Documentación del proyecto de *Organización de Computadoras - Ta-Te-Ti*

Nicolás Dato y Nicolás Medina

03/11/2019

Índice

1. Definiciones y especificación de requerimientos

1.1. Definición general del proyecto de software

Este proyecto se basa en la creación del famoso juego *Ta-Te-Ti*, el cual cuenta con 3 modos de juego, jugador vs jugador, jugador vs computadora y computadora vs computadora.

El modo jugador vs jugador consiste simplemente en turnos donde cada jugador realizará una jugada y se notificará si esta se realizó exitosamente, en caso de que no fuera así (como intentar ubicar una ficha en una parte del tablero en el cual ya había una) se pedirá realizar la jugada nuevamente. Las jugadas se ejecutarán decidiendo la posición en la grilla, es decir, decidiendo el lugar mediante coordenadas x e y comenzando a contar desde 0.

Si se desea ejecutar uno de los dos modos donde participa la computadora, cuando sea su turno se ejecutará el algoritmo de **búsqueda adversaria MIN-MAX** donde se buscará la mejor jugará, dadas las características del *Ta-Te-Ti* esto significa que la computadora nunca pierde, sólo gana o empata.

Cualquier persona que sepa las reglas del juego puede ser capaz de jugar, lo único que tendrá que hacer es indicar x e y para colocar la ficha en la posición que se quiera.

1.2. Especificación de requerimientos del proyecto

La implementación del proyecto se ha realizado en el lenguaje C, el modo de visualización del menú y tablero se ha realizado en la consola. Se ha implementado los *TDA LISTA* y *TDA ARBOL* los cuales se encontrara en una librería dinámica denominada **libliar** y esta se utilizara para el algoritmo de **búsqueda adversaria MIN-MAX** el cual es implementado en un *TDA IA* . También se hizo uso de un *TDA PARTIDA* para el manejo del tablero, turnos, etc. Por ultimo, el programa principal se encargara de de manejar tanto el *TDA PARTIDA* como el *TDA IA* e irá mostrando el estado actual del tablero.

El usuario al iniciar el programa tendrá la opción de iniciar una nueva partida o de salir del sistema, en caso de iniciar una nueva partida tendrá que elegir en qué modo es el que desea jugar y luego decidir los turnos, también se pedirá ingresar el nombre de los jugadores.

El proyecto es un desarrollo original cumpliendo las pautas de implementacion dada por la cátedra de la materia *organización de computadoras*, aún así dado el desarrollo de clases y de estructura de datos aprendido en materias

anteriores los *TDA* implementados en el lenguaje C pueden ser reutilizados en proyectos/desarrollos posteriores.

1.3. Especificación de los procedimientos

1.3.1. Procedimientos de desarrollo

Como se dijo anteriormente, la implementación se realizó en el lenguaje C y se ha usado como IDE y compilador de dicho lenguaje el dado por la cátedra llamado **Code::Blocks y MinGW**. Se han usado varias librerías de este lenguaje, las primeras dos son *stdlib.h* y *stdio.h* las cuales son las librerías primordiales para el manejo de memoria y de I/O. Una de las otras librerias utilizadas fue *time.h* utilizada en el caso de que el usuario decida que empieza un jugador aleatorio. Otra librería utilizada fue la de *string.h* usada para copiar cadena de caracteres de una manera más sencilla.

Para abarcar este proyecto se decidió primero la implementación del *TDA LISTA* para luego poder implementar el *TDA ARBOL* y así estar en condiciones de encarar el algoritmo de **búsqueda adversaria MIN-MAX**. Aun así primero decidimos completar el *TDA PARTIDA* antes de desarrollar *TDA IA*.

Una vez con todos los *TDA* completados se pasó a la implementación del programa principal, donde se implementa un menu facilitando la experiencia del usuario.

Ya con todo finalizado se implementĺa librería **libliar** con los *TDA LISTA* y *TDA ARBOL*.

1.3.2. Procedimientos de compilación y ejecución

Para el uso del mismo, el usuario, el cual es así mismo un desarrollador, contará con el archivo comprimido **.zip** conteniendo todos los archivos .cpb, .h y .c para la compilacón y ejecución del programa.

Dado que se genera una librería (*libliar*), se requiere abrir 2 proyectos con *Code::Blocks*, tp1.cbp y libliar.cbp. En las propiedades del proyecto tp1 se puede configurar que tp1 tiene de dependencia el proyecto libliar[1], para que al compilar tp1 también se compile libliar.

Para compilar el proyecto, primero que hay compilar libliar, que va a generar los archivos *bin\Debug\libliar.a* y *bin\Debug\libliar.dll*.

[1]Lamentablemente no se logró hacer que esta configuración quede guardada para que al abrir el proyecto ya esté la dependencia agregada

Con libliar compilado, se puede compilar el proyecto tp1, el cual va a linkearse con *bin\Debug\libliar.a*[2]

Una vez compilado el proyecto tp1, en *bin\Debug* se genera el archivo *tp1.exe* para ser ejecutado.

2. Arquitectura del sistema

2.1. Descripción jerárquica

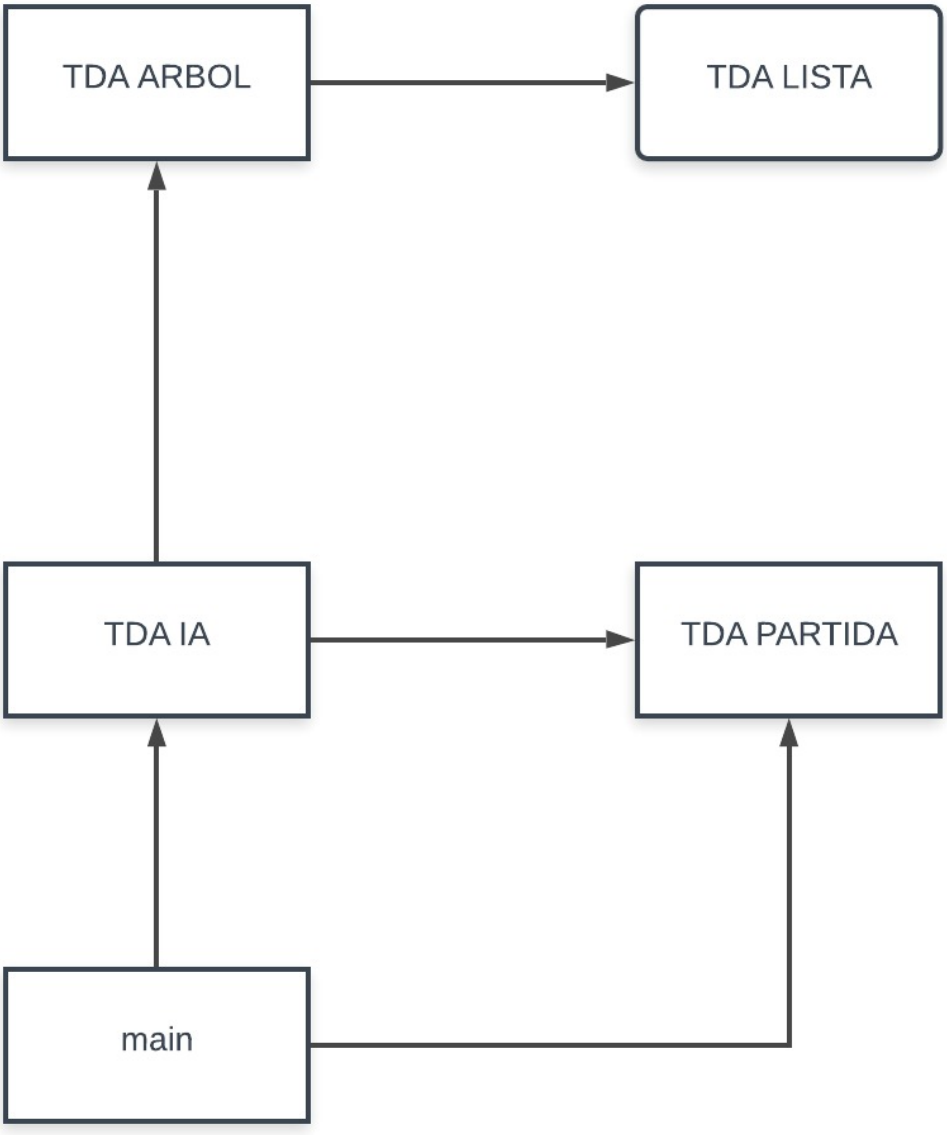
La arquitectura del sistema cuenta con un programa principal llamado **main** el cual se encarga de usar el *TDA PARTIDA* para la creación y administración de la misma, también en el **main** usará el *TDA IA* en casos de ser necesario, es decir, si el usuario desea iniciar uno de los dos modos donde juega la computadora.

El *TDA IA* utiliza el algoritmo de **búsqueda adversaria MIN-MAX**, también usará al *TDA PARTIDA* y *TDA ARBOL*.

Por último el *TDA ARBOL* usará al *TDA LISTA* para guardar una lista de sus hijos.

2.2. Diagrama de módulos

[2]Con *Code::Blocks* no se logró utilizar el archivo .dll, la única opción que había era linkear contra el archivo .a, de todas formas el archivo .dll se genera y está disponible.


TDA ARBOL
TDA LISTA
TDA IA
TDA PARTIDA
main

2.3. Descripción general de los módulos

2.3.1. Descripción general y propósito

- **TDA LISTA**: Implementa nodos enlazados los cuales apuntan al siguiente nodo y almacenan un elemento genérico. La posición es indirecta y tiene un nodo centinela.
- **TDA ARBOL**: Implementa *tnodos* los cuales apuntan a su *tnodo* padre (en caso de no ser la raiz), almacenan un elemento genérico y una lista de hijos.
- **TDA PARTIDA**: Implementa una partida con información actual de la misma. La partida guarda tanto los nombres de los jugadores, el turno del jugador al que le toca jugar, el estado actual de la partida (si alguien ganó, etc) y el modo de la partida que se está jugando. El tablero apunta a una grilla de 3x3 la cual guarda el estado del mismo teniendo en cada posición los valores 0, 1 o 2 para el caso de una casilla vacía, una ficha del jugador 1 o una ficha del jugador 2, respectivamente.
- **TDA IA**: Implementa un estado y una búsqueda adversaria. El estado guarda una grilla de 3x3 con la que fue llamada y la utilidad el cual representa el resultado de la grilla (si se sigue jugando, alguien ganó, etc). Busqueda adversaria apunta a un árbol y a dos enteros, el jugador max y el jugador min. Con esta información el algoritmo buscará la mejor jugada que puede realizar el jugador.
- **main**: Implementa un menú en la consola el cual administra el *TDA PARTIDA* y el *TDA IA* en caso de ser necesario.

2.3.2. Responsabilidad y restricciones

- **TDA LISTA**: La lista espera que cuando se le ingrese un elemento ya tenga un espacio en memoria asignado. La lista, al eliminar o destruir espera una función con la cual eleminará al elemento y liberará el espacio asignado a memoria del mismo. Al insertar un elemento lo asignará como siguiente a la posición recibida.
- **TDA ARBOL**: El arbol espera después de ser creado que se le asigne una raíz como primer elemento, y en caso de que se le intente asignar un nodo que no sea raíz se informará de un error. También cuando se desee insertar un elemento se le tiene que asignar un espacio en memoria al mismo previamente. Al eliminar o destruir, de igual manera que la lista se espera que se le pase una función por parámetro la cual se

encargara del eliminar y borrar de la memoria al elemento. Al insertar un elemento se esperan dos tnodos, el primero de ellos debe ser el padre al cual se le desea agregar un hijo y el segundo es el hermano derecho al nodo a insertar, en caso de que el segundo nodo sea NULL el elemento se asigna como el último hijo del tnodo padre.

- **TDA PARTIDA**: La partida lo que espera como modo y turno son enteros, asociados a constantes ya definidas en el TDA, en el caso del nombre se espera como máximo 49 caracteres. Si dado al caso se intenta hacer un movimiento en una casilla ya ocupada, retornará un error.

- **TDA IA**: La ia al momento de crear la búsqueda adversaria lo que espera es que se le haya pasado correctamente la partida en su estado actual, para así poder empezar con el algoritmo. En el caso de resultado esperado lo que espera la ia es que el resultado buscado siempre sea el mejor, es decir, gana max, ya que la ia nunca pierde, sólo gana o empata. También espera dos punteros a enteros los cuales va a modificar con la posición en la cual esta la mejor jugada posible.

- **main**: Las restricciones al programa principal son la longitud de los nombres de los jugadores en 49 caracteres y la elección de las opciones del menú espera un número entero.

2.3.3. Dependencias

- **TDA LISTA**: Este TDA no requiere de ningun paquete externo o librería ademas de las librerias esenciales proporcionadas por C.

- **TDA ARBOL**: Este TDA de la librería del *TDA LISTA* ya mencionado.

- **TDA PARTIDA**: Este TDA además de las librerías esenciales de C requiere de las librerías *time.h* y *string.h*.

- **TDA IA**: Este TDA requiere de los *TDA PARTIDA* y *TDA ARBOL* ya mencionados.

- **main**: El main solo requiere del *TDA IA* debido a que este ya tiene contenido todo el resto de los TDA y librerías.

2.3.4. Implementación

- **TDA LISTA**: Este TDA se encuentra implementado en la librería dinámina llamada **libliar**.

- **TDA ARBOL**: Este TDA se encuentra implementado en la librería dinámica llamada **libliar**.
- **TDA PARTIDA**: Este TDA se encontrará definido en un .h denominado *partida.h* e implementado en un .c llamado *partida.c*. Estos archivos se encontraran en el **.zip**.
- **TDA IA**: Este TDA al igual que el anterior se encuentra definido en *ia.h* e implementado en *ia.c*. Estos archivos se encontrarán en el **.zip**.
- **main**: Este se encuentra implementado en *main.c*. Este archivo se encontrará en el **.zip**.
- **libliar**: Librería dinámica con los *TDA LISTA* y *TDA ARBOL*, se encontrará en el **.zip**.

2.4. Dependencias externas

- **stdio.h**: Es la librería que contiene las definiciones de las macros, las constantes, las declaraciones de funciones de la biblioteca estándar del lenguaje de programación C para hacer operaciones de entrada y salida, así como la definición de tipos necesarias para dichas operaciones.
- **stdlib.h**: Es la librerá cabecera de la biblioteca estándar de propósito general del lenguaje de programación C. Contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otros.
- **time.h**: Es la librería relacionada con formato de hora y fecha, es un archivo de cabecera de la biblioteca estándar del lenguaje de programación C que contiene funciones para manipular y formatear la fecha y hora del sistema.
- **string.h**: Es una librería de la biblioteca estándar del lenguaje de programación C que contiene la definición de macros, constantes, funciones y tipos y algunas operaciones de manipulación de memoria relacionadas a cadena de caracteres.

3. Descripción de procesos y servicios ofrecidos por el sistema

3.1. Invocación del programa

Para ejecutar el programa sólo basta con ejecutar *tp1.exe*. No espera ningún parámetro.

3.2. TDA LISTA

3.2.1. Creación de la lista

La lista se crea en la función *crear_lista(tLista *l)*, la cual creará en l una nueva estructura y la inicializará con los valores de una lista vacía, es decir una lista con el elemento centinela pero sin más elementos.

3.2.2. Insertar elemento

Para insertar elemento está la función *void l_insertar(tLista l, tPosicion p, tElemento e)*, esta función crea una nueva celda y la inserta en la posición indicada. Dado que se trabaja con posición indirecta, para insertar se tiene que agregar la celda nueva en *p.siguiente* para que al recuperar el elemento en la posición p se obtenga el elemento insertado. El anterior elemento en p pasará a ser el siguiente del nodo insertado.

3.2.3. Eliminar elemento y destrucción de la lista

Las funciones *void l_destruir(tLista * l, void (*fEliminar)(tElemento))* y *void l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento))* destruyen la lista y eliminan un nodo, respectivamente. Al eliminar un nodo lo que se hace es eliminar *p.siguiente* dado que se trabaja con posición indirecta, y eliminando el elemento de la celda con $fEliminar$. En el caso de destruir la lista el procedimiento es similar, eliminando todos los elementos de la lista para luego eliminar la lista en sí. Los nodos y la lista se eliminan con *free()*.

3.2.4. Obtener posiciones de los nodos y la longitud de la lista

Para obtener la primera posición, la anterior y siguiente de una posicón, la última posición y el fin de la lista se tiene las funciones *tPosicion l_primera(tLista l)*, *tPosicion l_anterior(tLista l, tPosicion p)*, *tPosicion l_siguiente(tLista l, tPosicion p)*, *tPosicion l_ultima(tLista l)*, *tPosicion*

l_fin(tLista l). Teniendo en cuenta que se trabaja con posición indirecta estas funciones resultan triviales.

Para recuperar el elemento de una posición se utiliza la función *tElemento l_recuperar(tLista l, tPosicion p)*

La longitud de la lista se obtiene *int l_longitud(tLista l)*, una función recursiva que tiene de caso base la lista vacía (longitud 0) y sino suma 1 por cada elemento recursivamente hasta llegar al caso base.

3.3. TDA ARBOL

3.3.1. Creación del árbol y su raíz

El árbol se crea con la función *void crear_arbol(tArbol * a)*, que va a crear un árbol sin raíz. Para ingresarle la raíz hay que llamar a *void crear_raiz(tArbol a, tElemento e)*.

El árbol estará representado como nodos, donde cada nodo tiene un puntero a su padre y tiene una lista de hijos.

3.3.2. Ingresar elemento

Para ingresar elementos al árbol, existe la función *tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e)*, la cual ingresa el elemento e como hijo de np a la izquierda de nh, si nh es $NULL$ entonces se ingresa como último hijo de np.

Al ingresar un elemento se crea un nodo nuevo y con una lista vacía de hijos, en caso de que nh sea $NULL$ se ingresa el nuevo nodo al final de la lista de hijos de np; pero si nh no es $NULL$ entonces primero se busca a nh entre los hijos de np para poder ingresar el nuevo nodo antes de nh.

3.3.3. Eliminar elemento y destrucción del árbol

Las funciones *void a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento))* y *void a_destruir(tArbol * a, void (*fEliminar)(tElemento))* eliminan un elemento y destruyen el árbol respectivamente.

Para eliminar un elemento, hay 3 casos válidos: que sea la raíz sin hijos, que sea la raíz con sólo un hijo, que no sea la raíz; en todos los casos se destruye la lista de hijos y se elimina el elemento de la con $fEliminar$.

En el caso de que sea una raíz sin hijos, sólo se quita la raíz del árbol. Cuando sea la raíz con un único hijo, ese elemento pasa a ser la nueva raíz.

Si el elemento a eliminar es un nodo que no es raíz, entonces todos los hijos del nodo pasan a estar como hijos del padre, en el mismo orden y en el lugar donde estaba el nodo. Para realizar esta operación se tuvo que usar una

función *no_eliminar()* que no hace nada y pasar esa función a *l_eliminar()*. Esto se debe a que se necesita quitar los elementos del nodo sin eliminarlos para poder ingresarlos como hijos del padre, pero la interface del *TDA LISTA* no tiene una forma de quitar el elemento sin que se llame a una función de eliminado, entonces con *no_eliminar()* el nodo es eliminado de la lista pero sin destruir al elemento y se puede así recuperar un elemento de la lista, eliminarlo de la lista sin que se elimine el elemento en si, y luego colocar el elemento recuperado en otra lista.

Para la destrucción de todo el árbol, se utiliza una función auxiliar *n_eliminar(tNodo n, void (*fEliminar)(tElemento))* que va a eliminar al elemento del nodo n y luego recursivamente va a eliminar los nodos hijos de n para luego destruir la lista.

3.3.4. Obtener elementos

Para operar con el árbol y obtener elementos, están las funciones *tNodo a_raiz(tArbol a)*, *tLista a_hijos(tArbol a, tNodo n)* y *tElemento a_recuperar(tArbol a, tNodo n)* que van a devolver el nodo raíz, una lista de *tNodo* hijos de un nodo, y obtener el elemento del nodo respectivamente. Son funciones triviales.

3.3.5. Subárbol

Se puede obtener un subárbol utilizando *void a_sub_arbol(tArbol a, tNodo n, tArbol * sa)*. Esa función va a crear un nuevo árbol en *sa* donde n es raíz con todos sus hijos, y en el árbol original a se quita a n y sus hijos.

Para realizar esto se busca al nodo n en la lista de hijos de su padre, y se lo quita sin destruir al elemento (utilizando la funcón *no_eliminar()* tal como se hacía en *a_eliminar()*).

3.4. TDA PARTIDA

3.4.1. Crear partida nueva y finalizarla

Para crear una nueva partida hay que llamar a la función *void nueva_partida(tPartida * p, int modo_partida, int comienza, char * j1_nombre, char * j2_nombre)*, y se va a inicializar las estructuras con el modo de la partida, quién comienza, los nombres de los jugadores y un tablero sin fichas colocadas (todas las posiciones en 0). Si el jugador que comienza fuese aleatorio, se pide un número pseudo-aleatorio con *rand()*, si el resultado es par comienza el jugador 1, si no comienza el jugador 2.

Para finalizar la partida se utiliza la función *finalizar_partida(tPartida * p)*, la cual va a eliminar la memoria utilizada para la partida.

3.4.2. Realizar movimientos

Se pueden realizar movimientos en el tablero con la fución *int nuevo_movimiento(tPartida p, int mov_x, int mov_y)*, se va a corroborar que la posición se encuentre vacía y va a realizar la jugada colocando un 1 para el jugador 1 y un 2 para el jugador 2. Luego revisa el tablero para saber si con la jugada realizada hay un ganador, o es empate, o si se puede seguir jugando.

3.5. TDA IA

3.5.1. Creación y destrucción de búsqueda adversaria

La búsqueda adversaria, utilizada para saber el mejor movimiento a realizar, se crea con
*void crear_busqueda_adversaria(tBusquedaAdversaria * b, tPartida p)*. Se va a crear un árbol donde se va a ejecutar el algoritmo *Min-Max*, siendo la raíz del árbol el tablero actual y siempre arrancando a jugar como *Max* para maximizar la jugada. El árbol se crea de tal forma de que en cada nivel se vaya alternando entre *Min* y *Max*, se llama recusrivamente con cada uno de los posibles movimientos a realizar y luego se buscará la mejor jugada en el caso de ser *Max* o la peor jugada en el caso de ser *Min*. También se detendrá la búsqueda si se encuentra con una jugada ganadora o perdedora (según sea *Max* o *Min*) para no seguir procesando el árbol cuando no es necesario por que no podrá haber mejores resultados.

Para destruir la búsqueda adversaria hay que llamar a
*void destruir_busqueda_adversaria(tBusquedaAdversaria * b)* que va a destruir el árbol y liberar la memoria utilizada.

3.5.2. Obtención del próximo movimiento

Una vez creada la búsqueda adversaria, se utiliza
*void proximo_movimiento(tBusquedaAdversaria b, int resultado_esperado, int * x, int * y)* para obtener el mejor movimiento. Esta función va a revisar los hijos de la raíz (es decir, las posibles jugadas que se tienen a partir del estado actual) y se va a quedar con la mejor jugada de todas, luego comparando el tablero actual con el tablero de la mejor jugada se obtiene la diferencia y se indican en x e y las coordenadas a jugar. Dada la forma en que se crea el árbol para el algoritmo *Min-Max*, el valor de utilidad de la raíz va a ser la utilidad de la mejor jugada de sus hijos (dado que la raíz es siempre *Max*), por lo que buscar la mejor jugada entre los hijos es buscar al hijo que tenga la misma utilidad que la raíz.

3.6. Main - programa principal

El programa principal primero va a pedir toda la información de la partida (jugadores, modo, etc.) y va a crear una nueva partida. Luego va a iterar mientras no haya ganadores ni empate y va a pedir las coordenadas de la ficha al jugador y va a realizar la jugada. En caso de que sea el turno de la máquina va a crear la búsqueda adversaria del *TDA IA*, va a buscar la mejor jugada y la va a realizar.

Una vez terminado el juego se pregunta al usuario si se quiere seguir jugando o si se quiere salir, pudiendo jugar nuevamente sin reiniciar el programa.

4. Conclusiones

Lo más novedoso y que más costó entender fue el algoritmo de *Min-Max*, entre otras cosas por algunas diferencias entre el pseudocódigo del algoritmo y la interface que había que implementar donde el pseudocódigo proponía una función recursiva que retorna un valor mientras que la interface no retorna ningún valor, pero lo pudimos resolver guardando el valor que se debe retornar en la estructura del estado de la partida.

El análisis del tablero para saber si hay algún ganador lo intentamos hacer sólo revisando la posición jugada y sus alrededores (sin recorrer todo el tablero), pero luego de algunos intentos fallidos (andaba para algunos casos y para otros no), optamos por analizar todo el tablero para saber si hay algún ganador o empate.

Al implementar el *TDA ARBOL* nos encontramos con una limitación del *TDA LISTA*, no se puede quitar un elemento de la lista sin eliminarlo. Nos hubiera gustado que la interfaz incluya una función que retorne el elemento y al mismo tiempo lo quite de la lista, de esa forma se pueden mover elementos de una lista a otra sin tener que eliminarlos y recrearlos. Pero para sortear este tema optamos por hacer una función que no hace nada y pasar esta función al momento de querer eliminar un elemento de la lista, así podemos recuperar un elemento, eliminarlo de la lista sin que se elimine el elemento en si, y luego colocar el elemento en otra lista. Esto fue necesario al eliminar nodos y al crear un sub-árbol donde hay que operar con las listas y sus nodos y cambiarlos de lugar.

Si se ingresa el juego en modo *Computadora-vs-Computadora* el resultado es un empate, que es lo esperado dado que en *Ta-Te-Ti* si ambos jugadores juegan la mejor jugada es un empate. Notamos que la primer jugada de la computadora no es en el centro sino en una esquina, creemos que esto sucede

por que con el tablero vacío no hay ninguna jugada que sea ganadora, la mejor jugada va a ser un empate, y entonces jugando tanto en el centro como en la esquina el resultado sea empate y por algo de la implementación o de la forma que se recorre/genera el árbol la máquina se quede con la jugada de la esquina.