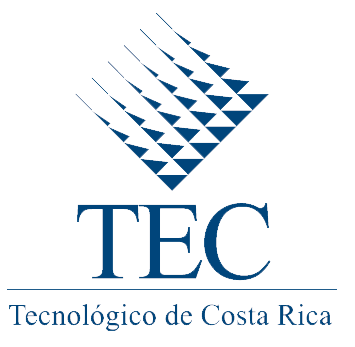
**Tecnológico de Costa Rica**

Sede San Carlos

****

**Escuela de Computación**

**Curso:** Lenguajes de Programación

**Profesor:**

Óscar Víquez Acuña

**Tarea Programada II:**

Juego de Othelo

**Estudiantes:**

Alonso Vega Brenes (201042592)

Kenneth Sancho Chacón (200941125)

Santa Clara, San Carlos

9 de Octubre, 2011

**Síntesis del problema**

Se debe realizar un Othelo, este es un juego que se desarrolla sobre un tablero cuyas dimensiones serán variantes según lo desee el jugador, al inicio deben aparecer cuatro fichas que estarán ubicadas en el centro del tablero, dos para cada jugador y estas se ubican transversalmente para cada jugador.

El juego consiste en encerrar las fichas del jugador contrario al menos entre dos fichas de las nuestras, mínimo debe encerrar una, estas en cualquier dirección (vertical, horizontal o transversal), lo que provoca que las fichas del jugador contrario se vuelvan nuestras. Una vez que se realice un movimiento válido se debe cambiar de jugador y este debe jugar cumpliendo con las medidas antes descritas y así hasta que se coloque la última ficha.

Existe la posibilidad de que un jugador no tenga opciones de “comer” una ficha del contrario por lo que la aplicación debe automáticamente transferir el turno al jugador contrario. Una vez que se coloque la última ficha sobre el tablero se hace un conteo de las fichas para ambos jugadores y se determina cual es el ganador.

**Solución y estructuras**

Jugador – Jugador

Para la solución de este problema desarrollamos funciones que a partir de un “x” y “y” de la matriz, comprueban las 8 direcciones en que se puede mover una ficha, si existe alguna jugada valida se ejecuta y se hacen los cambios respectivos en la matriz, en la dirección que se comprobó que había jugada.

Una vez que se hace el movimiento, se debe comprobar si el juego terminó, en caso de que no sea así, se hace el cambio de jugador y se comprueba que este tenga jugadas, ya que puede suceder en “x” momento que alguno de los jugadores no tenga movimientos válidos por lo que se debe realizar un cambio de jugador de nuevo.

Jugador – Máquina

Para permitir que un jugador pueda jugar en contra de la máquina, elaboramos varios algoritmos y usamos otros ya implementados en jugador contra jugador.

Este modo se activa cuando se selecciona “Jugar contra máquina”. Una vez activado, las fichas negras (jugador dos) serán puestas mediante una serie de algoritmos que elegirán la mejor jugada posible para el tablero actual.

Cuando el jugador humano realiza su jugada, se comprueba si el modo contra la máquina está activado. Si es así, se procede a buscar las posibles jugadas que tendría el jugador 2. Estas jugadas se obtienen revisando cada casilla del tablero, y seleccionando únicamente las que resulten en jugadas válidas. En general, esta función mencionada es una implementación de la función “Vecinos”, pues obtiene las posibles jugadas a realizar.

Cuando se tiene la lista de posibles jugadas, se realiza una búsqueda en profundidad para averiguar qué jugada puede traer el mayor beneficio al jugador, y esto se realiza contando la cantidad de fichas de cada jugador en cada posible nuevo estado de la matriz de juego (el tablero). El tablero en el que se obtenga un mayor índice de fichas con respecto al jugador enemigo, resulta siendo el mejor, por lo que se selecciona esa jugada para ser realizada.

La máquina ejecuta la jugada seleccionada y el control vuelve al humano, no sin antes realizar las comprobaciones de fin de turno, las cuales comprueban que el siguiente jugador tenga jugadas válidas. Si esto no ocurre, el control vuelve al otro jugador, y si vuelve a ocurrir que no quedan jugadas válidas, el juego termina y el ganador es quien tenga más fichas.

También se acaba el juego cuando el tablero está lleno, y se elige ganador el jugador con mayor cantidad de fichas.

**Comentarios finales**

La gran diferencia que se puede notar en un lenguaje de programación como lo es Scheme, se encuentra en el gran uso de la recursividad. Manejar esta técnica requiere cierta práctica, pues tiende a confundir. Junto a la recursividad, encontramos el manejo de listas, y en este caso, la ausencia de variables con restricciones de tipos. Todos estos elementos, junto al hecho de que el lenguaje sea interpretado implican una manera muy distinta de programar, con respecto a los lenguajes imperativos.

Sin duda, el programador puede abstraerse de una gran cantidad de funcionalidades, como lo son el manejo de memoria, de listas y de almacenamiento por tipos. Por otro lado, se puede destacar que el rendimiento de los programas interpretados es más bajo, y sumado al hecho de que la recursividad no siempre es el método más eficiente, puede ocasionar que los resultados obtenidos sean más lentos. La recursividad ocasiona en muchos casos que se realice varias veces un mismo cálculo, el cual no se almacena; en un programa imperativo se podría salvar ese valor a memoria y utilizarlo cuando sea necesario, sin necesidad de calcularlo cada vez.

Cambiando un poco el tema, dejando de lado la eficiencia del lenguaje, se puede mencionar el ambiente de desarrollo. No es del todo malo, pues posee varias funciones útiles, incluyendo la opción de depurar, pero una gran desventaja es el diseño de interfaces y controles. Es necesario escribir cada control y cada propiedad, lo que sí da cierto nivel de control, pero reduce enormemente la velocidad para crear interfaces.

Otro problema que se puede mencionar, es el hecho de tener que trabajar en un mismo archivo de código fuente. Esto reduce la facilidad de trabajo en equipo.

Uno de los pros más rescatables, es la documentación del lenguaje, la cual incluye desde las bases de las clases y métodos, hasta paquetes extra que permiten realizar muchos otros procedimientos. Se incluye además una lista de tutoriales introductorios al lenguaje, los cuales resultan muy útiles para aprenderlo.

**Bibliografía**

* Documentación de Racket. Version 5.1.2.