Programación Lógica
Laboratorio 02

Grupo 18

Joaquín Lezama, Diego Bartolomé, Sebastián Herrera

Introducción

Se presenta el problema de implementar ciertos predicados en Prolog para el juego conocido como "Paper soccer" donde dos jugadores se mueven por turnos dentro de un tablero (de tamaño configurable) con objetivo de hacer el gol.

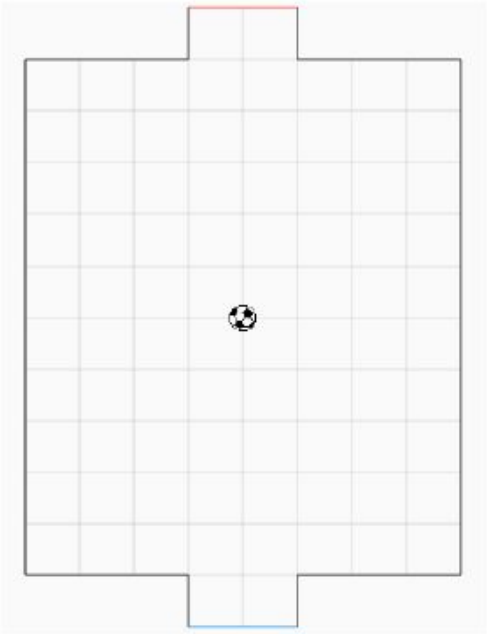


Figura 1: Estado inicial del juego

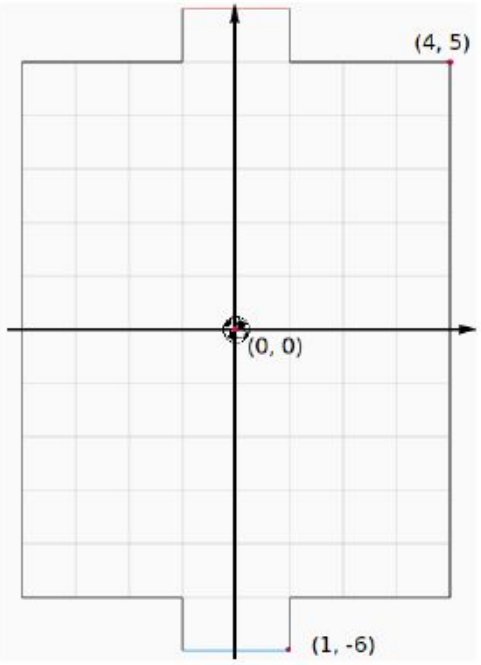


Figura 2: Sistema de coordenadas de las posiciones, con algunos puntos de ejemplo.

Se pide implementar en Prolog algunos predicados para el funcionamiento del juego, y otros predicados para jugar contra la máquina (denominada *Inteligencia*).

Los predicados para el juego son:

- **estado_inicial(?E)**: E es el estado inicial del juego, de tamaño AltoxAncho unificado por cantidad_casilleros(Ancho,Alto). La pelota comienza en la posición p(0,0), que es la posición central del tablero. El turno inicial es del jugador 1, que es el que patea hacia arriba. Los grupos son libres de modelar el estado del juego como quieran, mientras cumplan los requerimientos de los predicados.
- **posicion_pelota(+E,?P)**: P es la posición de la pelota para el estado E.
- **mover(+E,?LP,?E2)**: E2 el estado resultante de hacer un movimiento con la pelota, a través de las posiciones de la lista LP en el estado E y de cambiar el turno.
- **gol(+E,?Njugador)**: La pelota está en situación de gol a favor del jugador NJugador para el estado E.
- **turno(+E,?Njugador)**: NJugador es el jugador que tiene que mover en el siguiente turno para el estado E.

- **prefijo_movimiento(+E,+LP)**: LP es una lista no vacía de posiciones que constituyen el prefijo de un movimiento para el estado E, sin llegar a formar un movimiento. Este predicado permite validar jugadas del jugador humano mientras va eligiendo posiciones, para luego finalmente concretar el movimiento con mover.

Los predicados para la inteligencia son:

- **nombre(?Nombre)**: Nombre es el nombre de la inteligencia, que deberá ser "Grupo XX", siendo XX el número de grupo. Esto se usa en la web para saber contra quién se está jugando.
- **niveles_minimax(?MaxNiveles)**: MaxNiveles es la cantidad máxima de niveles de Minimax de la inteligencia.
- **hacer_jugada(+E,?LP,?E2)**: E2 es el estado resultante de mover según la lista de posiciones LP de un movimiento que la inteligencia elige jugar para el estado E. El turno de jugar en el estado E es el de la inteligencia, mientras que en E2 es del otro jugador.

Implementación

Estructura de la solución

La solución consta de dos módulos, uno llamado *juego.pl* donde se implementan los predicados necesarios para el funcionamiento del juego junto con varios predicados auxiliares, y otro llamado *inteligencia.pl* donde se implementan los predicados necesarios para jugar contra la máquina, también junto a otros predicados auxiliares.

Cada uno de estos módulos expone los predicados solicitados, y además *inteligencia.pl* consulta algunos de los predicados auxiliares implementados en *juego.pl*.

Representación de la estructura

La estructura elegida para representar el estado del juego es mediante un functor que contiene los siguientes tres argumentos:

1. el tablero
2. la posición actual de la pelota
3. el jugador que le toca mover

El tablero se representa mediante una matriz, donde cada entrada corresponde a un punto del tablero del cual se tienen dos datos:

- si es una posición inválida de la cancha toma el valor -1, si aún no fue visitado toma el valor 0, y si fue visitado toma el valor 1
- y también se tiene una lista, inicialmente vacía, que representa las aristas visitadas adyacentes al punto (esta lista puede tener como máximo 8 elementos, que sería un caso que el jugador se queda sin movimientos y pierde).

El tamaño de esta matriz depende del alto y ancho que se configuren para el juego.

La posición actual de la pelota se representa mediante las coordenadas en la matriz para la fila y la columna del vértice donde se encuentra la pelot.

El jugador que le toca mover inicialmente es el jugador 1, y este valor se alterna tomando el valor 2 o 1 cada vez que un jugador ejecuta la función mover.

Predicados del juego

estado_inicial(?E):

Se cargan los tres argumentos de la estructura inicial:

1. Se genera un tablero con Alto + 3 filas y Ancho + 2 columnas. En dicho tablero se cargan los vértices con su estado correspondiente (-1, 0 o 1) y la lista de adyacentes visitados vacía.
2. Se carga como posición actual de la pelota p(0,0)
3. Se inicializa el 3er argumento con 1, que representa el jugador al que le toca mover.

posicion_pelota(+E,?P):

La posición de la pelota se encuentra en el argumento 2 de la estructura E. Primero se obtiene el dato de la estructura E y luego se convierte al sistema de coordenadas manejado por la web utilizando un predicado auxiliar *coordenadasMatrizToWeb*.

mover(+E,?LP,?E2):

Es un predicado recursivo. Se recorre la lista de movimientos LP y para cada elemento H:

1. Se valida que dada la estructura E moverse a la posición H es un movimiento válido. Que sea un movimiento válido significa:
    a. Que la posición H sea adyacente a la posición P, que es la posición actual de la pelota
    b. Se verifica que la arista que une a P y H no haya sido visitada. Para esto se recorre la lista de adyacentes visitados del vértice que corresponde a la posición P y se verifica que H no se encuentra en la misma.
    c. Que no se esté moviendo de un vértice de un borde a otro vértice del mismo borde.
2. Se realiza el movimiento H en la estructura E. Esto significa:
    a. Se agrega la posición H a la lista de adyacentes visitados del vértice que corresponde a la posición P (P - posición actual de la pelota)
    b. Se agrega P a la lista de adyacentes visitados del vértice que corresponde a la posición H
    c. Se setea H' como segundo argumento de E.
3. [Si no quedan más elementos en LP] se verifica que el jugador no debe seguir moviendo. Para verificar esto se niega el predicado *hay_movimiento_siguiente(E),* este predicado consiste en:
    a. Verificar que la estructura E no esté en situación de gol
    b. Se verifica que no sea una esquina (en cuyo caso no hay más movimientos)
    c. [si *posicion_pelota* P es un borde] se verifica que el vértice que corresponde a P tenga menos de 3 elementos en su lista de adyacentes visitados
    d. [si *posicion_pelota* P no es un borde] se verifica que el vértice que corresponde a P tenga entre 1 a 7 elementos en su lista de adyacentes visitados
4. [Si aún quedan movimientos en LP] Se valida que dada la estructura E el jugador deba seguir moviendo (utilizando *hay_movimiento_siguiente*) y se llama al mover recursivo con el E resultante de aplicar el movimiento H y LP sin H (La cola de la lista).

gol(+E,?Njugador):

Un gol del jugador 1 se da en la situación que *posición_pelota* devuelve una de las siguiente 3 posiciones:

1. p(-1, $\frac{Alto}{2} + 1$)
2. p(0, $\frac{Alto}{2} + 1$)
3. p(1, $\frac{Alto}{2} + 1$)

Un gol del jugador 2 se da en la situación que *posición_pelota* devuelve una de las siguiente 3 posiciones:

1. p(-1, - $\frac{Alto}{2} + 1$)
2. p(0, - $\frac{Alto}{2} + 1$)
3. p(1, - $\frac{Alto}{2} + 1$)

turno(+E,?Njugador):

El jugador al que le toca mover es el tercer argumento de la estructura E.

prefijo_movimiento(+E,+LP):

La implementación de este predicado es muy similar a la de mover. Es un predicado recursivo, donde se recorre la lista de movimientos LP y para cada elemento H:

1. Se valida que dada la estructura E moverse a la posición H es un movimiento válido.
2. Se realiza el movimiento H en la estructura E.
3. [Si no quedan más elementos en LP] se verifica que el jugador sigue teniendo movimientos posibles en el estado que quedó E (se utiliza el predicado hay_movimiento_siguiente)
4. [Si aún quedan movimientos en LP]Se llama al mover *prefijo_movimiento_recursivo* con el E resultante de aplicar el movimiento H y LP sin H (La cola de la lista).

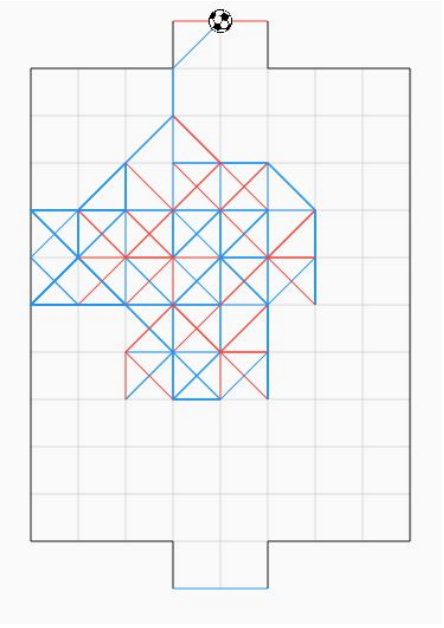
Predicados de la inteligencia

hacer_jugada(+E,?LP,?E2):

Este predicado recibe el estado del tablero E, y busca realizar el mejor movimiento posible LP, quedando el juego en el estado E2. La implementación del algoritmo solamente logra buscar un nivel hacia adelante, que se realiza de la siguiente forma:

1. Se obtiene el jugador J que le toca mover con el predicado *turno(E,J)*.
2. A partir de la posición actual de la pelota, y de las aristas visitadas, se exploran todos los posibles movimientos que puede realizar el jugador. Esto se realiza mediante el predicado *findall(LP, mover(E, LP, _), LPs)*.
3. Una vez que se tienen todos los posibles movimientos, se busca determinar cuál de ellos es el mejor, por lo cual se define una función de evaluación del estado de un tablero como sigue (se busca minimizar el resultado):
    a. Si es gol del jugador J, se asigna puntaje -100.
    b. Si es gol del otro jugador, se asigna puntaje +100.
    c. Si el jugador J se quedó sin movimientos, se asigna puntaje +100.
    d. De lo contrario se asigna como puntaje la distancia entre la posición de la pelota y el gol.
4. Con la función de evaluación definida en el punto anterior se evalúan todos los posibles movimientos para quedarse con el mejor. Si hay dos tableros diferentes que tienen el mismo puntaje, entonces se queda con el que evaluó primero.
5. Finalmente con el mejor LP obtenido se ejecuta el predicado *mover(E,LP,E2),* efectuando el movimiento y cambiando dejando el tablero en estado E2, donde es el turno del otro jugador.

A continuación se presenta una imagen del resultado de una partida entre dos “inteligencias” que utilizan el algoritmo implementado:



Conclusiones

Durante la implementación de la solución para los predicados planteados en el problema se presentaron diversas situaciones para reflexionar:

- La representación elegida en una primera instancia constaba de una lista de vértices visitados, y el jugador que había realizado cada movimiento. Esto tuvo que ser modificado rápidamente por la estructura final, pues iba a resultar en una degradación importante de la performance del sistema.
- Un aspecto importante de la implementación es la modularidad de las operaciones que se logró, que resultó beneficiosa para reutilizarlas en diferentes lugares, además de un mejor entendimiento del código.
- En las primeras etapas del desarrollo se implementaron algunos tests para verificar el correcto funcionamiento de las varias operaciones, principalmente aquellas más atómicas, y aquellas que se reutilizan dentro de otras operaciones. Esto sirvió para facilitar probar que al hacer arreglos sobre algunas operaciones todo siguiera funcionando correctamente.
- Se pudo observar que cuando se realizan copias (por ejemplo mediante el predicado *duplicate_term*) en operaciones recursivas, se puede presentar una degradación importante de la performance del sistema, al punto de fallar por quedarse sin memoria en el stack.
- El minimax implementado soporta solamente un nivel de búsqueda hacia adelante. Se intentó desarrollarlo de diferentes formas, pero no se tuvo éxito en superar dos problemas que se presentaron:
    - quedarse sin memoria en el stack
    - entrar en una recursión infinita

  Para el primero de los problemas un abordaje fue evitar realizar copias de la estructura, pero esto llevó a otro tipo de problemas, como por ejemplo que se viera afectada la misma estructura E durante la recursión completando aristas visitadas que no debería, y también se presentaba el problema de entrar en una recursión infinita.

  Para el segundo de los problemas se verificó de agregar todas las posibles condiciones de corte, y otro tipo de soluciones como mantener una copia de la estructura por cada nodo intermedio del árbol, pero no fue posible solucionarlo.

  Es por esto que finalmente se implementó un algoritmo que busca la mejor jugada contemplando solamente un turno hacia adelante.

Competencia

No deseamos participar de la misma.

Referencias

[1] Presentación “Recursión”. Curso de Programación Lógica. Facultad de Ingeniería. Universidad de la República.
https://eva.fing.edu.uy/pluginfile.php/157016/mod_resource/content/0/PL2017_02_conceptosBasicos3_rec%20%281%29.pdf (Extraído en junio de 2017).

[2] Presentación “Aritmética”. Curso de Programación Lógica. Facultad de Ingeniería. Universidad de la República.
https://eva.fing.edu.uy/pluginfile.php/158683/mod_resource/content/0/PL2017_05_conceptosBasicos-aritme.pdf (Extraído en junio de 2017).

[3] Presentación “Predicados metalógicos”. Curso de Programación Lógica. Facultad de Ingeniería. Universidad de la República.
https://eva.fing.edu.uy/pluginfile.php/161075/mod_resource/content/1/PL2017_12_metalogicos_terminos.pdf (Extraído en junio de 2017).

[4] Presentación “Resolución de juegos”. Curso de Programación Lógica. Facultad de Ingeniería. Universidad de la República.
https://eva.fing.edu.uy/pluginfile.php/163699/mod_resource/content/0/PL2017_15_resolucion_juegos.pdf (Extraído en junio de 2017).

[5] Sterling, L. Capítulo 20 “The Art of Prolog” 2da edición. The MIT Press. ISBN 0-262-19338-8.

[6] “Artificial Intelligence - Implementing Minimax with Prolog”. Mines Saint-Étienne.
http://www.emse.fr/~picard/cours/ai/minimax/ (Extraído en junio de 2017).