Programación Lógica

Laboratorio 2

Integrantes

Nombre	**CI**
Alexis Baladón	5574612-4
Ignacio Viscardi	5066666-2
Roberto de Armas	4886316-9

Índice

1. Descripción del problema

1.1. Requerimientos a implementar

Se debe implementar en Prolog una inteligencia artificial que juegue al 2048, con una estrategia buena y eficiente, que deberá describirse claramente en la documentación. Esta estrategia puede (pero no es necesario que así sea) basarse en una variante del algoritmo minimax (teniendo en cuenta que, en lugar de dos jugadores, luego de cada jugada hay un "movimiento" consistente en colocar la pieza aleatoria, según las reglas del juego).

El estado del juego se representa a través de una matriz con el tablero, que es un término Prolog con la siguiente forma (el tablero de ejemplo corresponde al primero de los presentados en la figura anterior):

$$m(f(8,4,-,-,),f(16,8,2,-),f(16,-,2,-),f(2,-,-,-))$$

Donde cada término `f(`x_0, x_1, x_2, x_3`)` representa una fila, y cada celda indica el valor de la pieza (si existe), o '-' en caso de estar el lugar libre.

Debe implementarse obligatoriamente el siguiente predicado:

$$mejor\_movimiento(+Tablero, +NivelMiniMax, +Estrategia, -Jugada)$$

Este predicado es el que implementa la IA. Recibe un tablero con el estado del juego, el nivel para el algoritmo minimax (en caso de que se lo utilice), y el nombre de la estrategia a utilizar (este argumento permite probar más de una estrategia), y devuelve la dirección del movimiento (up, down, left, right).

Se deberán entregarse las siguientes estrategias:

- Una estrategia, llamada `random`, que elige la dirección para el movimiento de forma aleatoria.
- Una estrategia, llamada `dummy`, que elige la dirección para el movimiento que permita sumar un mayor puntaje (y que sea un movimiento válido). El orden de preferencia, para el caso de tener dos direcciones que sumen lo mismo, es {up, down, left, right}. En caso de no haber ningún movimiento válido, devuelve `up`.
- Una estrategia, llamada `ia`, que implemente la mejor estrategia a la que puedan llegar.

2. IA Propuesta

Se plantea como estrategia una variación del conocido algoritmo minimax, adaptándolo al contexto específico. El algoritmo minimax es ampliamente utilizado para la toma de decisiones en juegos de adversario con información perfecta, con el objetivo de minimizar la pérdida esperada. Sin embargo, en este juego en particular, la suposición de que nuestro adversario juega de manera racional para obtener su mejor resultado, lo cual sería peor para nosotros, no se cumple.

En este juego, cada vez que se coloca una ficha en un espacio libre, existe una probabilidad del 80 % de que se coloque el número 2 y un 20 % de que se coloque el número 4. Esto indica que no se están tomando decisiones racionales que nos perjudiquen a largo plazo. Por lo tanto, implementar el algoritmo minimax sin modificaciones no daría buenos resultados.

Es necesario adaptar el algoritmo a las características específicas de este juego, teniendo en cuenta la probabilidad de las fichas que se colocan. De esta manera, se podrán tomar decisiones más acertadas que optimicen nuestro rendimiento en el juego.

Para abordar la problemática de la irracionalidad del oponente, surgen inicialmente dos opciones. Una posibilidad consiste en seleccionar un tablero aleatorio entre todos los generados al colocar una ficha en un lugar disponible. La segunda es elegir el promedio de los puntajes para todos los tableros obtenidos en ese nivel del subárbol. En nuestro caso, optamos por la segunda opción, ya que promete ser más precisa al aprovechar toda la información disponible, en lugar de dejar al azar la elección del tablero resultante.

Con esta variación, el único aspecto pendiente es definir la heurística seleccionada para evaluar un tablero. A continuación, se enumeran todas las que se aplican:

- Evaluación de la suma total de las fichas: Se considera la puntuación suma total en el tablero como medida de desempeño. Cuanto mayor sea la suma, mejor será la calificación del tablero.
- Utilización de las casillas libres: Se penaliza que use de mas lugares, es decir que se asigna mejor calificación al tablero con mas lugares libres.
- Suma generada en el movimiento: Esta heurística refuerza la anterior, dado que se califica mejor al tablero que procede de movimiento que unió la mayor cantidad de fichas para sumar el mayor puntaje posible.
- Diferencia de adyacentes. Tiene como objetivo lograr que el tablero resultante sea lo más "suave"posible, es decir, minimizar la diferencia entre una ficha y sus adyacentes. En esta heurística, existen diversas implementaciones posibles, pero optamos por una versión simple que consiste en calcular la diferencia entre una ficha y cada uno de sus adyacentes, y luego sumar estas diferencias para obtener una penalización en el puntaje total del tablero.

Estas heurísticas fueron ponderadas y combinadas hasta obtener un resultado razonablemente bueno en un tiempo razonable.

Fórmula utilizada:

$$Score = \begin{cases} \infty & \text{Si el } Tablero \text{ es perdedor} \\ -\infty & \text{Si el } Tablero \text{ es ganador} \\ \frac{8 \cdot \text{Suma} - 2 \cdot \text{RestaAdyacentes}}{Cantidad} & \text{Si no} \end{cases}$$

Donde:

$$\text{Suma} = \sum_{\text{i,j} \in \text{Tablero}} \text{v}_{ij}$$

$$\text{RestaAdyacentes} = \sum_{i=1}^{4} \sum_{j=1}^{3} |v_{i,j} - v_{i,j+1}|$$

Utilizando esta forma de evaluación de los tableros, se construye el árbol correspondiente al algoritmo minimax. En el caso del turno del jugador, se selecciona el puntaje máximo alcanzable a través del subárbol generado al elegir esa jugada. Por otro lado, en el turno de colocar una ficha, se calcula el promedio de los puntajes obtenidos para todas las posibles posiciones en las que se puede colocar una ficha. Además, se considera si la ficha a colocar es un 2 o un 4, ponderando la probabilidad de que sea efectivamente ese número.

2.1. Decisiones de diseño

En la imagen [2.1] puede visualizarse un diagrama de la solución propuesta. Esta fue realizada siguiendo una arquitectura en capas para facilitar su comprensión e implementación.

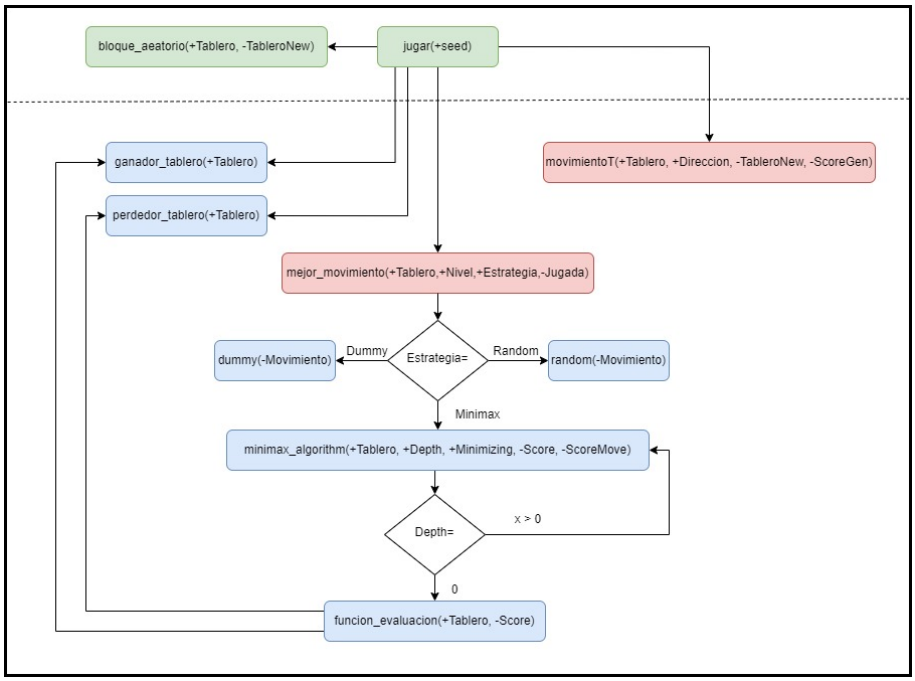

Figura 2.1: Diagrama de dependencia de predicados

En la parte superior pueden observarse en color verde los predicados que son de ayuda para simular una partida, aunque no fueron pedidos explícitamente en la tarea. Sin embargo, se requirió implementar predicados similares para resolver el problema en un script de prolog, y además en los archivos provistos dentro de la tarea estas ya estaban también implementadas utilizando un puente entre python y prolog.

En la parte inferior del andarivel pueden observarse en color rojo los predicados pedidos en la tarea junto a las funciones de las que se auxilian en color cian. En cuanto al diseño de la solución implementada, este se basa principalmente en la implementación de los predicados movimientoT y mejor_movimiento. El predicado movimientoT se encarga únicamente de generar el tablero resultado de aplicar un movimiento dado, con $movimiento \in \{up, down, left, right\}$.

En cuanto a mejor_movimiento, la implementación es más compleja e implica una capa extra en el diseño, ya que depende del modelo utilizado para determinar qué es un buen movimiento. Las heurísticas dummy y random son relativemente sencillas. Por otro lado, como puede observarse, la solución implementada muestra cierta complejidad agregada dado que, como se busca tanto minimizar como maximizar mediante el predicado minimax_algorithm, esta es implementada a modo de dos predicados (en realidad uno solo sobrecargado 2 veces) mutuamente recursivo, donde se intercala entre maximizar y minimizar posibles resultados.

2.2. Principales predicados

A continuación se presentan los principales predicados implementados para la resolución de mejor_movimiento con una breve descripción de su funcionamiento.

- mejor_movimiento(+Tablero,+NivelMiniMax,+Estrategia,-Jugada): devuelve la mejor jugada posible, a partir de un tablero y una estrategia
- minimax_algorithm(+Tablero, +Depth, +Player, -Score, +ScoreMove): devuelve el score de un tablero dado, a partir de un nivel de profundidad y un jugador (true = jugador, false = juego)

- movimiento_del_jugador(+Tablero, -NewTablero, -Direccion, -Score): dado un tablero, devuelve el tablero resultante de movimientos validos del jugador (movimientos up,left,right,down tales que hacen que el tablero cambie).
- movimiento_del_jugador(+Tablero, -NewTablero, -Direccion, -Score): dado un tablero, devuelve el tablero resultante de movimientos validos del jugador (movimientos up,left,right,down tales que hacen que el tablero cambie).
- movimiento_del_juego(+Tablero, -NewTablero, -Direccion, -Score): dado un tablero, devuelve el tablero resultante de agregar un valor 2 o 4 en alguna de las posiciones vacías del tablero dado
- funcion_evaluacion(+Tablero, -Score): devuelve el score de un tablero dado
- perdedor_tablero(+Tablero): devuelve true si el tablero no tiene movimientos posibles
- ganador_tablero(Tablero): devuelve true si el tablero tiene al menos un valor 2048
- addValue(+Tablero, +Posición, +Valor, -TableroNuevo): agrega un valor a un tablero en una posicion dada
- isValue(+Tablero, ?Posición, ?Valor): devuelve el valor de una posición de un tablero dado o la posición de un valor dado o ambos o true si el valor dado esta en la posición dada.

Para el caso de movimientoT, los principales predicados implementados son:

- movimientoT(+Tablero,+Direccion,-TableroNew,-ScoreGen): toma el tablero y mueve las piezas hacia la dirección correspondiente
- invertir_tablero(+Tablero, -TableroInvertido): invierte un tablero dado
- transponer_tablero(+Tablero, -TableroTranspuesto): transpone un tablero dado
- mover_tablero(+Tablero, -TableroNuevo, -ScoreTotal): mueve los valores de un tablero hacia la izquierda y devuelve el puntaje obtenido
- mover_fila(+Fila, -NuevaFila, -Puntaje): mueve los valores de una fila hacia la izquierda y devuelve el puntaje obtenido

3. Resultados

En esta sección se presentan los resultados obtenidos al correr 100 simulaciones por cada semilla de un conjunto de 10 semillas obtenidas aleatoriamente para cada estrategia (es decir que se corren 1000 simulaciones por estrategia). Esto fue realizado tanto para la estrategia random y dummy, como para la estrategia minimax implementada, en sus variantes de 1, 2 y 3 niveles de profundidad.

Adjunto a este informe se encuentran 3 archivos csv que contienen estadísticas de las partidas. Por un lado, se encuentra all_test_results.csv el cual contiene el resultado de todas las partidas realizadas para cada una de las estrategias. Entre la información que presenta se encuentra el ultimo tablero de cada partida, si gano o no, el puntaje final, la cantidad de movimientos, el tiempo empleado, el máximo valor alcanzado, entre otros.

Por otro lado, summary_per_seed_strategy_level.csv contiene un resumen de los resultados obtenidos por seed y estrategia, para las 100 partidas realizadas. Aqui se presenta la tasa de victorias para esa seed y estrategia en 100 partidas, el tiempo promedio, puntaje promedio, movimientos promedio, entre otras.

Finalmente, se presenta el archivo summary_per_strategy.csv, el cual da un resumen de los resultados obtenidos por estrategia para todas las seeds y partidas (resume lo obtenido por seed y estrategia en el csv mencionado

antes). Aqui se presentan estadisticas como, la tasa de victorias media entre las seeds para una estrategia, la varianza de la tasa de victorias, puntaje promedio, tiempo promedio, entre otros.

De estos resultados se puede observar que existe una clara diferencia en el rendimiento de las diversas estrategias planteadas.

Para comenzar, ninguna de las estrategias sencillas (dummy y random) pudieron ganar una sola partida. Por otra parte, como se observa en 1, la versión de minimax implementada (inclusive con profundidad 1) logra ganar al menos una partida en las semillas probadas. Viendo un poco mas en detalle se observa que la estrategia dummy logra para algunas partidas (un 0.3 % del total jugadas) alcanzar un valor máximo de 1024, lo cual mejora considerablemente el resultado de la estrategia random, la cual no logra pasar de los 256.

Cuadro 1: Valores máximos alcanzados

Estrategia	Max
Dummy	1024
Minimax_d1	2048
Minimax_d2	2048
Minimax_d3	2048
Random	256

Para ver los resultados de forma más transparente es de ayuda ver la distribución de los valores obtenidos, y no solo estudiar un máximo. Por un lado, en 2 se observan resultados numéricos que, entre otros, reportan medidas de tendencia central y de variación. Asimismo, se observa que el algoritmo con profundidad 3 logra una sorprendente tasa de aciertos (winrate) de 40 %, mientras que menores profundidades no alcanzan el 10 %.

Cuadro 2: Resumen de estrategias

Strategy	Depth	ScoreAvg	CantMovAvg	WinrateAvg	$\sigma(winrate)^2$	Tiempo (s)
Random	-	989.372	104.078	0	0	0.005962
Dummy	-	3026.756	238.118	0	0	0.014284
Minimax	3	16139.784	813.532	0.41	0.00111	6.563
Minimax	2	12390.572	697.783	0.094	0.00127	0.983165
Minimax	1	6799.848	444.878	0.004	2.67e-5	0.099461

Por otro lado, observando 3.1, se visualiza de forma más clara la distribución de los valores obtenidos. Es posible observar distribuciones multimodales con respecto a los puntajes obtenidos. Esto puede deberse a los saltos que realiza el puntaje al obtener un nuevo mayor puntaje en un bloque, o a la variedad de semillas probadas. Adicionalmente, se observa como la estrategia dummy y random tienden a estancarse en un puntaje pequeño, mientras que las demás estrategias tienden a distenderse considerablemente más.

Figura 3.1: Distribución de puntajes por estrategia

Es interesante observar la ganancia obtenida por aumentar la variable de profundidad. Sin embargo, esto también conlleva un costo computacional, el cual puede observarse tanto en la tabla de valores, como en el la gráfica en 3.2, donde puede observarse cómo el tiempo que conlleva un algoritmo de este tipo escala exponencialmente dada la cantidad de soluciones posibles que esta puntúa antes de decidir el mejor movimiento. Se destaca que para profundidad 4 el tiempo aproximado de una sola simulación de una partida era de unos 100 segundos, lo cual es 20 veces mas que lo que lleva una de profundidad 3 (por razones de tiempo de computo no se realizaron test extensivos para profundidades mayores a 3). Una solución para intentar evitar este crecimiento exponencial tan temprano podría lograrse utilizando métodos más complejos como aproximar un resultado a partir de una selección aleatoria de ramas del árbol de posibles soluciones.

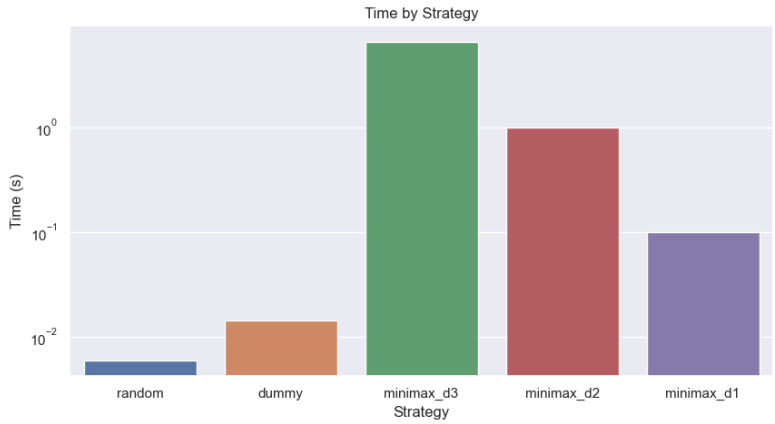

Figura 3.2: Tiempo promedio por estrategia

4. Conclusiones

Las estrategias como *random* y *dummy*, a pesar de no requerir mucho cómputo, demostraron ser muy ineficaces y no lograron llevar a la victoria en ninguno de los casos probados. Es evidente que la estrategia de minimax ofrece resultados más que aceptables para las profundidades que se evaluaron. Aunque no se exploraron profundidades superiores a 3, todo indica que al aumentar la profundidad, la tasa de victoria también aumenta. Sin embargo, el problema radica en que el costo computacional también crece de manera exponencial.

Este aumento en el costo computacional se debe a la creciente cantidad de tableros que deben evaluarse en cada nivel de profundidad. Este aspecto podría abordarse mediante el uso de estrategias de poda, las cuales no fueron experimentadas en este caso.