



Documentación de Proyecto

TaTeTi

Rodrigo Nicolás Herlein

3 de Noviembre de 2019

Organización de Computadoras

Universidad Nacional del Sur



Introducción	3
Librerías externas utilizadas:	3
1. "stdio.h": Contiene funcionalidad para hacer operaciones de entrada y salida.	3
2. "stdlib.h": Es una librería de propósito general.	3
Módulos Desarrollados	3
TDA LISTA	3
Operaciones:	4
TDA Arbol	6
Operaciones:	6
Operaciones Auxiliares:	9
TDA Partida	10
Operaciones:	10
TDA IA	11
Algoritmo MINMAX con podas α y β :	11
Operaciones:	13
Operaciones auxiliares:	14
Modo de Ejecución	18
Modo de Uso	20
Conclusiones/Otros	24
Representación de las fichas dentro del programa:	24
Situaciones Particulares en la ejecución del programa:	24

Introducción

El juego desarrollado se llama de tres en línea o TaTeTi y posee los modos de juego: usuario vs usuario, usuario vs computadora y computadora vs computadora. En particular, las últimas dos modalidades poseen una inteligencia artificial basada en el algoritmo minmax utilizando podas alfa-beta que definen el comportamiento de la computadora.

Librerías externas utilizadas:

1. "stdio.h": Contiene funcionalidad para hacer operaciones de entrada y salida.
2. "stdlib.h": Es una librería de propósito general.

Módulos Desarrollados

TDA LISTA

Estructura utilizada para representar la lista: Está conformada por los archivos "lista.h" y "lista.c". Se programó haciendo uso de posición indirecta y con una celda centinela al inicio de la lista. Para el correcto funcionamiento de este módulo, se requiere la presencia de las librerías "stdlib" y "stdio" provistas por C.

Se encapsula la lista, sus posiciones y los elementos de cada una con las estructuras tLista, tPosicion y tElemento respectivamente. Las primeras dos estructuras son definidas como punteros a una estructura de tipo celda y la última como un puntero a void.

Operaciones:

crear_lista(tLista * l):

parámetros:

- Un puntero de tipo tLista llamado l.

Inicializa la lista referenciada por l dejándola vacía.

l_insertar(tLista l, tPosicion p, tElemento e):

parámetros:

- Una lista llamada l.
- Una posición de la lista l llamada p
- Un elemento llamado e

Se inserta una celda con rótulo e en la posición de la lista correspondiente a p. Dejando a p en la posición siguiente.

l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento)):

Parámetros:

- Una lista llamada l.
- Una posición de la lista l llamada p
- Una función fEliminar que se libera la memoria ocupada por una variable de tipo tElemento

Elimina la posición p de la lista liberando la memoria de su rótulo con fEliminar.

Si P no es una posición válida finaliza indicando LST_POSICION_INVALIDA.

l_destruir(tLista * l, void (*fEliminar)(tElemento));

parámetros:

- Un puntero de tipo tLista llamado l.
- Una función fEliminar que se libera la memoria ocupada por una variable de tipo tElemento

Elimina la lista referenciada por l, liberando la memoria ocupada por ella y los elementos de cada posición.

int l_recuperar(tLista l, tPosicion p):

Parámetros:

- Una lista llamada l.
- Una posición de la lista l llamada p

Retorno:

- Retorna el elemento guardado en la posición p de la lista.

Excepciones:

- Si P no es una posición válida finaliza indicando LST_POSICION_INVALIDA.

Busca el elemento guardado en la posición p de la lista y lo retorna

tPosicion l_primera(tLista l):

Parámetros:

- Una lista llamada l.

Retorno:

- Retorna la primera posición de la lista l. En caso de que la lista este vacía retorna l.

Busca la primer posición de la lista y la retorna

tPosicion l_siguiente(tLista l, tPosicion p):

Parámetros:

- Una lista llamada l.
- Una posición de la lista l llamada p

Retorno:

- Retorna la posición siguiente a p en la lista l.

Excepciones:

- Si P es fin(L), finaliza indicando LST_NO_EXISTE_SIGUIENTE.

Busca la siguiente posición a p en la lista y la retorna.

tPosicion l_anterior(tLista l, tPosicion p):

Parámetros:

- Una lista llamada l.
- Una posición de la lista l llamada p

Retorno:

- Retorna la posición anterior a p en la lista l.

Excepciones:

- Si P es primera(L), finaliza indicando LST_NO_EXISTE_ANTERIOR.

Busca la anterior posición a p en la lista y la retorna.

tPosicion l_ultima(tLista l):*Parámetros:*

- Una lista llamada l.

Retorno:

- Retorna la última posición de la lista l. En caso de que la lista esté vacía retorna l
- Busca la última posición de la lista y la retorna.

tPosicion l_fin(tLista l):*Parámetros:*

- Una lista llamada l.

Retorno:

- Retorna la posición fin de la lista l. En caso de que la lista esté vacía retorna l
- Busca la última posición al final de la lista y la retorna.

TDA Arbol

Estructura utilizada para representar el Árbol General: Está conformada por los archivos “arbol.h” y “arbol.c”. Se programó haciendo uso de nodos con listas de hijos y referencias a su padre. Para el correcto funcionamiento de este módulo, se requiere la presencia de las librerías “stdlib” y “stdio” provistas por C.

Se encapsula el árbol, sus nodos y los elementos de cada uno con las estructuras tArbol, tNodo y tElemento respectivamente. La primera estructura es definida como un puntero a una estructura de tipo nodo, la segunda como un puntero a una estructura de tipo árbol y la última como un puntero a void.

Operaciones:

crear_arbol(tArbol * a):*Parámetros:*

- Un puntero de tipo tArbol llamado a
- .Inicializa el árbol referenciado por a dejándolo vacío.

crear_raiz(tArbol a, tElemento e):*Parámetros:*

- Un árbol llamado a.
- Un elemento llamado e

Excepciones:

- Si A no es vacío, es decir, ya tiene raíz, finaliza indicando ARB_OPERACION_INVALIDA

Crea la raíz del árbol a con rótulo e.

tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e):*Parámetros:*

- Un árbol llamado a.
- Un nodo padre llamado np
- Un nodo hijo de np llamado nh
- Un elemento llamado e

Retorno

- Retorna el nodo insertado.

Excepciones

- Si NH no corresponde a un nodo hijo de NP, finaliza indicando ARB_POSICION_INVALIDA.

Inserta un nuevo nodo al árbol con rotulo e, hermano izquierdo nh y padre np. y luego retorna el nodo insertado.

si nh es nulo, se inserta al nuevo nodo como hijo extremo derecho de np.

a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento)):*Parámetros:*

- Un árbol llamado a.
- Un nodo llamado n
- Una función llamada fEliminar que libera el espacio en memoria ocupado por una variable de tipo tElemento

Excepciones

- Si n es la raíz del árbol a y tiene más de un hijo se finaliza retornando ARB_OPERACION_INVALIDA.

Elimina del árbol el nodo n y libera la memoria ocupada por su rótulo usando fEliminar.

Si n es la raíz del árbol a y tiene un hijo, entonces ese hijo será la nueva raíz del árbol a.

Si n no es la raíz de a y tiene hijos, estos pasan a ser hijos del padre de n, en el mismo orden y a partir de la posición que ocupa n en la lista de hijos de su padre.

a_destruir(tArbol * a, void (*fEliminar)(tElemento)):

Parámetros:

- Un puntero a un árbol llamado a.
- Una función llamada fEliminar que libera el espacio en memoria ocupado por una variable de tipo tElemento.

Elimina el árbol referenciado por a, liberando la memoria de este, sus nodos y la memoria de los rótulos con fEliminar

Para recorrer el árbol referenciado por a y liberar su memoria utiliza la función auxiliar destruir_recursoivo.

tElemento a_recuperar(tArbol a, tNodo n):

Parámetros:

- Un árbol llamado a.
- Un nodo llamado n

Retorno

- Retorna el rótulo de n

Busca y retorna el rótulo del nodo n.

tNodo a_raiz(tArbol a):

Parámetros:

- Un árbol llamado a.

Retorno:

- El nodo raíz del árbol a

Retorna la raíz del árbol a.

tLista a_hijos(tArbol a, tNodo n):

Parámetros:

- Un árbol llamado a.
- Un nodo llamado n

Retorno

- Una lista compuesta de los hijos del nodo n

Retorna una copia de la lista de hijos del nodo n.

a_sub_arbol(tArbol a, tNodo n, tArbol * sa):

Parámetros:

- Un árbol llamado a.
- Un nodo llamado n
- Una referencia a un árbol llamado sa.

Inicializa el árbol sa, desvincula el subárbol con raíz n del árbol a y lo convierte en un árbol independiente referenciado por la variable sa.

Operaciones Auxiliares:

liberar_memoria(tElemento e);

Parámetros:

- Un elemento llamado e

Libera la memoria ocupada por e

anular (tElemento e):

Parámetros:

- Un elemento llamado e

Función dummy, no realiza ninguna acción

destruir_rekursivo(tLista hijos, void (*fEliminar)(tElemento) ,tNodo n);

Parámetros:

- Una lista Lista de hijos del nodo n llamada hijos.
- Una función llamada fEliminar que libera el espacio en memoria ocupado por una variable de tipo tElemento.
- Un nodo llamado n.

Recorre un árbol en posorden a partir del nodo n liberando la memoria de este nodo y de sus hijos.

void eliminaraux(tArbol a, tNodo n)

Parámetros:

- Un árbol llamado a
- Un nodo n del árbol a

Elimina el nodo n del árbol a sin eliminar la información guardada en este

TDA Partida

Estructura utilizada para representar la partida: “partida.h” y “partida.c” representan el estado de la partida, el modo de juego seleccionado, el estado actual del tablero, los jugadores, el turno de cada jugador y sus nombres.

La partida es encapsulada en una estructura llamada tPartida la cual está definida como una referencia a una estructura de tipo partida. El tablero de la partida es encapsulado en una estructura llamada tTablero el cual está definido como una referencia a una estructura de tipo tablero. Para el correcto funcionamiento de este módulo, se requiere la presencia de las librerías “stdlib” y “stdio” provistas por C.

Las constantes definidas para representar los modos de partida posibles y los resultados de esta se encuentran detallados en “partida.h”

Operaciones:

nueva_partida(tPartida * p, int modo_partida, int comienza, char * j1_nombre, char * j2_nombre);

Parámetros:

- Un puntero de tipo tPartida llamado p
- Un entero que representa el modo de la partida a jugar
- Un entero que representa que jugador comenzará jugando la partida
- Un puntero a char que representa el nombre del jugador 1
- Un puntero a char que representa el nombre del jugador 2

Inicializa la partida referenciada por p con un tablero vacío, los atributos de p pasados por parámetro y con estado de partida en curso

finalizar_partida(tPartida * p);

Parámetros:

- Un puntero de tipo tPartida llamado p

Finaliza una partida y libera toda la memoria que ocupa.

int nuevo_movimiento(tPartida p, int mov_x, int mov_y):

Parametros:

- Una partida p
- Un entero llamado mov_x que representa una fila de la grilla
- Un entero llamado mov_y que representa una columna de la grilla

Retorno

- Retorna PART_MOVIMIENTO_OK si el movimiento ingresado es valido sino retorna PART_MOVIMIENTO_ERROR

Realiza un movimiento en la coordenada (mov_x , mov_y) en la grilla de la partida p y actualiza el estado de la partida si es necesario.

Si el movimiento se pudo concretar retorna PART_MOVIMIENTO_OK sino retorna PART_MOVIMIENTO_ERROR

TDA IA

Estructura utilizada para representar la inteligencia artificial: "ia.h" y "ia.c" conforman la inteligencia artificial que define el comportamiento de la computadora al jugar.

La inteligencia artificial contiene una estructura llamada busqueda_adversaria encapsulada en una estructura de nombre tBusquedaAdversaria y otra estructura llamada estado encapsulada en una estructura de nombre tEstado. Para el correcto funcionamiento de este módulo, se requiere la presencia de las librerías "stdlib" y "stdio" provistas por C.

tBusquedaAdversaria representa el árbol de búsqueda creado con el algoritmo minmax con podas alfa beta y tEstado representa un estado de la partida, su tablero y su posible resultado llamado valor utilidad.

Algoritmo MINMAX con podas α y β :

El comportamiento de la inteligencia artificial está definido por un árbol de búsqueda formado utilizando el algoritmo minmax con podas alfabeta, en cada nodo del árbol hay un estado de la partida compuesto por una grilla de juego y un valor de utilidad asignado en función del estado de la grilla y su posición en el árbol.

Los valores de utilidad de cada estado son las siguientes constantes definidas dentro del programa:

- IA_PIERDE_MAX=1001
- IA_EMPATA_MAX =1002
- IA_GANA_MAX=1003

- IA_NO_TERMINO=1005

Pasos del algoritmo minimax:

1. Crear un arbol de busqueda con todas las posibilidades de juego tal que el tablero actual sea la raíz del árbol y las definiciones de este, las hojas del árbol.
2. El primer nivel del árbol será llamado MAX, el segundo será llamado MIN, el tercero MAX y así sucesivamente. Los niveles MAX representan los movimientos de la computadora y los niveles MIN los movimientos del oponente.
3. Calcular el valor utilidad de cada nodo hoja del árbol.
4. Calcular el valor utilidad de los nodos superiores a partir del valor de los inferiores. Según nivel si es MAX se elegirá como valor utilidad de un nodo n , el valor utilidad máximo de los hijos de n , si el nivel es MIN se elegirá como valor utilidad de un nodo n , el valor utilidad mínimo de los hijos de n .

Por último debemos elegir la jugada a realizar en función de los valores que han llegado al nivel superior. La mejor jugada será la que posea el valor utilidad necesario para la situación actual del tablero.

El orden del tiempo de ejecución de este algoritmo es n factorial. Para agilizar la ejecución del programa se utilizan las podas alfa-beta que buscan evitar que el algoritmo minimax recorra ramas del arbol de busqueda que no aporten a la solución del problema. Se calculará, para cada nodo del árbol, un valor alfa y un valor beta a partir de los nodos terminales del árbol de búsqueda.

El significado intuitivo de α y β en cada momento es:

- **Estados Max:** α es el valor de utilidad actual del estado N (que finalmente será igual o superior a α), y β es el valor de utilidad actual del padre del estado N (que finalmente será igual o inferior a β).
- **Estado Min:** β es el valor de utilidad actual del estado N (que finalmente será igual o inferior a β), y α es el valor de utilidad actual del padre del estado N (que finalmente será igual o superior a α).

La poda se produce si en algún momento $\alpha \geq \beta$, y en ese momento no hace falta analizar los hijos restantes del estado en el que me encuentro. En nodos Min, se denomina poda β , y en nodos Max, poda α .

Operaciones:

crear_busqueda_adversaria(tBusquedaAdversaria * b, tPartida p);

Parámetros

- Un puntero de tipo tBusquedaAdversaria llamado b
- Una partida p

Inicializa el arbol de busqueda apuntado por b haciendo uso del algoritmo minmax con podas alfa beta y los atributos jugador_min, jugador_max del mismo haciendo uso de los datos que contiene p.

Se asume que la partida tiene estado PART_EN_JUEGO.

proximo_movimiento(tBusquedaAdversaria b, int * x, int * y);

Parámetros:

- Un árbol de búsqueda llamado b
- Un puntero a un entero llamado x que representa una fila de la grilla de juego
- Un puntero a un entero llamado y que representa una columna de la grilla de juego

Computa el próximo movimiento a realizar por el jugador MAX haciendo uso del árbol creado por el algoritmo de búsqueda adversaria Min-max con podas Alpha-Beta llamado b y lo retorna.

Siempre que sea posible, se indicará un movimiento que permita que MAX gane la partida.

Si no existe un movimiento ganador para MAX, se indicará un movimiento que permita que MAX empate la partida.

En caso contrario, se indicará un movimiento que lleva a MAX a perder la partida.

destruir_busqueda_adversaria(tBusquedaAdversaria * b);

Parámetros

- Un puntero de tipo tBusquedaAdversaria llamado b

Libera el espacio ocupado por el arbol de busqueda adversaria apuntado por b

Operaciones auxiliares:

liberar_estado(tElemento e);

Parámetros:

- Un elemento llamado e

Libera la memoria ocupada por e

int esta_ganando(tEstado T, int ficha, int*x, int*y);

Parámetros:

- Un estado de juego llamado T
- Un entero que representa una ficha de juego (0 si es un círculo, 1 si es una x)
- Un puntero a un entero llamado x que representa una fila de la grilla de juego
- Un puntero a un entero llamado y que representa una columna de la grilla de juego

Retorno

- Retorna 1 si el jugador que usa la ficha pasada por parámetro puede ganar en el próximo movimiento, sino retorna 0

Verifica en el estado de juego T si el jugador que usa la ficha pasada por parámetro puede ganar en el próximo movimiento y retorna 1 si es así, junto con el movimiento más aconsejable a realizar en el siguiente turno. En el caso de que el jugador no esté por ganar retorna 0.

recorrido_random(int componentes[2][3]):

Parámetros:

- Un arreglo de 2 filas y 3 columnas llamado componentes

Inicializa la matriz pasada por parámetro donde:

La fila 0 representa las filas de la grilla de juego ordenadas de manera aleatoria

La fila 1 representa las columnas de la grilla de juego ordenadas de manera aleatoria

En el caso de que la primera columna ya esté inicializada la saltea.

tEstado clonar_estado(tEstado e):

Parámetros

- Un estado de la partida llamado e

Retorno

- Un clon del estado e pasado por parametro

Inicializa y retorna un nuevo estado que resulta de la clonación del estado E.

tLista estados_sucesores(tEstado e, int ficha_jugador):*Parámetros*

- Un estado de la partida llamado e
- Un entero que representa la ficha de un jugador

Retorno

- Retorna una lista con los estados sucesores de e

Computa y retorna una lista con aquellos estados que representan estados sucesores al estado e. Se llama estado sucesor a todos los estados de partida con un movimiento más que en e. Dicho movimiento es realizado usando ficha_jugador en algún espacio libre de la grilla de juego contenida en e.

La lista de estados sucesores se retorna de forma aleatoria, de forma tal que una doble invocación de la función retornara dos listas L1 y L2 tal que:

- L1 y L2 tienen exactamente los mismos estados sucesores de e a partir de jugar ficha_jugador.
- El orden de los estado en L1 será diferente al orden de los estados en L2 con una excepción, si el oponente puede ganar en su próximo movimiento entonces el estado donde se frustra dicha posibilidad de ganar siempre estará primero en la lista de sucesores.

int valor_utilidad(tEstado e, int jugador_max)*Parámetros:*

- Un estado de la partida llamado e
- Un entero que representa al jugador max

Retorno

- El valor utilidad del estado e

Computa el valor de utilidad correspondiente al estado e, y la ficha correspondiente al jugador max, retornando:

- IA_GANA_MAX(1003) si el estado E refleja una jugada en el que el jugador max ganó la partida.
- IA_EMPATA_MAX(1002) si el estado E refleja una jugada en el que el jugador max empató la partida.
- IA_PIERDE_MAX(1001) si el estado E refleja una jugada en el que el jugador max perdió la partida.
- IA_NO_TERMINO(1005) en caso contrario.

ejecutar_min_max(tBusquedaAdversaria b):

Parámetros:

- Un árbol de búsqueda llamado b

Ordena la ejecución del algoritmo Min-Max para la generación del árbol de búsqueda adversaria, considerando como estado inicial el estado de la partida almacenado en la raíz del árbol apuntado por B.

dummy(tElemento e)

Parámetros

- Un elemento llamado e

No hace nada.

int min2(int A, int B)

Parámetros

- Un entero llamado A
- Un entero llamado B

Retorno

- El número más chico entre A y B

Calcula el valor mínimo entre los parámetros A y B

int max2(int A, int B)

Parámetros

- Un entero llamado A
- Un entero llamado B

Retorno

- El número más grande entre A y B

Calcula el valor máximo entre los parámetros A y B

crear_sucesores_min_max(tArbol a, tNodo n, int es_max, int alpha, int beta, int jugador_max, int jugador_min)

Parámetros:

- Un árbol de búsqueda llamado a
- Un nodo del árbol a, llamado n a partir del cual se construirá el árbol de búsqueda
- Un entero llamado es_max que determina si n es un nodo max del árbol
- Un entero llamado alpha
- Un entero llamado beta
- Un entero llamado jugador_max que contiene el número reservado para el jugador max
- Un entero llamado jugador_min que contiene el número reservado para el jugador min

Implementa la estrategia del algoritmo Min-Max con podas Alpha-Beta, a partir del estado almacenado en N.

diferencia_estados(tEstado anterior, tEstado nuevo, int * x, int * y)

Parámetros:

- Un estado de la partida llamado anterior
- Un estado de la partida llamado nuevo
- Un puntero a un entero llamado x que representa una fila de la grilla de juego
- Un puntero a un entero llamado y que representa una columna de la grilla de juego

Computa la diferencia existente entre los estados anterior y nuevo.

Se asume que entre ambos existe sólo una posición en el que la ficha del estado anterior y nuevo difiere.

La posición en la que los estados difieren, es retornada en los parámetros *x e *y.

Modo de Ejecución

Compilación: Al momento de compilar el programa se deben tener dentro del proyecto los siguientes archivos .h “lista.h”, “arbol.h”, “partida.h”, “ia.h” y los siguientes archivos .c “lista.c”, “arbol.c”, “partida.c”, “ia.c” y “main.c”

Código fuente principal: El código contenido en “main.c” controla la interacción del usuario con el juego a través de la consola y utiliza las funciones implementadas en los otros módulos para llevar a cabo la ejecución del juego.

Relación entre módulos / flujo de ejecución: Como se mencionó anteriormente el TDAPartida es el encargado de inicializar, controlar y actualizar el estado de la partida en juego mientras que el TDAIA define el comportamiento de la computadora al momento de decidir el próximo movimiento.

Al comenzar la partida se creará una instancia de tPartida reservando la memoria necesaria para sus atributos utilizando **nueva_partida**.

Si es el turno de un jugador, se deberá hacer una llamada a tPartida para que verifique que el movimiento realizado por el jugador es válido o no y actualice el estado de la partida si esta terminó. Todo esto con la función **nuevo_movimiento**.

Si es el turno de la computadora, se deberá crear un nuevo árbol de búsqueda con **crear_busqueda_adversaria** que tendrá como raíz el estado actual del tablero. Luego de ejecutar la función, el algoritmo minmax habrá generado un arbol de busqueda con los movimientos más eficientes a realizar por la computadora y para seleccionar el más conveniente utilizaremos la función **proximo_movimiento**. Esta última función elegirá el movimiento más conveniente entre los sucesores del estado actual del juego y retornará dicha elección, la cual será utilizada para realizar un movimiento en la grilla de juego usando la función **nuevo_movimiento**.

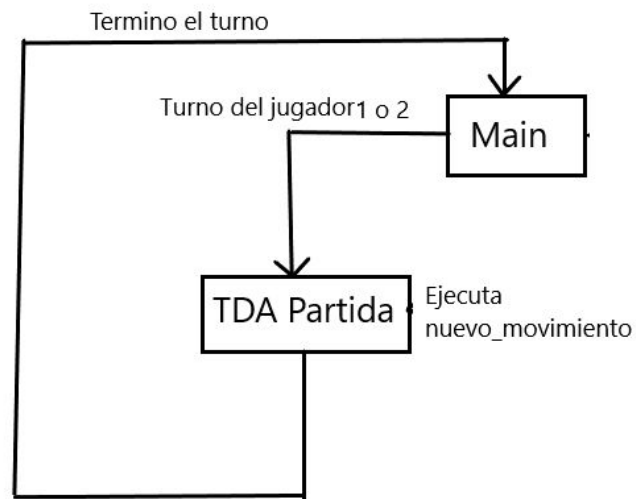
Flujo de ejecución: Usuario vs Usuario

Figura 1: Flujo de ejecución Usuario vs Usuario

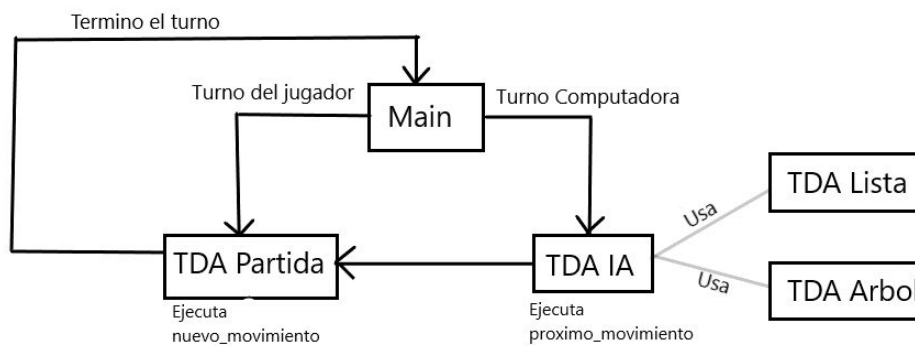
Flujo de ejecución: Usuario vs IA

Figura 2: Flujo de ejecución Usuario vs IA

Flujo de ejecución: IA vs IA

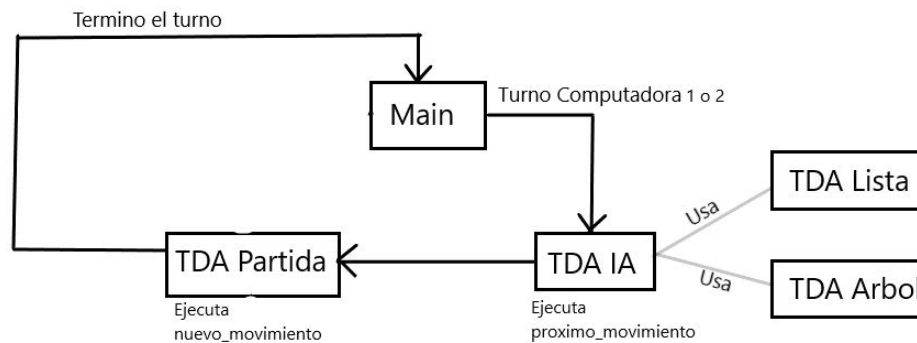


Figura 3: Flujo de ejecución IA vs IA

Modo de Uso

Elegir modo de juego: Al iniciar la aplicación se le presentará la siguiente situación

```
Elige un modo de juego
1. Usuario vs Usuario
2. Usuario vs Computadora
3. Computadora vs Computadora
4. Salir
```

Figura 4: Pantalla principal

Se deberá elegir un modo de juego utilizando los números 1, 2 y 3 o se podrá terminar la ejecución ingresando el número 4.

Una vez elegido un modo de juego se deberá ingresar el nombre de cada jugador(sin espacios) como en la siguiente imagen:

```
Elija un modo de juego
1. Usuario vs Usuario
2. Usuario vs Computadora
3. Computadora vs Computadora
4. Salir
1
Ingrese el nombre del jugador 1
Pedro
Ingrese el nombre del jugador 2
-
```

Figura 5: Ingreso de nombres de los jugadores

Por último se deberá elegir cual jugador tendrá el primer turno del juego:

```
Elija un modo de juego
1. Usuario vs Usuario
2. Usuario vs Computadora
3. Computadora vs Computadora
4. Salir
1
Ingrese el nombre del jugador 1
Pedro
Ingrese el nombre del jugador 2
Juan
Elija a continuacion el jugador que comenzara la partida
1. Pedro
2. Juan
3. Jugador al azar
```

Figura 6: Selección de primer turno

La opción 1 le otorga el primer turno al jugador 1, la opción 2 le otorga el primer turno al jugador 2 y la opción 3 elegirá de manera aleatoria un jugador y le otorgará el primer turno. Cada una se elige ingresando por teclado el número correspondiente a la opción.

Movimientos: Al ser nuestro turno debemos ingresar por teclado el movimiento que queremos hacer de la siguiente manera:

```

Ingrese el nombre del jugador 1
Pedro
Ingrese el nombre del jugador 2
Juan
Elija a continuacion el jugador que comenzara la partida
1. Pedro
2. Juan
3. Jugador al azar
3

      0   1   2
0  |  |  |  |
   -----
1  |  |  |  |
   -----
2  |  |  |  |
   -----

Es el turno del jugador Juan

Ingrese la coordenada horizontal del siguiente movimiento
2
Ingrese la coordenada vertical del siguiente movimiento

```

Figura 7: Ingreso de movimientos.

Los movimientos se realizan ingresando las coordenadas de la grilla de juego donde queremos poner una ficha. Dichas coordenadas están especificadas en el dibujo del tablero (que puede verse en la figura 7). Las coordenadas serán pares de valores (h,v) donde h es una fila del tablero y v es una columna del tablero. Cada fila y columna está numerada con dígitos del 0 al 2 inclusive. A modo de ejemplo colocaremos una ficha en la coordenada (2, 1) del tablero (es decir, fila 2, columna 1):

```
1  |  |  |  |
2  |  |  |  |
   -----

Es el turno del jugador Juan

Ingrese la coordenada horizontal del siguiente movimiento
2
Ingrese la coordenada vertical del siguiente movimiento
1

0  |  |  |  |
1  |  |  |  |
2  |  |x|  |
   -----
```

Figura 8: Ejemplo.

Si le corresponde a la computadora realizar un movimiento, este será realizado automáticamente sin intervención del usuario.

Finalizar Partida: Al terminar la partida aparecerá en la consola el resultado del juego y se retornará al menú principal.

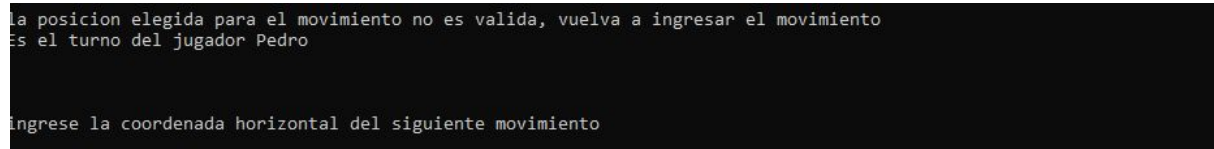
Conclusiones/Otros

Representación de las fichas dentro del programa:

Por decisión de los programadores, la ficha correspondiente al jugador 1 será representada por el número 0 (la cual se verá en el tablero como un círculo) y la ficha correspondiente al jugador 2 será representada por el número 1 (la cual se verá en el tablero como una equis). Un espacio vacío en el tablero será representado por el número 2.

Situaciones Particulares en la ejecución del programa:

1. En caso de ingresar un número de columna o fila inválido (si el numero ingresado es menor a cero o mayor a 2, o si elegimos una coordenada del tablero que ya tiene una ficha), aparecerá un cartel de error y deberemos ingresar una nueva jugada.



```
la posicion elegida para el movimiento no es valida, vuelva a ingresar el movimiento
es el turno del jugador Pedro

ingrese la coordenada horizontal del siguiente movimiento
```

Figura 9: El movimiento ingresado es inválido.

2. El nombre más largo del mundo está formado por 41 caracteres sin espacios por lo que el programa no contempla el caso en el que el nombre de un jugador exceda esa longitud.
3. En caso de no elegir una opción válida en el menú principal, se volverá a imprimir por pantalla las opciones de este hasta que se elija una opción entre las disponibles.
4. En caso de ingresar un número distinto a 3, 2 o 1 al momento de elegir el jugador que tendrá el primer turno de la partida, se elegirá un jugador al azar para ocupar dicho turno.