Práctica 6: Hilos y concurrencia

Fecha de entrega: viernes 2 de junio de 2017, a las 23:00

Objetivo: Hilos y concurrencia

Descripción de la Práctica

En esta práctica usarás hilos y concurrencia para mejorar el funcionamiento del jugador inteligente. Para ello, deberás completar las siguientes tareas:

1. Añade componentes a tu interfaz para que se pueda utilizar un nuevo jugador inteligentes automático, con y sin paralelismo. Cada vez que este jugador calcule un nuevo movimiento, muestra estadísticas detallando cuántas jugadas han conseguido evaluar en el tiempo del que disponían.

2. Modifica tu interfaz de jugador para que, mientras la IA piensa jugadas, no se bloquee el funcionamiento de la interfaz. Para ello, deberás usar un hilo separado, teniendo cuidado de nunca llamar a componentes gráficos desde fuera del hilo de eventos.

3. (Opcional) implementa una interfaz gráfica para poder jugar al ajedrez, y añade la posibilidad de elegirlo a tu Main. Las clases de modelo de este juego, y los recursos gráficos asociados, forman parte de los ficheros incluidos en esta entrega.

En las próximas secciones se proporcionan más detalles sobre cada una de las anteriores tareas.

Envío de la práctica Debes entregar un único zip (y no .rar, ni .7z, ni .tgz: un **.zip**) por grupo vía campus virtual, conteniendo exclusivamente los siguientes archivos y directorios: pom.xml, src/ (y todo lo que hay debajo), y un fichero llamado alumnos.txt. En alumnos.txt debes incluir el nombre de integrantes del grupo, así como cualquier comentario que quieras que tu profesor tenga en cuenta durante la corrección de la práctica.

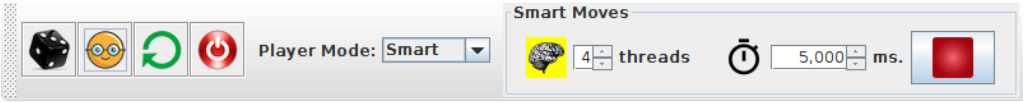

Figura 1: Nuevo subpanel con el jugador *smart* pensando.

Nuevos componentes

Añade un subpanel a la parte superior de tu interfaz, intentando imitar la de la figura 1. Sus componentes son los siguientes:

- Un icono para ilustrar que está pensando el jugador automático (*smart*). El fondo debe aparecer en amarillo cuando está pensando. Todos los iconos nuevos, incluyendo éste, se adjuntan en el zip que acompaña a este enunciado.
- Selector del número de hilos que puede utilizar el jugador automático. Ahora este jugador puede utilizar múltiples hilos en paralelo para buscar jugadas mucho más rápido que la versión no-concurrente. Usa un `JSpinner` para restringir valores al rango $[1, N]$ donde N es el número de procesadores disponibles.
- Icono para ilustrar el selector de tiempo máximo de búsqueda de jugada.
- Selector del tiempo máximo de búsqueda de jugada, en milisegundos. La búsqueda de jugada del jugador automático nunca deberá exceder de este valor. Usa un `JSpinner` para restringir valores al rango $[500, 5000]$, y permite modificaciones con las flechas en saltos de 500 ms.
- Botón para cancelar el proceso de búsqueda de jugada. Este botón debe estar activo (*enabled*) solo cuando el jugador automático está buscando una jugada. Cuando el usuario pulse este botón se debe parar la ejecución del hilo de ejecución de la búsqueda.

Nuevo jugador inteligente

En esta práctica, cada vez que el usuario pulse el botón de movimiento inteligente o tenga seleccionada alguna opción de jugador inteligente en el desplegable de movimientos automáticos, se deberá mostrar las estadísticas solicitadas a través del panel de mensajes:

```
322763 nodes in 5006 ms (64 n/ms) value = -0.0034
```

El zip que acompaña al enunciado incluye varias clases para utilizar jugadores inteligentes concurrentes. Por un lado, el zip incluye una versión actualizada de `MinMax` y una nueva clase `ConcurrentDeepeningMinMax` (CDMM) que permite lanzarlo de forma concurrente. En su constructor se pueden especificar cuántos hilos concurrentes lanzará como máximo. Ten en cuenta que, una vez que cada núcleo del procesador está ejecutando un hilo, ya no será posible mejorar el rendimiento intentando lanzar más hilos. Puedes saber (desde Java) cuántos hilos puedes usar como máximo llamando a `Runtime.getRuntime().availableProcessors()`.

Dado un CDMM, el método `chooseNode()` devuelve el nodo con la mejor acción encontrada y el valor que el algoritmo asocia a esta jugada para el jugador actual (positivo es bueno, negativo es malo). Una vez has llamado a este método, el método `getEvaluationCount()` devuelve el número de tableros que se han evaluado en total.

El zip también incluye una clase `ConcurrentAIPlayer` que implementa el interfaz `GamePlayer`. Esta clase utiliza internamente un CDMM y proporciona los siguientes métodos:

- `setMaxThreads(int nThreads)` Permite especificar el máximo número de hilos que el jugador concurrente va a utilizar al calcular una acción.
- `setTimeOut(int millis)` Permite especificar el tiempo máximo de ejecución del jugador automático concurrente.
- `requestAction(S state)` Calcula la mejor acción disponible para el jugador actual desde el estado `S`, usando de manera concurrente el número de hilos y el tiempo máximo especificados con los métodos anteriores.
- `getEvaluationCount()` Devuelve el número total de nodos evaluados en la última llamada a `requestAction`.
- `getValue()` Devuelve el valor asociado a la acción devuelta por la última llamada a `requestAction`.

Deberás reemplazar tu `SmartPlayer` por un `ConcurrentAIPlayer` que calculará la mejor acción disponible y proporcionará la información que necesitas para mostrar las estadísticas que se solicitan en este enunciado.

Cambios a la interfaz de jugador

La ventana del jugador debe hacer uso de los nuevos componentes a la hora de realizar jugadas inteligentes (las aleatorias y manuales no se ven afectadas), y deberá evitar cualquier congelamiento de la interfaz mientras se planifican jugadas. Tanto las jugadas utilizando el botón **Smart** como las jugadas realizadas en modo **Smart** deben calcular el siguiente movimiento utilizando tantos hilos como indique el `JSpinner` correspondiente y durante el tiempo indicado.

Para evitar congelar la interfaz, asegúrate de lanzar las llamadas a `ConcurrentAIPlayer.requestAction()` **desde un hilo distinto al de la interfaz gráfica** (el hilo de reparto de eventos, conocido por la abreviatura EDT). Ten en cuenta que desde el hilo que ejecuta `ConcurrentAIPlayer.requestAction()` **no debes acceder a los componentes de la interfaz gráfica**. En su lugar usa `SwingUtilities.invokeLater()` para cualquier acceso; por ejemplo, para mostrar estadísticas en el panel de mensajes:

```
// Usando una clase anónima
SwingUtilities.invokeLater( new Runnable() {
  public void run() {
    // cambios en la interfaz ; este código se ejecutará en el hilo EDT
    panelMensajes.append(interesantesEstadisticas);
  }
});
```

Si lo prefieres, también puedes utilizar una función anónima:

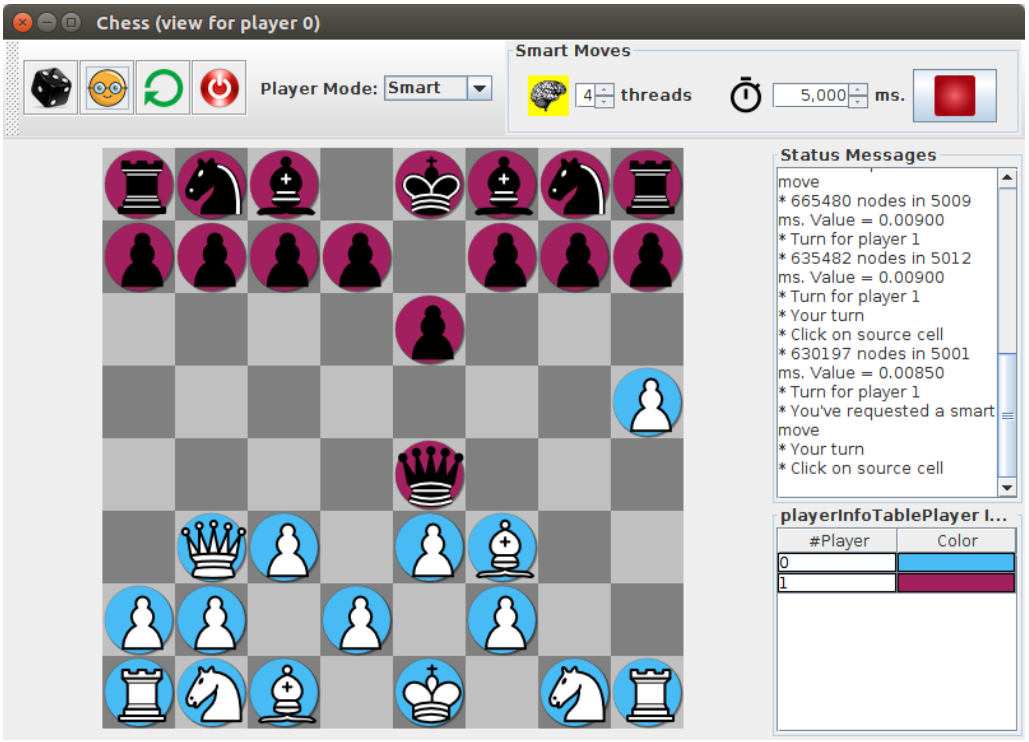

Figura 2: Tablero de ajedrez gráfico.

```
// Usando una función anónima
SwingUtilities.invokeLater(() -> {
    // cambios en la interfaz ; este código se ejecutará en el hilo EDT
    panelMensajes.append(interesantesEstadisticas);
});
```

En la práctica anterior toda la interfaz se ejecutaba desde el EDT. Para evitar complicaciones, es recomendable, tras calcular la mejor jugada, hacer las llamadas a los métodos de vuestro `GameTable` también desde el EDT, es decir, desde dentro de un `SwingUtilities.invokeLater()`.

El botón de realización de jugada inteligente también deberá llamar al nuevo mecanismo de búsqueda de jugadas inteligentes, usando la versión concurrente.

(Opcional) Ajedrez gráfico

Ninguno de los juegos de tu P5 son particularmente complejos. Si quieres un juego que realmente necesite potencia de cálculo, el ajedrez es una buena opción; y además se trata de un juego con una larga tradición informática.

Las clases de modelo del juego se entregan en el .zip que acompaña al enunciado, en el nuevo paquete `chess`. El directorio `src/main/resources/chess` contiene los iconos necesarios para mostrar las piezas.

Desde el punto de vista de implementación, lo más complicado de este apartado será modificar tu componente de tablero actual para que pueda, además de "colores", mostrar imágenes encima de los mismos (ver figura 2).

La generación de jugadas es muy similar a la de las ovejas del Wolf and Sheep (la mayoría sólo requieren de un origen y un destino). Eso sí, algunas jugadas (ver el **`enum`** `ChessAction.Special`) son especiales, y no bastará con que coincidan las coordenadas. Por ejemplo, un peón que llega a la última fila se puede convertir en reina, caballo, alfil o torre. Puedes elegir cualquier método para generar este tipo de jugadas (podrías mostrar un diálogo para permitir elegir), pero recomendamos escoger la primera jugada válida cuyos orígenes y destinos coincidan. De esta forma coronarás siempre como reina - pero a cambio el código quedará más sencillo.