INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

IMPLEMENTACIÓN DE UN JUEGO DE DOMINÓ CON BÚSQUEDA MINIMAX/PODA ALFA-BETA.

INTELIGENCIA ARTIFICIAL

Presenta:

DIEGO VILLALVAZO SULZER 155844

JOSÉ SÁNCHEZ AGUILAR 156190

SEBASTIÁN ARANDA BASSEGODA 157465

Profesor:

Andrés Gómez De Silva Garza

Dominó

Resumen

El presente documento explica la elaboración de un programa escrito es SWI-Prolog. Este consistía en programa y aplicar el algoritmo minimax con poda alfa-beta, de manera que se produjo un sistema que era capaz de jugar jugar dominó y tomar decisiones a partir de una función heurística que infiriera los diferentes estados del juego.

Este programa representó una solución de inteligencia artificial al ser genérica ya que permite ser adaptada para jugar otros juegos o resolver problemas probabilísticos, a partir de una investigación previa llegamos a una teoría formal, sistemática, y específica acerca del dominio de aplicación del sistema.

I. INTRODUCCIÓN Y MARCO TEÓRICO

El juego del dominó consiste en fichas divididas en dos cuadrados, cada uno numerado del 0 al 6 de manera similar a la cara de un dado. Las combinaciones nos presentan con un juego que contiene un total de 28 fichas.

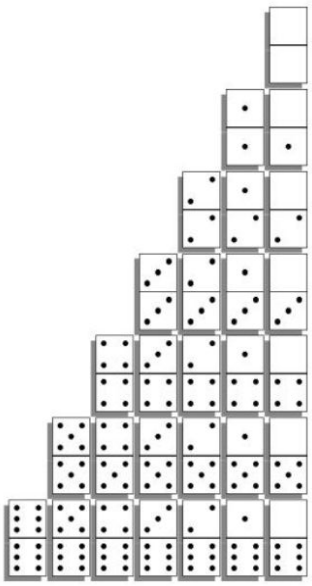

Fig. 1 Fichas del dominó enumeradas de forma ordenada

Al iniciar el juego cada jugador toma 7 fichas del pozo que consiste en las 28 fichas del juego colocadas boca abajo. Empieza uno de los jugadores y por turnos los jugadores colocan una de sus piezas restringidas por el hecho de que dos piezas solamente pueden colocarse juntas cuando los cuadrados adyacentes tengan el mismo valor. En caso de que un jugador no pueda colocar alguna ficha tiene que tomar fichas del pozo hasta poder tirar o pasar en caso de que el pozo haya quedado sin fichas.

El juego continúa hasta que no haya más movimientos posibles o alguno de los jugadores se haya quedado sin fichas, en cuyo caso será ganador.

Una vez definido el juego era necesario investigar la manera en la que se podría implementar un programa que pudiera jugar Dominó de la mejor manera posible.

El lenguaje de Prolog es un lenguaje de programación lógico. El desarrollo de una solución consistía en tres partes esenciales. *Primero, la implementación de todos los aspectos técnicos del juego*, es decir, cómo se podían representar las fichas, cómo se iban a expresar los extremos, cómo se guardaría el tablero y cualquier otra información que pensáramos pudiera ser relevante y que estuviera al acceso del programa.

La segunda parte de la solución consistía en la implementación de una función heurística. Una función heurística consiste en maximizar o minimizar los aspectos de un problema, debido a la naturaleza del problema donde no se jugaba por puntos, se tuvo que determinar alguna manera alterna de cuantificar los diferentes estados de juego. Con este objetivo en mente se pensó en las prioridades que tendría un jugador durante una partida y asignarle valor a estos casos. En una partida, la manera más efectiva y tangible de sacar ventaja al oponente es cuando el oponente no puede tirar, haciéndolo comer del pozo y que de esta manera te quedes sin fichas antes que el. A partir de los estados de juego, es decir, la mano y los extremos del tablero, se calculaba un valor a cada estado de juego según que tan probable era que el rival tuviera que pasar en la ronda, ya sea a partir de los turnos en los que ha pasado el rival anteriormente o estimando la probabilidad de que el rival no cuente con alguna ficha determinada.

La tercera parte de la solución consistía en hacer que el sistema encuentre la jugada óptima dependiendo del estado actual del juego. Para esto se utilizó el método de búsqueda minimax con poda Alfa-beta, el cual consiste en disminuir el número de nodos que son evaluados por la búsqueda minimax en el árbol de búsqueda. Es un algoritmo de búsqueda adversarial que se usa comúnmente para juegos de dos jugadores (Tic-tac-toe, Ajedrés, Go, etc.). La ventaja de este tipo de búsqueda es que deja de evaluar una serie de movimientos posibles cuando encuentra al menos una posibilidad que demuestra que el movimiento es peor que un movimiento examinado previamente, estos movimientos no necesitan ser evaluados más a fondo. Cuando se aplica a un árbol minimax estándar, devuelve el mismo resultado que entregaría minimax, pero elimina las ramas que posiblemente no puedan influir en la decisión final.

II. DESARROLLO DE LA SOLUCIÓN

1.Implementación del Dominó y main.

Empezamos por implementar un dominó a partir del cual se se pudieran crear los diferentes estados del juego. Empezamos por definir que cada una de las fichas del juego podía ser representada como una lista de dos elementos.

Se creó una lista para cada una de las fichas y se metieron las 28 listas en una lista que se llamaría *'desconocidas'*, de esta lista el programa podría consultar cualquiera de las 28 fichas en caso de que no estuvieran en la mano o hubieran sido jugadas previamente.

Para comenzar el juego, se llamaba el main. En ese momento el programa le preguntaba al usuario que 7 fichas había tomado del pozo, era a partir de esta información que el programa las sacaba de la lista de fichas desconocidas y las metía en una nueva lista de listas bautizada mano. Posteriormente, preguntaba quién hacía el primer movimiento, ya sea el programa o el oponente. A partir de este primer movimiento se determinaba el valor de dos predicados dinámicos:

extremoDerecho

extremoIzquierdo

Estos predicados le permiten al programa identificar los extremos, a partir de estos es que puede determinar los posibles movimientos y es más práctica la manipulación del tablero. Hay que remarcar que para el programa no es esencial tener las fichas del tablero en orden para obtener el valor de los extremos, el uso de los predicados dinámicos hace todo más sencillo y eficiente que leer el tablero para obtener los valores.

Para que la jugabilidad fuera lo más sencilla y rápida, desde la consola solo se tiene que ejecutar una regla *main*. Después, solo tenemos que ingresar la información que te pide el programa, por ejemplo, qué ficha tiró el oponente, o en caso de tener que robar del pozo, qué ficha robó el programa. Cada que el oponente tira una ficha, inmediatamente nuestro programa despliega en la consola nuestro siguiente movimiento, y espera el siguiente del oponente.

2. Función Heurística

Como fue explicado en la introducción, al desarrollar la función heurística le asignamos valor a los diferentes estados de juego a partir de qué tan probable era que el oponente tuviera que pasar o tomar fichas del pozo. Para hacer posible estimar la función se tuvieron que implementar un par de listas. La primera lista se llama *noTiene([]).* En esta lista se agregan los extremos cada vez que el rival toma del pozo.

La regla *rivalPaso(),* buscaba los extremos de cada estado de juego en la lista y asignaba un valor entre 0 y 4, 0 cuando ninguno de los extremos en un estado particular se encontraban en la lista, 2 cuando solamente uno de los extremos estaba en la lista y 4 cuando estaban ambos. La lógica es que mientras este número sea mayor es más probable que el rival tenga que comer y por lo tanto se obtenga ventaja.

En caso de que el oponente no hubiera pasado previamente se desarrolló una regla que estimaría la probabilidad de que el oponente no tuviera alguna ficha determinada que se pudiera colocar en algún extremo. Esta probabilidad iba entre 0 y 2, aumentando según era más probable que no tuviera la ficha. Para poder hacer esta estimación hizo falta implementar una par de cosas. Se empezó por hacer una lista llamada *numeros([8, 8, 8, 8, 8, 8 ,8]).* En la lista cada índice representaba un número del 0 al 6 existentes en las fichas y mostraba cuántas fichas de cada una eran desconocidas, es decir no estaban en la mano o habían sido jugadas en el tablero y por lo tanto tenían que estar en el pozo o la mano del oponente. Se implementó el predicado *Pozo,* utilizando ambos se utilizó la siguiente fórmula para determinar qué ficha era más probable que tuviera el oponente en su mano.

$$F(X, Y) = 1 - (X/Y)$$

Donde *X* era el número de fichas desconocidas de algún número y se obtenía de la lista y *Y* el número total de fichas desconocidas. Esta fórmula era efectiva porque permite obtener un número mayor en cuanto es más probable que el oponente no tenga alguna ficha determinada, aumentando la probabilidad de que el rival tenga que pasar y asignándole un peso mayor al estado de juego.

Es importante hacer énfasis que durante el juego, en cada tiro, nuestro o del rival, se actualiza la base de conocimiento del programa para que la función heurística pueda estimar de la mejor manera posible el peso del estado.

3. Algoritmo Minimax con Poda Alfa-Beta

Para encontrar el movimiento óptimo en cada jugada, se utilizó la búsqueda Mínimax con Poda Alfa-Beta. Para su implementación en Prolog, se dividió en tres reglas:

evalua(Turno, FichasPosibles, Tablero, Profundidad, Alfa, Beta, Record, Mejor)

poda(Turno, Ficha, Peso, Profundidad, Alfa, Beta, Posición, Record, Mejor)

alfa_beta(Turno, Profundidad, Tablero, Alfa, Beta, (Ficha, Peso)).

La regla *alfa_beta* permite inicializar la búsqueda con la profundidad deseada, la regla tiene dos casos. Un caso ocurre cuando el nodo que está evaluando no se encuentra en la frontera del árbol, realiza una llamada recursiva a *evalua* para crear el siguiente nivel. En el otro caso, cuando el nodo actual se encuentra en la frontera (bajó a profundidad) se regresa el peso que tiene ese nodo con ayuda de la función heurística.

La regla *evalúa* se encarga de crear nuevos niveles del árbol a partir de un nodo, dependiendo del turno de ese nivel (nuestro o del rival) se realiza la llamada recursiva a *alfa_beta* con las nuevas fichas compatibles de ese estado en particular y así sucesivamente hasta que la profundidad sea cero, obtenga el peso del nodo compatible y llame a la regla *poda* para evaluar las cotas alfa y beta.

La regla *poda* se encarga de realizar comparar los pesos proporcionados dependiendo de si en ese nivel se está maximizando nuestro tiro o minimizando el del rival. En caso de que se cumplan las condiciones dependiendo el estado se realiza un 'corte' en la ejecución (backtracking) para regresar al nodo padre y evitar que se calculen otros estados innecesarios.

Para obtener la ficha óptima se realizó la búsqueda a una profundidad 2, este nivel nos permitió obtener la ficha óptima rápidamente.

III. RESULTADOS

En las pruebas nos encontramos con un programa funcional que ganaba un poco por encima del 50% de las ocasiones, Cabe resaltar que en el dominó, al ser jugado por dos personas depende mucho de la suerte del jugador, con qué fichas se empieza y que si en algún momento tienes que tomar del pozo no sea necesario tomar demasiadas fichas.

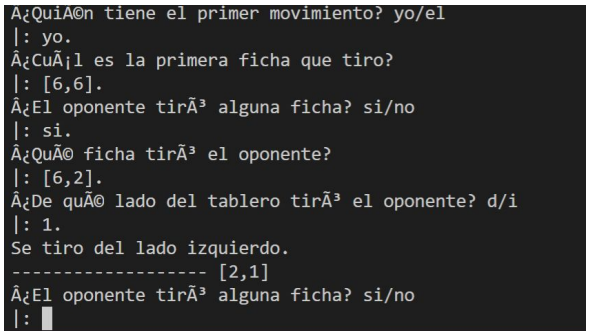

Fig. 2 Terminal mostrando la ejecución del programa

El programa no tuvo ningún problema para diferenciar los extremos del tablero y tomaba decisiones tomando en cuenta el extremo en el que quería colocar la ficha, expresándose claramente en la terminal como es mostrado en la figura 2.

IV. TRABAJO FUTURO

El proyecto consistió en implementar el dominó para un juego de dos personas, uno contra uno. Sin embargo, es posible implementar el modo de juego para tres o cuatro jugadores, ya sea en equipo o cada uno de forma individual.

Una forma de implementarlo sería sería tener diferentes reglas de *tiroOponente* para cada uno de los oponentes, con el propósito de poder ingresar a la base de conocimiento quién hizo cada tiro, y con qué fichas

pasaron. En general, sería una implementación muy similar a la que tenemos.

En cambio, para jugar en equipos (dos contra dos), tendríamos que simultáneamente intentar cerrar el juego de nuestros contrincantes e intentar abrir el juego de nuestro compañero. Se tendrían que modificar los parámetros que utiliza nuestra función heurística, ya que es necesario conocer qué fichas carece el compañero de equipo. Se guardaría en la base de conocimiento las fichas que nuestro compañero tiró bajo el supuesto que juega de manera racional, intentando abrir su juego.

V. CONCLUSIONES

Al final del proyecto los resultados fueron satisfactorios, además de que se logró afianzar los conocimientos obtenidos en clase sobre Prolog y Árboles de búsqueda. También se pudo observar y comprender la eficiencia que presenta la búsqueda minimax por poda alfa beta respecto a la minimax tradicional.

Empezando el proyecto, mucho antes de sentarse a redactar código, como equipo empezamos por discutir el juego de dominó, lo que lo compone y cómo podía ser expresado para que pudiera ser interpretado en prolog. Conocimiento que no solamente era respecto al juego, también del programa en el que lo íbamos a laborar, prolog.

Elaborar el programa fue laborioso y frustrante, constantemente el equipo tenía que parar e investigar cómo funcionaba algún aspecto de prolog, a veces para implementar algo nuevo y otras para buscar maneras alternativas de implementar cosas que ya se habían implementado para volverlas a hacer.

Como equipo concluímos que no existían muchas alternativas a la manera en la que trabajamos, proyectos de este tipo son largos y complicados, nos organizamos desde un inicio y parecía que estábamos siendo responsables y cumpliendo con nuestras diferentes metas en los plazos deseados, a pesar de eso vivimos un par de días tensos previos a la entrega que parecen, hasta cierto grado, inevitables.

Al presentar nuestro proyecto, ganamos en diferentes ocasiones, dejando claro que hicimos bien las cosas, no solamente por el éxito competitivo del programa, nos encontramos con que nuestro trabajo también era bastante más amigable con el usuario de lo que eran otros.

V. REFERENCIAS

1] Celko, J. (2001, September 9). Dice and Dominoes. Retrieved March 23, 2019, from https://www.pagat.com/tile/wdom/dice_dom.html

2] Russel, S and Norvig, P, Artificial Intelligence: A Modern Approach, Prentice Hall, 2010.