

Universidad Nacional
ARTURO JAURETCHE

Trabajo Práctico Final

Instituto de Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Asignatura: Complejidad Temporal, Estructuras de Datos y Algoritmos.

Alumno: Salinas, Matías Nahuel.

DNI: 35.189.786

Legajo: 46607.

Docente: Dr. Ing. Amet, Leonardo Javier.

El objetivo del presente informe es describir el funcionamiento de un juego de cartas de dos jugadores, de los cuales, uno de ellos funciona mediante una inteligencia artificial, la cual es implementada basada en árboles MiniMax. Para ello, resulta fundamental explicar en primer lugar en que consiste la lógica MiniMax y cuales son las reglas del juego.

Durante la cursada de Complejidad Temporal, Estructuras de Datos y Algoritmos, hemos estudiado diferentes formas de solucionar problemáticas que consisten en problemas de búsqueda en estructuras de datos, tales como árboles o grafos. La teoría de juegos proporciona soluciones a problemas utilizando los métodos de búsqueda estudiados.

MiniMax es un método de toma de decisiones, el cual tiene como finalidad minimizar las pérdidas en juegos con oponente y con información perfecta, es decir, en un contexto en donde ambos jugadores conocen de antemano todos los movimientos posibles de su oponente y sus consecuencias, se supone que ambos jugadores optarán por una estrategia que minimice la pérdida máxima esperada. Su funcionamiento puede resumirse en “Tomar la mejor decisión para tí, teniendo en cuenta que tu oponente tomará la peor decisión para ti”. Para ello, se suele trabajar con una función que retorna valores positivos para indicar situaciones favorables para el jugador que hace uso del algoritmo y valores negativos para indicar situaciones favorables para el contrincante. Dicha función se conoce como Función Heurística.

Todos los estados posibles del juego se representan utilizando un árbol general, en el cual los nodos representan una situación del juego y un nivel contiene todas las situaciones posibles para ambos jugadores.

El juego de cartas con el cual se trabajó se desarrolla de la siguiente manera:

- 1) Se dispone de un mazo con doce cartas numeradas del uno al doce, que se reparten en cantidades iguales y de manera aleatoria entre dos jugadores.
- 2) El juego fija un límite máximo del cual los jugadores no pueden pasarse.
- 3) Los jugadores juegan una vez por turno y en cada uno se tiene que descartar una carta.
- 4) El descarte va formando un montículo cuyo valor es la suma de las cartas que lo integran. El montículo de descarte inicialmente está vacío y su valor es cero.
- 5) El jugador que incorpore la carta al montículo que haga que el valor del mismo supere el límite fijado es aquel que pierde el juego.

Se nos solicita encargarnos de la implementación del jugador con inteligencia artificial basado en árboles MiniMax. Para realizar dicha implementación, se utilizará el editor de código Visual Studio Code en su versión 1.46.1 para Ubuntu y el lenguaje de programación C#. A continuación, pasaremos a describir los detalles de implementación pertinentes.

Detalles de implementación:

El armado del árbol general que contiene todas los posibles estados del juego se hace mediante el método “inicializar” correspondiente a la clase “ComputerPlayer”. Además, el mismo método se encarga de asignar la función heurística correspondiente la cual permitirá determinar que jugador se encuentra en una situación favorable en cada nodo hoja. Una vez que el proceso de creación del árbol finaliza, este queda almacenado en el atributo estado de la clase “Game”.

El método “descartarUnaCarta” de la clase “ComputerPlayer” se encarga de determinar que carta resulta conveniente descartar dependiendo del estado actual del juego. Para ello, verifica cual es la función heurística de cada hijo del atributo “estado”.

El método “cartaDelOponente”, también correspondiente a la clase “ComputerPlayer”, recibe como parámetro una variable de tipo “int”, la cual representa una carta que se busca en los hijos del atributo “estado”, de esta forma se actualiza el estado del juego.

El usuario también cuenta con la posibilidad de realizar consultas durante la ejecución del juego. Para ello, se realizó la implementación de un método al que, de forma muy creativa y original, llamamos “consultas”. Dicho método recibe por parámetro una variable de tipo string, con la cual el método ejecuta el código correspondiente a cada consulta utilizando una estructura de datos de tipo “Switch Case”. Las consultas que el usuario puede realizar son las siguientes:

- a) Imprimir todos los posibles resultados.
- b) Ingresar jugadas para obtener resultados.
- c) Ingrese una profundidad para obtener las posibles jugadas.
- r) Reiniciar el juego.

Imprimir todos los posibles resultados:

Se recorre el árbol “estado” y se imprime por pantalla el valor correspondiente a cada nodo, permitiendo de esta forma visualizar todas las posibles jugadas que pueden realizarse a partir de ese punto. Cabe mencionar que con la implementación actual, el programa imprime los resultados en pantalla de forma tal que resulta difícil hacer un seguimiento detallado de cada jugada, debido a la considerable cantidad de datos que deben imprimirse. Poder mostrar todas las posibles jugadas de forma más legible para el usuario sería una de las posibles mejoras a realizar.

Ingresar jugadas para obtener resultados:

El jugador controlado por el humano procede a ingresar una secuencia de cartas por teclado, y se prosigue a buscar dicha secuencia en el árbol estado. A continuación, se verifica la función heurística de la última carta ingresada y se retorna su valor (1 gana la inteligencia artificial o -1 si gana el humano).

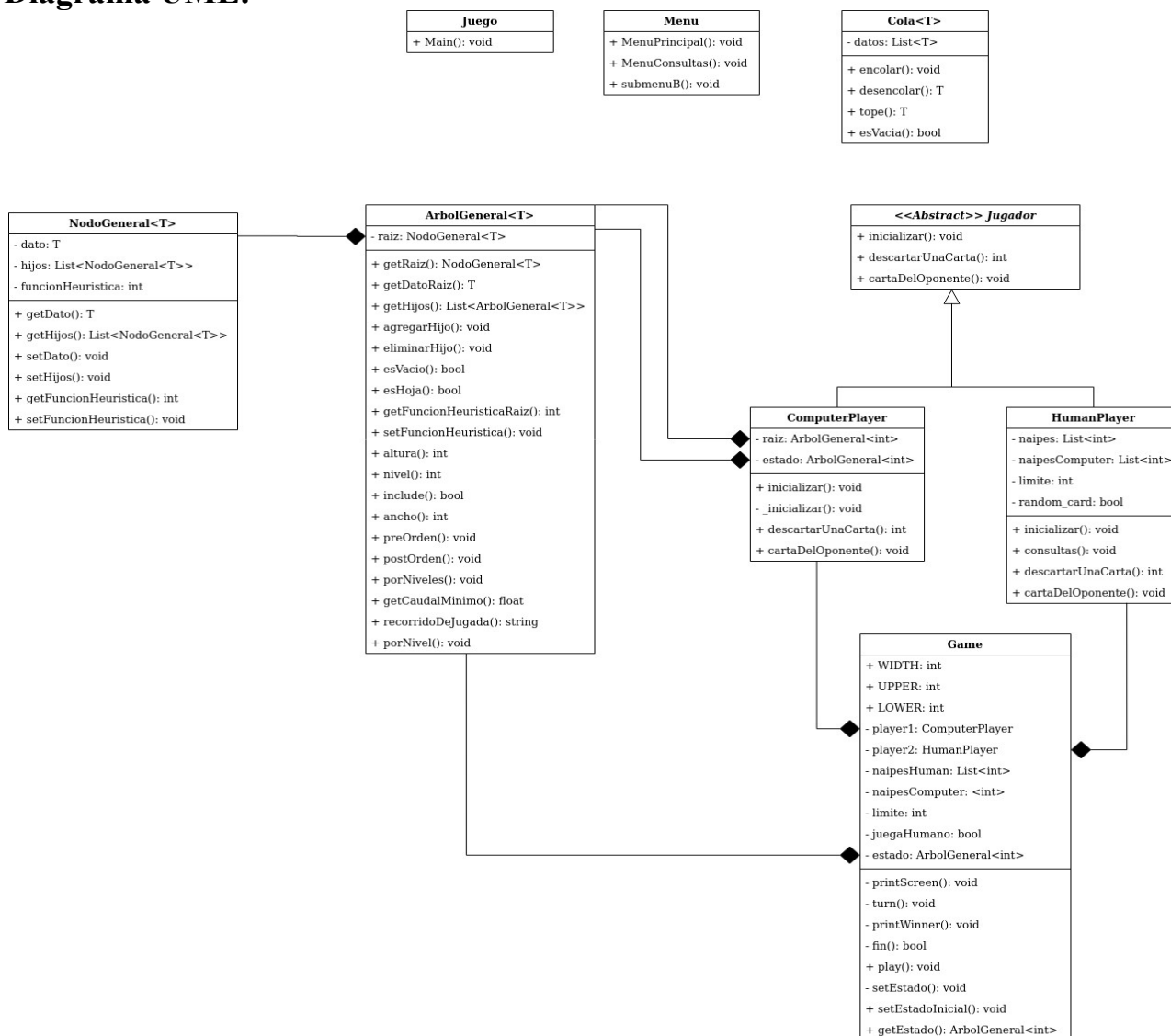
Ingrese una profundidad para obtener las posibles jugadas:

Se imprimen por pantalla los valores de todos los nodos en la profundidad ingresada por teclado (o en su defecto, se le informa al usuario que ingresó una profundidad inválida). Dicha profundidad se busca a partir del árbol “estado”.

Reiniciar el juego:

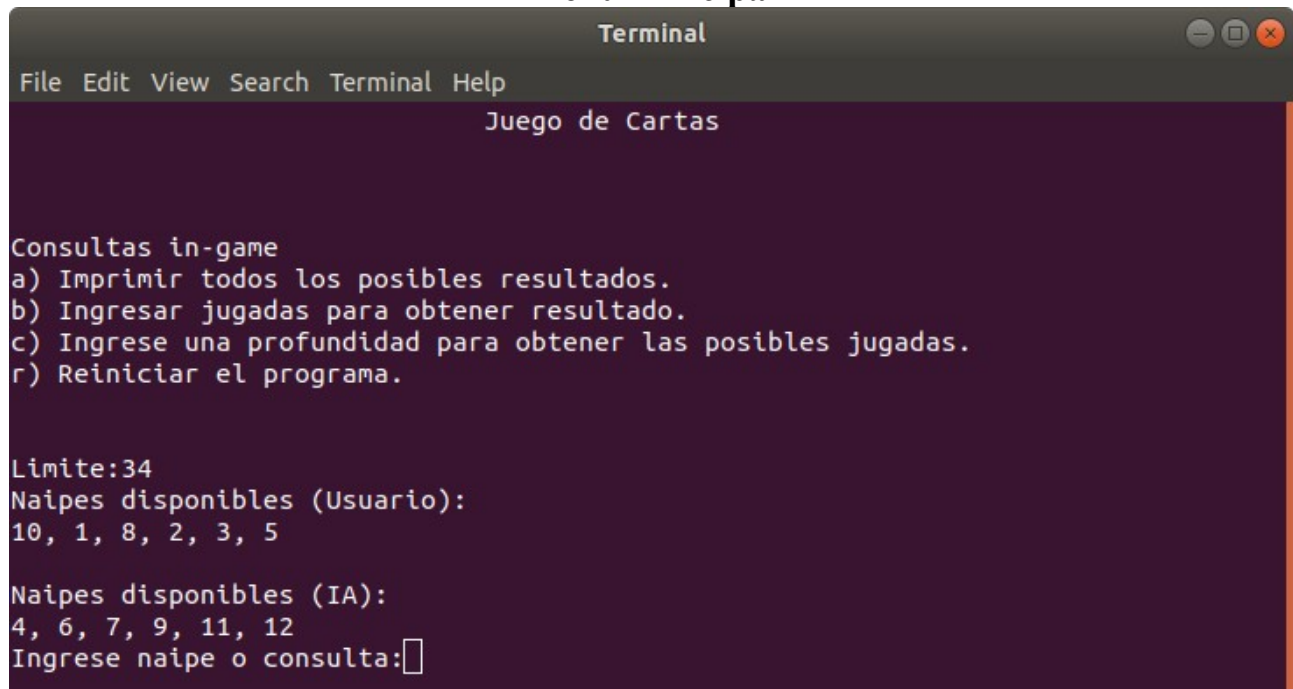
Se invoca al método “Main”, logrando de esta forma que se de comienzo a un nuevo juego.

Diagrama UML:



Capturas de pantalla con programa en funcionamiento:

Menú Principal



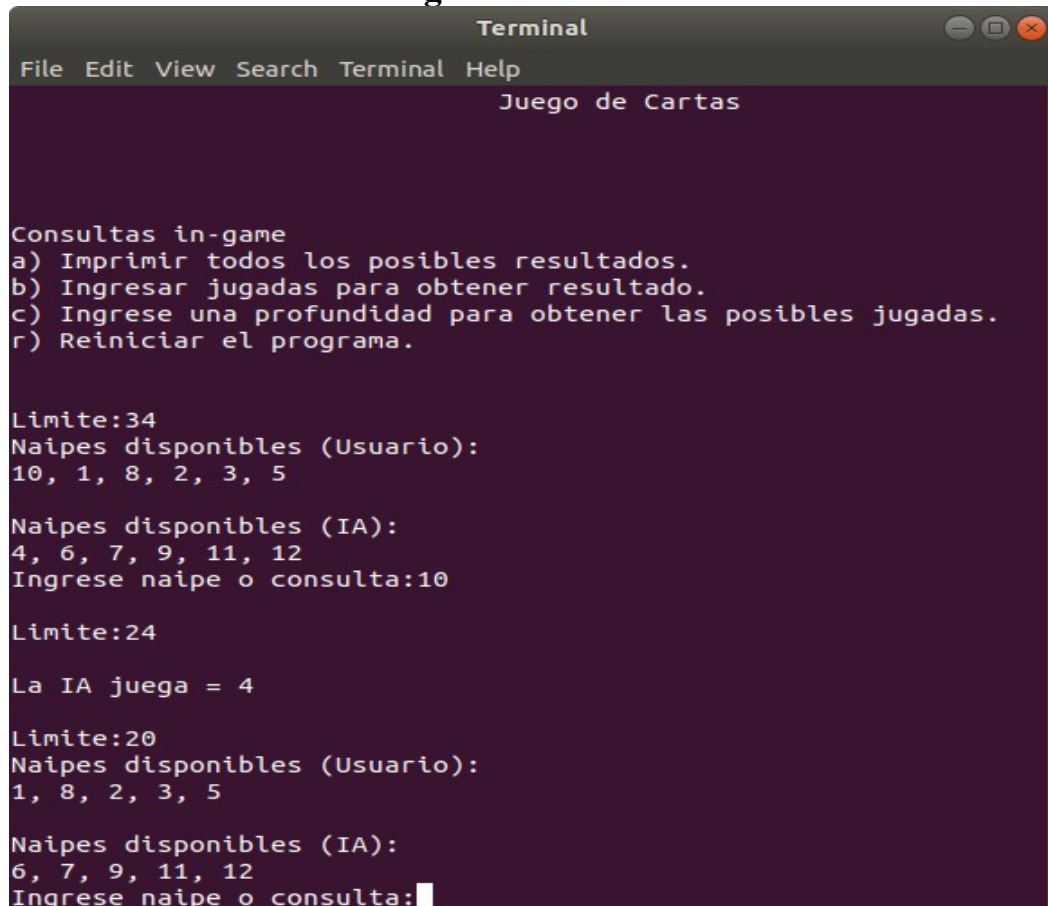
```
Terminal
File Edit View Search Terminal Help
Juego de Cartas

Consultas in-game
a) Imprimir todos los posibles resultados.
b) Ingresar jugadas para obtener resultado.
c) Ingrese una profundidad para obtener las posibles jugadas.
r) Reiniciar el programa.

Limite:34
Naipes disponibles (Usuario):
10, 1, 8, 2, 3, 5

Naipes disponibles (IA):
4, 6, 7, 9, 11, 12
Ingrese naipe o consulta:
```

Jugando una carta



```
Terminal
File Edit View Search Terminal Help
Juego de Cartas

Consultas in-game
a) Imprimir todos los posibles resultados.
b) Ingresar jugadas para obtener resultado.
c) Ingrese una profundidad para obtener las posibles jugadas.
r) Reiniciar el programa.

Limite:34
Naipes disponibles (Usuario):
10, 1, 8, 2, 3, 5

Naipes disponibles (IA):
4, 6, 7, 9, 11, 12
Ingrese naipe o consulta:10

Limite:24

La IA juega = 4

Limite:20
Naipes disponibles (Usuario):
1, 8, 2, 3, 5

Naipes disponibles (IA):
6, 7, 9, 11, 12
Ingrese naipe o consulta:
```

Imprimir todos los posibles resultados

(Debido a la considerable cantidad de resultados posibles, múltiples capturas de pantalla serían necesarias para poder ver todas las jugadas)

[illegible]

Distintas jugadas según profundidad ingresada

```

Terminal
File Edit View Search Terminal Help
Ingrese naípe o consulta:c

Inserte la profundidad (Menor o igual a 7) =1
(6,1)(7,1)(3,1)(2,1)(8,1)(12,1)
Ingrese naípe o consulta:c

Inserte la profundidad (Menor o igual a 7) =2
(1,1)(4,1)(5,1)(9,1)(10,1)(11,1)(1,1)(4,1)(5,1)(9,1)(10,1)(11,1)(1,1)(4,1)(5,1)(9,1)(10,1)(11,1)(1,1)(4,1)(5,1)(9,1)(10,1)(11,1)(1,1)(4,1)(5,1)(9,1)(10,1)(11,1)
Ingrese naípe o consulta:c

Inserte la profundidad (Menor o igual a 7) =3
(7,-1)(3,1)(2,1)(8,1)(12,1)(7,1)(3,1)(2,1)(8,1)(12,1)(7,1)(3,1)(2,1)(8,1)(12,1)(7,-1)(3,1)(2,1)(8,1)(12,1)(7,-1)(3,1)(2,1)(8,1)(12,1)(6,-1)(3,1)(2,1)(8,1)(12,1)(6,1)(3,1)(2,1)(8,1)(12,1)(6,1)(3,1)(2,1)(8,1)(12,1)(6,-1)(3,1)(2,1)(8,1)(12,1)(6,1)(3,1)(2,-1)(8,1)(12,1)(6,1)(7,1)(2,1)(8,1)(12,1)(6,1)(7,1)(2,1)(8,1)(12,1)(6,1)(7,1)(2,-1)(8,1)(12,1)(6,1)(7,1)(2,1)(8,1)(12,1)(6,1)(7,1)(3,1)(8,1)(12,1)(6,1)(7,1)(3,-1)(8,1)(12,1)(6,1)(7,1)(3,1)(8,1)(12,1)(6,1)(7,1)(3,1)(8,1)(12,1)(6,1)(7,-1)(3,-1)(8,1)(12,1)(6,1)(7,1)(3,1)(2,1)(12,1)(6,1)(7,1)(3,1)(2,1)(12,1)(6,1)(7,1)(3,1)(2,1)(12,1)(6,1)(7,1)(3,1)(2,1)(12,1)(6,1)(7,1)(3,1)(2,1)(8,1)(6,1)(7,1)(3,1)(2,1)(8,1)(6,0)(7,1)(3,1)(2,1)(8,1)(6,1)(7,1)(3,1)(2,1)(8,1)(6,1)(7,1)(3,1)(2,1)(8,1)
Ingrese naípe o consulta:

```

Posibles mejoras:

Como ya se mencionó con anterioridad, una posible (y muy necesaria) mejora sería buscar una forma de imprimir todas las jugadas posibles por pantalla para facilitarle al usuario su visualización. Adicionalmente, podría incluirse algún mecanismo que decida de manera aleatoria que jugador comienza el juego. Finalmente, sería ideal poder implementar una interfaz gráfica de manera tal de poder visualizar mejor el desarrollo del juego. Algunos sitios web, como por ejemplo visualgo.net (ctrl+click para abrir enlace) permiten visualizar paso a paso como se hace la inserción, eliminación y los distintos tipos de recorridos en árboles mientras se describe paso a paso la lógica del algoritmo. Lograr una implementación con estas características para este juego de cartas sería ideal (Aunque naturalmente, excede a los alcances de la asignatura).

Conclusiones:

Durante el desarrollo de este trabajo práctico, surgieron bastantes problemas, en su mayoría, pertinentes a la lógica de programación requerida para implementar los métodos de la inteligencia artificial (en particular, el método de inicialización con el cual se crea el árbol general). Dichos problemas permitieron al autor de este informe enfrentarse a un problema de considerable complejidad y trabajar en su capacidad de análisis para buscar la solución al mismo. Por otra parte, tanto la cursada de Complejidad Temporal, Estructuras de Datos y Algoritmos como la realización de este trabajo, dieron la posibilidad de aprender en profundidad acerca de las distintas estructuras de datos que son comunes a la mayoría de los lenguajes de programación, pudiendo de esta forma conocer las ventajas que estas ofrecen y desarrollar el criterio necesario para poder elegir la más adecuada según el problema que se presente. El desarrollo de este criterio resulta ser una habilidad fundamental para todo profesional que se desenvuelva en el área del desarrollo de software. Finalmente, pero no menos importante, se tuvo la oportunidad de poder aprender los conceptos básicos de GIT. Hoy en día, cualquier proyecto serio en un entorno profesional utiliza un controlador de versiones, por lo cual, aprender a utilizarlos durante la carrera es algo que el autor de este informe considera muy positivo.