Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5438 - Inteligencia Artificial I
Trimestre Abril - Julio 2021
Prof. Carlos Infante
Amin Arriaga 16-10072, David Segura 13-11341, Wilfredo Graterol 15-10639

Árboles de juego

July 4, 2021

othello_t

Tomando como base la implementación incompleta de la clase `state_t` del profesor Blai Bonet, la cual renombramos como `othello_t`, realizamos los siguientes cambios que consideramos convenientes[1]:

- Se creó una método `valid_moves` que, dado un color, retorna un vector con las posibles jugadas de dicho color en el estado actual. Si no se consigue ningún movimiento válido, retorna el vector `{36}`, indicando que la única jugada válida es pasar. No se verifica si el estado es terminal pues esa verificación se realiza en los algoritmos antes de llamar a esta función, por lo que, para no realizar cálculos innecesarios, se hace la suposición de que el estado no es terminal.
- Se modificó la función `print` para que imprimiera una representación del tablero mucho más agradable. Sin embargo, debe ejecutarse en una terminal que soporte los caracteres especiales de BASH para darle estilo al output. También se creó una función `print_board` que imprime prácticamente lo mismo, pero enumerando las filas y columnas.
- Se agregaron las verificaciones sobre las diagonales en las funciones `outflank` y `move`, siendo prácticamente iguales a las verificaciones de filas y columnas pero usando los vectores `dia1` y `dia2` en lugar de `rows` y `cols`.

Algoritmos

Las funciones `negamax` (con y sin poda alpha-beta) y `scout` fueron implementados guiándonos por el pseudocódigo visto en clases, mientras que `negascout` fue implementado guiándonos por la siguiente página de Wikipedia, ya que el de las láminas tenía un error que fue corregido luego de finalizar nuestra implementación[2]. La única diferencia con los pesudocódigos y nuestras implementaciones es que en lugar de iterar sobre los sucesores de un estado, se itera sobre sus posibles movimientos, usando la función `valid_moves`, y en cada iteración se calcula el nuevo estado sucesor. Esto es un poco más eficiente en memoria pues no se mantienen almacenados todos los sucesores de un estado en cada llamada recursiva.

Cada algoritmo tiene su análogo `best_move` que, además de retornar el valor de juego del estado pasado como argumento, también retorna el movimiento que permite conseguir dicho valor[3]. La razón de esto es poder verificar que los movimientos obtenidos coinciden con los de la variación principal.

Para contar el numero de estados generados, se aumentaba en uno una variable global `generated` cada vez que se creaba una nueva instancia de `othello_t`. Mientras que para el número de estados expandidos, se aumentaba en uno una variable global `expanded` cada vez que se calculaban los sucesores de un estado.

[1] En el repositorio se encuentra documentado todos los cambios realizados, así como la explicación de la implementación original. En esta sección solo realizaremos un resumen de las modificaciones.
[2] Sin embargo, posteriormente se comprobó que el rendimiento del algoritmo que aparece en Wikipedia y el de las clases son iguales.
[3] Si varios movimientos permiten conseguir la mejor puntuación, se retorna cualquiera de ellos.

Tablas de transposición

Decidimos implementar las tablas de transposición para aumentar la eficiencia de los algoritmos. En particular, creamos dos clases:

- `othello_TT_t`, el cual es básicamente un diccionario que mapea tuplas `(bool, unsigned char, unsigned, unsigned)` a enteros, recordemos que un estado de othello es codificado usando 3 variables `unsigned char t_` para las 4 casillas centrales, `unsigned free_` para determinar las casillas libres y `unsigned pos_` para determinar el color de las casillas no libres; mientras que el booleano indica si el valor que se va a almacenar es para el negro (`true`) o el blanco (`false`). Ademas, cada tabla tiene un atributo `type_` que indica la frecuencia con la que se almacenarán los estados del juego, y puede tener alguno de los siguientes valores:
  - `TOTAL`. Todos los estados son almacenados.
  - `DEPTH`. Todos los estados hasta cierta profundidad son almacenados.
  - `RANDOM`. Los estados son almacenados con una determinada probabilidad.

  Esta tabla se usa en los algoritmos que no tienen poda alpha-beta.
- `othello_TTab_t` funciona prácticamente igual al anterior, pero en lugar de mapear la tupla a enteros, lo hace a pares `(int, tt_flag_t)`, donde `tt_flag_t` puede tomar alguno de los siguientes valores:
  - `EXACT`. Esto significa que el valor almacenado representa el valor exacto del estado, es decir, fue calculado mientras se encontraba dentro del rango (alpha, beta).
  - `UPPERBOUND`. Esto significa que cuando se almacenó el par, el score calculado era menor que alpha, por lo que no representa el valor de juego exacto del estado, sino una sobreestimación de este. Por esta razón se le denomina `upperbound`, pues al consultarse, este valor almcaneado representará una cota superior del valor real del estado. Debido a que beta también representa una cota superior, entonces se toma el mínimo entre ambas.
  - `LOWERBOUND`. Esto significa que cuando se almacenó el par, el score calculado era mayor que beta, por lo que no representa el valor de juego exacto del estado, sino una subestimación de este. Por esta razón se le denomina `lowerbound`, pues al consultarse, este valor almcaneado representará una cota inferior del valor real del estado. Debido a que alpha también representa una cota inferior, entonces se toma el máximo entre ambas.

Luego, cada algoritmo para la resolución del juego tomaba como argumento una variable booleana `use_tt` que indicaba si se usaba una tabla de transposición.

main.cpp

Para compilar el main simplemente se debe ejecutar `make` en la raiz del repositorio del proyecto. Una vez compilado, se creará un ejecutable `othello.out` dentro del directorio `bin/` cuya sintaxis es

```
othello.out [ VERBOSE ] ALGORITHM [ SECONDS [ TT ] ]
```

donde

- `VERBOSE` puede ser `-v`, indicando que solo se imprimirá el resultado global de todos los estados de la variación principal; `-vv`, para imprimir también los resultados de cada estado (este es el valor por defecto); y `-vvv` para imprimir la representación del tablero de cada estado.
- `ALGORITHM` puede toma valores del 0 al 3, indicando que el algoritmo a usar es *negamax*, *negamax con poda alpha-beta*, *scout* o *negascout* respectivamente.
- `SECONDS` es el número máximo de segundos que se esperará por cada estado de la variación principal. Su valor por defecto es 60.
- `TT` indica el tipo de tabla de transposición, pudiendo tener los siguientes valores:

- `0`, no se usará la tabla.
- `1`, se usará una tabla de tipo `TOTAL`.
- `2 [ DEPTH ]`, se usará una tabla por profundidad, siendo `DEPTH` la profundidad máxima (valor por defecto: 0).
- `3 [ PROB ]`, se usará una tabla probabilística, siendo `PROB` la probabilidad de almacenar un estado (valor por defecto: 0).

Al ejecutarse el script, se recorren los estados de la variación principal comenzando por el estado final. Antes de calcular el mejor movimiento del estado de una iteración, se libera la memoria de las tablas de transposición, se inicializan en 0 las variables globales de los estados generados y expandidos, y se inicia un cronómetro, el cual se ejecuta en un proceso hijo, recibe como argumento el tiempo a esperar y el pid del proceso padre, y al pasar ese tiempo, le envia una señal `SIGINT` para interrumpirlo. En caso de que el algoritmo calcule el mejor movimiento antes que el cronómetro, entonces el proceso padre interrumpe y finaliza a ese proceso hijo. Al final de la ejecución se imprimen tres listas: Nodos generados, nodos expandidos y tiempo de ejecución por cada estado de la variación principal que se logró calcular.

Resultados

Para estudiar los distintos algoritmos colocamos un tiempo máximo de 20 minutos por estado para cada algoritmo. Como vemos en las *Figuras 1 y 2*, negamax fue el que tuvo el peor rendimiento, llegando sólo hasta el estado 14, en el cual genera poco más de 10^9 nodos, y tardando alrededor de 10^3 segundos. Negamax con poda alpha-beta no se le aleja demasiado, pues llegó hasta el estado 15 generando poco menos de 10^9 nodos en 300 segundos aproximadamente. Mientras que scout y negascout tuvieron un rendimiento mucho mejor que los anteriores, llegando hasta el estado 22 expandiendo alrededor de 10^8 nodos en 300 segundos aproximadamente. Sin embargo, entre ellos 2 la diferencia es prácticamente despreciable (aunque negascout es ligeramente superior), tanto en nodos generados como en tiempo de ejecución, dando a entender que la poda alpha-beta no es muy efectiva en el algoritmo scout para este juego. Luego, en todos los algoritmos, el número de nodos expandidos es prácticamente igual al número de nodos generados, aunque obviamente siempre es menor. Esto ocurre ya que todo estado generado es expandido a menos que sea un estado terminal.

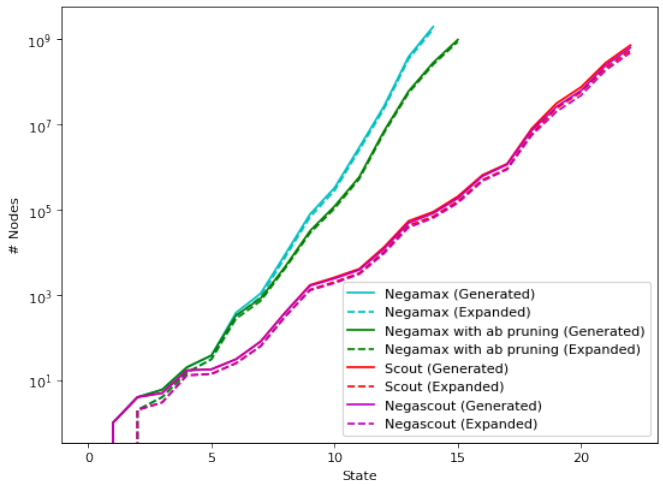
Al usar las tablas de transposición, negamax obtuvo una gran mejoría. Aunque negamax sin poda alpha-beta seguía siendo la más ineficiente, ahora alcanzaba al estado 17 expandiendo menos de 10^8 nodos en alrededor de 10^3 segundos. Mientras que negamax con poda alpha-beta fue el que obtuvo los mejores beneficios, al alcanzar el estado 21 en alrededor de 10^3 segundos, e incluso expandiendo casi tantos nodos como scout. Sin embargo, los algoritmos scout y negascout no tuvieron una mejora significativa. Ambos siguen llegando hasta el estado 22, e incluso negascout tardó más. Y considerando la cantidad de memoria RAM necesaria para usar las tablas de transposición, entonces para estos dos algoritmos no vale nada la pena usarlas. La diferencia entre nodos generados y expandidos en todos los algoritmos es mayor que cuando no se usaban las tablas de transposición, esto es debido a que precisamente dichas tablas se usan para no expandir estados cuyo valor de juego fue previamente calculado en otra iteración.

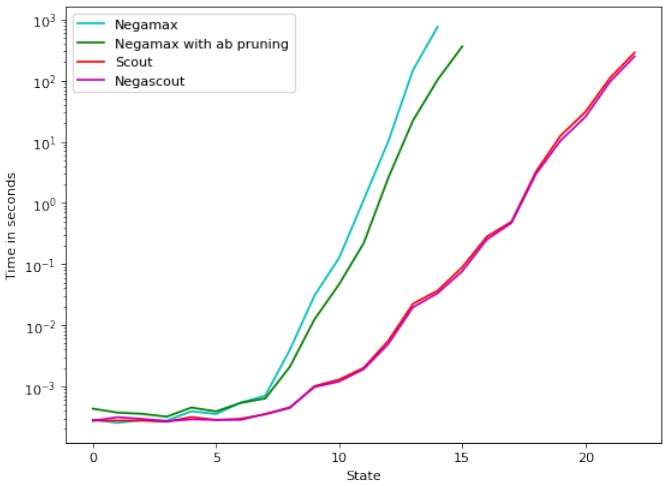
Extra: game.cpp

Adicionalmente, creó este archivo para poder jugar othello 6x6 en contra de la IA. Para compilarlo se debe ejecutar `make game` en la raiz del repositorio del proyecto. Una vez compilado, se creará un ejecutable `game.out` dentro del directorio `bin/` cuya sintaxis es

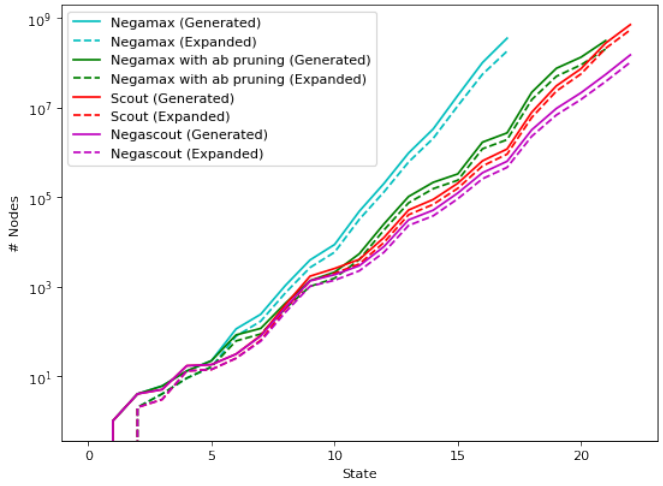
```
game.out DEPTH BLACKS
```

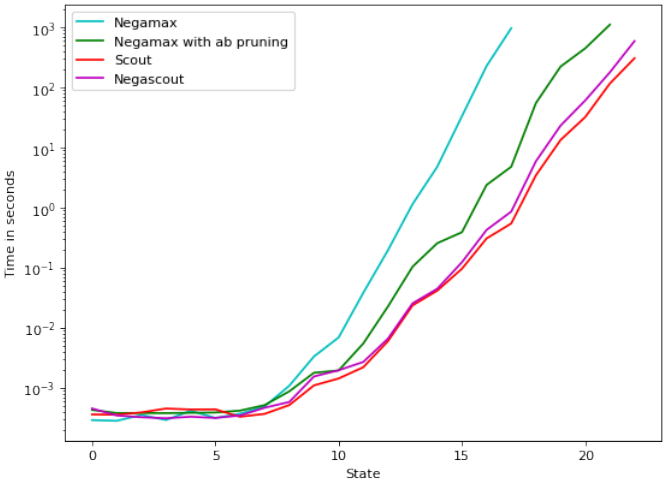
donde `DEPTH` es la profundidad máxima para la búsqueda en el árbol de juego, y `BLACKS` indica si el jugador será las negras (`1`) o las blancas (`0`). Se usa siempre `scout` sin tabla de transposición. Cada jugada debe indicarse según la fila (enumeradas del 1 al 6) y la columnas (identificadas de la a a la f), por ejemplo, `3c`. En caso de que se deba pasar, se escribe `P`.





Figuras 1 y 2. A la izquierda, nodos generados y expandidos por cada algoritmo en cada estado de la PV sin usar TT. A a la derecha, tiempo de ejecución por cada algoritmo en cada estado de la PV sin usar TT.





Figuras 3 y 4. A la izquierda, nodos generados y expandidos por cada algoritmo en cada estado de la PV usando TT. A la derecha, tiempo de ejecución por cada algoritmo en cada estado de la PV usando TT

Conclusiones

- Negamax es el más ineficiente de los algoritmos probados.
- La poda alpha-beta ayuda bastante al algoritmo negamax.
- Los algoritmos scout y negascout tienen un rendimiento muy similar, por lo que la poda alpha-beta no es muy efectiva en scout.
- El número de nodos expandidos no se aleja mucho del número de nodos generados.
- Las tablas de transposición mejoran bastante el rendimiento de negamax, más aun si utiliza poda alpha-beta.
- Las tablas de transposición no mejoran a los algoritmos scout y negascout, incluso puede hacerlos más ineficientes en tiempo, además de que obviamente es mucho más ineficiente en memoria debido a la gran cantidad de estados que debe almacenar.
- Para implementar un juego con IA, las mejores opciones son scout y negascout sin tablas de transposición.