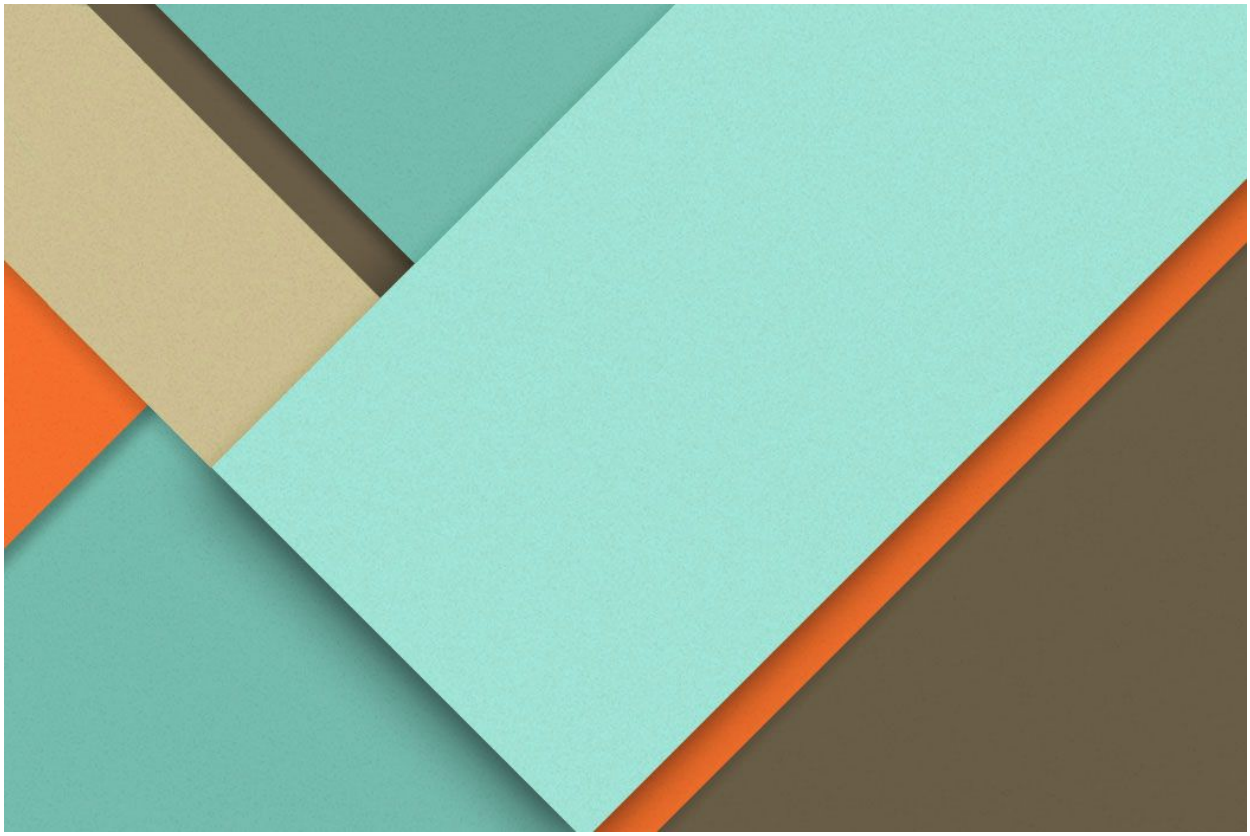


TA-TE-TI Inteligente.

Matias Pippig-Santiago Rosales Guinzburg
118548-106460

.

TDA Lista

Estructura utilizada para representar la lista.

Se utiliza el modelo de representación de una lista simplemente enlazada con celda centinela utilizando el concepto de posición indirecta.

Operaciones del TDA:

- **void crear_lista(tLista * l);**
  - **Uso:** Inicializa una lista vacía.
  - **Como:** Guardo espacio en memoria para una variable de estructura celda y asignó a su elemento y puntero su puntero a siguiente posición como nulo, creando así la celda centinela.

- **void l_insertar(tLista l, tPosicion p, tElemento e);**
  - **Uso:** Inserta en la lista 'l', y en la posición 'p' una posición con elemento 'e'.
  - **Como:** Guardo espacio en memoria para una variable de estructura celda y asignó a su elemento el elemento 'e' y a su siguiente el puntero que tiene la posición 'p', luego actualizo el siguiente de la posición 'p' con la reciente creada posición.

- **void l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento));**
  - **Uso:** Elimina de la lista 'l' la posición 'p' utilizando una función pasada por parámetro para destruir el elemento contenido en la posición a eliminar.
  - **Cómo:** Actualizo los punteros a la siguiente posición de la posición anterior a la posición 'p' con el puntero a la siguiente posición de la posición a eliminar, luego destruyó el elemento con la función pasada por parámetro, asigno NULL a los punteros en la posición a eliminar y libero la memoria asignada a la posición a eliminar.

- **void l_destruir(tLista* l, void (*fEliminar)(tElemento));**
  - **Uso:** Destruye la lista 'l' utilizando la función pasada por parámetro para destruir los elementos contenidos en sus posiciones.
  - **Como:** Recorro la lista y mientras la recorro eliminó los elementos de las posiciones con la función pasada por parámetro, asigno NULL a los punteros en la posición actual y libero el espacio en memoria ocupado por dicha posición.

- **tElemento l_recuperar(tLista l, tPosicion p);**
  - **Uso:**Recupera el elemento de la posición ‘p’ de la lista ‘l’.
  - **Cómo:**Retorno el puntero del elemento de la posición ‘p’.

- **tPosicion l_primera(tLista l);**
  - **Uso:**Retorna la primer posición de la lista ‘l’.
  - **Cómo:**Cómo es posición indirecta simplemente retorna la posición del centinela.

- **tPosicion l_siguiente(tLista l, tPosicion p):**
  - **Uso:**Retorna la siguiente posición de la posición ‘p’ de la lista ‘l’.
  - **Cómo:**Retorna el puntero a la siguiente posición encontrado en la posición ‘p’.

- **tPosicion l_anterior(tLista l, tPosicion p);**
  - **Uso:**Retorna la posición anterior a la posición ‘p’ de la lista ‘l’.
  - **Cómo:**Recorro la lista hasta que encontrar a la posición anterior a la posición ‘p’ y la retorno.

- **tPosicion l_ultima(tLista l);**
  - **Uso:**Retorna la última posición de la lista ‘l’.
  - **Cómo:**Recorro la lista hasta encontrar la última posición de la lista ‘l’ y la retorno.

- **tPosicion l_fin(tLista l);**
  - **Uso:**Retorna la posición fin de la lista ‘l’.
  - **Cómo:**Recorro la lista hasta encontrar la posición fin de la lista ‘l’ y la retorno.

- **int l_longitud(tLista l);**
  - **Uso:**Retorna la longitud de la lista ‘l’.
  - **Cómo:**Recorre la lista ‘l’ hasta recorrer todas su posiciones y retorna la cantidad de posiciones.

TDA Arbol

Estructura utilizada para representar el árbol.

Se utiliza el modelo de representación de nodos enlazados, cada nodo mantiene una referencia a un nodo considerado padre del mismo dentro del árbol mediante un puntero, un puntero referenciando a una lista de nodos que representa los nodos hijos del mismo y un puntero a un elemento genérico que representa el rótulo de dicho nodo.

Operaciones del TDA:

- **void crear_arbol(tArbol * a);**
  - **Uso:**Inicializa un arbol vacio.
  - **Cómo:**Guardo espacio en memoria para una variable de estructura arbol y asigno a su raiz NULL.

- **void crear_raiz(tArbol a, tElemento e);**
  - **Uso:**Crea la raiz del arbol ‘a’ con rótulo ‘e’.
  - **Cómo:**Guardo espacio en memoria para una variable de estructura nodo, inicializo su lista y asignó a su padre NULL.

- **tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e);**
  - **Uso:**Inserta en el árbol ‘a’ un nodo con rótulo ‘e’ con padre el nodo ‘np’ y hermano el nodo ‘nh’.
  - **Cómo:**Guardo espacio en memoria para una variable de estructura nodo. le asignó el elemento ‘e’ ,le asignó a su padre el nodo ‘np’ y si el nodo ‘nh’ pasado por parámetro es NULL lo inserto al final de la lista de hijos del nodo ‘np’ con el nodo creado como elemento, sino recorro la lista de hijos de ‘np’ hasta encontrar la posición con elemento ‘nh’ e insertó una nueva posición con el nuevo nodo con elemento, luego retorno el nuevo nodo.

- **void a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento));**
  - **Uso:**Elimina del árbol ‘a’ el nodo ‘n’ y utiliza la función pasada por parámetro para eliminar su elemento.
  - **Cómo:**Si el nodo es raíz y no tiene hijos, elimino la raíz, si tiene un hijo elimino la raíz y el hijo será la nueva raíz; si no es raíz, inserto los hijos del nodo a eliminar en la lista del padre en la posición anterior al nodo a eliminar y luego elimino el nodo.

- **void a_destruir(tArbol * a, void (*fEliminar)(tElemento));**
  - **Uso:**Destruye el árbol ‘a’ utilizando la función pasada por parámetro para destruir los elementos de los nodos de esta.
  - **Cómo:**Utilizo la función auxiliar ‘auxDestruir’ pasando como parámetro la lista de hijos de la raíz, luego elimino el elemento de la raíz y libero el espacio de memoria que ocupa la raíz.

- **tElemento a_recuperar(tArbol a, tNodo n);**
  - **Uso:**Recupera el elemento del nodo ‘n’ del árbol ‘a’.
  - **Cómo:**Retorno el elemento del nodo ‘n’.

- **tNodo a_raiz(tArbol a);**
  - **Uso:**Recupera la raiz del arbol ‘a’.
  - **Cómo:**Retorno el nodo raiz del arbol ‘a’.

- **tLista a_hijos(tArbol a, tNodo n);**
  - **Uso:**Recupera la lista de hijos del nodo ‘n’ del árbol ‘a’.
  - **Cómo:**Retorno la lista de hijos del nodo ‘n’.

- **void a_sub_arbol(tArbol a, tNodo n, tArbol * sa);**
  - **Uso:**Crea un árbol ‘sa’, con raíz ‘n’, del árbol ‘a’.
  - **Cómo:**Luego de inicializar el arbol ‘sa’, asignó a el nodo ‘n’ como raíz; Si el padre del nodo ‘n’ no es NULL, busco su posición en la lista de su padre y lo eliminó y asignó a su padre como NULL, sino asignó a su padre como NULL.

Operaciones auxiliares:

- **void auxDestruir(tElemento elemento);**
  - **Uso:**Destruye un árbol .
  - **Cómo:**Mediante un recorrido preorden se van eliminando todos los elementos del árbol.

Pseudocódigo Aux Destruir:

Tomo el elemento que me dan

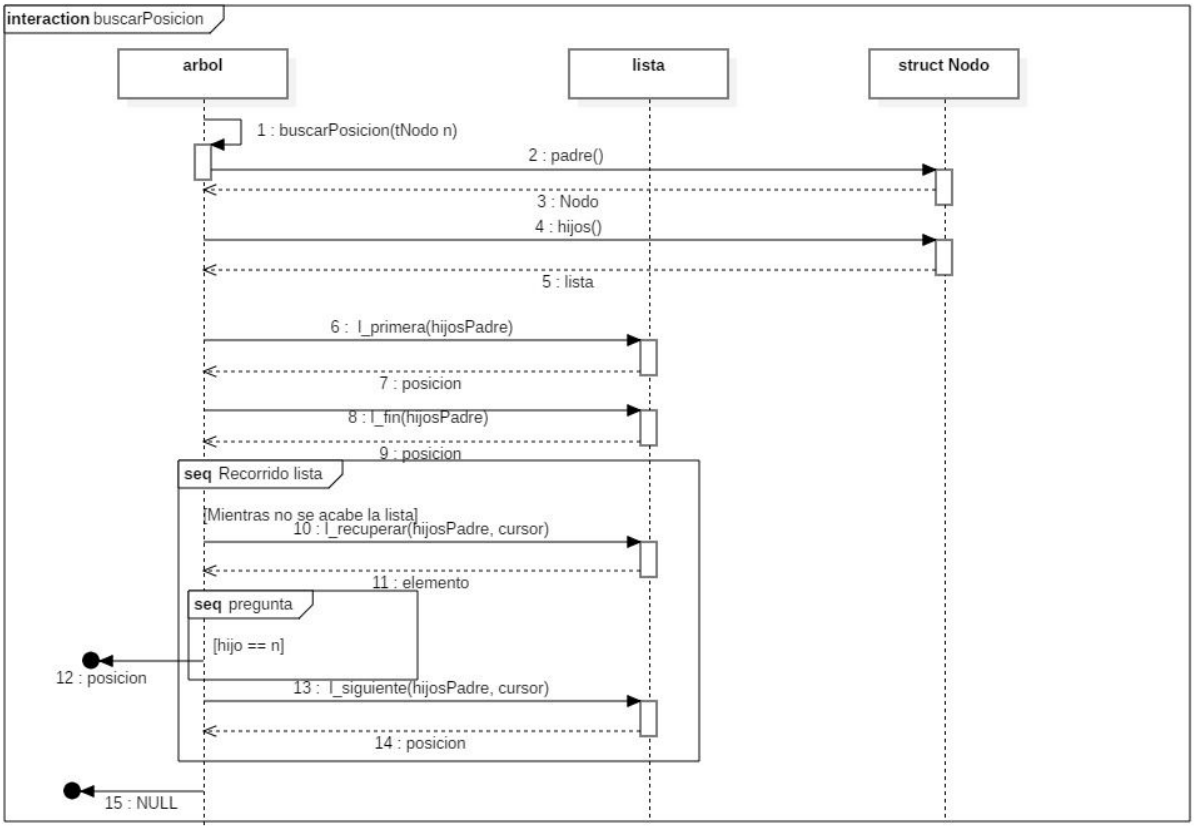
para cada hijo que tengo el elemento

Llamó Aux Destruir con el hijo

Luego elimino el elemento que me dieron

➔ **tPosicion buscarPosicion(tNodo n);**

- **Uso:**Encuentro la posición del nodo
- **Como:** A través de recorrer la lista de hermanos del nodo busco el mismo y devuelvo la posición que tiene.



Juego TA-TE-TI

Introducción:

Lo abordado en este proyecto , es el desarrollo de un TA-TE-TI que permita jugar entre dos personas y además permite jugar versus una inteligencia artificial.

Además se permite ver una partida entre dos inteligencias artificiales.

Módulos desarrollados:

- **Partida:**
  - **Funcionalidad:** Módulo que permite la creación de una partida, generar un nuevo movimiento en la partida, y finalizar.

    Se encuentra en el archivo fuente partida.c y partida.h
  - **Responsabilidad:** Mantiene el estado de la partida, sabiendo si está terminó con la victoria de algún jugador, empate o si no termino.

    También comprueba que los movimientos a realizar sean válidos.
  - **Operaciones:**
    - **void nueva_partida (tPartida * p, int modo_partida, int comienza, char * j1_nombre, char * j2_nombre)**
      - **Uso:**

        Inicializa una nueva partida, indicando:

        Modo de partida (Usuario vs. Usuario, Usuario vs. Agente IA o Agente IA vs Agente IA), jugador que comienza la partida (Jugador 1, Jugador 2, o elección al azar), nombre que representa al Jugador 1, nombre que representa al Jugador 2.
    - **int nuevo_movimiento(tPartida p, int mov_x, int mov_y)**
      - **Uso:**

        Actualiza, si corresponde, el estado de la partida considerando que el jugador al que le corresponde jugar, decide hacerlo en la posición indicada (X,Y).

        En caso de que el movimiento a dicha posición sea posible, retorna PART_MOVIMIENTO_OK; en caso contrario, retorna PART_MOVIMIENTO_ERROR.

Las posiciones (X,Y) deben corresponderse al rango [0-2];

X representa el número de fila, mientras Y el número de columna.

- **void finalizar_partida(tPartida * p)**
  - **Uso:**

    Finaliza la partida referenciada por P, liberando toda la memoria utilizada.

- **Operaciones Auxiliares:**
  - **int ganoP(tTablero t, int ficha)**
    - **Uso:**

      calcula si gana la ficha pasada por parametro, mirando filas columnas y diagonales buscando la ficha tres veces seguidas.

➔ **Inteligencia Artificial:**

- **Funcionalidad:** Módulo que permite ,crear una inteligencia artificial ,encontrar el próximo movimiento que debe realizar, y destruir la inteligencia.
- **Operaciones:**
  - **void crear_busqueda_adversaria(tBusquedaAdversaria * b, tPartida p);**
    - **Uso:**

      Inicializa la estructura correspondiente a una búsqueda adversaria, a partir del estado actual de la partida parametrizada.

      Los datos del tablero de la partida parametrizada son clonados, por lo que P no se ve modificada.

      Una vez esto, se genera el árbol de búsqueda adversaria siguiendo el algoritmo Min-Max con podas Alpha-Beta.

  - **void proximo_movimiento(tBusquedaAdversaria b, int * x, int * y)**
    - **Uso:**

      Computa y retorna el próximo movimiento a realizar por el jugador MAX.

      Para esto, se tiene en cuenta el árbol creado por el algoritmo de búsqueda adversaria Min-max con podas Alpha-Beta.

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.

- **void destruir_busqueda_adversaria(tBusquedaAdversaria * b)**
  - **Uso:**

    Libera el espacio asociado a la estructura correspondiente para la búsqueda adversaria.

◆ **Operaciones Auxiliares:**

- **void ejecutar_min_max(tBusquedaAdversaria b)**
  - **Uso:**

    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 el árbol almacenado en B.

- **void crear_sucesores_min_max(tArbol a, tNodo n, int es_max, int alpha, int beta, int jugador_max, int jugador_min)**
  - **Uso:**

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

    ES_MAX indica si N representa un nodo MAX en el árbol de búsqueda adversaria.

    ALPHA y BETA indican sendos valores correspondientes a los nodos ancestros a N en el árbol de búsqueda A.

    JUGADOR_MAX y JUGADOR_MIN indican las fichas con las que juegan los respectivos jugadores.

- **int valor_utilidad(tEstado e, int jugador_max)**
  - **Uso:**

    Calcula el valor utilidad de un estado de partida respecto del jugador max.

    Retorna IA_GANA_MAX si gana max, IA_PIERDE_MAX si pierde y IA_EMPATA_MAX si hay empate.

- **tLista estados_sucesores(tEstado e, int ficha_jugador)**
  - **Uso:**

    Computa y retorna una lista con aquellos estados que representan estados sucesores al estado E.

    Un estado sucesor corresponde a la clonación del estado E, junto con la incorporación de un nuevo movimiento

    realizado por el jugador cuya ficha es FICHA_JUGADOR por sobre una posición que se encuentra libre en el estado E.

    La lista de estados sucesores se debe ordenar de forma aleatoria.

- **tEstado clonar_estado(tEstado e)**
  - **Uso:**

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

    Para esto copia en el estado a retornar los valores actuales de la grilla del estado E, como su valor

    de utilidad.

- **void diferencia_estados(tEstado anterior, tEstado nuevo, int * x, int * y)**
  - **Uso:**

    Computa la diferencia existente entre dos estados.

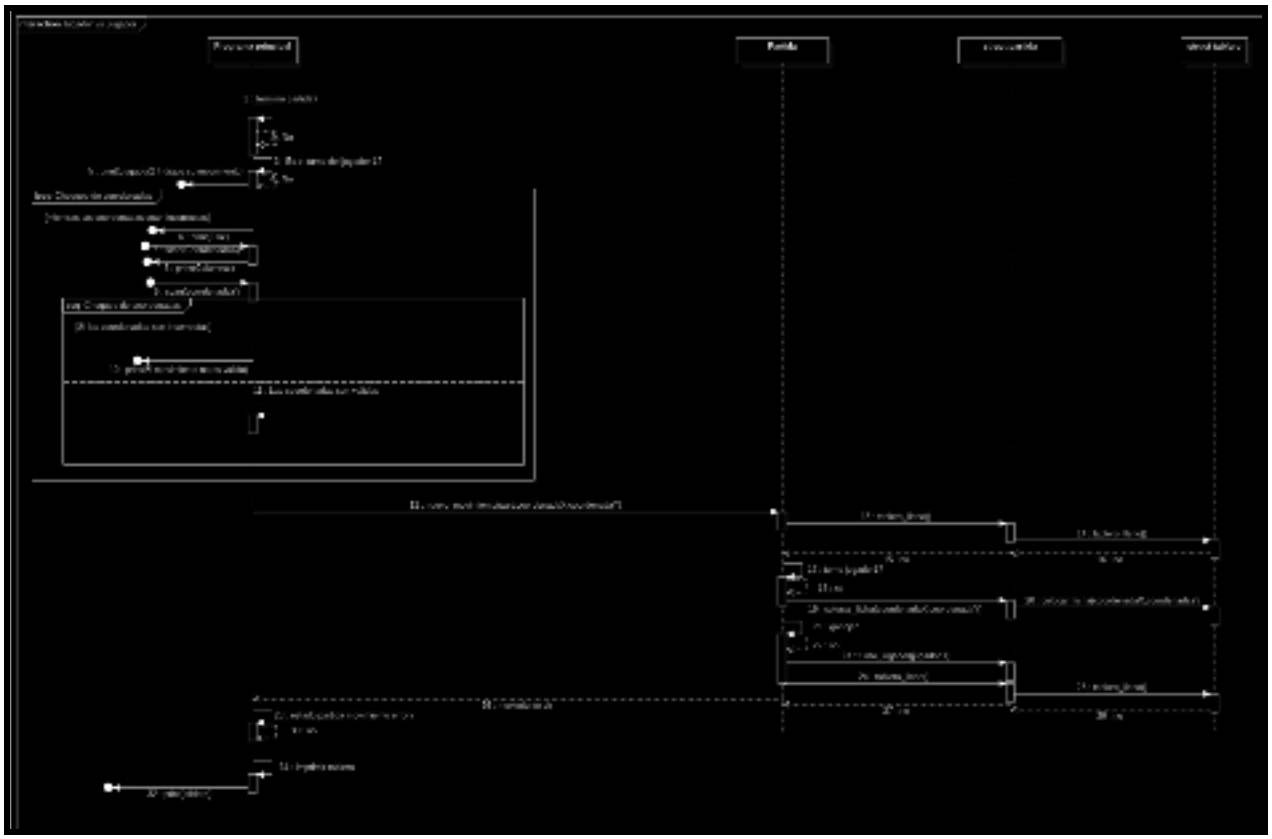
    Se asume que entre ambos existe sólo una posición en el que la ficha del estado anterior y nuevo difiere.

    La posición en la que los estados difiere, es retornada en los parámetros *X e *Y.

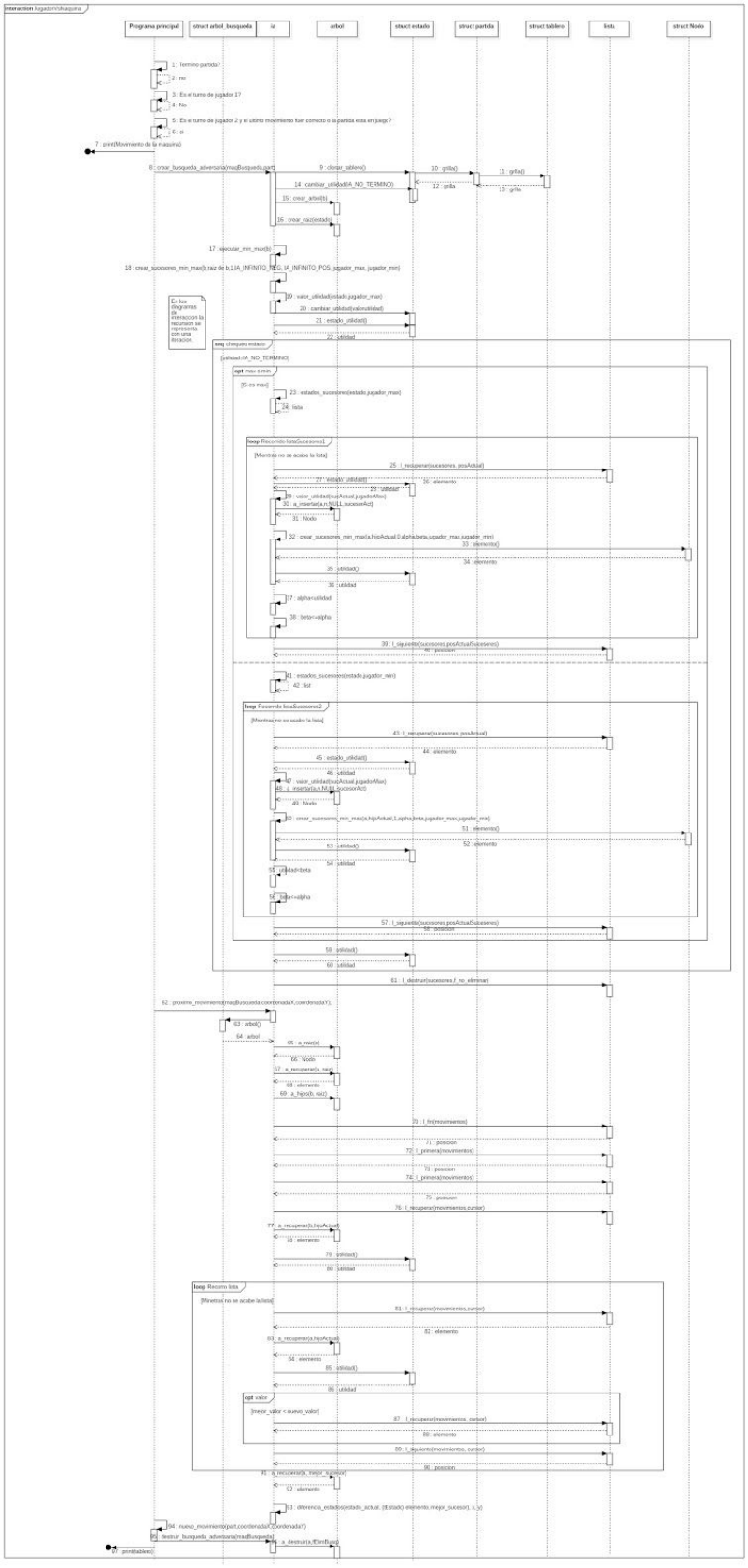
Relación entre módulos\Flujo de ejecución:

Diagramas de interacción :

➢ Situación, la partida ya terminó de ejecutar el 3er movimiento y es el turno del jugador 2.



➢ Situación, la partida la partida ya terminó de ejecutar el 3er movimiento y es el turno de la máquina.


interaction JugadorVsMaquina
Programa principal
struct arbol_busqueda
ia
arbol
struct estado
struct partida
struct tablero
lista
struct Nodo
1 : Termino partida?
2 : no
3 : Es el turno de jugador 1?
4 : No
5 : Es el turno de jugador 2 y el ultimo movimiento fuer correcto o la partida esta en juego?
6 : si
7 : print(Movimiento de la maquina)
8 : crear_busqueda_adversaria(maxBusqueda,sup)
9 : clonar_tablero()
10 : grilla()
11 : grilla()
12 : grilla
13 : grilla
14 : cambiar_utilidad(IA_NO_TERMINO)
15 : crear_arbol(ib)
16 : crear_raiz(estado)
17 : ejecutar_min_max(ib)
18 : crear_sucesores_min_max(b.raiz de b,2,IA_INFINITO_NEG,IA_INFINITO_POS,jugador_max,jugador_min)
19 : valor_utilidad(estado,jugador_max)
20 : cambiar_utilidad(valorutilidad)
21 : estado_utilidad()
22 : utilidad
En los diagramas de interaccion se representa con una iteracion.
seq chequeo estado
[utilidad!=IA_NO_TERMINO]
opt max o min
[Si es max]
23 : estados_sucesores(estado,jugador_max)
24 : lista
loop Recorrido listaSucesores1
[Mientras no se acabe la lista]
25 : l_recuperar(sucesores, posActual)
26 : elemento
27 : estado_utilidad()
28 : utilidad
29 : valor_utilidad(hijoActual,jugadorMax)
30 : a_insertar(a,n,NULL,sucesorAct)
31 : Nodo
32 : crear_sucesores_min_max(a,hijoActual,0,alpha,beta,jugador_max,jugador_min)
33 : elemento()
34 : elemento
35 : utilidad()
36 : utilidad
37 : alpha=utilidad
38 : beta=+alpha
39 : l_siguiente(sucesores,posActualSucesores)
40 : posicion
41 : estados_sucesores(estado,jugador_min)
42 : lst
loop Recorrido listaSucesores2
[Mientras no se acabe la lista]
43 : l_recuperar(sucesores, posActual)
44 : elemento
45 : estado_utilidad()
46 : utilidad
47 : valor_utilidad(hijoActual,jugadorMax)
48 : a_insertar(a,n,NULL,sucesorAct)
49 : Nodo
50 : crear_sucesores_min_max(a,hijoActual,1,alpha,beta,jugador_max,jugador_min)
51 : elemento()
52 : elemento
53 : utilidad()
54 : utilidad
55 : utilidad<beta
56 : beta=+alpha
57 : l_siguiente(sucesores,posActualSucesores)
58 : posicion
59 : utilidad()
60 : utilidad
61 : l_destruir(sucesores,f_no_eliminar)
62 : proximo_movimiento(maxBusqueda,coordenadaX,coordenadaY);
63 : arbol()
64 : arbol
65 : a_raiz(a)
66 : Nodo
67 : a_recuperar(a,nai)
68 : elemento
69 : a_hacer(b,raiz)
70 : l_fin(movimientos)
71 : posicion
72 : l_primera(movimientos)
73 : posicion
74 : l_primera(movimientos)
75 : posicion
76 : l_recupera(movimientos,cursor)
77 : a_recuperar(b,hijoActual)
78 : elemento
79 : utilidad()
80 : utilidad
loop Recorro lista
[Mientras no se acabe la lista]
81 : l_recupera(movimientos,cursor)
82 : elemento
83 : a_recupera(a,hijoActual)
84 : elemento
85 : utilidad()
86 : utilidad
opt valor
[mejor_valor < nuevo_valor]
87 : l_recuperar(movimientos, cursor)
88 : elemento
89 : l_siguiente(movimientos, cursor)
90 : posicion
91 : a_recupera(a, mejor_sucesor)
92 : elemento
93 : diferencia_estados(estado_actual, (Estado) elemento, mejor_sucesor), x, y)
94 : nuevo_movimiento(port,coordenadaX,coordenadaY)
95 : destruir_busqueda_adversaria(maxBusqueda)
96 : a_destruir(a,IElimBool)
97 : print(tablero)

Modo de ejecución

Compilación:

Se necesita crear una libreria dinamica que contenga los archivos fuente lista.c, arbol.c, ia.c, partida.c y luego generar un archivo .exe con la libreria dinamica y el archivo fuente ProgamaPrincipal.c

y se necesita que la libreria se encuntre junto con el codigo fuente principal

Codigo fuente principal:

El codigo fuente es programa.exe

Modo de uso:

1. Elijo el modo de juego ingresando un número del 1 al 3.
2. Luego se pide el nombre del jugador 1 independientemente del modo.
3. Elijo quien empieza a jugar con un número del 1 al 3.
4. Cuando sea el turno del jugador humano deberá elegir una fila del 1 al 3
5. Luego deberá elegir una columna del 1 al 3 y se verá el movimiento realizado
6. En caso de colocar filas o columnas invalidas se mostrara un error y se pedirá que juegue nuevamente
7. Luego verá cómo juega la máquina.
8. Repetir el proceso desde el paso 4 hasta llenar el tablero o que algún jugador gane.
9. Luego de terminada la partida se mostrará el resultado y se deberá escribir cualquier cosa y apretar enter para terminar la ejecución.

Conclusiones

Si en el paso 1 no se pasa un int falla

Si en el paso 3 no se pasa un int falla

si en el paso 4 o 5 no se pasa un int falla