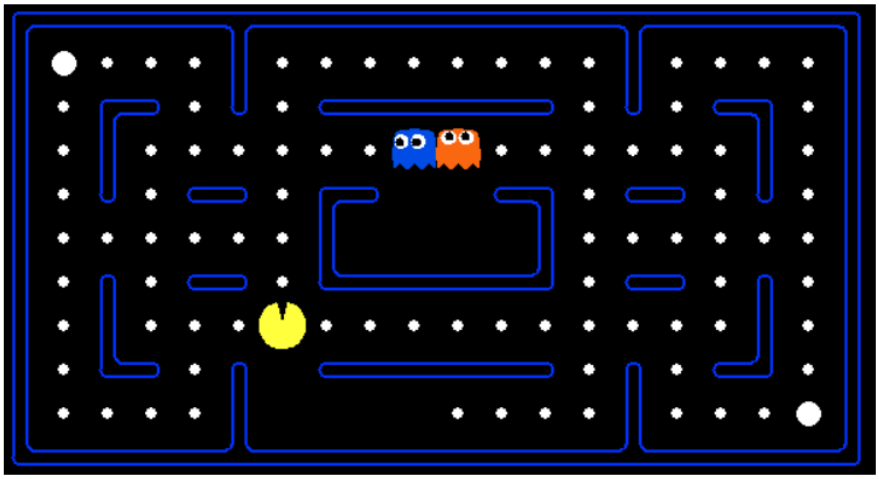
python pacman.py −p MinimaxAgent −l openClassic −a depth = 3

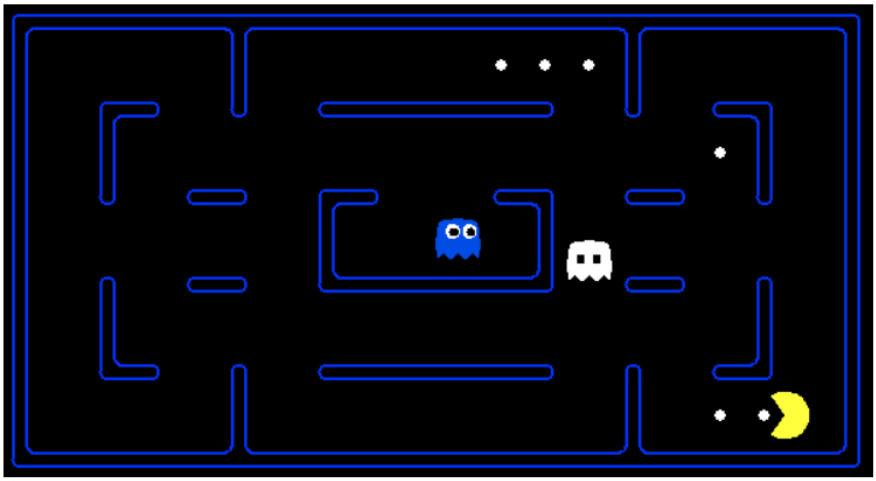




python pacman.py −p MinimaxAgent −l mediumClassic −a depth = 3

```
python pacman.py -p MinimaxAgent -l mediumClassic -a depth=3
Pacman emerges victorious! Score: 1562
Average Score: 1562.0
Scores:        1562.0
Win Rate:      1/1 (1.00)
Record:        Win
```





Conclusiones

La implementación correcta de minimax lleva a Pacman a perder el juego en varios debido a que cuando Pacman cree que su muerte es inevitable intentara terminar el juego lo antes posible debido a la penalización constante por vivir

En tableros más grandes como openClassic y mediumClassic 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.

Podado Alfa Beta

Se trata de una optimización del algoritmo mini Max mediante la cual descartamos los caminos que nos llevarían a soluciones peores en el momento en que nos damos cuenta de que hemos llegado a un máximo local en una etapa de minimización o a un mínimo local en una etapa de maximización

Al realizar podado sobre el mini Max

- No alteramos el resultado óptimo previsto debido a que los valores que descartamos no tenían ninguna posibilidad de ser elegidos en la etapa que se deberían haber analizado porque la etapa anterior ya había tomado una decisión acerca de la solución
- Reduce considerablemente el coste en tiempo y en memoria al reducir el factor de ramificación
- Una buena ordenación de los hijos mejoraría la efectividad del podado debido a que alcanzaríamos antes el mínimo o máximo local

```
class AlphaBetaAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):
        accion = self.AlphaBetaUtiliti( gameState, 1 , 0 , -9999999999999999999 , 9999999999999999999 )
        return accion

    def AlphaBetaUtiliti(self, gameState, depth, agent, alpha, beta):

        # Variables
        max_depth = self.depth
        next_depth = depth

        num_agents = gameState.getNumAgents()
        next_agent = agent + 1

        legalMoves = [action for action in gameState.getLegalActions(agent) if action!='Stop']

        puntuacion = 0
        accion = None

        # Caso Basico
        if( gameState.isLose() or gameState.isWin() or depth > max_depth ):
            return self.evaluationFunction(gameState)

        # Restablecer los indices
        if ( next_agent >= num_agents ):
            next_agent = 0
            next_depth = next_depth + 1

        # Primer Movimiento --> Seleccionamos la accion final de Pacman
        if agent == 0 and depth == 1:
            actual = -9999999999999999999
            index = -1
            chosenIndex = 0
            for action in legalMoves:
                index = index + 1
                scores = self.AlphaBetaUtiliti( gameState.generateSuccessor(agent, action) , next_depth , next_agent , alpha, beta)
                if ( scores > actual ):
                    actual = scores
                    chosenIndex = index
                if actual > beta:
                    break
                else:
                    alpha =  max(alpha, actual)

            accion = legalMoves[chosenIndex]
            return accion

        # Movimientos siguientes
        else:
            # Pacman Maximiza
            if ( agent == 0 ):
                # inicializamos el valor actual en -Inf
                actual = -9999999999999999999

                for action in legalMoves:
                    # Actualizamos el valor si es mayor que actual
                    actual = max(actual, self.AlphaBetaUtiliti( gameState.generateSuccessor(agent, action) , next_depth , next_agent , alpha, beta)
                    # si actual es mayor que el maximo local Podamos y sino actualizamos alpha
                    if actual > beta:
                        return actual
                    else:
                        alpha =  max(alpha, actual)


            # Fantasmas Minimizan
            else:
               # inicializamos el valor actual en Inf
                actual = 9999999999999999999

                for action in legalMoves:
                    # Actualizamos el valor si es menor que actual
                    actual = min(actual, self.AlphaBetaUtiliti( gameState.generateSuccessor(agent, action) , next_depth , next_agent , alpha, beta)
                    # si actual es menor que el minimo local Podamos y sino actualizamos beta
                    if actual < alpha:
                        return actual
                    else:
                        beta =  min(beta, actual)

            return puntuacion
```

La función **getAction(gameState)** inicia el problema llamando a la función recursiva **AlphaBetaUtiliti** estableciendo el valor inicial de las variables de recursión:

- **depth:** Es la profundidad actual a la que estamos evaluando los movimientos que deberían efectuar los agentes.
  Se inicializa en uno debido a que la recursividad comienza desde el primer movimiento y finaliza al alcanzar la profundidad máxima especificada por el juego
- **agent:** Es la el agente que está actualmente analizando cuál es su mejor movimiento posible
  - Pacman es el agente número Cero
  - Los fantasmas son los agentes comprendidos del uno al cuatro

  Se inicializa en cero porque el primer movimiento siempre es de Pacman
- **alpha** Es el mínimo local. Se inicializa como menos infinito
- **beta** Es el máximo local. Se inicializa como infinito

La función **AlphaBetaUtiliti (gameState , deph , agent)** va a calcular para cada profundidad y cada agente el mejor movimiento posible de dicho agente.

Para efectuar los cálculos necesitaremos definir las siguientes variables locales

- **max_depth :** Es la profundidad máxima a la que vamos a evaluar.
  Viene especificada por el juego: $self.deph$
- **next_depth :** Es la siguiente profundidad que vamos a evaluar el problema.
  La inicializamos al valor de la profundidad actual a la que estamos evaluando
- **num_agents :** Es el número de agentes que están participando en el problema
  Viene especificada por el estado actual del juego: $self.gameState.getNumAgents()$
- **next_agent :** Es el siguiente agente al que le tocara analizar su mejor movimiento posible
  Lo inicializamos al valor del agente actual más uno

Lo primero que deberemos hacer es actualizar los índices:

- Si el índice del siguiente agente al que le toca jugar es superior al número de agentes que está jugando es porque ya han jugado todos los agentes que tenían que hacerlo por lo que:
  - Definimos que el siguiente agente será otra vez Pacman (el primer agente)
  - Consideramos que ya hemos explorado todas las opciones de esta profundidad y pasamos a la siguiente actualizando **next_depth**

Esta función está dividida en tres pasos:

- **Caso recursivo básico**: devuelve la solución de la hoja del árbol.

  Cuando nuestro algoritmo explore todas las profundidades previstas, o bien el juego haya finalizado porque Pacman ha perdido o ganado. Consideraremos que hemos llegado al último nodo explorable del árbol y devolveremos el peso que le asigna nuestra función de evaluación a dicho estado de juego.
  - Nuestra función de evaluación por defecto será **evaluationFunction(gameState)**

- **Primer movimiento**: La primera recursión es distinta de las demás debido a que Pacman debe decidir cuál será la acción que va a realizar en función de los pesos que reciba de cada una de las ramas que llegan hasta él.

  Para identificar la primera recursión definimos que debemos estar en la deph uno y el agente que debe estar jugando es Pacman

  Antes de realizar el movimiento debemos expandir el árbol. En el mini Max podíamos hacer una expansión común e independiente de la capa porque expandíamos siempre pero ahora no siempre vamos a expandir. La expansión del árbol sigue el mismo procedimiento:
  - **gameState** deberá ser el nuevo estado de juego generado al moverse el agente que está siendo evaluado en cada una de las direcciones posibles en las que puede hacerlo
    $gameState.generateSucessors(agent , action) for\ action\ in\ legalMoves$
    - $legalMoves$ son todos los estados de juego validos excepto el de Stop debido a que generaba algunos problemas $gameState.getLegalActions(agent)$
  - **deph** deberá ser siguiente profundidad a la que tenemos que evaluar que ya habíamos definido en **next_depth**
  - **agent** deberá ser el nuevo agente al que le toca jugar a continuación que ya habíamos definido en **next_agent**
  - **alpha** hereda el que venía por parámetro de la función que la llama a menos que el algoritmo determine que debe modificarse
  - **beta** hereda el que venía por parámetro de la función que la llama a menos que el algoritmo determine que debe modificarse

  Como el primer movimiento lo realiza el Pacman es una maximización. Seguiremos el mismo algoritmo que en cualquier otra maximización con las siguientes diferencias
  - **Debemos devolver el índice de la acción que genera la mayor puntuación no la puntuación en sí**: Hemos utilizado un contador para determinar la posición de cada acción, de modo que cuando la puntuación recibida como resultado de los nodos sucesores supere al último máximo localizado actualizaremos la acción que queremos devolver y el valor de la puntuación máxima
  - **Ya no existe un nodo padre al que devolver el peso máximo** por lo que utilizaremos un **break** en vez de un **return** para efectuar la poda si fuera necesaria

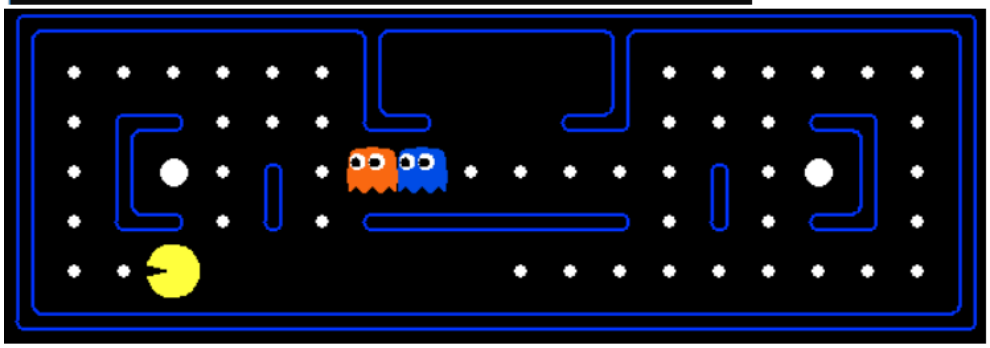
- **Movimientos recursivos:** Una vez que alcancemos las hojas del árbol comenzaremos a devolver valores hacia la copa en función del agente que esté jugando:
  - Pacman maximiza su movimiento
    - Definimos la variable local **actual** que se inicializa como **menos infinito** ya que va a representar el valor máximo encontrado por Pacman en esta capa.
    - Realizamos la expansión del árbol en cada una de las acciones legales posibles de la misma forma que ya se ha explicado en el primer movimiento
    - Se actualiza la variable actual como el **máximo** entre el valor que tenía antes y el máximo de todos los valores que devuelven sus hijos
    - Comprobamos si **actual** es **mayor que el máximo local** **beta** que venía como parámetro desde el nodo padre
      - **Si es mayor** efectuaremos la poda devolviendo con **return** el valor **actual** que acabamos de encontrar debido a que sabemos con certeza que el nodo anterior, que era de minimización, ha encontrado un valor inferior al que acabamos de encontrar. Por lo que aun que el que acabamos de encontrar no va a tener ninguna relevancia. La etapa anterior ya ha definido un mínimo inferior al valor que acabamos de encontrar por lo que lo máximo que podemos hacer es no darle para seleccionar un valor más pequeño. De este modo el valor actual, aunque no sea el máximo posible, es mayor que el máximo local por lo que es tan bueno lo seria cualquier otro que cumpla esa condición.
      - **Si es menor** actualizamos el mínimo local **alpha** con el valor máximo del **actual** que acabamos de encontrar y el **alpha** anterior
  - Los fantasmas minimizan nuestro movimiento
    - Definimos la variable local **actual** que se inicializa como **infinito** ya que va a representar el valor máximo encontrado por Pacman en esta capa.
    - Realizamos la expansión del árbol en cada una de las acciones legales posibles de la misma forma que ya se ha explicado en el primer movimiento
    - Se actualiza la variable actual como el **mínimo** entre el valor que tenía antes y el máximo de todos los valores que devuelven sus hijos
    - Comprobamos si **actual** es **menor que el mínimo local** **alpha** que venía como parámetro desde el nodo padre
      - **Si es menor** efectuaremos la poda devolviendo con **return** el valor **actual** que acabamos de encontrar debido a que sabemos con certeza que el nodo anterior, que era de maximización, ha encontrado un valor superior al que acabamos de encontrar. Por lo que aun que el que acabamos de encontrar no va a tener ninguna relevancia. La etapa anterior ya ha definido un máximo superior al valor que acabamos de encontrar por lo que lo máximo que podemos hacer es no darle para seleccionar un valor más grande. De este modo el valor actual, aunque no sea el mínimo posible, es menor que el mínimo local por lo que es tan bueno lo seria cualquier otro que cumpla esa condición.
      - **Si es mayor** actualizamos el mínimo local **alpha** con el valor **mínimo** del **actual** que acabamos de encontrar y el **alpha** anterior

Ejecuciones

Para que el Pacman juegue automáticamente contra uno o dos fantasmas ejecutaremos los siguientes comandos:

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

```
python pacman.py -p AlphaBetaAgent -a depth=5 -l smallClassic
Pacman emerges victorious! Score: 1273
Average Score: 1273.0
Scores:        1273.0
Win Rate:      1/1 (1.00)
Record:        Win
```



Conclusiones

Si hacemos pruebas con profundidades muy altas en minimax y alpha beta si podemos observar una diferencia en el rendimiento

Concretamente hemos observado como el Pacman tomaba decisiones considerablemente más rápido cuando tenía un fantasma cerca. Probablemente porque en estas circunstancias se daban más situaciones donde podía efectuar la poda

Expectimax

Mini Max y alpha-beta son geniales pero ambos asumen que estás jugando contra un adversario que toma decisiones óptimas.

Debido a que este no siempre es el caso Expectimax es un algoritmo que nos permite modelar el comportamiento probabilístico de los agentes para que puedan tomar decisiones subóptimas que nos acerquen más a la realidad de nuestro adversario

En esta ocasión nuestros Pacman no asumirá que los fantasmas vallan a elegir el peor resultado posible para nosotros y estimara el coste que tendría tomar una decisión en función de la probabilidad que tengan los adversarios de tomar una decisión ponderada con el peso que tendría que la tomaran.

Debido a que nuestros fantasmas son aleatorios las probabilidades de tomar cada acción son equiponderadas.

El valor esperado de una función de una variable aleatoria es la media ponderada por la distribución de probabilidad de los resultados

```
class ExpectimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):
        accion = self.ExpectimaxUtiliti( gameState, 1 , 0 )
        return accion

    def ExpectimaxUtiliti(self, gameState, depth, agent):

        # Variables
        max_depth = self.depth
        next_depth = depth

        num_agents = gameState.getNumAgents()
        next_agent = agent + 1

        legalMoves = [action for action in gameState.getLegalActions(agent) if action!='Stop']

        puntuacion = 0
        accion = None

        # Caso Basico
        if( gameState.isLose() or gameState.isWin() or depth > max_depth ):
            return self.evaluationFunction(gameState)

        # Restablecer los indices
        if ( next_agent >= num_agents ):
            next_agent = 0
            next_depth = next_depth + 1

        # acciones posibles
        scores = [self.ExpectimaxUtiliti( gameState.generateSuccessor(agent, action) , next_depth, next_agent) for action in legalMoves]

        # Primer Movimiento --> Seleccionamos la accion final de Pacman
        if agent == 0 and depth == 1:
            bestMove = max(scores)
            bestIndices = [index for index in range(len(scores)) if scores[index] == bestMove]
            chosenIndex = random.choice(bestIndices) # Pick randomly among the best
            accion = legalMoves[chosenIndex]
            return accion

        # Movimientos siguientes
        else:
            # Pacman Maximiza
            if ( agent == 0 ):
                puntuacion = max(scores)
                print("PUNTUACIONES A ELEGIR = ",scores," Maximo -->", puntuacion)
            # Fantasmas Minimizan
            else:
               puntuacion = sum(scores)/len(scores)
               print(" PUNTUACIONES A ELEGIR = ",scores," Minimo -->", puntuacion)
            return puntuacion
```

La función **getAction(gameState)** inicia el problema llamando a la función recursiva **MinMaxUtiliti** estableciendo el valor inicial de las variables de recursión:

- **depth:** Es la profundidad actual a la que estamos evaluando los movimientos que deberían efectuar los agentes.
  Se inicializa en uno debido a que la recursividad comienza desde el primer movimiento y finaliza al alcanzar la profundidad máxima especificada por el juego
- **agent:** Es la el agente que está actualmente analizando cuál es su mejor movimiento posible
  - Pacman es el agente número Cero
  - Los fantasmas son los agentes comprendidos del uno al cuatro

  Se inicializa en cero porque el primer movimiento siempre es de Pacman

La función **MinMaxUtiliti (gameState , deph , agent)** va a calcular para cada profundidad y cada agente el mejor movimiento posible de dicho agente.

Para efectuar los cálculos necesitaremos definir las siguientes variables locales

- **max_depth :** Es la profundidad máxima a la que vamos a evaluar.
  Viene especificada por el juego: $self.deph$
- **next_depth :** Es la siguiente profundidad que vamos a evaluar el problema.
  La inicializamos al valor de la profundidad actual a la que estamos evaluando
- **num_agents :** Es el número de agentes que están participando en el problema
  Viene especificada por el estado actual del juego: $self.gameState.getNumAgents()$
- **next_agent :** Es el siguiente agente al que le tocara analizar su mejor movimiento posible
  Lo inicializamos al valor del agente actual más uno

Lo primero que deberemos hacer es actualizar los índices:

- Si el índice del siguiente agente al que le toca jugar es superior al número de agentes que está jugando es porque ya han jugado todos los agentes que tenían que hacerlo por lo que:
  - Definimos que el siguiente agente será otra vez Pacman (el primer agente)
  - Consideramos que ya hemos explorado todas las opciones de esta profundidad y pasamos a la siguiente actualizando **next_depth**

Esta función está dividida en cuatro pasos:

- **Caso recursivo básico**: devuelve la solución de la hoja del árbol.

  Cuando nuestro algoritmo explore todas las profundidades previstas, o bien el juego haya finalizado porque Pacman ha perdido o ganado. Consideraremos que hemos llegado al último nodo explorable del árbol y devolveremos el peso que le asigna nuestra función de evaluación a dicho estado de juego.
  - Nuestra función de evaluación por defecto será **evaluationFunction(gameState)**
- **Expansión del árbol:** Si el nodo que estamos evaluando no es una hoja del árbol deberemos expandir los siguientes nodos.

  Hasta que no lleguemos a una hoja del árbol no podremos saber que puntuaciones están asignando los agentes en cada una de las ramas, por ello debemos ir bajando hasta que termine la recursión.
  Para efectuar el proceso recursivo llamamos a nuestra función recursiva con los siguientes parámetros:
  - **gameState** deberá ser el nuevo estado de juego generado al moverse el agente que está siendo evaluado en cada una de las direcciones posibles en las que puede hacerlo
    $gameState.generateSucessors(agent , action) for action in legalMoves$
    - $legalMoves$ son todos los estados de juego validos excepto el de Stop debido a que generaba algunos problemas $gameState.getLegalActions(agent)$
  - **deph** deberá ser siguiente profundidad a la que tenemos que evaluar que ya habíamos definido en **next_depth**
  - **agent** deberá ser el nuevo agente al que le toca jugar a continuación que ya habíamos definido en **next_agent**
- **Primer movimiento**: La primera recursión es distinta de las demás debido a que Pacman debe decidir cuál será la acción que va a realizar en función de los pesos que reciba de cada una de las ramas que llegan hasta él.
  Para identificar la primera recursión definimos que debemos estar en la deph uno y el agente que debe estar jugando es Pacman
  Para identificar la acción que devuelve el mejor coste he utilizado las mismas funciones que ya estaban en el primer ejercicio
- **Movimientos recursivos:** Una vez que alcancemos las hojas del árbol comenzaremos a devolver valores hacia la copa en función del agente que esté jugando:
  - Pacman maximiza su movimiento por lo que cuando el agente actual sea el cero devolveremos a la función padre el máximo de los pesos recogidos por los nodos anteriores
  - Los fantasmas ya no minimizan nuestro movimiento por lo que cuando el agente actual no sea el cero devolveremos a la función padre la media de los pesos recogidos por los nodos anteriores

Ejecuciones

Para ver cómo se comporta el agente Expectimax en Pacman ejecutamos:

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

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
Pacman emerges victorious! Score: 499
Average Score: 499.0
Scores:        499.0
Win Rate:      1/1 (1.00)
Record:        Win
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
Pacman died! Score: -496
Average Score: -496.0
Scores:        -496.0
Win Rate:      0/1 (0.00)
Record:        Loss
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Investiga los resultados de estos dos escenarios:

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

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

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

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

Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Average Score: -88.4
Scores:        532.0, -502.0, -502.0, 532.0, -502.0, -502.0, 532.0, -502.0, -502.0, 532.0
Win Rate:      4/10 (0.40)
Record:        Win, Loss, Loss, Win, Loss, Loss, Win, Loss, Loss, Win
```

Conclusiones

Con el Expectimax podemos observar que Pacman no siempre se suicida cuando piensa que está atrapado, sino que intenta ver si puede escapar y lo consigue aproximadamente la mitad de las veces debido a que los fantasmas son aleatorios

Función de Evaluación

```
def betterEvaluationFunction(currentGameState):

    newPos = currentGameState.getPacmanPosition()
    newFood = currentGameState.getFood()

    newGhostStates = currentGameState.getGhostStates()
    newGhostPosition = [ghostState.getPosition() for ghostState in newGhostStates]
    newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

    puntuacion = currentGameState.getScore()
    margenSeguro = 2
    velocidad = 1000

    for gost in newGhostPosition:
        distRiesgo = manhattanDistance(newPos,gost)
        if ( newScaredTimes[0] > 0 ):
            #print("Persiguiendo fantasmas")
            puntuacion = puntuacion + newScaredTimes[0]*velocidad
        else:
            if ( distRiesgo <= margenSeguro ):
                #print("Riesgo de fantasma a", distRiesgo)
                puntuacion = puntuacion - velocidad*margenSeguro

    for food in newFood.asList():
        distComida = manhattanDistance(newPos,food)
        puntuacion = puntuacion + 1/distComida

    return puntuacion
```

Para el estado del juego dado sacamos la nueva posición del Pacman, de las comidas y los fantasmas así como el tiempo que les queda estando asustados.

Definimos e inicializamos las siguientes variables:

- **Puntuación**: contendrá la puntuación que le vamos a asignar al movimiento legal que estamos analizando. Se inicializa como la puntación que establece el estado de la partida.
- **distRiesgo**: es la distancia real a la que se encuentra cada fantasma de nuestro Pacman
- **distComida**: es la distancia real a la que se encuentra cada comida de nuestro Pacman

Definimos las siguientes constantes:

- **margenSeguro**: es la distancia a partir de la cual comenzaremos a huir de los fantasmas
- **velocidad**: es el factor por el que multiplicamos la puntuación que sustraeremos al Pacman por elegir un movimiento que nos acerque a una distancia menor que el margen seguro o le incrementaremos por elegir un movimiento que nos acerca a un fantasma asustado.

Comprobamos si los fantasmas están asustados y si lo están alentamos al Pacman a que se acerquen a él tanto más cuando más asustados estén.

Si los fantasmas no están asustados comprobamos la distancia de cada uno de los fantasmas a nuestro Pacman y si se supera la distancia del margen seguro penalizamos al Pacman a acercarse a ellos tanto más cuando más cerca estén

Calculamos la distancia a cada comida del mapa y sumamos al Pacman una puntuación por cada comida que tenga cerca, cuando más cerca este la comida mayor puntuación le añadiremos.

- Obsérvese que dos comidas a una distancia de 2 valen lo mismo que una comida a una distancia de 1.

Ejecuciones

$$\textbf{python pacman.py -p MinimaxAgent -l openClassic -a depth = 3}$$

/*TODO*/

$$\textbf{python pacman.py -p MinimaxAgent -l mediumClassic -a depth = 3}$$

/*TODO*/

Conclusiones

El valor asignado a las constantes ha sido establecido en base a numerosas pruebas de las que hemos obtenido las siguientes conclusiones:

- **margenSeguro**:
  - Un valor demasiado bajo hace que el Pacman no detecte a los fantasmas hasta que es demasiado tarde
  - Un valor demasiado alto hace que el Pacman se quede bloqueado, incluso cuando el fantasma se acerca
- **velocidad**:
  - Un valor demasiado bajo provoca que el Pacman cometa errores cuando la partida está muy avanzada debido a que la puntuación obtenida del estado del juego va creciendo conforme pasa el tiempo
  - Un valor demasiado alto provoca que el Pacman cometa errores al principio de la partida

Como ahora tenemos una función de evaluación el minimax debería de haber solucionado los problemas que habíamos encontrado en los tableros más grandes como openClassic y mediumClassic

/*TODO*/

No he sabido como ejecutarlo para hacer esta prueba

Apéndices:

Resultados del Autograder

Resultados

```
Finished at 18:13:38

Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 5/5
Question q5: 6/6
------------------
Total: 25/25
```

Apartado Q1

```
Question q1
===========

Pacman emerges victorious! Score: 1229
Pacman emerges victorious! Score: 1227
Pacman emerges victorious! Score: 1226
Pacman emerges victorious! Score: 1226
Pacman emerges victorious! Score: 1229
Pacman emerges victorious! Score: 1229
Pacman emerges victorious! Score: 1216
Pacman emerges victorious! Score: 1227
Pacman emerges victorious! Score: 1233
Pacman emerges victorious! Score: 1210
Average Score: 1225.2
Scores:        1229.0, 1227.0, 1226.0, 1226.0, 1229.0, 1229.0, 1216.0, 1227.0, 1233.0, 1210.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q1\grade-agent.test (4 of 4 points)
***     1225.2 average score (2 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (0 of 0 points)
***         Grading scheme:
***          < 10:  fail
***         >= 10:  0 points
***     10 wins (2 of 2 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  0 points
***         >= 5:  1 points
***         >= 10:  2 points

### Question q1: 4/4 ###
```

Apartado Q2

```
Question q2
===========

*** PASS: test_cases\q2\0-eval-function-lose-states-1.test
*** PASS: test_cases\q2\0-eval-function-lose-states-2.test
*** PASS: test_cases\q2\0-eval-function-win-states-1.test
*** PASS: test_cases\q2\0-eval-function-win-states-2.test
*** PASS: test_cases\q2\0-lecture-6-tree.test
*** PASS: test_cases\q2\0-small-tree.test
*** PASS: test_cases\q2\1-1-minmax.test
*** PASS: test_cases\q2\1-2-minmax.test
*** PASS: test_cases\q2\1-3-minmax.test
*** PASS: test_cases\q2\1-4-minmax.test
*** PASS: test_cases\q2\1-5-minmax.test
*** PASS: test_cases\q2\1-6-minmax.test
*** PASS: test_cases\q2\1-7-minmax.test
*** PASS: test_cases\q2\1-8-minmax.test
*** PASS: test_cases\q2\2-1a-vary-depth.test
*** PASS: test_cases\q2\2-1b-vary-depth.test
*** PASS: test_cases\q2\2-2a-vary-depth.test
*** PASS: test_cases\q2\2-2b-vary-depth.test
*** PASS: test_cases\q2\2-3a-vary-depth.test
*** PASS: test_cases\q2\2-3b-vary-depth.test
*** PASS: test_cases\q2\2-4a-vary-depth.test
*** PASS: test_cases\q2\2-4b-vary-depth.test
*** PASS: test_cases\q2\2-one-ghost-3level.test
*** PASS: test_cases\q2\3-one-ghost-4level.test
*** PASS: test_cases\q2\4-two-ghosts-3level.test
*** PASS: test_cases\q2\5-two-ghosts-4level.test
*** PASS: test_cases\q2\6-tied-root.test
*** PASS: test_cases\q2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q2\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q2\7-2c-check-depth-two-ghosts.test
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q2\8-pacman-game.test

### Question q2: 5/5 ###
```

Apartado Q3

```
Question q3
===========

*** PASS: test_cases\q3\0-eval-function-lose-states-1.test
*** PASS: test_cases\q3\0-eval-function-lose-states-2.test
*** PASS: test_cases\q3\0-eval-function-win-states-1.test
*** PASS: test_cases\q3\0-eval-function-win-states-2.test
*** PASS: test_cases\q3\0-lecture-6-tree.test
*** PASS: test_cases\q3\0-small-tree.test
*** PASS: test_cases\q3\1-1-minmax.test
*** PASS: test_cases\q3\1-2-minmax.test
*** PASS: test_cases\q3\1-3-minmax.test
*** PASS: test_cases\q3\1-4-minmax.test
*** PASS: test_cases\q3\1-5-minmax.test
*** PASS: test_cases\q3\1-6-minmax.test
*** PASS: test_cases\q3\1-7-minmax.test
*** PASS: test_cases\q3\1-8-minmax.test
*** PASS: test_cases\q3\2-1a-vary-depth.test
*** PASS: test_cases\q3\2-1b-vary-depth.test
*** PASS: test_cases\q3\2-2a-vary-depth.test
*** PASS: test_cases\q3\2-2b-vary-depth.test
*** PASS: test_cases\q3\2-3a-vary-depth.test
*** PASS: test_cases\q3\2-3b-vary-depth.test
*** PASS: test_cases\q3\2-4a-vary-depth.test
*** PASS: test_cases\q3\2-4b-vary-depth.test
*** PASS: test_cases\q3\2-one-ghost-3level.test
*** PASS: test_cases\q3\3-one-ghost-4level.test
*** PASS: test_cases\q3\4-two-ghosts-3level.test
*** PASS: test_cases\q3\5-two-ghosts-4level.test
*** PASS: test_cases\q3\6-tied-root.test
*** PASS: test_cases\q3\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\q3\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q3\7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q3\8-pacman-game.test
```

Apartado Q4

```
Question q4
===========

*** PASS: test_cases\q4\0-eval-function-lose-states-1.test
*** PASS: test_cases\q4\0-eval-function-lose-states-2.test
*** PASS: test_cases\q4\0-eval-function-win-states-1.test
*** PASS: test_cases\q4\0-eval-function-win-states-2.test
*** PASS: test_cases\q4\0-expectimax1.test
*** PASS: test_cases\q4\1-expectimax2.test
*** PASS: test_cases\q4\2-one-ghost-3level.test
*** PASS: test_cases\q4\3-one-ghost-4level.test
*** PASS: test_cases\q4\4-two-ghosts-3level.test
*** PASS: test_cases\q4\5-two-ghosts-4level.test
*** PASS: test_cases\q4\6-1a-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1b-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-1c-check-depth-one-ghost.test
*** PASS: test_cases\q4\6-2a-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2b-check-depth-two-ghosts.test
*** PASS: test_cases\q4\6-2c-check-depth-two-ghosts.test
*** Running ExpectimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running ExpectimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\q4\7-pacman-game.test

### Question q4: 5/5 ###
```

Apartado Q5

```
Question q5
===========

Pacman emerges victorious! Score: 960
Pacman emerges victorious! Score: 974
Pacman emerges victorious! Score: 968
Pacman emerges victorious! Score: 970
Pacman emerges victorious! Score: 970
Pacman emerges victorious! Score: 1168
Pacman emerges victorious! Score: 960
Pacman emerges victorious! Score: 1155
Pacman emerges victorious! Score: 1162
Pacman emerges victorious! Score: 960
Average Score: 1024.7
Scores:        960.0, 974.0, 968.0, 970.0, 970.0, 1168.0, 960.0, 1155.0, 1162.0, 960.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\q5\grade-agent.test (6 of 6 points)
***     1024.7 average score (2 of 2 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  1 points
***         >= 1000:  2 points
***     10 games not timed out (1 of 1 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 10:  1 points
***     10 wins (3 of 3 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 5:  2 points
***         >= 10:  3 points

### Question q5: 6/6 ###
```

Referencias

Todo el contenido teórico así como las imágenes utilizadas para este guion de prácticas han sido obtenidos de las siguientes fuentes:

Apuntes de la asignatura de IA (Ekaitz Jauregi, Eneko Agirre, Juanma Pikatza)

https://inst.eecs.berkeley.edu/~cs188/sp19

Guion de prácticas de los laboratorios