Instituto Tecnológico de Costa Rica

Inteligencia Artificial

Tarea 2 - Othello bajo Minimax

Authors:
Carlos Jenkins - 200769999
Pablo Rodriguez - 200767344

26 de octubre de 2012

Índice

Índice de cuadros

Índice de figuras

1. Descripción del problema

Los juegos de dos contrincantes han sido una de las tareas principales en el desarrollo inicial de la IA. En esta tarea se deberá programar un agente para jugar Othello implementando el algoritmo Minimax o alguna de sus variantes.

El Othello (Reversi o Yang) es un juego entre dos personas, que comparten 64 fichas iguales, de caras distintas, que se van colocando por turnos en un tablero dividido en 64 escaques. Las caras de las fichas se distinguen por su color y cada jugador tiene asignado uno de esos colores, ganando quien tenga más fichas sobre el tablero al finalizar la partida.

A continuación se presenta la descripción de la solución (estrategias y variante de Minimax implementada) y el análisis de resultados de la implementación.

2. Descripción de la solución

A continuación se describe la solución implementada para el proyecto.

2.1. Minimax

Según la investigación realizada sobre Minimax y bajo recomendación del profesor se decidió implementar un algoritmo Minimax con poda alfa-beta. Dicho algoritmo tiene las siguiente características:

- Alpha/Beta prunning.
- Profundidad primero.
- Configuración básica de profundidad según tiempo disponible.

Para esto se estudiaron varios pseudocódigos disponibles e información de cómo implementarlo en varios artículos, lo cual llevo a que se diseñara un pseudocódigo propio adaptado al contexto del proyecto. Éste pseudo-código permitió implementar el algoritmo en Erlang.

2.1.1. Pseudocódigo

```
1. Alpha_Beta
    Si profundidad es 0, evalua y retorna el valor del tablero.
    Cualquier otro caso, llama a Escoger_Jugada y le manda como parametro
        las jugadas posibles.

2. Escoger_Jugada (funcion que retorna cual es la mejor jugada)
    Toma una jugada posible y actualiza el tablero con dicha jugada.
    Al tablero se aplica Alpha_Beta, con profundidad -1, lo cual regresa
        el valor de dicha jugada.
    Se llama a la funcion cut_off, que se encarga de eliminar las ramas
        que no son relevantes y de devolver la mejor jugada.

3.Cut_Off
    Se encarga de ver si el valor de la jugada recibida es buena, si lo es
        , la guarda y sino la desecha.
    Vuelve a llamar a Escoger_Jugada, para que este continue con la
        siguiente jugada posible.
```

2.1.2. Funciones principales

Las funciones principales son:

1. `alpha_beta`
2. `evaluate`
3. `choose`
4. `update_board`
5. `cut_off`

A continuación se describe cada una de ellas:

1) `alpha_beta`
Es la función central del código, ya que es la que verifica el valor del tablero en una profundidad de 0, o sino llama a `choose`, para que esta busque entre las jugadas posibles cual es la mejor, según el valor del tablero.

2) `evaluate`
Es la función que se encarga de evaluar cual seria el valor del tablero, para un jugador, según el valor de sus fichas menos el valor de da las fichas del oponente.

3) `choose`
Es la función que se encarga de la recursividad, ya que para cada jugada posible, realiza `alpha_beta` para cada una de estas y así obtiene cual seria el mejor valor para hacer la siguiente jugada.

4) `update_board`
Esta función se encarga tomar una jugada y devolver un tablero con esa jugada implementada.

5) `cut_off`
Esta función es la encargada de seleccionar cual jugada debe ir guardando y cual se elimina durante la implementación del `alpha_beta`.

2.1.3. Funciones auxiliares

- `color`, retorna cual es el valor en el tablero asignado al color.
- `swap`, es la que se encarga de cambiar el valor del jugador: white <-> black o 1 <-> -1.
- `wait_turn`, es la encargada de esperar su turno para realizar una jugada.
- `dead_lock`, es la función que analiza si es conveniente o no hacer una jugada con posibilidades de caer en un dead lock, o sea que ninguno de los dos jugadores pueda jugar. `win_or_lose`, se encarga de ver quien gana en un dead lock.
- `pieces`, se encarga de contar cuantas fichas tiene un jugador de su color.
- `posible_moves`, devuelve una lista con los posible movimientos de un jugador.
- `eval` y las funciones `eval_corner` son las auxiliares de `evaluate` para obtener el valor del tablero en un momento dado.
- `verify_line` y la función `count`, son auxiliares de `eval`, `eval_corner` y `evaluate`.
- `valid`, determina si una posición en el tablero es valida.
- `turn`, la función `view_line` y la función `change_line` son encargadas de actualizar el tablero como auxiliares de `update_board`.
- `score`, es la encargada de devolver el valor que tiene una casilla en el tablero.

2.1.4. Tablero de valores

Se utilizó el siguiente tablero de valores, (Florian Cruz, Carranza Athó):

```
120 -40  20   5   5  20 -40 120
-40 -06  -5  -5  -5  -5 -60 -40
 20  -5  15   3   3  15  -5  20
  5  -5   3   3   3   3  -5   5
  5  -5   3   3   3   3  -5   5
 20  -5  15   3   3  15  -5  20
-40 -06  -5  -5  -5  -5 -60 -40
120 -40  20   5   5  20 -40 120
```

2.2. Cambios en el servidor

Se realizaron los siguientes cambios en el servidor:

1. Se implementó la notificación a los jugadores cuando una proceso decide pasar el turno y no hacer una jugada (por ejemplo, en el caso de un deadlock).

```
123,131d122
<          Turn = NewState#game.current,
<          if
<              Turn == white ->
<                  White = NewState#game.white,
<                  notify(White, {your_turn, NewState});
<              true ->
<                  Black = NewState#game.black,
<                  notify(Black, {your_turn, NewState})
<          end,
```

3. Análisis de resultados

Para le presente proyecto se implementó el algoritmo exitosamente. A continuación se presenta una tabla con los resultados obtenidos de las pruebas.

3.1. Profundidades

Profundidad	Tasa de éxito	Tiempo	Notas
1	0 %	$<$ 1s	Poca visión a futuro
2	10 %	$<$ 1s	Rápido.
3	50 %	$<$ 1s	Profundidad interesante.
4	50 %	˜ 1s	Genera deadlocks. Los deadlocks provocan perder.
5	90 %	˜ 3s	Excelente profundidad pero lenta.
6	80 %	˜ 40s	Mala profundidad, muy lenta y genera deadlocks.
7	100 %	˜ 120s	Excelentes resultados pero excesivamente lenta.

Cuadro 1: Resultado de las pruebas según profundidad.

3.2. Ejecucion del programa

Para ejecutar el programa realice lo siguiente:

```
$ erl
1> c(server.erl).
2> server:start().
3> client:start().
4> client:connect(black).
5> c(player.erl).
6> player:start(white).
```

El programa está diseñado para funcionar en Erlang R15B02 bajo Ubuntu GNU/Linux y Windows 7. Puede también conectar el cliente a las blancas y el jugador a las negras.

3.3. Capturas de pantalla

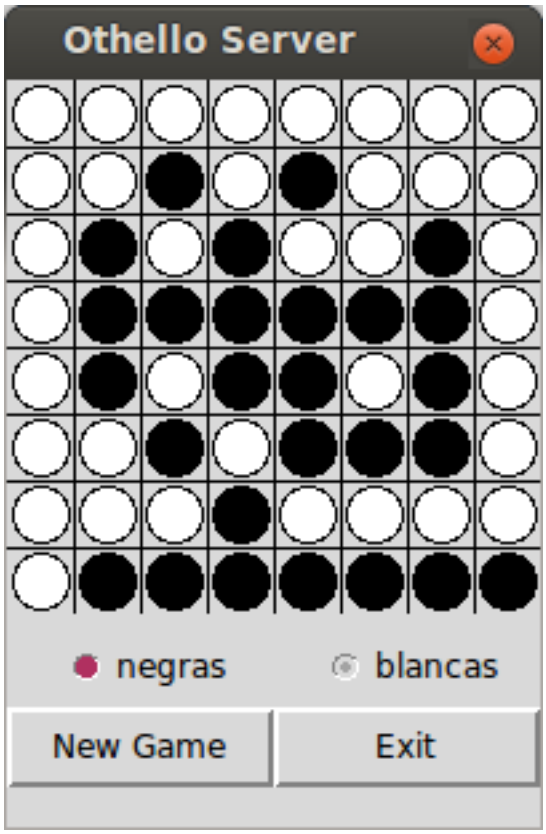

Figura 1: Juego ganado por el algoritmo Minimax programado.

4. Conclusiones

- Cómo se pudo observar en las pruebas realizadas un nivel más de profundidad aumenta considerablemente el tiempo de ejecución del algoritmo, incluso con poda alfa-beta.
- Para saber cuantos niveles hacer se pueden tomar dos enfoques que se mencionan a continuación. Se implementó una versión dummy de la primera, pero existe muchas posibilidades de mejorar el rendimiento del algoritmo considerando éstos enfoques.
  - Predicción de duración: es una función que mapea el tiempo disponible con un número entero con la profundidad a explorar. Dicha función no es nada trivial y supone integrales y derivadas de segundo grado.
  - Profundidad adaptativa: permite explorar el árbol de posiciones de forma que si ha terminado un nivel y no se ha vencido el tiempo continúe con el siguiente nivel.

5. Referencias

- Florian Cruz, L., and Carranza Athó, F. (2008). *Othello*. Escuela Académico Profesional de Informática, Universidad Nacional de Trujillo, Peru.
- Aguilar Fruto, P. (2008). *IA - Algoritmos de Juegos*. Departamento de Lenguajes y Sistemas Informáticos, Universidad Politécnica de Cataluña, España.
- Fröberg, M., and Törnkvist T. (1998). *othello-1.0.tgz*.
  Source code available at http://www1.erlang.org/contrib/othello-1.0.tgz