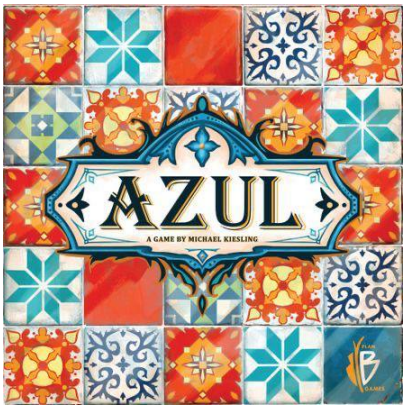
PROGRAMACIÓN DECLARATIVA

PROYECTO # 1: PROLOG.
JUEGO AZUL



Ejecutores:
Marcos Manuel Tirador del Riego
Laura Victoria Riera Pérez

Grupo: C-311

Date: 29 de mayo de 2022.

Nuestra solución esta separada en 4 módulos fundamentales, *game.pl*, *strategy.pl*, *board.pl* y *wall.pl*, que se consultan todos a la vez en cada sección del programa. El programa comienza consultando *game.pl* y se hace uso del predicado predefinido de prolog *consult* para cargar todas las cláusulas de los otros módulos. Se cuenta además con el archivo adicional *tools.pl*. Las funcionalidades que agrupan cada uno de estos ficheros serán explicadas a continuación.

board.pl:

El archivo *board.pl* es un pilar fundamental de este proyecto pues contiene todos los hechos que definen los elementos del juego, así como predicados dedicados a su manipulación según avance la partida. Almacenamos la información del tablero común a todos los jugadores en los predicados `center`, `factory`, `bag` and `discardPile`. En el encabezado de este archivo hay una descripción más explicativa de los mismos.

Tenemos además aquí la información individual de cada jugador, donde podemos encontar la puntuación, el muro que le corresponde, así como las filas de preparación. Finalmente, hay predicados para la obtención de información relevante de estas propiedades y para la manipulación de las mismas. Para actualizar el valor de una propiedad o hecho se hará uso de la metaprogramación, específicamente de los métodos `retract` y `assert` para cambiar algunos hechos por otros, que representarán estados actualizados del juego.

strategy.pl:

En *strategy.pl* encontramos las diversas estrategias utilizadas para implementar los jugadores y su elección de jugada.

tools.pl:

En *tools.pl* encontramos una serie de predicados que nos ayudan a tener ciertas funcionalidades generales en una forma más simple. Por ejemplo, contamos con predicados para el trabajo sencillo con listas (indexar, contar o remover elementos, comparar dos listas), para simular una cadena de sentencias if-else, llamar a otro predicado sin imprimir nada en el output, entre otros varios.

Wall.pl:

Wall.pl contiene una serie de predicados dedicados a brindarnos información sobre los estados de los muros. Aquí se puede saber si hay filas o columnas completas, si ya la pieza de un color de una fila está, cuánto se obtendrá al final del juego por completar filas y columnas, entre otros.

Game.pl:

El archivo *Game.pl* contiene los predicados encargados de controlar la simulación del juego. Para comenzar la misma debemos escribir el siguiente objetivo

```
?- startGame(N).
```

donde `N` debe ser un entero entre 2 y 4, que indica la cantidad de jugadores de la partida.

La partida se ejecutará automáticamente hasta el final, mostrando una gran cantidad de resultados intermedios que no alcanzarán a verse en la pantalla completamente. Por tanto, si queremos ejecutar el juego por rondas podemos comentar el subobjetivo `nextStep(_)` dentro de la tercera de las cláusulas `nextStep(_)` definidas (aproximadamente en la línea 137), y luego preguntar por él manulamente cada vez que se termine una ronda.

Alternativamente, se puede comentar el objetivo de igual nombre en la segunda de las cláusulas definidas de `nextStep(_)` (aproximadamente en la línea 104), para ejecutar el juego una jugada a la vez.

También podemos preguntar por

```
?- reset(_).
```

para reiniciar el estado del juego y volver a jugar de cero con `startGame(N)`.

El flujo general del programa: Al comenzar la ejecución con *startstartGame(N)* creamos todos los hechos que determinan el estado de las fábricas inicialmente, del centro, de los tableros de los jugadores y sus puntuaciones. Todos las cláusulas para crear este contenido están definidas en *board.pl* y comentadas para su entendimiento. Los jugadores estarán determinados por un número del 1 al N, y se sucederán sus turnos cíclicamente en el orden natural.

Luego se pasa por el subobjetivo *nextStep/1* que comienza el flujo de la partida realizando al menos la primera jugada. Aquí primeramente se obtienen todas los grupos de ficha posibles a seleccionar, en forma de tuplas que contienen el color de las fichas, la procedencia (número de fábrica o el centro) y la cantidad. Esto se logra llamando a un predicado definido para esto en *board.pl*. A continuación se imprime información relativa al estado del juego antes de la jugada. Por último se comienza a procesar la siguiente jugada.

Para procesar la jugada se usa *makeMove* definido en *strategy.pl*. Este decide que estrategia se va a usar para calcular la siguiente jugada en dependencia del jugador que sea. Para esto accede a la información predefinida en los hechos *strategy/2*, los cuales deberán ser modificados a mano si se desea cambiar la estrategia de los jugadores. Lo próximo será calcular la siguiente jugada. Esto dependerá de la estrategia seleccionada y la cual determinará la cláusula de *calculateMove* que unifique correctamente. Finalmente se realiza esta jugada y se procede al próximo llamado a *nextStep/1*.

Si no quedan movimientos que hacer la cláusula de *nextStep* que unifique será la tercera. Aquí se actualiza la puntuación de los jugadores con *actualizeAllScores* en *board.pl*, el cual adiciona a cada uno el resultado de las fichas insertadas en el muro, y substrae lo relativo a los azulejos en la línea del suelo.

Luego se comprueba si el juego termino, imprimiendo los resultados del mismo si así es. De lo contrario se comienza la siguiente ronda y se continúa el ciclo hasta que termine la partida. **La estrategia:**

En principio se implementó un jugador aleatorio, que escogía de una lista de jugadas válidas, una al azar. Luego se implementó otro jugador, llamado *minmax*, el cual se tratará de explicar con detalle.

El objetivo del jugador de forma general es analizar todas las posibles situaciones que lleven a terminar la ronda, y darle una calificación a cada una de estas, para luego escoger en cada caso la mejor. Note que esto es una algoritmo de tipo min-max ya que si fijamos un jugador X, en cada paso se maximiza la puntuación de la situacion escogida por un jugador Y; si Y es diferente de X, esto de cierta forma minimiza la puntuación para X (se ve mejor cuando solo hay dos jugadores), y si no, maximiza la de X.

Dado que esta solución se hace extremadamente costosa en tiempo de ejecución, alternativamente definimos una profundidad máxima (`depth(X).`) que puede tener el subárbol por el cual se están buscando posibles situaciones.

Para analizar posibles situaciones lo que hacemos es remplazar los jugadores existentes por unos falsos, que jueguen como esperamos, y luego seguir la ejecución del juego normalmente. Cuando se alcance la profundidad deseada, entonces se calcula el movimiento que más nos aporte, y se procede a restablecer la situación del juego a como estaba antes de calcular esta jugada. Lo ideal sería poder hacer y revertir los movimientos sin cambiar mucha información, ya que al guardarla como hechos es muy costoso remplazarla muchas veces. Sin embargo, dada la dificultad del lenguaje y que implementamos este jugador sin cambiar nada de lo hecho en el juego, la única solución factible fue almacenar y restaurar en cada jugada todo el estado del juego. Esto hace que con 2 niveles de profundidad ya sea suficiente para que el programa se tome un tiempo en pensar cada jugada y por tanto demore en ejecutarse. Pero para profundidad 1 no demora nada. Si se desea cambiar la profundidad se puede modificar manualmente el hecho de nombre *depth* en *strategy.pl*.

Como poda a esta búsqueda de la mejor jugada, memorizamos algunos estados del juego en el hecho *snapShot* para luego comprobar si ya se calculó antes la mejor jugada para una situación y no tener que hacerlo nuevamente. Note que esto es efectivo ya que cuando se quiere calcular la mejor jugada en una situación, primeramente hay que calcular la mejor jugada para las situaciones que se derivan inmediatamente de esta luego de probar cada posible movida. Entonces cuando se decida que jugada realizar, el resto de jugadas en niveles de profundidad mayores al de la situación antes de la jugada, ya estarán fijos y no habŕa que recalcularlos.

Para calcular la puntuación a asignar a una situación del juego, la idea que seguimos se rige por algunas estrategias que nos hicimos luego de jugar varias veces Azul:

- No es conveniente que caigan fichas al suelo
- Llenar una fila en la próxima jugada es siempre una buena opcion cuando no interfiere con alguno de los otros enunciados aquí.
- Siempre que se pongan fichas en el muro, que sean lo mas agrupadas posible, o sea, no deben estar muy separadas.
- No se debe comenzar a llenar una fila de un color si hay alta posibilidad de que no podamos terminarla en este turno.
- No es conveniente tener varias filas sin completar a la vez.
- Si se dejan filas incompletas preferiblemente han de ser de un color que aporte bastante en el futuro, ya sea por completar requisitos para bonos o por aportar muchos puntos por adyacencias.

Finalmente, lo que hacemos es darle alguna puntuación positiva o negativa a cada una de estas situaciones de acuerdo a nuestro criterio. No se pudieron tener en cuenta todas, tal cual las mencionamos, pero sí varias de ellas.

No obstante los inconvenientes, se puede apreciar claramente como este jugador incluso con profundidad uno, tiene resultados muy altos respecto al de tipo *random*.

En *strategy.pl* encontarmos uns predicados `strategy/2` que podemos cambiar manualmente para asignar el tipo de jugador que será cada uno de los jugadores del 1 al 4 (si hay menos no importa).