Proyecto del curso de Programación Funcional 2020

El proyecto de curso de Programación Funcional en 2020 será implementar en Haskell una versión del juego Tak. Se trata de un juego de tablero de estrategía abstracta para dos jugadores, creado por James Ernest y Patrick Rothfuss, y publicado por Cheapass Games en 2016.

Figure 1: Tak banner

Tak

El juego se juega colocando y moviendo piezas sobre un tablero con forma de damero. El tablero se puede jugar sobre tableros de diferentes dimensiones. Para este proyecto se usarán los tableros de 3x3 (con 10 piezas) y 4x4 (con 15 piezas). Las piezas son piedras planas de color blanco o negro.

Figure 2: Partida de Tak

Los jugadores toman turnos o colocando piezas o moviendo las piezas que ya están colocadas en el tablero. Las piezas deben siempre colocarse en una casilla vacía. Las piedras planas que pueden ser colocadas acostadas o paradas (llamadas *muros*). Las piezas se mueven a una casilla adyacente en horizontal o vertical, pero no diagonal. Las piedras planas se pueden mover a una casilla ocupada, formando una pila, siempre que la casilla no tenga en el tope de su pila un muro. Las pilas de piezas pueden moverse por el jugador dueño de la pieza en el tope.

El juego puede terminar de dos maneras:

- Cuando un jugador logra hacer una cadena de piezas planas entre dos bordes opuestos del tablero gana la partida.
- Cuando un jugador se queda sin piezas, o cuando el tablero se queda sin casillas vacías, la partida termina. El jugador con más piezas planas en los topes de las pilas de cada casilla gana la partida. Si ambos jugadores tienen la misma cantidad, la partida termina en empate.

Planteo

Primero se debe definir una estructura de datos para almacenar la información del estado del juego en cualquier momento de la partida. Dicha estructura de datos se implementará como un tipo data de Haskell llamado `TakGame`. También se debe definir un tipo data de Haskell llamado `TakAction` para representar los posibles movimientos de los jugadores. Los jugadores se definen de la siguiente forma:

Figure 3: Partida de Tak

```
data TakPlayer = WhitePlayer | BlackPlayer deriving (Eq, Show, Enum)
```

Las funciones a implementar son las siguientes:

- `beginning3x3 :: TakGame`: El estado inicial del juego Tak con un tablero de 3x3, con el tablero vacío.
- `beginning4x4 :: TakGame`: El estado inicial del juego Tak con un tablero de 4x4, con el tablero vacío.
- `activePlayer :: TakGame -> TakPlayer`: Esta función determina a cuál jugador le toca mover, dado un estado de juego.
- `actions :: TakGame -> [(TakPlayer, [TakAction])]`: La lista debe incluir una y solo una tupla para cada jugador. Si el jugador está activo, la lista asociada debe incluir todos sus posibles movimientos para el estado de juego dado. Sino la lista debe estar vacía.
- `next :: TakGame -> (TakPlayer, TakAction) -> TakGame`: Esta función aplica una acción sobre un estado de juego dado, y retorna el estado resultante. Se debe levantar un error si el jugador dado no es el jugador activo, si el juego está terminado, o si la acción no es realizable.
- `result :: TakGame -> [(TakPlayer, Int)]`: Si el juego está terminado retorna el resultado de juego para cada jugador. Este valor es 1 si el jugador ganó, -1 si perdió y 0 si se empató. Si el juego no está terminado, se debe retornar una lista vacía.
- `score :: TakGame -> [(TakPlayer, Int)]`: Retorna el puntaje para todos los jugadores en el estado de juego dado. Esto es independiente de si el juego está terminado o no.
- `showGame :: TakGame -> String`: Convierte el estado de juego a un texto que puede ser impreso en la consola para mostrar el tablero y demás información de la partida.
- `showAction :: TakAction -> String`: Convierte una acción a un texto que puede ser impreso en la consola para mostrarla.
- `readAction :: String -> TakAction`: Obtiene una acción a partir de un texto que puede haber sido introducido por el usuario en la consola.

La cátedra entregará código de referencia para facilitar las pruebas del código solicitado. Éste contiene: definiciones de algunos tipos, esqueletos de funciones a implementar, y una función `main` para poder probar la implementación.

El trabajo debe realizarse en equipo. Se entregará vía Webasignatura hasta el 10 de julio a las 19:00 horas. Inmediatamente luego de la entrega se tomará una defensa a todos los miembros de cada equipo.

Extras

Las siguientes tareas extra pueden ser realizadas por los equipos para agregar puntos a su calificación, por encima de los 100 puntos que vale el proyecto sin éstos.

- *Tak 5x5 y 6x6 (15 puntos).* Una tarea extra consiste en agregar los tableros de dimensiones 5x5 (con 21 piezas) y 6x6 (con 30 piezas) (funciones `beginning5x5` y `beginning6x6`). Para estos tableros se agrega

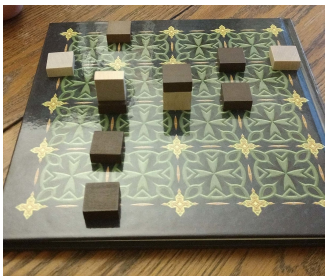

Figure 4: Partida de Tak

una pieza llamada *la piedra angular* o *capstone*. La *capstone* puede moverse encima de un muro, acostando la pieza del muro como consecuencia, aunque en este caso la *capstone* debe moverse sola.

- *Jugador MCTS (10 puntos).* La plantilla de código entregada contiene una mínima implementación de juego. Se define un tipo para agentes jugadores, y se implementan dos: el *jugador aleatorio* (que elige sus movimientos al azar) y el *jugador de consola* (que toma sus movimientos de la entrada estándar). Una tarea extra consiste en implementar un jugador basado en *Monte Carlo Tree Search* plano. Se considerará correcto si el jugador le gana significativamente al jugador aleatorio.

Referencias

- *Tak* @ Wikipedia.
- *Tak* @ Board Game Geek.
- Cameron Browne *et al*, *"A Survey of Monte Carlo Tree Search Methods"*, IEEE Transactions on Computational Intelligence and AI in Games - March 2012.
