Tarea Programada № 2

José Castro, Instituto Tecnológico de Costa Rica 24/2/2017

Búsqueda en Juegos

Esta tarea es sobre búsqueda en juegos y se enfoca sobre el juego de “Connect Four” o Conectar Cuatro en español. Este juego existe desde hace mucho tiempo y se conoce por nombres distintos, fue comercializado recientemente por Milton Bradley.

Consta de un tablero de 7x6 posiciones. El tablero esta organizado verticalmente en 7 columnas, 6 posiciones por columna de la siguiente manera:

```
.  0  1  2  3  4  5  6
0  *  *  *  *  *  *  *
1  *  *  *  *  *  *  *
2  *  *  *  *  *  *  *
3  *  *  *  *  *  *  *
4  *  *  *  *  *  *  *
5  *  *  *  *  *  *  *
```

Dos jugadores se turnan para agregar fichas al tablero. Las fichas se pueden agregar a cualquier columna que todavía no se encuentre llena (tenga menos de 6 fichas). Cuando se agrega una ficha, inmediatamente cae hasta la posición desocupada que se encuentra más abajo en la columna.

El juego lo gana el primer jugados que logre poner cuatro fichas en linea, esto puede ser:

verticalmente

```
.  0  1  2  3  4  5  6
1
2              0
3              0        X
4              0        X
5              0        X
```

horizontalmente

```
.  0  1  2  3  4  5  6
1
2
3
4
5  X  X  X  0  0  0  0
```

o diagonalmente

```
.  0  1  2  3  4  5  6
1
2                    0
3                 0  X
4              0  X  X
5           0  X  X  X
```

```
.  0  1  2  3  4  5  6
1
2           X
3           0  X  X  X
4           0  0  X  X
5  X        0  0  X  X
```

Jugando el juego

Se puede experimentar como funciona el juego compitiendo contra la computadora, por ejemplo, eliminando el comentario de la siguiente linea usted puede jugar con las blancas (de primero) mientras que la computadora hace búsqueda minimax a una profundidad de 4 con las negras.

```
run_game(basic_player, human_player)
```

Para cada movida, el programa le solicitará indicar en cual columna quiere poner su ficha. La interacción puede ir de la siguiente manera:

```
Player 1 (X) puts a token in column 0
```

```
.  0  1  2  3  4  5  6
1
2
3
4
5  X

Pick a column #: -->
```

En este juego, el jugador 1 acaba de agregar una ficha a la columna 0. El juego le esta solicitando a usted, como jugador 2, por el número de la columna en a que quiere agregar su ficha. Supongamos que desea agregar la ficha en la columna 1, entonces debe digitar '1' y presionar Enter.

Mientras tanto, la computadora esta calculando la mejor movida que puede hacer revisando el arbol de búsqueda a una profundidad de 4 (dos movidas para el y dos para usted). Si lee mas adelante en este pdf, se explica como se pueden crear jugadores que efectuen la búsqueda a profundidades arbitrarias.

El Código

Aquí se mecionan los archivos que ocupa para la tarea. El código tiene documentación interna también, sientase en confiaza de leerla.

ConnectFourBoard

`connectfour.py` contiene una clase llamada `ConnectFourBoard`. Como se puede imaginar la clase encapsua la noción de el tablero de Connect Four.

Los objetos de tipo `ConnectFourBoard` son *inmutables*. Si no ha estudiado mutabilidad, no se preocupe: esto solo significa que dada una instancia de la clase ConnectFourBoard, incluidas la posición de las fichas, nunca cambiará después de ser creada. Para hacer una movida en el tablero (o mas bien, el código complementario que se provee para usted) usted debe crear un nuevo tablero en un nuevo objeto `ConnectFourBoard` con su nueva ficha en la posición correcta. Esto hace que sea mucho mas sencillo hacer un arbol de búsqueda de tableros: Usted puede tomar su tablero inicial e intentar movidas distintas sin modificar el estado, pero antes de decidir que

hacer exáctamente. La búsqueda por `minimax` proveida toma ventaja de esto. Revise las funciones `get_all_next_moves, minimax` y `minimax_find_board_value` que se encuentran en `basicplayer.py`

Asi que para hacer una movida en el tablero usted puede digitar lo siguiente:

```
>>> myBoard = ConnectFourBoard()
>>> myBoard

   0  1  2  3  4  5  6
0
1
2
3
4
5

>> myNextBoard = myBoard.do_move(1)
>> myNextBoard

   0  1  2  3  4  5  6
0
1
2
3
4
5     X

>> myBoard # Solo para mostrar que no ha cambiado

   0  1  2  3  4  5  6
0
1
2
3
4
5

>>>
```

Hay bastantes métodos en la objeto ConnectFourBoard. esta bienvenido a utilizar cualquiera de ellos, mucho son métodos de ayuda que se utilizan por el código del tester, solo esperamos que necesite los siguientes métodos:

- `ConnectFourBoard()` (el constructor) – Crea una nueva instancia de ConnectFourBoard. La puede llamar sin argumentos y creara un nuevo tablero en blanco.
- `get_current_player_id()` – Devuelve el número de ID del jugador que tiene el turno actualmente.
- `get_other_player_id()` – Devuelve el número de ID del jugador que no tiene el turno actualmente.
- `get_cell(row, col)` – Devuelve el número de ID que tiene una ficha en la posición indicada, o 0 si posición esta vacia.
- `get_top_elt_in_column(column)` – Devuelve el ID del jugador que tiene la ficha mas arriba en la columna. Retorna 0 si la columna esta vacia.
- `get_height_of_column(column)` – Devuelve el número de fila del de la primer fila desocupada en la columna. Retorna -1 si la columna esta llena, y 6 si la columna esta vacia. NOTA: este es el índice de la fila no el verdadero “alto” de la columna, los indices cuentan desde 0 en la fila mas arriba hasta 5 en la mas abajo.
- `do_move(column)` – Devueve un nuevo tablero con la ficha del jugador actual en la columna indicada. El nuevo tablero indicará que ahora es el turno del otro jugador.
- `longest_chain(playerid)` – Devuelve el largo de la cadena de fichas contigua más larga del jugador especificado. Una cadena se define por as regas de Connect Four lo que significa que el primer jugador en construir una cadena de 4 fichas gana el juego.
- `chain_cells(playerid)` – Devuelve un conjunto de tuplas de Python para cada cadena de fichas de largo 1 o mas que estan controladas por el jugador actual.
- `is_win()` – Retorna el número de ID del jugador que ha ganado, 0 en caso que aún no exista un ganador.
- `is_game_over()` – Devuelve true si el juego ha terminado. Utillize `is_win` para encontrar el ganador.

Note que, como los objetos de ConnectFourBoards son inmutables pueden ser utilizados como llaves en los diccionarios y pueden insertarse en objets set() de Python.

Otras funciones útiles

Hay otro conjunto de funciones útiles en esta tarea que no son miembros de ninguna clase. Son las siguientes:

- `get_all_next_moves(board)` (`basicplayer.py` – Devuelve un generador de todas las movidas que se pueden hacer en el tablero actual.
- `is_terminal(depth, board)` (`basicplayer.py`) – Retorna True si: o bien se obtiene la profundidad de 0, o bien el juego esta en el estado de terminado (game over).
- `run_search_function(board, search_fn, eval_fn, timeout)` - (`util.py`) – Corre la función de búsqueda con profundidad iterativa, por el tiempo especificado. Descrita en detalle mas adelante.
- human_player(`connectfour.py`) – Un jugador especial que solicita por medio de consola dónde es que se debe ubicar la ficha. Vea abajo para la documentación de jugadores.
- `count_runs()` (`util.py`) – Este es un Decorator de Python que cuenta cuantas veces la función que decora ha sido llamada. Vea la definición del decorador para las instrucciones de como se debe utilizar. Puede ser util para confirmar que usted ha implementado poda alpha-beta de manera correcta: puede decorar su función de evaluación y verificar que se esta llamando la cantidad correcta de veces.
- `run_game(player1, player2, board = ConnectFourBoard())` (`connectfour.py`) – Corre el juego de Connect Four utilizando los dos jugadores especificados. 'board' puede ser especificada si se quiere empezar el juego en algún estado inicial específico.

Escribiendo su algoritmo de búsqueda

Esta tarea es acumulativa con la anterior, en es sentido que ocupara los procedimientos de búsqueda que fueron proveidos en la tarea anterior. En particular los archivos `search.py` y `graphs.py`

Funciones de "evaluación"

En esta tarea usted implementará dos funciones de evaluación para la búsqueda minimax y alpha-beta: `focused_evaluate` y `better_evaluate`. Funciones de evaluación toman como argu-

mento una instancia de la clase ConnectFourBoard. Retornan un indice entero indicando qué tan favorable es el tablero para el jugador de turno.

La intención de `focused_evaluate` es simplemente que empieze a explorar el mundo de las funciones de evaluación. Asi que la función se supone que debe ser muy simple. Usted debe lograr que su jugador gane mas rápidamente, o pierda mas lentamente.

La intención de `better_evaluate` es ir mas allá de funciones simples de evaluación estática y lograr que su función le gane a la función por defecto: `basic_evaluate`. Hay muchas maneras de hacer esto, pero las soluciones más comunes involucran conocer qué tan lejos ha llegado en el juego en un determinado momento. También note que en cada turno se agrega una ficha al tablero. Hay unas cuantas funciones en la clase ConnectFourBoard que le dan información útil sobre las fichas en el tablero; tiene libertad de utilizarlas. Puede también revisar el código fuente de `def basic_evaluate(board)` en `basicplayer.py` de la función que esta tratando de vencer.

Funciones de búsqueda

Como parte de esta tarea debe implementar un algoritmo de búsqueda de alpha-beta. Puede utilizar como base el algoritmo de `minimax` que se encuentra en `basicplayer.py`

Su función de `alpha_beta_search` debe tener los siguientes parámetros:

- `board` – La instancia de la clase ConnectFourBoard que representa el estado actual de juego.
- `depth` – La profundidad máxima que puede alcanzar el árbol de búsqueda.
- `depth` – La función de evaluación que debe usar para evaluar tableros.

Debe permitir que su función acepte opcionalmente otros dos argumentos:

- `get_next_moves_fn` – una función que dado un tablero/estado retorna los tableros succesores. Por defecto `get_next_moves_fn` toma como valor la función `basicplayer.get_all_next_moves`.
- `is_terminal_fn` – una función que dado una profundidad y un tablero/estado, retorna True o False. True si el tablero/estado es terminal y que se debe hacer evaluación estática. En caso de no indicarse este `is_termina_fn` debe tomar e valor de `basicplayer.is_terminal`.

Debe utilizar estas funciones en su implementación para buscar los próximos tableros y revisar condiciones de finalización.

La búsqueda debe retornar el `número de columna` en la cual debe agregar la ficha. Si empieza a tener errores masivos del tester, revise que esta retornando el número de columna y no el tablero completo!

TIP: Se agregó un archivo llamado `tree_searcher.py` para ayudarle a despulgar su implementación de la búsqueda alpha-beta. Contiene codigo que revisará su búsqueda de alpha-beta en 'arboles de juego estáticos, el tipo que usted puede revisar a pie. Para despulgar su búsqueda alpha-beta debe ejecutar: `python tree_searcher.py` y revisar visualmente si la salida de su programa retorna la movida correcta en árboles de juego simples. Sólo después de que haya superado los tests de `tree_searcher` debería ir y ejecutar el tester completo.

Creando un Jugador

Para poder jugar el juego, usted debe convertir su algoritmo de búsqueda en un `jugador`. Un jugador es una función que toma como parámetro un tablero y retorna la columna en la cual se desea agregar una ficha.

Note que estos requerimientos son bastante similares a los de la función de búsqueda, asi que puede definir su jugador básico de la siguiente manera:

Listing 1: Ejemplo de Jugador Simple.

```
1 def mi_jugador(board):
2     return minimax(board, depth=3, eval_fn=focused_evaluate, timeout=5)
```

O mas succintanete (pero equivalente):

Listing 2: Jugador Simple en declaración lambda.

```
1 mi_jugador = lambda board: minimax(board, depth=3, eval_fn=focused_evaluate)
```

Sin embargo este código asume que desea evaluar solo a una profundidad específica. En clase discutimos el concepto de profundidad iterativa. Se provee una función de ayuda llamada `run_search_function` para crear un jugador que hace profundidad iterativa basado en una función gen'erica de búsqueda. Usted puede crear un jugador de profundidad iterativa de la siguiente manera:

Listing 3: Jugador con profundidad iterativa.

```
1 mi_jugador = lambda board: run_search_function(board, search_fn=minimax, ↩
    eval=focused_evaluate)
```

Puede notar que cuando juega contra la computadora, pareciera que hace movidas “estúpidas”. Si usted obtiene la ventaja tal que esta garantizado a ganar si hace las movidas correctas, la computadora puede que parezca desinteresarse y dejar que usted gane sin siquiera intentar defenderse. O bien, si la computadora esta en una posición donde claramente gana, puede que empiece a “jugar” con usted y hacer movidas irrelevantes que no cambian el desenlace del juego.

Esto no es “estúpido” desde el punto de vista de búsqueda minimax. Si todas las movidas conducen al mismo desenlace, ¿Qué importa cuáles movidas ejecute la computadora primero?

Esta no es la manera en que personas generalmente juegan. Las personas buscan ganar de la manera más rápida posible cuando ven que pueden ganar, y perder lentamente contra un oponente dandole así varias oportunidades de cometer un error. Un pequeño cambio en la función estática de evaluación hará que la computadora juega de esta manera también.

- En `lab3.py` escriba una función de evaluación: `focused_evaluate`, que prefiere posiciones de gane que sucedan más rápido y posiciones perdedoras que sucedan mas tarde.

Le servirá seguir las siguientes indicaciones:

- No modifique los valores “normales” (valores que son como 1 o -2, 1000 o -1000). No necesita cambiar cómo el procedimiento evalúa posiciones que no son un gane o una pérdida garantizada.
- Indique un gane seguro con un valor que es mayor o igual a 1000, y una pérdida segura con un valor que es menor o igua a -1000.
- Recuerde que `focus_evaluate` debe ser muy simple, asi que no introduzca heurísticas elaboradas aqui. Guarde sus ideas para cuando mas tarde implementa `better_evaluate`.

Los jugadores computarizados que ha utilizado hasta ahora calzan bien en el Museo Británico - están evaluando todas las posiciones hasta una cierta profundidad, aún aquellas que se sabe que son inútiles. Puede hacer la búsqueda mucho mas eficiente si utiliza búsqueda alpha-beta.

- Escriba un procedimiento `alpha_beta_search` que trabaja como `minimax` excepto que utiliza poda alpha-beta para reducir el espacio de búsqueda.

- `lab3.py` define dos valores `INFINITY` y `NEG_INFINITY` que debe utilizar como los valores de ∞ y $-\infty$ respectivamente.

Este procedimiento se llama por `alphabeta_player` definido en `lab3.py`

Su procedimiento será revisado con árboles de juego, no se sorprenda si la entrada no siempre parece un tablero de Connect 4.

Pistas

En clase describimos el alpha-beta en términos de un jugador maximizando un valor y el otro minimizándolo, sin embargo, es probable que sea más facil escribir su código si cada jugador lo que intenta es *maximizar* el valor desde su punto de vista. Asi cada vez que el algoritmo ve hacia adelante a la movida de su contrincante, *niega* el valor resultado para obtener el valor real desde su punto de vista, esta variante del algoritmo minimax se le llama *negmax*, y así es como trabaja la función de minimax que se provee en el código de la tarea.

En su búsqueda por el árbol necesitará darle seguimiento con cuidado al rango alpha-beta, porque necesita negar éste rango (si utiliza negmax) cuando lo propaga al bajar en el árbol. Si sus valores, por ejemplo, determinados para el rango son [3,5] – en otras palabras, alpha=3 y beta = 5 – entonces los valores válidos para el otro jugador serán [-5,-3]. Si utiliza este truco de negación, sólo deberá implementar una función en vez de dos (minmax y maxmin), y debe propagar los valores de la siguiente manera:

```
newalpha = -beta
newbeta = -alpha
```

Esto es mas compacto que la representación que utilizamos en clase, y captura la noción de que el jugador evalúa las opciones del oponente tal como si él estuviera tomando dichas decisiones.

Un error común es permitir que el juego continue mas allá del fin del juego. Asegúrese de utilizar is_terminal_fn para determinar bien el fin.

Una mejor función de evaluación

Este problema va a ser un poco diferente, no hay una única manera correcta de hacerlo - tomará un poco de creatividad y de pensamiento sobre el funcionamiento del juego.

Su objetivo es escribir un procedimiento estático de evaluación que le gane a`basic_player` que se provee en la tarea. Es evaluado por el test `run_test_game_1` que juega `your_player` contra `basic_player` en un torneo de 4 juegos. Claramente si sólo juega basic_player contra sí mismo cada jugador ganará aproximadamente la misma cantidad de juegos que pierde. Queremos aquí que usted mejore esto y que diseñe un jugador que en el torneo de cuatro juegos gane por lo menos el doble de veces que las que pierde.

Sobra decir que su algoritmo debe ganar legítimamente y no interfiriendo de alguna manera con el rendimiento del otro jugador. Esto no es Spassky vs Fischer.

Una advertencia adicional con respecto a este tema: hay bastante de investigación e publicada e información en linea sobre “Connect Four”, y sobre algoritmos de búsqueda en Python. Esta bienvenido a leer los documentos que quiera, siempre y cuando los cite incluyendo el comentario apropiado en la parte relevante de código. Sin embargo, no debe utilizar implementaciones completas de los algoritmos de búsqueda elaboradas por terceros, y no debe escribir código que acceda a recursos en linea mientras esta ejecutando, la pena es 0 en a nota si copia o plagia código en linea o de sus compañeros.

Consejo

Para una función de evaluación, la simplicidad puede ser mejor! Es mucho mas útil tener tiempo para explorar mas a fondo el árbol de búsqueda que expresar de manera perfecta el valor de una posición. Éste es el motivo por el cual muchos juegos (no es el caso de Connect 4) toma algo de esfuerzo ganarle a la heurística “cantidad de movidas disponibes”, no se puede obtener algo más simple y todavía tener información que es útil para ganar el juego.

Si escribe una función de evaluación muy complicada, no tendrá tiempo de explorar tan a fondo en el árbol como `basic_evaluate`. Mantenga esto siempre en mente.

NOTAS IMPORTANTES

Después de impementar `better_evaluate` por favor cambie la linea en su `lab3.py` de

```
better_evaluate = memoize(basic_evaluate)
```

a

```
better_evaluate = memoize(better_evaluate)
```

El código original estaba puesto para permitir que pueda jugar el juego antes de escribir esta función, pero se vuelve incorrecta e innecesaria una vez que implementó su versión de `better_evaluate`.

TIP: Para ahorrar tiempo, cambie `if True` a `if False` en la linea 308 de tests.py. Esto deshabilita esta prueba que usualmente toma algunos minutos para terminar. Asegúrese de volverlo a poner en True una vez que ha pasado todos los tests.

Listing 4: Código a cambiar.

```
1 # Ponga False en este if para deshabilitar temporalmente esta prueba
2 if True:
3    make_test(type = 'MULTIFUNCTION',
4              getargs = run_test_game_1_getargs,
5              testanswer = run_test_game_1_testanswer,
6              expected_val = "You must win at least 2 more games than you lose↩
                   to pass this test",
7              name = 'run_test_game'
8              )
```

Torneo

Por favor asigne True a “COMPETE” si quiere participar en un torneo. Si hay suficiente interes e IAs capaces entonces se hara una competencia con sus compañeros y tal vez un aún no especificado premio. El primer round elimina las tarea que no pueden ganarle a `basic_player` asi que asigne False a “COMPETE” si no ha logrado pasar esta prueba.

Encuesta

Por favor conteste las preguntas al final de `lab3.py`

- ¿Cuantas horas le tomo hacer esta tarea?
- ¿Cuales son las partes, si es que hay, que encontró interesantes?
- ¿Cuales son las partes, si es que hay, que encotró aburridas o tediosas?

FAQ

La eficiencia de su jugador mejora en un juego que ve el mismo tablero muchas veces en distintas ramas del árbol de búsqueda cuando descomenta la lindea de código que memoiza a su evaluador. Sin embargo, esto no toma en cuenta la identidad del jugador, así que si usted quiere probar su función de evaluación contra los tableros de prueba proveidos entonces debe eliminar la memoización comentando esa linea de código.