Universidad Católica del Norte
Departamento de Ingeniería de Sistemas y Computación

Inteligencia Artificial

Taller 1

Búsqueda con adversarios

Autor(es):
Pablo Fontecilla
Claudio González

Profesor Responsable:
Juan Bekios Calfa

11 de noviembre de 2017

1. Introducción

Cuando un agente se encuentra en un entorno en el que existe al menos un agente del cual dependa su acción, se le llama entorno multiagente. En el caso de que además de que un multiagente esté en competencia con otro por ver quién maximiza su rendimiento, se les llama búsqueda con adversarios. Estas búsquedas pueden ser utilizadas para resolver diversos tipos de juegos.

En este informe se detalla cómo fueron implementados los algoritmos Minimax y Poda alfa-beta con una heurística adicional para encontrar las mejores jugadas en el juego de Nim de forma eficiente.

El taller fue implementado en el lenguaje de programación Python, en la versión 2.7.

2. Formulación del problema

Con el fin de entender de mejor manera cómo funcionan las búsquedas con adversario, se ha encargado implementar algoritmos de Minimax y Poda alfa-beta para encontrar las mejores jugadas para ganar contra un adversario en el juego de Nim.

Este juego consiste en que dos jugadores, colocan cerillas sobre una superficie separadas en filas o grupos (en este caso 3 filas). Cada fila contiene una cantidad de cerillas (En una 7, 5 y en otra 3). El primer jugador toma cualquier número de cerillas de una fila, entre uno y el total de la fila, pero sólo de una fila. El segundo jugador hace su jugada de manera similar, retirando algunas de las cerillas que quedan, y los jugadores van alternándose en sus jugadas. Pierde el jugador que retire la última cerilla.

En base a las reglas del juego, se procedió a analizar cómo funcionan los algoritmos y cómo deberían adaptarse para seguir reglas.

3. Diseño de los algoritmos

Ambos algoritmos tienen:

- Nodo: El nodo está compuesto por:
  - El estado, que contiene una lista con tres valores, donde cada uno representa la cantidad de palitos en una fila.
  - El valor asignado por la función heurística.
- Estado Inicial: El estado inicial del algoritmo depende de quien juegue primero. En el caso de jugar primero la máquina el estado inicial será siempre todos los palos de fósforo dispuestos en la manera que el juego inicia, es decir, un lista con los valores [7, 5, 3]. Mientras que si el jugador comienza la partida, el estado inicial puede ser cualquiera de los sucesores del estado mencionado anteriormente.
- Estado objetivo: El estado objetivo es aquel en el que solo queda un palito de fósforo, es decir, una lista con los valores [1,0,0], [0,1,0] o [0,0,1].
- Función de sucesor: Los sucesores son solo la mejor y peor jugada posible. Para esto se obtienen todos los hijos del estado que se está expandiendo y se les asigna un valor de acuerdo a la heurística (La cual se explicará más adelante). Con este valor se comparan todos los sucesores y se conserva solo los que tengan el mayor y menor valor.

- Función de término: El algoritmo termina de ejecutarse cuando se encuentra en el estado objetivo o cuando ya no se pueden realizar más movimientos, ya que no quedan palos.
- Función heurística: Para determinar qué tan bueno es un estado se utilizó un método que consiste en convertir cada uno de los valores de la lista a números binarios y sumarlos sin acarreo, si el resultado es cero quiere decir que esa jugada es una posición que llevará a la victoria, de lo contrario llevará a la derrota (Si la persona sabe jugar bien, sino igual puede equivocarse). Además, para tener una mayor chance de que el oponente se equivoque, si se está en una posición de derrota el resultado entregado por la función de heurística es la cantidad total de palitos en juego. Del mismo modo si se está en una posición de victoria el resultado entregado será 40 menos el total de palitos, con el objetivo de terminar la partida más rápido. Existen algunos casos donde este método no funciona, estos fueron tratados de manera particular:
  - La función objetivo entrega un valor de -50.
  - Si se encuentra en el estado de termino entrega un valor de 50.
  - En caso de que la cantidad de ceros sea igual a dos, por ejemplo, [0, 3, 0] se entrega un valor de -45.
  - En caso de que la cantidad de unos sea igual a dos, por ejemplo, [1, 3, 1] se entrega un valor de -45.
  - Si todos los valores son unos, es decir, [1, 1, 1] entrega un valor de 45.

Con lo mencionado anteriromente se implementaron el algoritmo Minimax y Poda alfa-beta de manera recursiva.

4. Evaluación de los algoritmos

4.1. Evaluación del orden de magnitud en tiempo y espacio requerido por ambos algoritmos.

Ambos algoritmos tienen un orden de magnitud de tiempo de:
$O(n^2)$
Y un orden de magnitud de espacio de:
$O(n^2)$
Esto es debido a que siempre expande solo 2 nodos.

4.2. Evaluación empírica del costo computacional de cada algoritmo.

El algoritmo minimax en el peor caso, es decir, el primer movimiento toma un tiempo de 0.00906 segundos en promedio; mientras que el algoritmo poda alfa-beta demora 0.00881 en promedio.

Para realizar las pruebas del algoritmo se utilizó un computador con:

- Procesador: Intel Pentium(R) CPU B960 @ 2.20GHz $\times$ 2.
- Memoria: 3,8 GiB.

5. Resultados obtenidos.

En un principio, al implementar el algoritmo Minimax normal, el agente logró deducir la mejor jugada, sin embargo tardaba bastante en encontrarla (sobretodo si jugaba en primer turno). Por ello, con la heurística modificada se pudo reducir las búsquedas, logrando deducir la jugada de forma casi instantánea.

El algoritmo de Poda alfa-beta normal implementado desde un comienzo sí funcionaba de forma eficiente (tiempo máximo de respuesta era de 3 segundos), pero modificando la heurística de la misma forma que se modificó la del minimax normal se consiguió una respuesta igual de instantánea.

6. Conclusiones

Una vez aplicando la heurística apropiada, el agente es casi invencible. De hecho, si juega contra un oponente que posea la misma heurística, siempre ganará el agente que juegue el primer turno. Se puede concluir que los algoritmos de búsqueda contra adversarios, aunque simples, pueden ser muy poderosos a la hora maximizar su rendimiento. Para este problema sería posible hacer un algoritmo que siempre encuentre la solución correcta sin necesidad de realizar una búsqueda, ya que siempre es posible determinar cuál será la mejor jugada evaluando los sucesores con la heurística implementada.

7. Apéndice: Código de ambos algoritmos

7.1. Minimax

```
def maximo(estado_actual, jugadas):
    """
    Retorna la jugada con mayor valor de la heuristica.
    :param estado_actual: Jugada a evaluar.
    :param jugadas: Variable donde se guarda la jugada a realizar.
    :return: Valor de laheuristica.
    """
    if estado_actual.estado_objetivo():
        return estado_actual.heuristica
    if estado_actual.estado_final():
        return estado_actual.heuristica
    # Obtener los sucesores.
    valor_max = -100
    aux = nodo.Nodo([0, 0, 0])
    for sucesor in obtener_sucesores(estado_actual.estado):
        valor = minimo(sucesor, jugadas)
        # Guardar la mejor jugada.
        if valor >= valor_max:
            aux = sucesor
            valor_max = valor
```

```
    # Guardar la posible jugada a realizar.
    jugadas[0] = aux.estado
    return valor_max


def minimo(estado_actual, jugadas):
    """
    Reorna la jugada con menor valor de la heuristica.
    :param estado_actual: Jugada a evaluar.
    :param jugadas: Variable donde se guarda la jugada a realizar.
    :return: Valor de laheuristica.
    """
    if estado_actual.estado_objetivo():
        return estado_actual.heuristica * -1
    if estado_actual.estado_final():
        return estado_actual.heuristica * -1
    # Obtener los sucesores.
    valor_min = 100
    aux = nodo.Nodo([0, 0, 0])
    for sucesor in obtener_sucesores(estado_actual.estado):
        # Guardar la peor jugada.
        valor = maximo(sucesor, jugadas)
        if valor <= valor_min:
            aux = sucesor
            valor_min = valor
    # Guardar la posible jugada a realizar.
    jugadas[0] = aux.estado
    return valor_min
```

7.2. Poda alfa-beta

```
def maximo(estado_actual, alfa, beta, jugadas):
    """
    Retorna la jugada con mayor valor de la heuristica.
    :param beta: Valor para determinar cuando podar en min.
    :param alfa: Valor para determinar cuando podar en max.
    :param estado_actual: Jugada a evaluar.
    :param jugadas: Variable donde se guarda la jugada a realizar.
    :return: Valor de laheuristica.
    """
    if estado_actual.estado_objetivo():
        return estado_actual.heuristica
    if estado_actual.estado_final():
        return estado_actual.heuristica
    # Obtener los sucesores.
    for sucesor in obtener_sucesores(estado_actual.estado):
```

```
        valor = minimo(sucesor, alfa, beta, jugadas)
        # Verificar si se debe modificar el valor de alfa
        if valor > alfa:
            alfa = valor
        # Guardar la posible jugada a realizar.
        jugadas[0] = sucesor.estado
        # Verificar si se debe hacer poda.
        if valor >= beta:
            return alfa
    return alfa


def minimo(estado_actual, alfa, beta, jugadas):
    """
    Reorna la jugada con menor valor de la heuristica.
    :param beta: Valor para determinar cuando podar en min.
    :param alfa: Valor para determinar cuando podar en max.
    :param estado_actual: Jugada a evaluar.
    :param jugadas: Variable donde se guarda la jugada a realizar.
    :return: Valor de laheuristica.
    """
    if estado_actual.estado_objetivo():
        return estado_actual.heuristica * -1
    if estado_actual.estado_final():
        return estado_actual.heuristica * -1
    # Obtener los sucesores.
    for sucesor in obtener_sucesores(estado_actual.estado):
        valor = maximo(sucesor, alfa, beta, jugadas)
        # Verificar si se debe modificar el valor de alfa
        if valor < beta:
            beta = valor
        # Guardar la posible jugada a realizar.
        jugadas[0] = sucesor.estado
        # Verificar si se debe hacer poda.
        if valor <= alfa:
            return beta
    return beta
```