Programación Lógica

Laboratorio 2

Árabe

Integrantes

Nombre	CI	Correo
Gonzalo Marco	4.854.892-9	gonzalo.marco.mohotse@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy
Ramiro Pombo	4.725.542-8	ramiro.pombo@fing.edu.uy

Índice

1. Lógica del juego

A continuación se tratan todos los temas relacionados al desarrollo de la lógica del juego y de tres de los cuatro predicados principales de la consigna.

1.1. Predicados auxiliares

Se implementaron múltiples predicados auxiliares de manera tal que se facilite tanto el diseño como la lectura de los predicados principales, dividiendo los problemas bajo un enfoque de *divide and conquer*. Teniendo en cuenta que la lista de predicados es extensa y el funcionamiento de muchos de ellos es trivial, se procede a listar los distintos grupos de predicados en base a su funcionalidad:

- **Tablero:** Los predicados de uso más extendido dentro del programa en sí, permiten el manejo del tablero para realizar diversas tareas básicas como la lectura o escritura de celdas, el conteo de piezas, entre otras.
- **Chequeo:** Predicados de uso auxiliar para los predicados principales de chequeo, revisando si existen movimientos posibles o capturas posibles para cada celda del tablero.
- **Captura:** Predicados de uso auxiliar para el predicado principal de movimiento, realizando las capturas correspondientes en cada posible dirección.
- **Movimiento:** Predicados de uso auxiliar para los predicados principales de movimiento y elección de mejor movimiento, explorando el espacio de potenciales movimientos en base a estados particulares. Se destaca que dentro de estos predicados, hay uno con la misma estructura y función que `hacer_movimiento`, el cual tiene como única diferencia la generación de una bandera de captura, utilizada por otros predicados para explorar los posibles movimientos y diferenciar aquellos que realizan capturas de los que no.

Se destaca que la mayoría de estos predicados son usados dentro de la lógica del juego, pero varios de ellos también son empleados por la lógica de la máquina.

1.2. Predicados principales

A continuación se adjunta una explicación superficial del funcionamiento de los tres predicados principales de manejo de juego:

- `hay_movimiento(+Estado,+Jugador)`
  Comprueba que el jugador `Jugador` tiene algún movimiento posible con alguna de sus piezas en el tablero indicado en `Estado`. Para ello, determina todas aquellas piezas que le pertenecen al jugador y para cada una de ellas, comprueba que tengan al menos una casilla libre adyacente. Ni bien encuentre la primer casilla que cumple esto, se termina la ejecución pues se sabe que el predicado es `true`.
- `hay_posible_captura(+Estado,+Jugador)`
  Comprueba que el jugador `Jugador` puede hacer al menos una captura con alguna de sus piezas en el tablero indicado en `Estado`. Para esto, el predicado busca alguna pieza del rival que sea "capturable" en un movimiento (la pieza rival tiene de un lado una pieza del jugador, y la otra casilla adyacente en el mismo sentido se encuentra vacía; además esa última casilla es alcanzada por una pieza del jugador). Por regla del juego, no se toma en cuenta a la pieza en el centro del tablero.

- `hacer_movimiento(+Estado, ?FilaOrigen,?ColumnaOrigen,?FilaDestino,?ColumnaDestino, +TipoMovimiento,-Estado2)`
  Partiendo de un estado original `Estado`, el predicado permite la realización de un movimiento para una pieza en las coordenadas `(FilaOrigen, ColumnaOrigen)` hacia la celda con coordenadas `(FilaDestino, ColumnaDestino)`. El estado resultante se retorna en `Estado2`.

  La lógica del predicado se divide en tres grandes secciones:

  1. **Validación:** Se realizan las validaciones pertinentes respecto a si el movimiento es posible o no (que la celda origen no esté vacía, que la celda destino sí lo esté, que las celdas sean adyacentes entre sí y que si el movimiento es con captura, que haya captura posible).
  2. **Ejecución:** Se modifica el tablero para realizar el movimiento en cuestión.
  3. **Captura:** En caso de que producto al movimiento realizado se produzca una o más capturas, se modifica el tablero para reflejar el resultado. Si en `TipoMovimiento` se indica que sea *con_captura* pero no se realizó ninguna, se invalida el movimiento.

2. Lógica de la máquina

A continuación se tratan todos los temas relacionados al desarrollo de la lógica de la inteligencia artificial y del cuarto de los predicados principales de la consigna.

2.1. Predicados auxiliares

En este escenario el enfoque fue un poco distinto, resolviéndose la implementación de varios predicados auxiliares tanto para la elección de movimiento en la fase 1 como en la fase 2.

Con respecto a la fase 1, se implementaron predicados para la elección de piezas siguiendo dos estrategias bien distintas: una genérica (denominada *dummy* en el código) y una basada en una heurística (denominada *minimax* en el código, con el fin de agruparla con la estrategia minimax de la fase 2). Se destaca que la estrategia genérica recorre el tablero de izquierda a derecha y de arriba a abajo, buscando las primeras dos casillas libres. Por el contrario, la estrategia basada en heurística, parte de la premisa que considera mejores a las celdas cercanas al centro. Teniendo esto en cuenta, la cercanía a la casilla central está determinada la distancia Manhattan, y la heurística buscará la casilla de menor distancia para cada instancia del tablero.

Con respecto a la fase 2, se siguió la misma idea y se implementaron predicados para la elección de piezas siguiendo también dos estrategias: una genérica (denominada *dummy* en el código) y una basada en el algoritmo minimax (denominada *ia_06* en el código). La primera, recorre el tablero de izquierda a derecha y de arriba a abajo, buscando la primera pieza del jugador que es capaz de moverse, y acto seguido, ejecuta el movimiento. En caso de capturar, ignora la posibilidad de seguir capturando y pasa. Por otra parte, la segunda estrategia considera todos los movimientos posibles y se elije la mejor en base a una heurística. En caso de que `Nivel` sea mayor a 1, se continuará anailizando movimientos entre turnos, y luego se calculará el puntaje con la heurística preestablecida. Debido a esto, esta estrategia presenta una complejidad bastante mayor, por lo que se desarrollaron predicados auxiliares de distintas índoles.

Por una parte, se implementaron tres predicados principales para la ejecución recursiva de minimax:

- `minimax(+Nivel,+Alpha,+Beta,+Jugador,+EstadoBase,-EstadoFinal,-Puntaje)`
  Llamada principal del algoritmo, la cual sigue una estrategia recursiva generando el árbol minimax y calculando los puntajes correspondientes. El predicado determina que una hoja es cualquier nodo que cumple alguna de las siguientes tres condiciones: `Nivel = 0`, `EstadoBase` es un estado final (es decir, el juego terminó) o `Jugador` no tiene movimientos posibles. En cualquiera de estos casos, se calcula el puntaje de la hoja llamando `calcular_puntaje_minimax_hoja`. En el caso contrario, se determina que el nodo actual es parte de una rama, se calculan todas las posibles jugadas con `calcular_posibles_estados` y se llama a `calcular_puntaje_minimax_rama` para seguir con la recursión.
- `calcular_puntaje_minimax_hoja(+Estado,-Puntaje)`
  Calcula el puntaje del estado pasado por parámetro tomando como heurística la diferencia de piezas de cada jugador. Sin importar que jugador este anallizando el tablero final, el puntaje se calcula como las piezas de `X` menos las piezas de `O`.

- `calcular_puntaje_minimax_rama(+Nivel,+Alpha,+Beta,+MejorEstado,+Jugador, +Estados,-EstadoFinal,-Puntaje)`
  Teniendo en cuenta que `Estados` es la lista con posibles jugadas a partir del nodo padre, este predicado explora el espacio de soluciones tanto en profundidad como en amplitud. La lista de estados se recorre uno por uno, explora en profundidad calculando su puntaje al ejecutar `minimax` de manera recursiva y actualizando los valores de `Alpha` y `Beta` en caso de que así sea necesario. Se realiza la poda necesaria en caso de que la condición `Alpha >= Beta` se cumpla, y si no, se sigue explorando en amplitud la lista de jugadas para el mismo nodo.
  Como detalle final, al llegar a la última jugada de la lista, se calcula si dicha jugada tiene un subárbol con mejor puntaje que el anterior, sustituyéndose el parámetro `MejorEstado` en tal caso. Este procedimiento se realiza comparando cada elemento de la lista de adelante hacia atrás, hasta llegar a la llamada original y arrastrando la mejor jugada en `MejorEstado`, pudiendo así pasarsela al nodo padre. De esta manera, se evita que la llamada original de `minimax` devuelva el estado final resultante de toda la simulación (el nodo hoja en el análisis), devolviendo únicamente la primer jugada.

Luego, se implementaron algunos predicados de uso más puntual dentro del ciclo de generación del árbol minimax, como por ejemplo predicados para la inicialización de valores de *alpha* y *beta*, el calculo de dichos valores según cierto puntaje obtenido, el chequeo de un estado final y el cálculo deñ puntaje de un tablero, entre otros.

2.2. Predicados principales

En este caso, el único predicado principal es el siguiente:

- `mejor_movimiento(+Estado,+Jugador,+NivelMinimax,+Estrategia,-Estado2)`
  Partiendo del estado `Estado`, elige el mejor movimiento para el jugador `Jugador` siguiendo la estrategia `Estrategia` y retornando el estado resultante en `Estado2`.
  Se destaca que existen dos versiones de este predicado: la versión genérica o *dummy* y la versión *minimax*; y cada uno de estos distingue la fase del juego según el `Estado` pasado como parámetro. La implementación con la `Estrategia` *minimax* se sirve de los predicados mencionados en la sección anterior, para elegir el mejor movimiento en base al puntaje obtenido de la exploración del árbol minimax.

2.3. Diseño

A continuación, se adjuntan comentarios respecto a la implementación del algoritmo minimax en el escenario particular del laboratorio, centrándose en las decisiones de diseño tomadas.

Tal como se comentó anteriormente, debido a la forma en que se implementó el algoritmo, este es capaz de identificar los movimientos donde captura piezas y estos tienen preferencia en contraste con los que que no realizan ninguna captura. Además, también es capaz de encontrar la mayor cantidad de capturas que puede realizar en un turno. Si un jugador en su turno no tiene movimientos de captura, se optará por alguno de los otros movimientos según el puntaje calculado. Se debe comentar que en esta situación, no es muy capaz de hacer jugadas "defensivas", como mover una pieza suya para evitar que el rival la capture.

En general se implementa el algoritmo minimax donde el estado inicial del juego, el jugador lleva adelante el rol de *maximización*, y por tanto, el siguiente el de *minimización*. En este caso para facilitar la implementación del algoritmo, independientemente quien invoque *mejor_movimiento* los turnos del jugador X se buscará **maximizar**, mientras que el jugador O buscará **minimizar**. Para que esta perspectiva se ejecute correctamente, el puntaje siempre será la resta de la cantidad de piezas X menos las piezas O.

El predicado `calcular_posibles_estados` es el usado para que el algoritmo minimax explore las opciones posibles en cada nivel. El resultado del predicado es una lista de estados donde se hizo algún movimiento. Según las reglas del juego, el jugador tiene la posibilidad de pasar o seguir capturando (siempre que sea posible) una vez que se capturó una pieza. Si bien esto es una opción, se optó por retornar un estado donde se capture todo lo posible, considerando que capturar es mejor que pasar según la heurística empleada para el cálculo de puntaje dentro de minimax.

Por el mecanismo del juego, puede suceder que el jugador haga dos movimientos distintos con la misma pieza donde estos capturen piezas. La implementación realizada solo considerará la primer opción de movimiento, ignorando todas las demas.

Finalmente, resta comentar sobre el uso del predicado *cut (!/0)*. Este predicado fue de gran utilidad en el desarrollo de toda la tarea. Ya sea para simplificar la implementación de los predicados de forma tal que no se tengan que realizar tantos controles, asegurar una única invocación del predicado cuando se podría haber unificado en más de una instancia (asegurándose que se retorne solo un valor posible), retornar valores de respuesta lo antes posible (por ejemplo, cuando se valida que existe un posible movimiento, el predicado devuelve éxito al encontrar la primer pieza del jugador que tenga un movimiento válido y no espera a recorrer todas las posibilidades), y lo máas importante, hacer los cortes de poda en la ejecución de la recorrida del árbol producido por el minimax.

2.4. Performance

Como pautas generales, se realizaron pruebas de tres índoles, con el objetivo de determinar un funcionamiento apto para la entrega. Dichas características son:

- **Funcionalidad:** Pruebas básicas dentro de prolog, comprobando el funcionamiento de la máquina en casos borde donde podría intentar ejecutar acciones erróneas o no encontrar los caminos más obvios.
- **Tiempo:** Pruebas de tipo experimental, ejecutando el algoritmo minimax contra humanos y contra sí mismo, y midiendo el tiempo de ejecución obtenido empíricamente.
- **Calidad:** Pruebas de tipo experimental, corriendo el algoritmo minimax contra humanos y contra sí mismo, y observando las estrategias y enfoques tomados por la máquina, así como los resultados obtenidos al finalizar las partidas.

En relación al tiempo de ejecución, teniendo en cuenta que para nivel 5 de minimax no se detectaron demoras importantes, se asume correcto el comportamiento en dicha área.

```
?- time((mejor_movimiento(estado(m(f(-,-,o,o,x),f(x,-,x,-,o),f(x,x,-,x,x),f(-,-,x,-,x),f(x,x,o,x,o)),0,0,0,2,2), o, 1, minimax, Estado2), fail; true)).
% 1,698 inferences, 0.001 CPU in 0.001 seconds (100% CPU, 1320239 Lips)
true.

?- time((mejor_movimiento(estado(m(f(-,-,o,o,x),f(x,-,x,-,o),f(x,x,-,x,x),f(-,-,x,-,x),f(x,x,o,x,o)),0,0,0,2,2), o, 3, minimax, Estado2), fail; true)).
% 156,494 inferences, 0.054 CPU in 0.058 seconds (93% CPU, 2889540 Lips)
true.

?- time((mejor_movimiento(estado(m(f(-,-,o,o,x),f(x,-,x,-,o),f(x,x,-,x,x),f(-,-,x,-,x),f(x,x,o,x,o)),0,0,0,2,2), o, 5, minimax, Estado2), fail; true)).
% 3,775,776 inferences, 0.644 CPU in 0.654 seconds (98% CPU, 5866576 Lips)
true.
```

En relación a las podas, no se contrastó el funcionamiento del algoritmo sin las mismas, ya que se consideraron parte crucial del algoritmo en cuestión y no sólo se trataron como forma de optimizar la velocidad del código. No obstante, los índices de velocidad obtenidos dan a entender la optimización en el uso de las podas

En relación a la calidad de la máquina, corriendo 10 partidas en tableros aleatorios, se observó que los resultados se encontraron bastante equilibrados. El jugador `X` tuvo una pequeña ventaja sobre el jugador `O` al ser el que comenzaba, traduciéndose esto en resultados relativamente mejores (en general, un par de partidas más ganadas con respecto a `O`). Por otra parte, se observó una gran cantidad de empates, la cual aumentó considerablemente al bajar el nivel de `X` y aumentar el de `O`. Se concluye que esto refiere a que la capacidad de ganar de `O` es bastante limitada teniendo en cuenta la calidad actual del algoritmo y las condiciones de comienzo de la partida.

En relación a la calidad detectada de manera empírica, se observó una convergencia al empate al entrar en un bucle infinito de movimientos inútiles. Debido a la forma en que el algoritmo está implementado, este favorece en decidir movimientos de captura a los que que no lo hace. Además la heurística es bastante sencilla ya que considera sólo la cantidad de piezas de cada jugador, y no toma en cuenta estrategias a más largo plazo como la capacidad de atrapar casillas o jugar a la defensiva evitando que el rival logre capturas. Cuando los jugadores se encuentran en un escenario donde no tienen capturas en un futuro cercano, el juego se convierte en un bucle infinito de ambos jugadores realizando siempre el mismo movimiento uno y otra vez, llevando esto a que la mayoría de juegos resulten en empate. No obstante, la cantidad de empates entre cada simulación rondó el 60 %.