Tarea Programada № 2

José Castro, Instituto Tecnológico de Costa Rica 13/02/2019

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) putsa 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 cambiara despues de ser creada. Para hacer una movida en el tablero usted (o mas bien, el código complementario que proveemos para usted) 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 desde 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 esta 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 creará 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 dell jugador que no tiene el turno actualmente.
- `get_cell(row, col)` – Devuelve el número de ID del jugador que tiene una ficha en la posición indicada, o 0 si la posición se encuentra vacia.
- `get_tio_elt_in_column(column)` – Devuelve el número de ID del jugador que tiene la ficha más arriba en la columna. Retorna 0 si la columna se encuentra vacia.
- `get_height_of_column(column)` – Devuelve le número de fila de la primer fila desocupada en la columna. Retorna -1 si la columna esta llena, y 6 si la columna esta vacia. NOTA: éste es el índice de la fila y no el verdadero "altoöde la columna, los índices cuentan desde 0 en la fila más arria hasta 5 en la más baja.
- `do_move(column)` – Devuelve 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(ID)` – Devuelve el largo de la cadena de fichas contigua más larga del jugador especificado. Una cadena se define por las reglas de Connect Four lo que significa que el primer jugador en construir una cadena de 4 fichas gana el juego.
- `chain_cells(ID)` – Devuelve un conjunto de tuplas de Python para cada cadena de fichas de largo 1 o más que están controladas por el jugador actual.
- `is_win()` – Retorna el número del jugador que ha ganado, 0 en caso de que aún no exista un ganador.
- `is_game_over()` – Devuelve true si el juego ha terminado. Utilice `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 objetos tipo 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 un tablero.
- `is_terminal(depth, board)` (`basicplayer.py`) –Retorna True si: o bien se obtiene la profundidad de 0, o bien el juego está 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 más adelante.
- `human_player()` (`connectfour.py`) – Un jugador especial que solicita por medio de consola donde es que se debe ubucar la ficha.
- `count_runs()` (`util.py`) – Este es un decorador de Python que cuentas cuantas veces la función que decora ha sido llamada. Vea la definición del decorador para las instrucciones de cómo 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 el sentido que ocupa 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`. Las funciones de evaluación toman como argumento una instancia de la clase ConnectFourBoard y retornan un indice entero indicando qué tan faborable es el tablero para el jugador de turno.

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

La intención de `better_evaluate` es ir mas allá de funciones de evaluación estática y lograr que su función le gane a la función proveida: `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 cuántas funciones en la clase ConnectFourBourd que le dan información útil sobre las fichas en el tablero; tiene libertad para utilizarlas. Puede también revisar el código fuente de `basic_evaluate(board)` en `basicplayer.py` de la función que está tratando de vencer.

Funciones de búsqueda

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

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

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

Deber permitir que su función acepte optionalmente 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 válida la función `basicplayer.get_all_next_mo`
- `is_terminal_fn` – una función que ado una profundidad y un tablero/estado, retorna True o False. True si el tablero es termina y se debe hacer evaluación estática. En caso de no indicarse textttis_terminal_fn debe tomar el valor de `basicplayer.is_terminal`.

Debe utilizar éstas 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 está 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 código que revisa su búsqueda en árboles de juego estáticos, el tipo que usted puede corroborar 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 a 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, así que puede definir su jugador básico de la siguiente manera:

Listing 1: Ejemplo de un Jugador simple

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

O más succintamente (pero equivalente):

Listing 2: Ejemplo de un Jugador simple en declaración lambda

```
1 mi_jugador = lambda board: minimax(board, depth=3, eval_fn=focused_evaluate,↩
      timeout=5)
```

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 haga profundidad iterativa basado en una función genérica de búsqueda. Usted puede crear un jugador de profundidad iterativa de la siguiente manera.

Listing 3: Ejemplo de un Jugador simple

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

Podría notar que cuando juega contra la computadora, pareciera que hace movidas *estúpidas*. Si usted obtiene la ventaja tal que está 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 está 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 es normal 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 juegas. Ellas buscan ganar de la manera más rápida posible cuando ven que pueden ganar, y tratan de perder lentamente contra un oponente que los tiene acorralados, dándole así más oportunidades a su contrincante de que cometa

un error. Un pequeño cambio en la función de evaluación hará que la computadora juegue 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 más 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 seguro con un valor que es menor o igual a-1000.
- Recuerde que `focus_evaluate` debe ser muy simple, así que no introduzca heurísticas elaboradas aquí. Guarde sus ideas para cuando más tarde implementa `better_evaluate`.

Los jugadores computarizados que ha utilizado hasta ahora son bastante rudimentarios. Evalúan todas las posiciones hasta cierta profundidad, aún cuando se sabe que algunas son inútiles. Puede hacer búsqueda mucho más 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 búqueda.
- `lab3.py` define dos valores `INFINITY` y `NEG_INFINITY` que debe utilizar como los valores de $+\infty$ y $-\infty$ respectivamente.

Este procedimiento se llama por `alpha_beta_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 que maximiza y otro que minimiza, sin embargo es probable que sea más facil escribir el código si cada jugador lo que intenta es *maximizar* el valor desde su punto de vista. Así cada ve que el algoritmo ve hacia adelante a la movida del 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 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, aplha=3 y beta=5

– entonces los valores válidos para el otro jugador serán [-5,-3]. Si utiliza este truco de negación, sólo tendrá que implementar una función en ves de dos (minmax y maxmin).

Un error común es permitir que el juego continúe más allá del final del juego. Asegúrese de utilizar `is_terminal_fn` para determinar bien esto.

Una mejor función de evaluación

Esta parte de la tarea será un poco diferente, no hay una manera única correcta de hacerlo - tomará un poco de creatividad y de pensamiento sobre el funcionamiento del juego.

Su objetivo es escribir un procedimiento de evaluacióno estática 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 las 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 debe interferir de alguna manera con el rendimiento del otro jugador y su código.

Una advertencia adicional con respecto a este tema: hay bastante investigación publicadada 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 incluyeto el comentario apropiado en la parte relevante del código. Sin embargo, no debe utilizar implementaciones completas de los algoritmos de búsqueda elaborados por terceros, y no debe escribir código que acceda a recursos en linea mientras ejecuta. La pena es 0 en la 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 más útil tener tiempo para explorar más a fondo el árbol de búsqueda que expresar de manera perfecta el valor de una posición. Este es el motivo por el cual muchos juegos (no el caso de Connect 4(toma algo de esfuerzo ganarle a la heurśtica *cantidad de movidas disponibles*, no se puede obtener algo más simple y todavía tener información que esútil para ganar el juego.

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

NOTAS IMPORTANTES

Después de implementar `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 una prueba que usualmente toma algunos minutos en terminar. Asegúrese de volverlo a poner en True una vez que haya pasado todos los demás tests.

Listing 4: Código a cambiar.

```
1 if True:
2     make_test(type = 'MULTIFUNCTION',
3         getargs = run_test_game_1_getargs,
4         testanswer = run_test_game_1_testanswer,
5         expected_val = "You must win at least 2 more games than your ↩
              opponent to pass this test"
6         name = 'run_test_game'
7         )
```