Informe
Trabajo Práctico Especial

Fuga del Rey: Aplicación de algoritmo Minimax

MATERIA:

- ✓ Estructura de datos y algoritmos

INTEGRANTES:

- ✓ Arlanti, María de los Ángeles (53373)
- ✓ Castaño Gómez, Nicolás (53384)
- ✓ Kulesz, Sebastián (54045)

FECHA DE ENTREGA:

- ✓ 4/11/2014

Índice

1. MODELO DE DATOS

En un principio implementamos el tablero usando una matriz de la clase Ficha, donde esta representaba una pieza genérica del tablero. Cada una de las que estan representadas en el juego heredaban de ella. Nos encontramos con un problema a la hora de comparar entre piezas del mismo bando, para saber cuales eran aliadas entre sí y cuales enemigas. Otro problema que iba a ser aparente era el uso de memoria. Al tener una instancia de cada posicion del tablero para cada estado actual posible el uso del heap al momento de calcular la mejor jugada iba a ser demasiado alto, y en algunos casos suficiente para crashear la máquina virtual de java.

Luego de discutirlo decidimos utilizar un Enum para representar las piezas, y una matriz de dicha clase para representar el tablero. Este cambio nos permitió comparar piezas por referencia, y mantener una única instancia en memoria.

Este cambio nos acerca más a un proyecto del tipo imperativo, pero lo consideramos indispensable para mantener una buena calidad en el código, y asegurar el buen funcionamiento del juego.

Para escribir el archivo .dot que describe el árbol de minimax calculado utilizamos un buffer de escritura para no saturar al sistema y que la ejecución del algoritmo, y el tiempo que le lleve calcular la movida no se vean limitados por la velocidad de escritura.

2. IMPLEMENTACIÓN DE MINIMAX

Para implementar el algoritmo Minimax, abstraemos el estado del juego, junto con la movida que nos llevó a ese estado en un nodo. Luego, el estado actual es el nodo raiz del arbol minimax, el cual siempre es guardado. Para obtener la mejor movida a partir de un estado, analizamos sobre todas las fichas del jugador, todas las movidas posibles que puede realizar, y ponderamos el estado del juego al cual nos lleva.

Es posible ejecutar el juego por limite de tiempo, o limitando la profundidad del análisis. Al limitarlo por tiempo, la única opción para poder garantizar una respuesta es calcular el arbol de nivel 1. Si queda tiempo calcular un nivel más de profundidad. Si en este cálculo nos quedamos sin tiempo, nos vemos obligados a retornar el árbol calculado en el paso anterior. Si aún nos queda tiempo, seguimos con el cálculo siempre aumentando en 1 la profundidad y guardando la respuesta del nivel anterior.

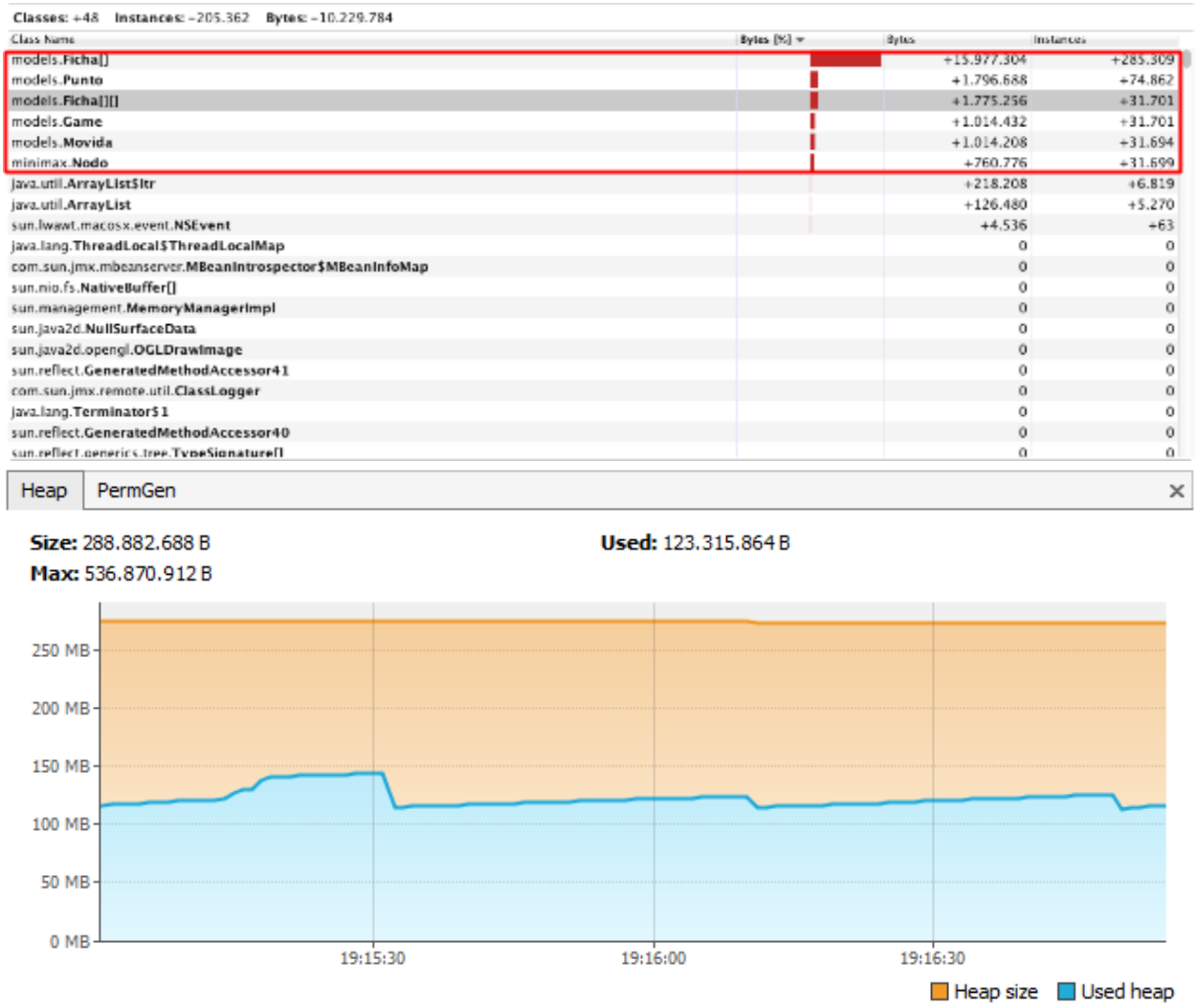
Al no estar especificado, nuestra implementación de minimax siempre jugará a nombre de los enemigos, mientras que la persona juega a nombre de los guardias.
Es posible revertir esta situación muy fácilmente, pero la heurística que usamos no es compatible con esta situación. Cambiandola, es posible hacer que la computadora juegue por ambos bandos.

3. CAMBIOS EN LA IMPLEMENTACIÓN DEL ALGORÍTMO MINIMAX

Versión 1:

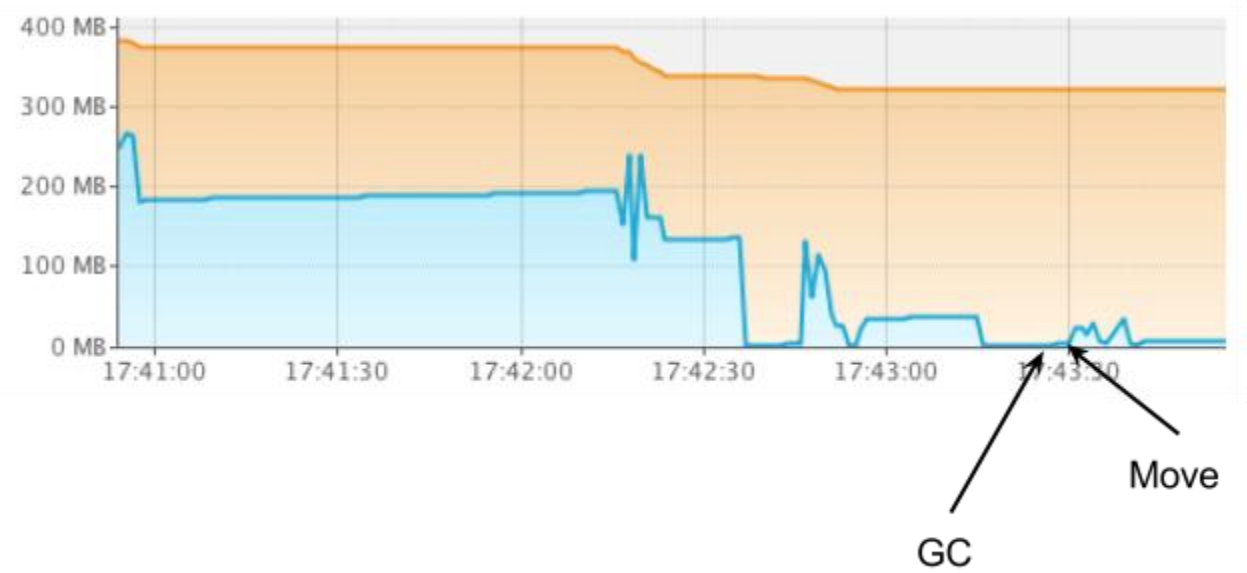
Primero se calculan todas las posibles jugadas para todos los estados posibles, dada una profundidad o tiempo. Con ellas, se arma el árbol de jugadas y recién después se busca la mejor jugada por profundidad.
Desventaja: La complejidad del algoritmo es muy alta, hay demasiadas instancias de la clase Punto (Posición del tablero) lo que genera que la memoria se sature para profundidad mayor que dos. Hasta este momento no habíamos implementado la poda, y nos dimos cuenta que de esta forma iba a ser imposible implementarla.

A continuación se puede observar el estado del Heap y la cantidad de instancias de objetos durante la ejecución del juego.

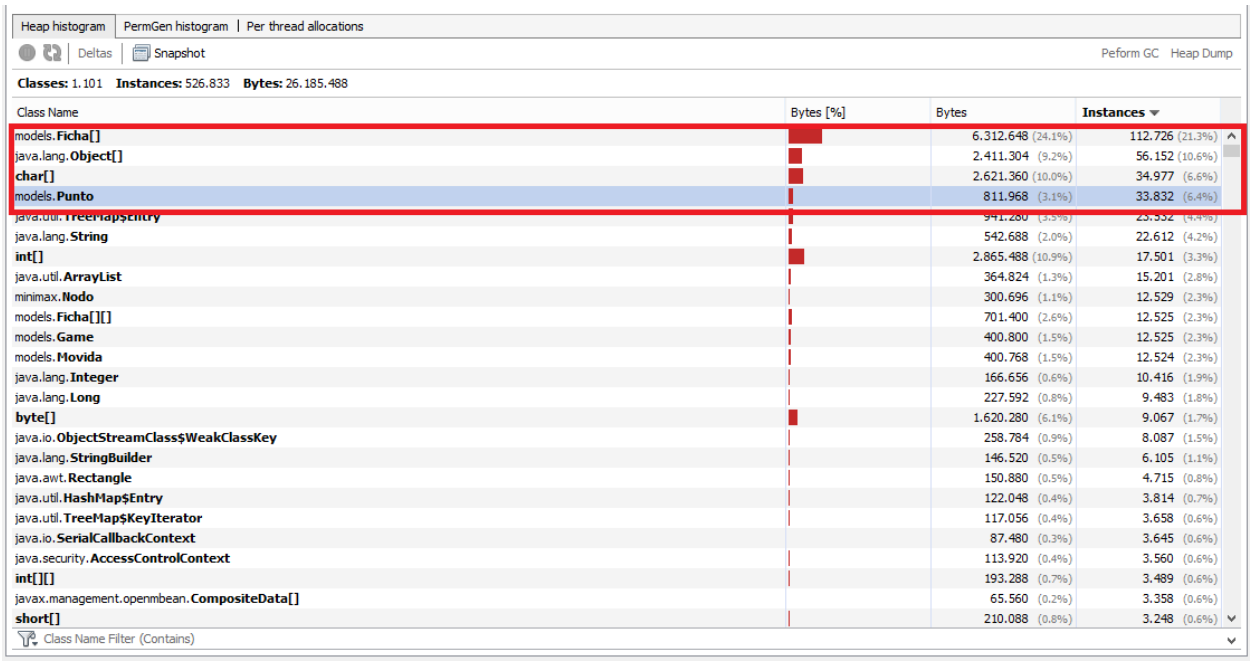


Versión 2:

Cambiamos el algoritmo para no tener todo el árbol en memoria cargado. Se calculan todas las posibles movidas para un nodo y se itera sobre ellas. Se crea un nodo para la primera movida y se llama de manera recursiva a la misma función hasta que devuelve el mejorHijo. Ese nodo se guarda en el nodo padre que generó la recursión y se sigue iterando sobre las movidas comparando contra este mejorHijo. A lo último se devuelve el mejorHijo con la mejor jugada.
De esta manera evitamos mantener todo el árbol de jugadas en memoria. Se puede observar en las capturas como el uso de la memoria bajó considerablemente, de la misma forma que lo hicieron las instancias de la clase Punto.
Ventajas: Ahora se puede llegar hasta profundidad 4.



En la imagen se observa un llano ya que el Garbage Collector estaba actuando sobre las instancias. Entre los dos llanos se observan tres picos que están mostrando el desarrollo de toda una jugada (Move): ejecuta movida del jugador y luego la movida de la máquina.



Versión 3:

Se mantienen casi todos los cambios que se realizaron para la versión 2, excepto en el método gerMejorEnProfundidad (método que elige la mejor movida por profundidad). El cambio consiste en la manera de recorrer el tablero, ya que antes se lo recorría mediante dos ciclos “for” anidados(filas y columnas del tablero), mientras que ahora se lo recorre utilizando un iterador. Este iterador que nos permite recorrer el tablero de fichas, se obtiene de los métodos, fichas() o fichasDelBandoActual(), ambos ubicados en la clase Game:

fichas():

Método hasNext(): retorna true mientras la columna en la que me encuentre sea menor a el ancho del tablero. En caso de que la columna haya llegado al final, hacemos que vuelva a la columna inicial mediante la operación columna %size , incrementamos la fila y volvemos a llamar a hasNext(). En caso de que se hayan recorrido todas las filas, devuelve false.

Método next(): devuelve una instancia de Punto dependiendo en la posición del tablero en la que se esté ubicado.

FichasDelBandoActual():

Método hasNext(): retorna true si la posición actual en la que me ubico es una ficha que corresponde al jugador al que le corresponde el turno, caso contrario incrementa la columna para poder seguir moviéndose en el tablero. Una vez que recorridas todas las columnas de la fila actual, setea la columna a la columna inicial,incrementamos la fila y volvemos a llamar al método.

Método next(): devuelve una instancia de Punto dependiendo en la posición del tablero en la que se esté ubicado.

Ventajas: es más claro el código a la hora de recorrer el tablero, ya que recorremos objetos de tipo Punto, que serían las coordenadas de una ficha en el tablero.

Desventajas: no es tan eficiente, ya que el método hasNext() se llama recursivamente.

4. CAMBIOS EN LA HEURÍSTICA

Prueba 1 (Versión 1 de Minimax):

El valor heurístico es calculado restándole la cantidad de guardias a la cantidad de enemigos.

Ventajas: De ésta forma se priorizan los estados de tablero en los que se contribuye a la eliminación de guardias, dado que la resta entre enemigos y guardias da mayor.
Desventajas: El problema es que ante la posibilidad de eliminar al rey, la computadora no toma ninguna decisión al respecto.

Prueba 2 (Versión 1 de Minimax):

Se intenta mejorar el valor calculado en la prueba anterior. Se cuenta la cantidad de fichas enemigas que hay en un radio de dos casilleros alrededor del rey, llamamos a esta cantidad "bloqueos". Se le suma al anterior valor heurístico los bloqueos multiplicados por dos.

Ventajas: el valor heurístico es una combinación lineal que da más peso a los tableros que posee mayor cantidad de fichas enemigas cerca del rey. Es decir que se priorizan los tableros que "persiguen" al rey.
Desventajas: Ante la posibilidad de matar al rey cuando queda una posición libre en un radio de un casillero alrededor del rey, la computadora no toma ninguna decisión.

Prueba 3 (Versión 1 de Minimax):

Se mejora la heurística anterior contando la cantidad de enemigos posicionados en un radio de 1 casillero alrededor del rey. Llamamos a esta cantidad "cuadranteMenor".

Ventajas: Ayuda a encerrar al rey.
Desventajas: Ante la posibilidad de matar al rey, en algunos casos elige estados de tablero que acercan al primer cuadrante, dado que los dos estados tienen igual valor heurístico. El problema es que no le se está dando mayor peso a las posiciones que permiten matar al rey.

Prueba 4 (Versión 1 de Minimax):

Se mejora la heurística anterior contando la cantidad de enemigos posicionados en los casilleros adyacentes horizontal y verticalmente al rey. Llamamos a esta cantidad "matarAlRey". Sumamos al valor heurístico que multiplica por 6 la cantidad de matarAlRey.

Ventajas: Las fichas enemigas persiguen al rey y lo matan de presentarse la oportunidad.

Desventajas: Hay situaciones en las que queda un casillero habilitado para que una filla se pueda acercar al rey (por mas que no lo pueda comer), pero no lo hace.

Prueba 5 (A partir de la versión 2 de Minimax):

Al cambiar el algoritmo minimax, comienza a priorizar matar guardias, que perseguir al rey. No podemos encontrar explicación a este comportamiento. Es confuso ya que el valor heurístico es una combinación lineal.

5. ANÁLISIS DE RENDIMIENTO

La poda del árbol de jugadas nos permitió explorar un nivel mas en un tiempo aceptable para un jugador humano