ESCUELA SUPERIOR DE INFORMÁTICA

UNIVERSIDAD DE CASILLA-LA MANCHA

Inteligencia Artificial e Ingeniería del Conocimiento

Práctica – Segunda Parte

Primer hito: descripción en pseudo-código

Jose Domingo López López
josed.lopez1@alu.uclm.es
Univ. de Castilla – La Mancha

24 de Marzo del 2009

Índice de contenido

1. Introducción

El objetivo de esta práctica es realizar un bot (agente racional) que pueda jugar de forma autónoma contra un adversario, permitiendo la posibilidad de realizar una pequeña competición. La estrategia de juego será dirigida por un algoritmo Mini-Max con poda alfa-beta. Para simplificar el diseño de dicho algoritmo, se hará una primera versión de éste sin implementar la poda y, por último, se hará la versión definitiva que incorporará la poda alfa-beta.

En un algoritmo Mini-Max se consideran dos jugadores a los que llamaremos Max y Min ya que, una vez que evaluamos un nodo, se supone que los valores altos son buenos para Max y los malos para Min. De esto se deduce que, dado que queremos que nuestro equipo consiga las mejores puntuaciones, nosotros seremos el equipo Max y nuestro rival será el equipo Min.

Considerando un árbol de juegos, la estrategia óptima puede determinarse examinando el **valor minimax** de cada nodo (si es un estado terminal es solamente su utilidad). Un algoritmo minimax calcula la decisión minimax del estado actual. La recursión avanza hacia las hojas del árbol, y entonces los valores minimax retroceden por el árbol cuando la recursión se va deshaciendo. Como se puede deducir, un algoritmo minimax realiza una **exploración primero en profundidad** completa del árbol de juegos, es decir, si la profundidad máxima de árbol es *m*, y hay *b* movimientos legales en cada punto, entonces la complejidad en tiempo del algoritmo minimax es $O(b^m)$ y la complejidad en espacio es *O(bm)*.

Como hemos visto, el problema de la búsqueda minimax es que el número de estados que tiene que examinar es exponencial en el número de movimientos. Lamentablemente no podemos eliminar el exponente, pero podemos dividirlo, con eficacia, en la mitad. La jugada es que es posile calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juegos aplicando la ténica de **poda alfa-beta**. Esta poda puede aplicarse en árboles de cualquier profundidad y, a menudo, es posible podar subárboles enteros. Además, es necesario tener en cuenta que los estados repetidos en el árbol de búsqueda pueden causar un aumento exponencial del coste de búsqueda. Esto se debe a permutaciones diferentes de la secuencia de movimientos que terminan en la misma posición. Para resolver este problema se utiliza una **tabla de transposición**, que tradicionalmente es una *tabla hash* idéntica a la *lista cerrada* que utilizamos en el algoritmo A* de la primera práctica. La forma de trabajar con dicha tabla es guardando la evaluación de cada posición la primera vez que se encuentre, de modo que no tenemos que volver a calcularla las siguientes veces.

Por último, hay que tener en cuenta que la **función de utilidad** (que comprueba si un estado es objetivo y le asigna un valor de utilidad) hay que modificarla ligeramente si no podemos explorar completamente el árbol de búsqueda (y no podremos ya que los turnos están limitados en tiempo), ya que no podemos explorar las ramas hasta las hojas y debemos cortar a una determinada profundidad. Esta nueva función recibe el nombre de **función de evaluación**, que devuelve una estimación de la utilidad esperada de una posición dada. En este momento, podemos plantearnos la posibilidad de implementar nuestro algoritmo mediante **profundidad iterativa** para que cuando se agote el tiempo, el programa nos devuelva el movimiento seleccionado por la búsqueda completa más profunda. El problema que surge con esta aproximación, es que el la función de evaluación debe aplicarse sólo a posiciones que son estables (es decir, improbablemente expuestas a grandes oscilaciones en un futuro próximo). A esta búsqueda suplementaria se le llama **búsqueda de estabilidad o reposo**.

2. Conceptos importantes

Para la realización de esta práctica es necesario tener en cuenta los siguientes conceptos:

- **Estado**. En la primera práctica un estado estaba compuesto por una lista de jugadores (ya que únicamente jugaba un equipo) y una lista de casillas del tablero que se habían ido modificando a lo largo de la partida. En esta segunda práctica, en la que juegan juegan dos equipos (el nuestro y el rival), es necesario hacer un pequeño ajuste para que el estado esté compuesto por una lista de equipos (cada equipo tendrá una lista con sus jugadores) y una lista de casillas modificadas. Pero, como hemos visto en la sección anterior, la complejidad en espacio de un algoritmo minimax es *O(bm)*. Dicho esto, puede ser interesante plantearse la posibilidad de almacenar el estado del tablero completo para no tener que recalcularlo cada vez que lo necesitemos aplicando los cambios indicados por la lista de casillas modificadas.

- **Función sucesor**. Esta función es similar a la que utilizamos en la primera práctica, solo que añadiremos un séptimo movimiento para poder cavar. De modo que serán dos bucles anidados en los que el primero recorrerá la lista de jugadores y el segundo hará cada uno de los siete posibles movimientos, generando así todos los posibles sucesores de un nodo.

- **Test terminal**. Esta función determina si un nodo contiene un estado objetivo. Un estado será objetivo cuando se hayan capturado todas las banderas del equipo contrario o no nos quede ningún jugador con vida.

- **Función de utilidad**. Nuestro juego es un **juego de suma no cero**. Esta función evaluará un nodo terminal y le asignará un valor de utilidad en función del número de banderas capturadas y la energía total de cada equipo, tratando así de maximizar las banderas que captura nuestro equipo y su energía; y de minimizar las banderas que captura el equipo rival y su energía. En esta función no es necesario tener en cuenta las distancais de los jugadores a las banderas restantes ya que estamos hablando de **nodos terminales**.

- **Función de evaluación**. Convierte los nodos no terminales en hojas terminales. Es necesario sustituir test terminal por un **test-límite** que decide cuando aplicar la función de evaluación (que sustituye a la función de utilidad), que nos devolverá una estimación de la utilidad esperada de una posición dada (revisar el concepto de *posición estable* para evitar evaluar posiciones expuestas a grandes oscilaciones en su valor en un futuro próximo).

3. Algoritmo minimax

A continuación se muestra el pseudo-código del algoritmo minimax:

```
# accion es una tupla de enteros (jugador,movimiento)


function decision-minimax(nodo)
   v=-INFINITO
   (accion,v)=max-valor(nodo)
   return accion


function max-valor(nodo):
   if test-terminal(nodo.estado,MAX):
      return utilidad(nodo.estado,MAX)
   else:
      v=-INFINITO
      accion=(-1,-1)
      sucesores=expandir(nodo,MAX)
      for s in sucesores:
         (accion_aux,v_aux)=min-valor(s)
         if v<v_aux:
            accion=s.accion
            v=v_aux
   return (accion,v)


function min-valor(nodo):
   if test-terminal(nodo.estado,MIN):
      return utilidad(nodo.estado,MIN)
   else:
      v=INFINITO
      accion=(-1,-1)
      sucesores=expandir(nodo.estado,MIN)
      for s in sucesores:
         (accion_aux,v_aux)=max-valor(s)
         if v>v_aux:
            accion=s.accion
            v=v_aux
   return (accion,v)
```

```
function test-terminal(estado,equipo):
    terminal=True
    # Si se han capturado todas las banderas, es terminal
    if estado.equipos[equipo].banderasCapturadas==banderasTotales:
        pass
    else:
        # Sino, comprobamos si hay algun jugador vivo
        # Si queda alguno vivo, no es terminal
        for jug in estado.equipos[equipo].jugadores:
            if jug.energia<=0:
                termnal=False
                break
    return terminal


function utilidad(estado,equipo):
    eq_aux=estado.equipos[equipo]
    utilidad=0
    utilidad+=valoracionEnergia(eq_aux,energiaTotal)
    utilidad+=valoracionCapturas(eq_aux.banderasCapturadas,banderasTotales)
    return utilidad
```

4. Poda alfa-beta

A continuación se muestra el pseudo-código del algoritmo minimax con poda alfa-beta:

```
# alfa: valor de la mejor alternativa (el valor más alto) encontrada
#hasta el momento para un nodo MAX
# beta: valor de la mejor alternativa (el valor más bajo) encontrada
#hasta el momento para un nodo MIN
# accion es una tupla de enteros (jugador,movimiento)


function busqueda-alfa-beta(nodo)
   v=-INFINITO
   (accion,v)=max-valor(nodo,-INFINITO,INFINITO)
   return accion


function max-valor(nodo,alfa,beta):
   if test-terminal(nodo.estado,MAX):
      return utilidad(nodo.estado,MAX)
   else:
      v=-INFINITO
      accion=(-1,-1)
      sucesores=expandir(nodo,MAX)
      for s in sucesores:
         (accion_aux,v_aux)=min-valor(s,alfa,beta)
         if v<v_aux:
            accion=s.accion
            v=v_aux
         if v>=beta: return(accion,v)
         alfa=max(alfa,v)
   return (accion,v)


function min-valor(nodo,alfa,beta):
   if test-terminal(nodo.estado,MIN):
      return utilidad(nodo.estado,MIN)
   else:
      v=INFINITO
      accion=(-1,-1)
      sucesores=expandir(nodo,MIN)
      for s in sucesores:
         (accion_aux,v_aux)=max-valor(s,alfa,beta)
         if v>v_aux:
            accion=s.accion
            v=v_aux
         if v<=alfa: return(accion,v)
         beta=min(beta,v)
   return (accion,v)
```

```
function test-terminal(estado,equipo):
    terminal=True
    # Si se han capturado todas las banderas, es terminal
    if estado.equipos[equipo].banderasCapturadas==banderasTotales:
        pass
    else:
        # Sino, comprobamos si hay algun jugador vivo
        # Si queda alguno vivo, no es terminal
        for jug in estado.equipos[equipo].jugadores:
            if jug.energia<=0:
                termnal=False
                break
    return terminal


function utilidad(estado,equipo):
    eq_aux=estado.equipos[equipo]
    utilidad=0
    utilidad+=valoracionEnergia(eq_aux,energiaTotal)
    utilidad+=valoracionCapturas(eq_aux.banderasCapturadas,banderasTotales)
    return utilidad
```

5. Función de evaluación

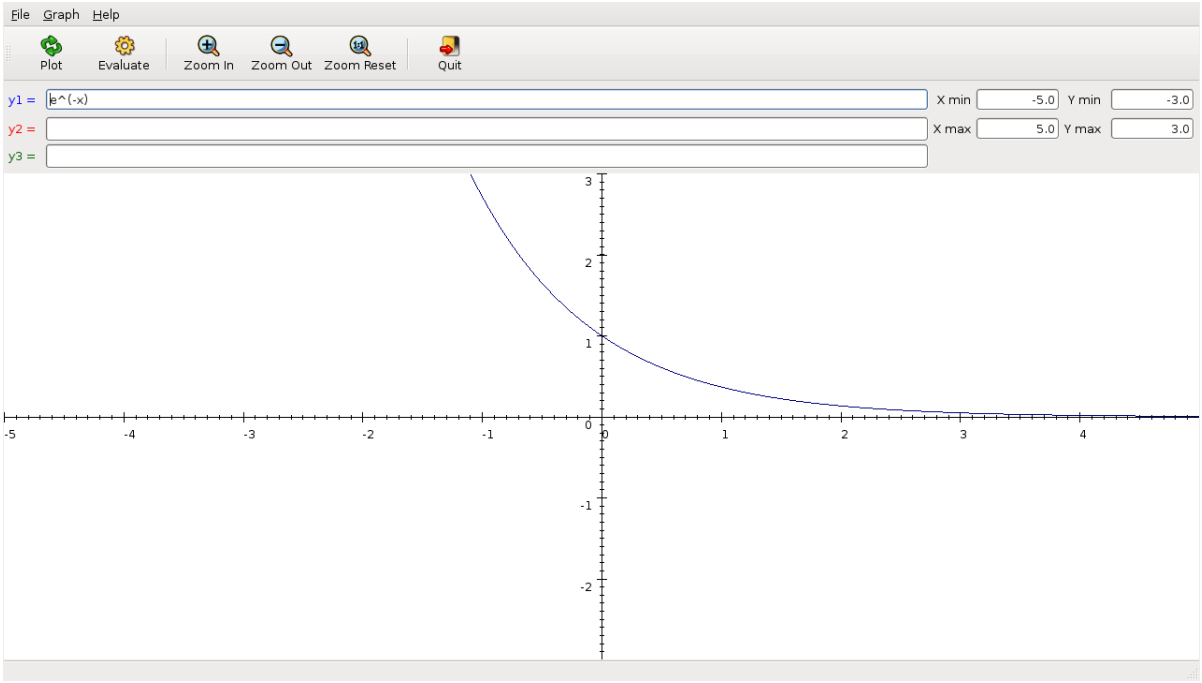
La función de evaluación (que sustituye a la función de utilidad), nos devolverá una estimación de la utilidad esperada de una posición dada y será aplicada cuando el test-límite (que sustituye al test-terminal) lo decida. De esto se deduce que la función de evaluación será similar a la función de utilidad pero tendrá que estudiar ciertos aspectos que en la función de utilidad no se tenían en cuenta por aplicarse únicamente a nodos terminales. También es necesario decir que la función de evaluación se aplicará teniendo en cuenta únicamente los parámetros del equipo que la lanza. De un primer vistazo podemos pensar que estos parámetros son:

- Número de banderas capturadas por el equipo.
- Distancia total de los jugadores del equipo a las banderas restantes.
- Energía total de los jugadores del equipo.

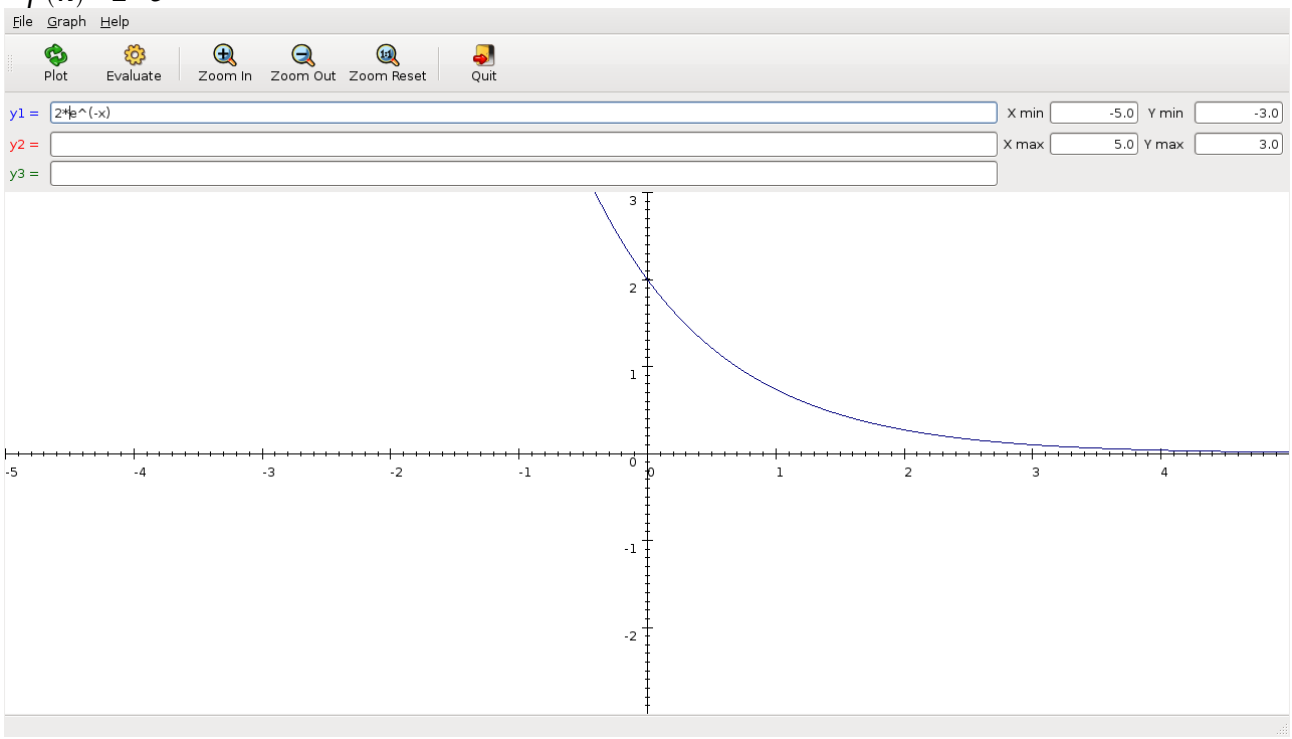
Para cada uno de estos parámetros, será necesario calcular su bondad. Esta bondad vendrá determinada en función de características específicas de la partida. Por ejemplo, tendrá una mayor bondad capturar una bandera en un mapa que alberga únicamente dos banderas, que capturar esa misma bandera en un mapa que alberga diez banderas.

Pero los valores de bondad no es algo trivial ya que MAX siempre buscará valores de bondad altos y MIN siempre buscará valores de bondad bajos. El problema viene cuando tratamos de calcular la utilidad de la distancia total de los jugadores del equipo a las banderas que no han sido capturadas: cuanto mayor sea la distancia, menor será su utilidad. Como podemos apreciar, es una relación inversa en la que los valores más prometedores son los más bajos. Para expresar esto, podemos suponer un eje de coordenadas en el que representamos la distancia a las banderas en el eje de abcisas y el valor de bondad en el eje de ordenadas. Para ello podemos hacerlo mediante la función $f(x)=e^{(-x)}$ la cual nos da unos valores de utilidad altos para distancias cortas, y por el contrario, unos valores de utilidad bajos para distancias largas. Pero esta función está muy acotada por lo que debemos añadir dos constantes que tendrán un valor u otro en función de las características específicas del mapa. De este modo, nuestra ecuación quedaría de la siguiente forma: $f(x)=v*e^{(-1/d*x)}$, donde para mayor *v* más importante será la distancia a las banderas y para mayor *d* más despacio decrecerá el valor de utilidad de la distancia. Podemos verlo más claro en las siguientes imágenes:

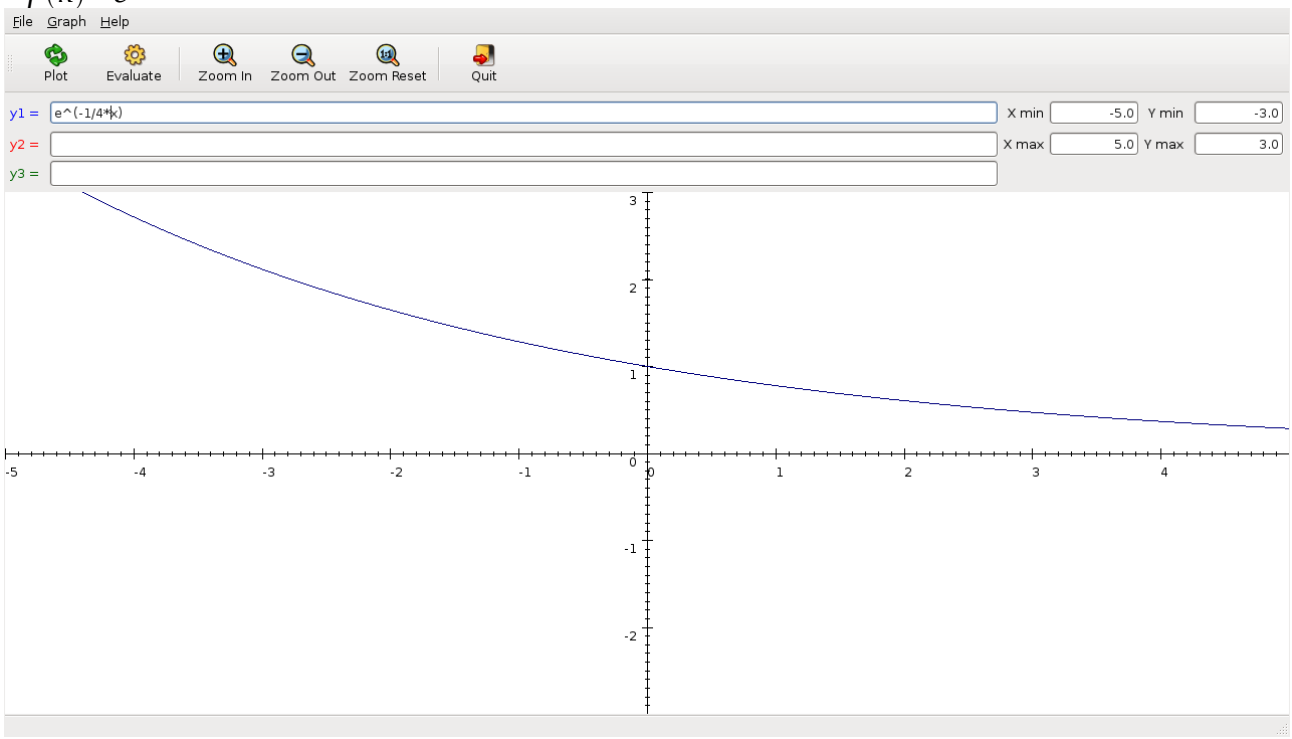
$f(x)=e^{(-x)}$



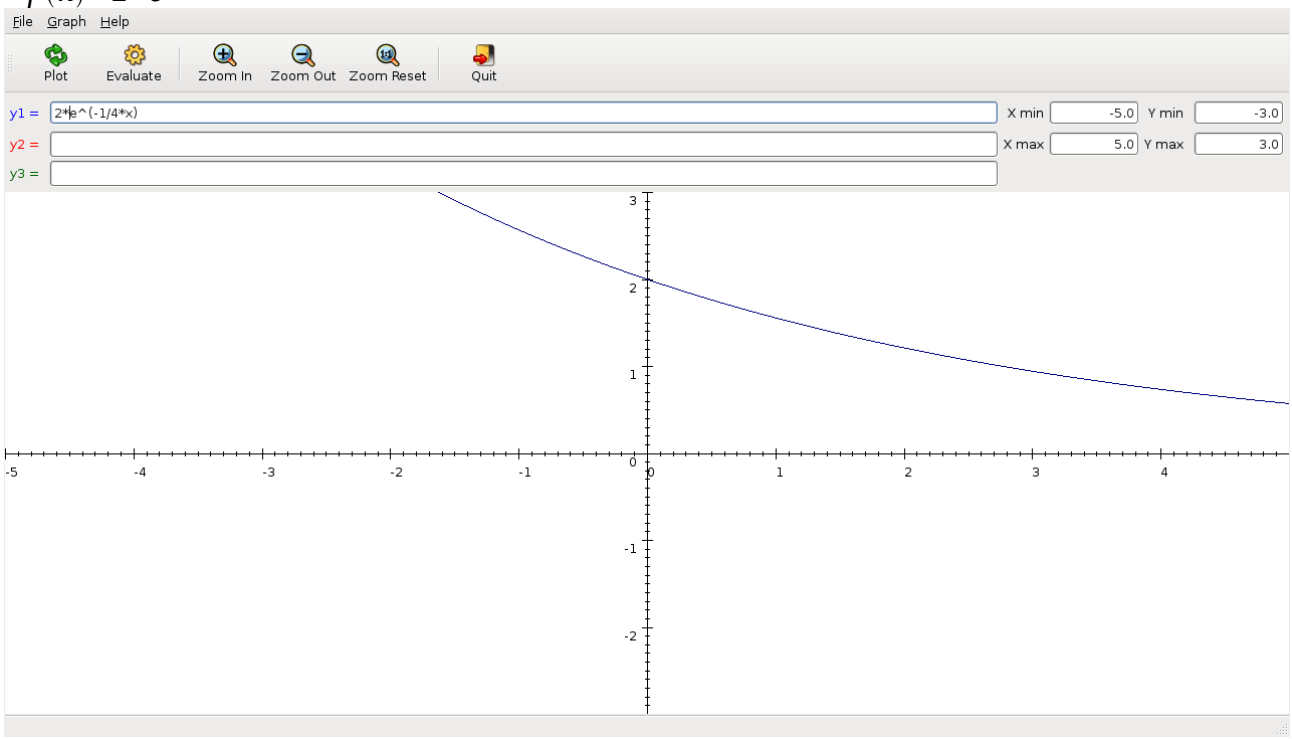
$f(x)=2*e^{(-x)}$



$f(x)=e^{(-1/4*x)}$



$$f(x)=2*e^{(-1/4*x)}$$



El método de evaluar ambos equipos, tanto MAX como MIN, será el mismo. Para saber quien tiene una posición ganadora, simplemente restaremos eval(MAX)-eval(MIN): cuanto mayor sea el resultado, mejor para MAX y viceversa, cuando menor sea el resultado, mejor para MIN.

A continuación se muestra un sencillo pseudocódigo perteneciente a la función de evaluación:

```
function eval(estado,equipo):
    # Calculamos los parametros especificos de cada componente en
    #funcion de las caracteristicas del mapa
    (banderas_param,distancia_param,energia_param)=calcularParametrosEspecificos(estado)

    # Calculamos los valores de bondad de cada componente
    bondad_banderas=banderas_param*evaluarBanderas(estado,equipo)
    bondad_distancia=distancia_param*evaluarDistancia(estado,equipo)
    bondad_energia=energia_param*evaluarEnergia(estado,equipo)

    # Calculamos la utilidad de cada componente dandoles un peso especifico
    utilidad_banderas=peso_banderas*bondad_banderas
    utilidad_distancia=peso_distancia*bondad_distancia
    utilidad_energia=peso_energia*bondad_energia

    # Calculamos la utilidad total
    utilidad=utilidad_banderas+utilidad_distancia+utilidad_energia

    return utilidad
```

6. Bibliografía

- Russell, S. J.; Norvig, P. *“Inteligencia Artificial: Un Enfoque Moderno”, 2ª edición.*