Universidad de la Habana, MATCOM
Programación Declarativa (2021)

Juego Hive

Samuel David Suárez Rodríguez C412
Enmanuel Verdesia Suárez C411

1 Detalles de Implementación

El juego fue creado usando `SWI-Prolog 8.2.4`. En dicho lenguaje se implementó toda la lógica para la ejecución de la Inteligencia Artificial (IA).

1.1 Sistema de Coordenadas

Se empleó un sistema axial de coordenadas para representar la posición de las piezas en el tablero, donde la posición de cada pieza está determinada por dos componentes (q, r) como se muestra en la imagen a continuación.

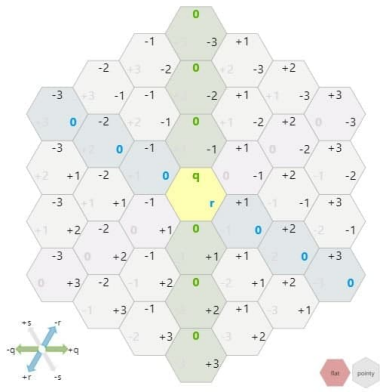

Figure 1: Cuadrícula hexagonal.

1.2 Estados

Para representar una pieza se usó un predicado de la siguiente forma:

```
piece(Class, Color, Id, Q, R, S).
```

- `Class`: es el tipo de la pieza dentro del juego de Hive: reina (`'q'`), saltamontes (`'g'`), araña (`'s'`), mariquita (`'l'`), hormiga (`'a'`), escarabajo (`'b'`) o mosquito (`'m'`).
- `Color`: el color del jugador a quien corresponde la pieza (`'b'` o `'w'`).
- `Id`: el identificador de la cada pieza para diferenciarla de otras con la misma clase.
- `Q` y `R`: las componentes correspondientes en el sistema de coordenadas.
- `S`: índice de apilación que tiene la pieza en caso de estar encima de otra.

Estos predicados son almacenados en la Base de Datos (BD) local de Prolog, manteniendo así el estado actual del sistema. Por tanto modificar el estado implica agregar y/o eliminar predicados en la BD. Mediante el siguiente predicado se consulta dicho estado.

```
get_pieces(Pieces) :-
  findall(
    piece(Class, Color, Id, Q, R, Stacked),
    piece(Class, Color, Id, Q, R, Stacked),
    Pieces
  ).
```

1.3 Consistencia de la Colmena

En todo momento se mantiene la consistencia de la colmena, de tal forma que esta no se divida en una o más partes motivo del movimiento de una pieza a una nueva posición. Por tanto si una pieza es punto de articulación (eliminarlo causaría que el grafo formado por las piezas deje de ser conexo) esta no puede ser movida a otra posición hasta que deje de serlo. El predicado a continuación realiza dicha validación.

```
articulation_point(piece(Class, Color, Id, Q, R, S)) :-
  get_pieces(Pieces),
  length(Pieces, LPieces),
  Remaining is LPieces - 1,
  piece_neighbours(Q, R, [Start|_]),
  remove_piece(piece(Class, Color, Id, Q, R, S)),
  connected_component([Start], [], Component),
  add_piece(piece(Class, Color, Id, Q, R, S)),
  length(Component, LComponent),
  Remaining \= LComponent.

connected_component([], Marked, Component) :- append([], Marked, Component).
connected_component([piece(Class, Color, Id, Q, R, S) | Stack], Marked, Component) :-
  piece_neighbours(Q, R, Neighbours),
  member(Next, Neighbours),
  \+ member(Next, Marked),
  append(Stack, [Next], NStack),
  append(Marked, [Next], NMarked),
  connected_component([piece(Class, Color, Id, Q, R, S) | NStack], NMarked, Component), !.
connected_component([_ | Stack], Marked, Component) :-
  connected_component(Stack, Marked, Component).
```

Donde `piece_neighbours` es un predicado que dada la posición de una pieza obtiene todas la piezas que son adyacentes a esta y `connected_component` devuelve la lista de elementos en una componente conexa comenzando en alguna pieza. Si al eliminar la pieza y llamar `connected_component` obtenemos menos de la cantidad de piezas que restan en el tablero, podemos decir que es un punto de articulación y por tanto rompe la colmena.

1.4 Transiciones de estados

Un juego no es más que una lista de acciones secuenciales que van modificando los estados, en nuestro caso una acción se representa como

```
action(
  Piece1,
  Piece2,
  Side
)
```

En donde `Piece1` y `Piece2` son piezas y `Side` puede tomar uno de estos posibles valores:

- `N` (Al norte de)
- `NW` (Al noroeste de)
- `SW` (Al suroeste de)
- `NE` (Al noreste de)
- `SE` (Al sureste de)
- `S` (Al sur de)
- `O` (Sobre)

Esto indica la posición donde colocar la pieza `Piece1` respecto a `Piece2`.
Un ejemplo de acción puede ser

```
action(
  piece(a, b, 3, 1, -3, 0),
  piece(q, w, 1, 0, 1, 0),
  NW
)
```

que se traduce como

```
'La hormiga negra con id 3 se posiciona al noroeste de la reina blanca con id 1.'
```

Los edge cases aquí son los movimientos de adición de una pieza al tablero y la primera acción del juego que también es de adición pero sin ninguna pieza respectiva, cuando esto pasa decidimos por convenio igualar los valores de `Piece1` y `Piece2`, sin importar el valor de `Side`, y para el segundo edge case hacemos una diferenciación si nos encontramos en el turno 1.

Como detalle cabe aclarar que si una pieza no se encuentra en el tablero actualmente entonces los valores de su posición `Q, R, S` son -1. Esta es una posición válida si solo tenemos en cuenta `Q, R`, sin embargo como `S` debe ser mayor o igual que cero nos sirve para diferenciar los casos.

Con respecto a la implementación, definimos un predicado `step` que recibe una acción y un turno y en caso de que la acción sea correcta, actualiza el estado actual. Para las validaciones implementamos predicados especificos para cada tipo de pieza, en el que cada tipo tiene un predicado `<tipo>_can_move` que recibe la pieza y una posición y chequea si esa pieza se puede mover. De esta manera, en caso de que la acción sea de movimiento, llamamos al predicado correspondiente según la pieza que se quiera mover en la acción, y si el resultado triunfa, movemos la pieza. En caso de la adición las validaciones son las mismas para cualquier tipo de pieza, por lo que implementamos un predicado que lo valide.

Este tipo de organización en el código para validar las acciones nos permite de una manera limpia validar especialmente el movimiento del mosquito, cuya validación cambia en dependencia de las piezas adyacentes, el código para esto es el siguiente

```
mosquito_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS), C) :-
  (C = q, queen_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)));
  (C = a, ant_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)));
  (C = l, ladybug_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)));
  (C = s, spider_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)));
  (C = g, grasshopper_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)));
  (C = b, beetle_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS))).

mosquito_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS)) :-
  piece_neighbours(Q, R, Neighbours),
```

```
member(piece(NeighbourClass, _, _, _, _, _), Neighbours),
mosquito_can_move(piece(Class, Color, Id, Q, R, S), position(NQ, NR, NS), NeighbourClass).
```

por lo que un mosquito puede moverse a una posición cuando posee un vecino tal que alguno de los predicados anteriores triunfe.

1.5 Interacción con el usuario

Para iniciar el juego se debe llamar el archivo `main.pl`, ejecutar `main().` y seguir las instrucciones.

El usuario interactúa en el juego escribiendo de una manera codificada la acción que quiere realizar, por ejemplo

`ba3 NW wq1`

corresponde a la acción descrita anteriormente donde la hormiga negra se mueve al noroeste de la reina blanca. El primer caracter de cada pieza indica el color, luego el tipo de pieza seguido por el id, y en el medio y separado por espacios se indica el valor de `Side`. En caso de la primera jugada solo se escribirá la pieza que se quiere colocar, por ejemplo

`wq1`

En cada momento luego de realizar una acción se devuelve en forma de lista los valores de cada `piece` que hay en el tablero, de esa manera sabremos el estado actual, en caso de que la jugada sea inválida se le notificará al usuario un mensaje con la descripción del error que cometió.

1.6 Inteligencia artificial

El juego posee una variante `pve (Player vs Environment)` en el que el usuario puede jugar contra un agente. Este agente implementa una heurística sencilla basada en asignarle valores a cada estado y en cada turno observar qué acción maximiza el valor del siguiente estado.

La función de estado se califica de la manera siguiente

- Si el estado es ganador, asignamos un valor lo suficientemente grande para ser mayor que los demás estados (500), puesto que llegar a este estado implica una victoria.
- Si el estado es una derrota, asignamos el menor valor posible (-500), puesto que es el estado que queremos evitar.
- Si el estado es un empate, asignamos el valor 5. En un principio podemos pensar en asignar el valor medio de los estados ganadores y perdedores (250), sin embargo esto conlleva a que nuestro agente priorize empatar que realizar otras acciones que pueden llevarlo a ganar, por eso le dimos menos prioridad a los empates.
- En cualquier otro caso calculamos el valor de un estado con la siguiente fórmula.

```
10 * (OponentPiecesAroundQueen - PlayerPiecesAroundQueen)
 + 2 * (LengthPlayerFreePieces - LengthOponentFreePieces)
 + (LengthPlayerPiecesOnBoard - LengthOponentPiecesOnBoard)
```

  es una suma de términos donde indicamos a qué detalles del estado le damos prioridad. El primer término es la diferencia entre la cantidad de piezas que rodean a la reina del oponente menos las del agente, es el término con más prioridad, y que hace que el agente rodee la reina enemiga y mueva su reina cuando tiene muchos vecinos que la bloquean. El segundo término mide la diferencia entre la cantidad de piezas que no estan rodeadas en el tablero entre el agente y el oponente, lo que prioriza tener una mayor libertad entre las piezas en el tablero. Por último el tercer término es la diferencia entre la cantidad de piezas entre el agente y el oponente, lo que permite agregar piezas en el tablero en el early game, luego esta decisión va perdiendo prioridad debido a que los otros casos aportan mayor valor, pero permiten una mayor flexibilidad en los primeros turnos.

Este agente encapsula todo el algoritmo en un predicado `choose_action(Color, Action, Turn)` que decide cuál acción realizar dado el estado actual, para probar cada acción llamamos la función `step` de la que hablamos anteriormente, y también definimos un predicado `undo_action` que revierte la acción realizada, este predicado es usado exclusivamente por el agente.