

# Proyecto N°1

---

PROGRAMACIÓN EN EL LENGUAJE PROLOG

Chia, Guido Giuliano (L.U.: 117825)

Frizzera, Ignacio (L.U.: 117815)

Comisión de Cursado N°9

Lógica para Ciencias de la Computación – 17/05/2019

## Contenido

Introducción .....	3
Uso del Sistema .....	4
Ejecución del Programa.....	4
Uso del Programa.....	4
Interfaz del programa.....	5
Colocación de una piedra en una posición invalida .....	6
Finalización de partida y cálculo de puntajes.....	7
Desarrollo del Programa .....	8
checked(Player, Index) .....	8
encerradoActual(Player, Index).....	8
noEncerrado(Index).....	8
emptyBoard(Board).....	8
goMove(+Board, +Player, +Pos, -RBoard) .....	8
replace(?X, +XIndex, +Y, +Xs, -XsY).....	9
esValida(+Board, +Player, +Index) .....	9
buscarEliminarVecinosEncerrados(+Board, +Index, +Player, -NBoard) .....	9
checkEncerrado(+Board, +Index, +Player, +Opponent, +Liberty, -NBoard) .....	9
checkEncerradoCascara(+Board, +Player, +Opponent, +Liberty, -NBoard, +Index).....	9
estaEncerrado(+Board, +Player, +Opponent, +Liberty, +Index) .....	9
getVecinos(+Index, -IndexesVecinos) .....	10
vecinoEstaEncerrado(+Board, +Player, +Opponent, +Liberty, +ValuesVecinos, +IndexVecinos) .....	10
noEstaEncerrado(+Board, +Player, +Opponent, +Index) .....	10
getValueListOnBoard(+Board, +Indexes, -Values) .....	10
getValueOnBoard(+Board, +Index, -Value).....	10
getValueLista(+Columna, +Lista, -Value).....	10
getListIndex(+Index, +Board, -Lista).....	11
eliminarEncerradosActuales(+Board, -NBoard) .....	11
winPoints(+Board, -PWhite, -PBlack) .....	11
calcularPuntos(+Board, +Player, +Opponent, +Liberty, -P) .....	11
calcularPuntosAux(+Board, +Player, +Opponent, +Liberty, +Index).....	11
checkEncerradoSinUncheck(+Board, +Player, +Opponent, +Liberty, +Index) .....	11
getNext(+Index, -NextIndex) .....	12
checkedToEncerrado.....	12
checkedToNoEncerrado .....	12

Comunicación entre la interfaz web y Prolog .....	12
Conclusión .....	13

## Introducción

En el presente informe se detallarán todos los aspectos relevantes asociados al desarrollo del Proyecto N°1 de la asignatura *Lógica para Ciencias de la Computación* de la Universidad Nacional del Sur para la comisión del Primer Cuatrimestre del año 2019.

Para su resolución se debió implementar el juego GO: un juego de mesa para dos jugadores que consiste de un tablero de 19 líneas horizontales (filas) por 19 líneas verticales (columnas) y piezas llamadas piedras (fichas) de color blanco y negro.

El instructivo de cómo se juega a GO se encuentra en el archivo PDF del enunciado del proyecto.

El desarrollo del programa se dividió en dos partes:

- **Interfaz gráfica**: está consiste de una interfaz web, la cual fue implementada mediante tres lenguajes de desarrollo web: HTML, CSS y JavaScript. Esta interfaz fue desarrollada en su mayor parte por la cátedra, excepto un par de detalles que fueron modificados.
- **Lógica del juego**: fue implementada utilizando el lenguaje Prolog. Esto incluye todos los aspectos relacionados al funcionamiento de GO. Esta lógica correrá mediante un servidor, el cual será un servidor web Pengines.

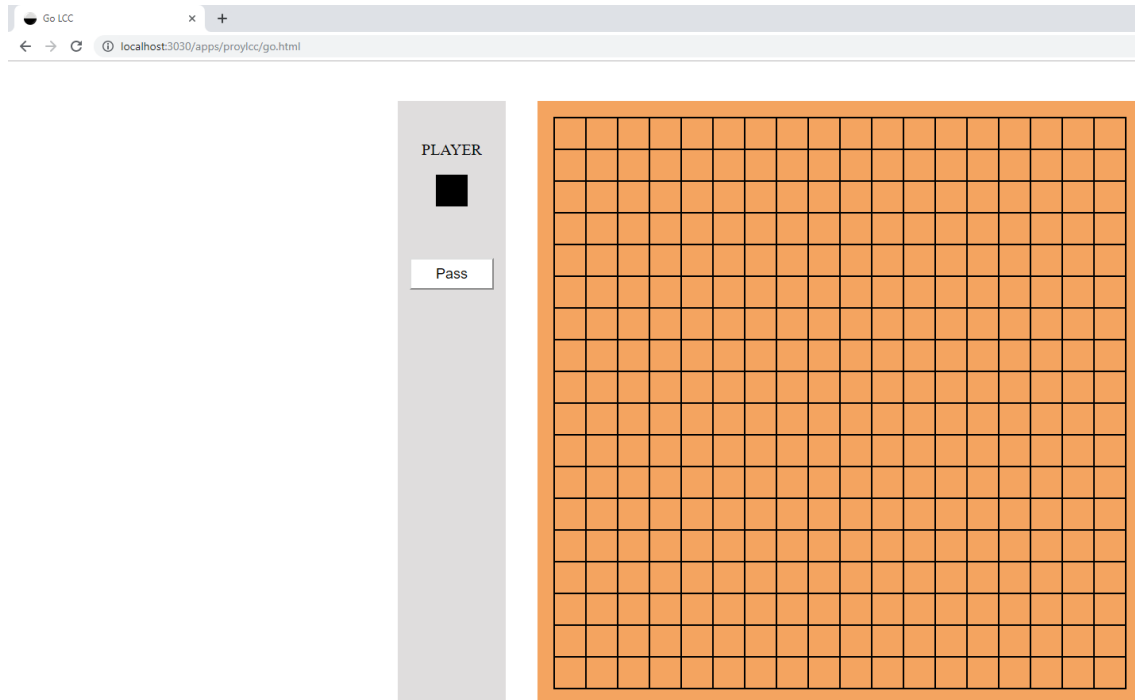
## Uso del Sistema

### Ejecución del Programa

Para comenzar, se debe tener instalado el programa *SWI-Prolog*, como también la implementación del servidor *Pengines*. Hecho esto se deberá ejecutar el archivo “run.pl” lo que levantará el servidor a utilizar, el cual escuchará en el puerto 3030 del localhost.

Una vez realizado lo mencionado anteriormente se deberá abrir en un buscador la siguiente url: <http://localhost:3030/apps/proylcc/go.html>, lo cual comenzará la ejecución del programa.

Una vez inicializado el programa se presentará la siguiente venta:



Hecho esto se podrá comenzar a jugar.

### Uso del Programa

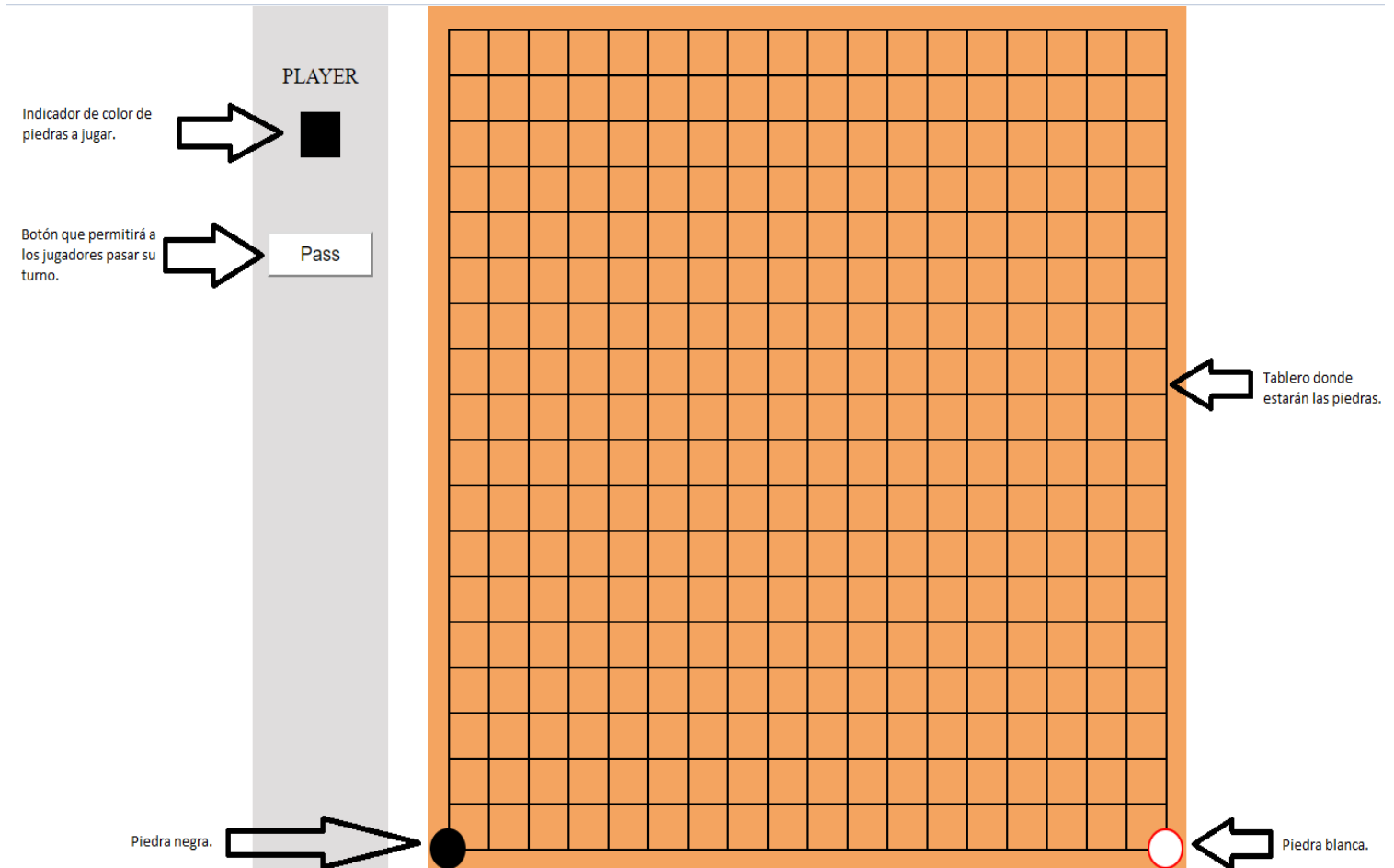
En principio, el programa presentará una interfaz la cual contendrá un tablero vacío donde se llevará acabo el desarrollo del juego. A medida que avance la partida este tablero se comenzará a llenar de piedras ya sean de color blanco o negro.

Contará con un cartel que diga “PLAYER” y de bajo de este un cuadro, el cual indicará que color de piedras toca jugar.

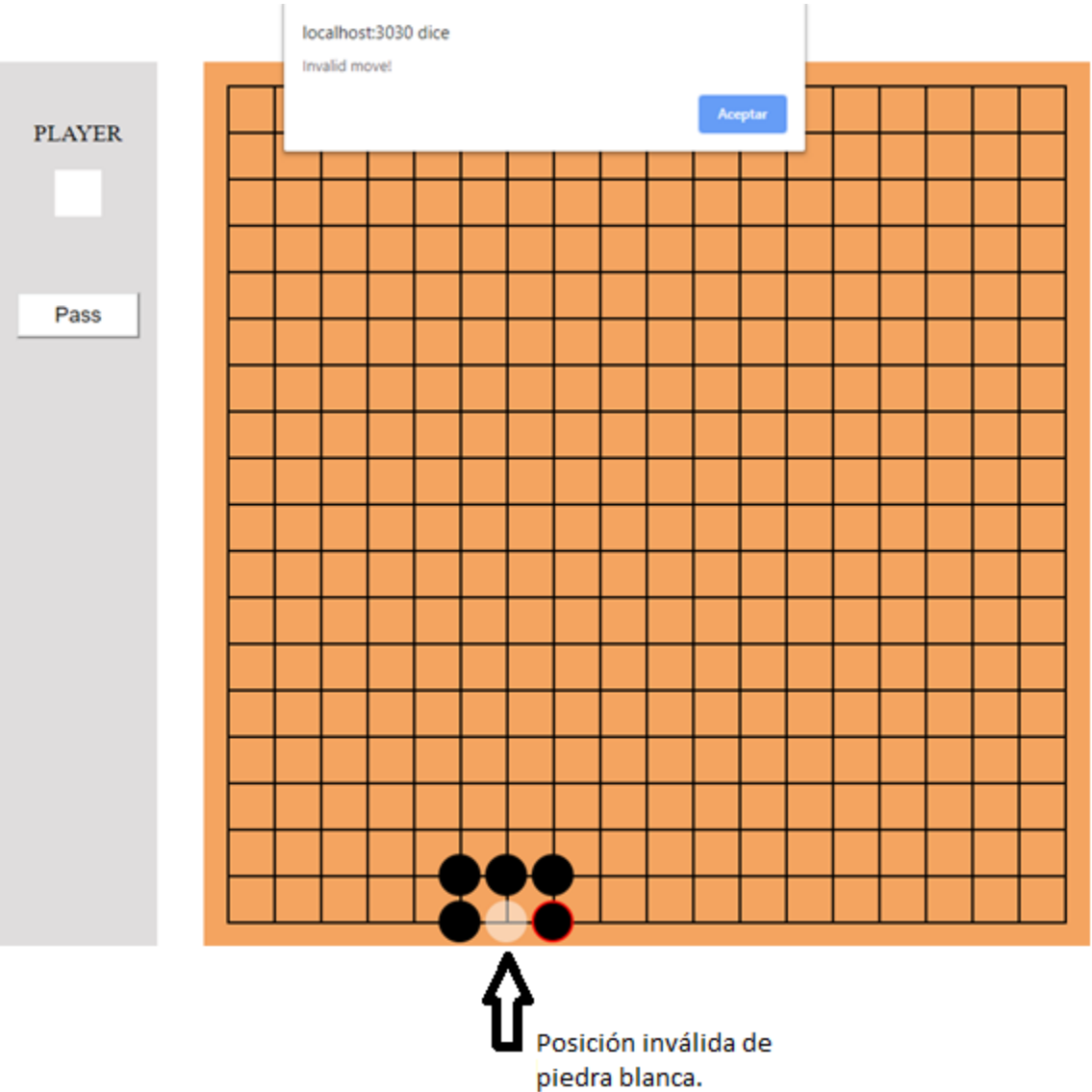
Para poder colocar una piedra se deberá hacer click en algún lugar del tablero que permita colocarla. En el caso de que el jugador no desee colocar una piedra, podrá hacer click en el botón “Pass” que le permitirá saltar su turno. En el caso de que tanto el jugador de piedras negras y el jugador de piedras blanca pasen de turno, la partida finalizará y se mostrará una notificación con los puntajes de cada jugador. Si se desea volver a jugar se deberá hacer click en el botón “Aceptar” de la notificación y así poder comenzar una nueva partida del juego.

En el caso que se desee colocar una piedra en una posición inválida el programa notificará al jugador de que está realizando un movimiento inválido y se anulará la colocación de la piedra, manteniendo el turno del jugador al que le correspondía jugar.

### Interfaz del programa




Colocación de una piedra en una posición invalida



Finalización de partida y cálculo de puntajes

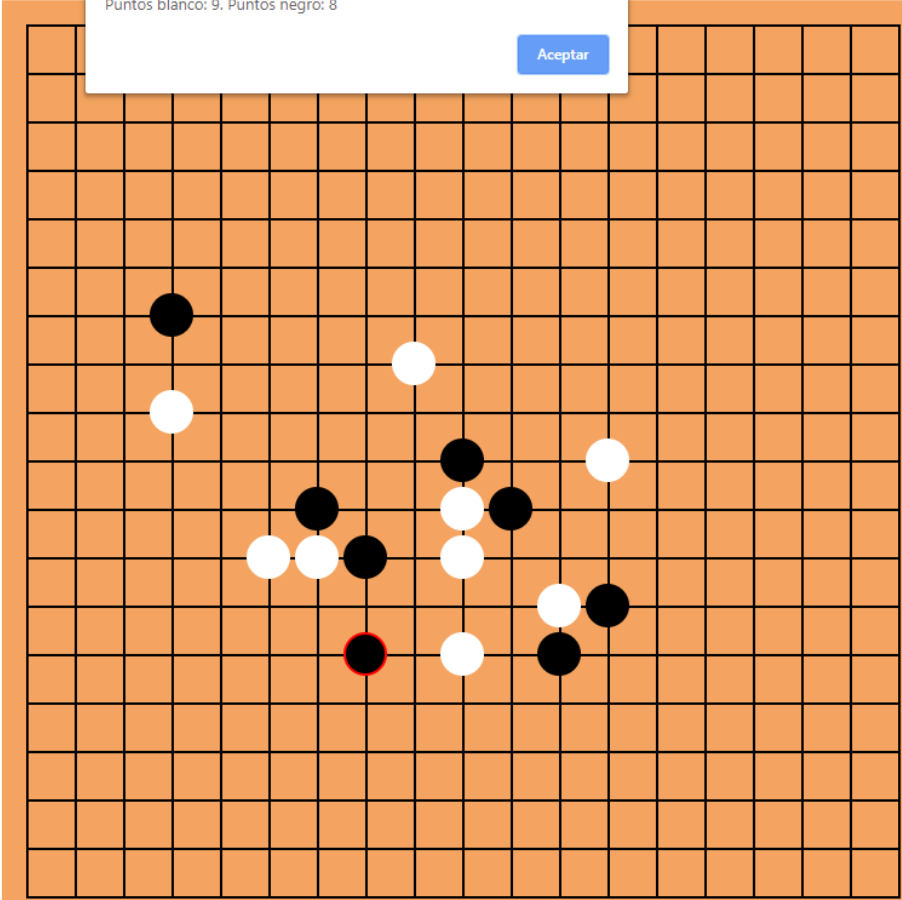
PLAYER



Pass

localhost:3030 dice  
Puntos blanco: 9. Puntos negro: 8

Aceptar





## Desarrollo del Programa

En esta sección se hablará puramente de la implementación en Prolog haciendo referencia solo a el programa en sí y no en cómo interactúa con la interfaz web.

### **Sea una regla o predicado a modo de ejemplo: *predicado(?X, +Y, -Z)***

El parámetro ?X representa que X podrá ser utilizado tanto como un dato de entrada como también un dato de salida.

El parámetro +Y representa que Y será utilizado como dato de entrada y será utilizado dentro de la regla o predicado.

El parámetro -Z representa que Z será utilizado como dato de salida y contendrá un valor de interés una vez finalizada la regla o el predicado.

### **Definición de algunos valores utilizados en el programa:**

- “o” : se utilizó para representar una posición que se encuentre fuera del tablero de juego. El objetivo es representar los vecinos de las piedras que se encuentren colocadas en los bordes de dicho tablero.
- “b” : se utilizó para representar al jugador que utiliza las piedras de color negro.
- “w” : se utilizó para representar al jugador que utiliza las piedras de color blanco.

Cabe destacar que cuando se mencionan los términos *posición* o *index*, se hace referencia a un lugar del tablero de la forma [FILA,COLUMNA].

Se optó la convención de describir cada regla o predicado presentados en el programa, con el objetivo de realizar una descripción clara y concisa de cada uno.

### checked(Player, Index)

Predicado definido de forma dinámica. Se utiliza para marcar como chequeada una posición visitada del tablero de juego mientras el jugador, el parámetro Player, está buscando qué posiciones ha encerrado ya. En este caso la posición visitada será Index.

### encerradoActual(Player, Index)

Predicado definido de forma dinámica. Se utiliza para marcar que una posición del tablero ha sido encerrada por el jugador, en este caso el parámetro Player. La posición a encerrar en si es el parámetro Index.

### noEncerrado(Index)

Predicado definido de forma dinámica. Se utiliza para marcar que una posición del tablero efectivamente no está encerrada, dicha posición es el parámetro Index. El objetivo de este predicado es facilitar y efectivizar el conteo de puntos, ya que se utiliza una vez haya finalizado la partida y se tiene la certeza de que el tablero no volverá a cambiar.

### emptyBoard(Board)

Predicado destinado a generar un tablero vacío. Genera un tablero vacío en el parámetro Board.

### goMove(+Board, +Player, +Pos, -RBoard)

Predicado destinado a reflejar lo causado por el movimiento de un jugador. Se reciben como parámetros el tablero en su estado actual, en este caso Board, el jugador a realizar el movimiento, recibido como Player, y la posición en la que Player desea colocar una piedra, la

cual se recibe como Pos. Una vez colocada la piedra de Player en la posición, Pos, que este haya elegido, se procede a analizar si con este movimiento Player encerró a alguna piedra del jugador contrario. En caso de ser así, se procede a eliminar la/s piedra/s y generar un nuevo tablero con estos cambios reflejados, dicho tablero es RBoard, en el caso contrario se devuelve en RBoard un nuevo tablero con la jugada realizada. Luego se verifica que la posición en la que se deseó colocar la piedra no haya sido un suicidio, en caso de ser así se devuelve en RBoard un nuevo tablero con los cambios reflejados, caso contrario el predicado devuelve false.

#### `replace(?X, +XIndex, +Y, +Xs, -XsY)`

Predicado utilizado para reemplazar en la lista Xs, en el índice XIndex, al elemento X por el elemento Y, devolviendo el resultado en la lista XsY. Dicho predicado fue implementado por la cátedra.

#### `esValida(+Board, +Player, +Index)`

Predicado destinado a analizar si la posición (Index) en la que el jugador (Player) desea colocar una piedra es válida, es decir no está cometiendo suicidio. Se utiliza el tablero (Board) para recorrerlo e ir chequeando las posiciones recorridas. Una vez finalizado el predicado se proceden a deschequear todas las posiciones marcadas.

#### `buscarEliminarVecinosEncerrados(+Board, +Index, +Player, -NBoard)`

Predicado utilizado para analizar si los vecinos, es decir las posiciones adyacentes, de una dada posición(Index) están encerrados, de ser así se los marcará como chequeados, de la forma encerrado. Se utilizará el tablero (Board) para analizar y recorrer todas las piedras, y se utilizará al jugador (Player) para decidir si el jugador que está encerrando es el de piedras blancas o negras. Una vez finalizado el predicado se procederá a eliminar todas las piedras marcadas como encerradas y se retornará un nuevo tablero con los cambios reflejados, el cual es el parámetro NBoard.

#### `checkEncerrado(+Board, +Index, +Player, +Opponent, +Liberty, -NBoard)`

Predicado destinado a analizar si los vecinos de una posición (Index) se encuentran encerrados. Se utilizará el tablero actual (Board) para recorrer y analizar las posiciones, como también al jugador (Player) y el jugador que juega en contra (Opponent), además se utilizará un valor de libertad para analizar si la posición se encuentra libre, este valor se pasa como parámetro con el nombre de Liberty. Si analizando los vecinos se encuentra con un valor de Liberty se procede a finalizar el recorrido ya que una vez encontrado esto se concluye que no se encuentra encerrado el Index. En el caso de eliminar piedra/s se generará un nuevo tablero con los cambios efectuados con el nombre de NBoard, caso contrario se devolvera el tablero original (Board) dentro del parametro NBoard.

#### `checkEncerradoCascara(+Board, +Player, +Opponent, +Liberty, -NBoard, +Index)`

Predicado destinado a analizar si la piedra (Index) del jugador contrario (Opponent) se encuentra encerrada por el jugador (Player). Determina si el index pasado por parametro esta encerrado, en el caso de que index este encerrado, todas las posiciones recorridas previamente se marcan como encerradas. En el caso de que no este encerrado se realiza un retract sobre todas las posiciones recorridas desmarcandolas.

#### `estaEncerrado(+Board, +Player, +Opponent, +Liberty, +Index)`

Predicado destinado a verificar que una piedra de un jugador contrario (Opponent) esté encerrada por el jugador (Player). Comienza verificando si la posición (Index) está marcada

como encerrada, de ser así retornará true, caso contrario comienza recursivamente a analizar si los vecinos de Index se encuentran encerrados. En el caso de encontrar algún vecino de Index con el valor Liberty, lo que significa que algún vecino no está encerrado, se concluye que Index no se encuentra encerrado por Player y el predicado retorna false, caso contrario si todos los vecinos de Index se encuentran encerrados o son piedras de Player, se concluye que Index si se encuentra encerrado y devuelve true. Se utilizó el predicado predefinido *member* para verificar que el parámetro Liberty no pertenezca a la lista de valores de los vecinos de Index y así concluir si está encerrado o no.

#### `getVecinos(+Index, -IndexesVecinos)`

Predicado utilizado para obtener los vecinos, es decir posiciones adyacentes, de una posición dada (Index). Dichos vecinos se retornarán en una lista llamada IndexesVecinos.

#### `vecinoEstaEncerrado(+Board, +Player, +Opponent, +Liberty, +ValuesVecinos, +IndexVecinos)`

Predicado destinado a analizar de forma recursiva si los vecinos (IndexVecinos) de una piedra del jugador contrario (Opponent) se encuentran encerrados por el jugador (Player). El parámetro Liberty será el valor que indique que una piedra no se encuentra encerrada.

Al utilizar predicados definidos de forma dinámica, en este caso el predicado *checked*, para realizar el marcado de posiciones, aunque esté utilizado de forma que sea mutuamente excluyente, *checked* y *not(checked)*, a la hora de buscar soluciones alternativas esto puede cambiar ya que es dinámico y generar así caminos que no son correctos. Por esta razón se vio la necesidad de utilizar un cut asegurando así que no será posible obtener caminos de solución alternativos que no son correctos al momento que el programa realiza backtracking.

#### `noEstaEncerrado(+Board, +Player, +Opponent, +Index)`

Predicado destinado a analizar si en la posición Index no se encuentra una piedra del jugador contrario (Opponent) encerrada por el jugador (Player). En caso de que no se encuentre, se procede a desmarcar todas las posiciones que habían sido marcadas y el predicado devuelve true, caso contrario el predicado no desmarca las posiciones y devuelve false.

#### `getValueListOnBoard(+Board, +Indexes, -Values)`

Predicado destinado a generar una lista de valores (Values) mediante una lista de posiciones (Indexes) de las cuales se obtendrán los valores. Se pasan como parámetro el tablero actual (Board), el cual será recorrido, y las posiciones de las cuales se quiere obtener el valor (Indexes). Una vez finalizado el predicado se retornarán los valores deseados en una lista, la cual será el parámetro Values.

#### `getValueOnBoard(+Board, +Index, -Value)`

Predicado destinado a la obtención del valor de una posición dada (Index). Se utiliza el tablero actual (Board) para acceder a dicha posición. El valor obtenido se retornará en el parámetro Value.

#### `getValueLista(+Columna, +Lista, -Value)`

Predicado utilizado para la obtención del valor de la posición de la columna pasada por parametro ubicada en la lista (Lista). En el caso de que se haya encontrado una posición valida, se procede a retornar su valor en el parámetro Value, caso contrario se procede a devolver el valor "o" en Value.

#### `getListaIndex(+Index, +Board, -Lista)`

Predicado destinado a la obtención de la lista correspondiente a la fila representada por el parámetro Index. Se utilizará el tablero actual (Board) para recorrer la fila y obtener sus respectivos elementos. En el caso de ingresar un Index válido, se procederá a retornar la correspondiente lista en el parámetro Lista, caso contrario se retornará en el parámetro Lista una lista vacía.

#### `eliminarEncerradosActuales(+Board, -NBoard)`

Predicado utilizado para eliminar todas las posiciones marcadas como encerrados actuales del tablero actual (Board), luego se retorna un nuevo tablero que refleje todos los cambios aplicados en NBoard.

#### `winPoints(+Board, -PWhite, -PBlack)`

Predicado destinado a calcular el puntaje de ambos jugadores una vez finalizada la partida. El puntaje de cada jugador se calcula contando las posiciones en el tablero ocupadas por cada uno, sumado a la cantidad de posiciones que se encuentren vacías y que estén capturadas por un jugador. Se recibe el tablero actual (Board) como parámetro para poder recorrerlo y calcular dichos puntajes. El puntaje de las piedras de color blanco se retorna en el parámetro PWhite y el puntaje de las piedras negras se retorna en el parámetro PBlack.

#### `calcularPuntos(+Board, +Player, +Opponent, +Liberty, -P)`

Predicado destinado a calcular el puntaje de un jugador, en este caso del parámetro Player, buscando donde encierra piedras del jugador contrario (Opponent) y donde Player ocupa lugares. Este predicado luego llama a *calcularPuntosAux* brindándole la posición inicial del tablero para que luego *calcularPuntosAux* lo recorra y calcule recursivamente el puntaje. Una vez finalizado el predicado *calcularPuntosAux* se utilizará el predicado predefinido *findAll* para generar una lista con todas las posiciones que se encuentren encerradas actualmente por Player, una vez generada la lista el puntaje de Player será la longitud de dicha lista. El puntaje se retornará en el parámetro P.

#### `calcularPuntosAux(+Board, +Player, +Opponent, +Liberty, +Index)`

Predicado utilizado para calcular el puntaje de un jugador (Player), recorriendo el tablero actual (Board) de manera recursiva, comenzando por una posición dada (Index). El parámetro Liberty será el valor que indique que una piedra no se encuentra encerrada. El predicado recorre el tablero en busca de piedras enemigas y posiciones que se encuentren encerradas por Player utilizando los predicados definidos de forma dinámica para marcar dichas posiciones. Una vez recorrido todo el tablero finaliza, dejando marcadas así todas las posiciones marcadas por Player.

#### `checkEncerradoSinUncheck(+Board, +Player, +Opponent, +Liberty, +Index)`

Predicado destinado a analizar si en la posición brindada (Index) se encuentra una piedra del jugador contrario (Opponent) encerrada por el jugador (Player). En el caso de que la piedra se encuentre encerrada, lo cual se verifica mediante el predicado *estaEncerrado*, se procede a llamar al predicado *checkedToEncerrado*, el cual se encargará de marcar todas las posiciones previamente marcadas como encerradas. En el caso contrario, es decir la piedra no se encuentra encerrada, se utilizará el predicado *noEstaEncerrado* para verificarlo y luego el predicado *checkedToNoEncerrado* se encargará de marcar todas las posiciones previamente marcadas como no encerradas.

### `getNext(+Index, -NextIndex)`

Predicado utilizado para obtener el index o posición siguiente a un Index pasado como parámetro. En el caso de que Index no se encuentre en el extremo derecho de su fila, se retornará en NextIndex la posición a la derecha de Index, caso contrario se retornará la primera posición de la fila siguiente en NextIndex.

### `checkedToEncerrado`

Predicado destinado a marcar aquellas posiciones que estaban marcadas como chequeadas a encerradas. Finaliza una vez no encontradas más posiciones marcadas como chequeadas.

### `checkedToNoEncerrado`

Predicado destinado a marcar aquellas posiciones que estaban marcadas como chequeadas a no encerradas. Finaliza una vez no encontradas más posiciones marcadas como chequeadas.

## Comunicación entre la interfaz web y Prolog

El objetivo de esta sección es describir de forma general la interacción entre la interfaz web y el programa realizado en Prolog.

Durante la ejecución hay una constante comunicación entre la interfaz web, la cual está escrita en JavaScript, y la lógica del programa, la cual fue desarrollada en Prolog, esta comunicación se realiza mediante llamadas de la interfaz hacia la lógica, donde luego la interfaz realiza determinadas funciones en base a lo que obtiene de su comunicación con la lógica.

En la función *handleCreate()* la interfaz solicita un tablero vacío a la lógica, la cual mediante su predicado *emptyBoard* lo genera. Una vez obtenido el tablero la interfaz procederá a mostrarlo por pantalla.

En la función *handleClick(row, col)* la interfaz se encargará de manejar un click de un jugador sobre una posición del tablero, dicha posición como también todos los parámetros necesarios para realizar el predicado *goMove* son enviados a la lógica para que esta se encargue de analizar si el movimiento del jugador fue válido o no. Una vez analizado el movimiento la interfaz procederá a actualizar el tablero con el movimiento realizado.

En la función *pass()* la interfaz se encargará de analizar si la partida ha finalizado, es decir ambos jugadores han pasado de turno, o no. En el caso de no haber finalizado la ejecución del programa sigue su flujo normal, caso contrario la interfaz notifica a la lógica que se deben calcular los puntos, la cual los calcula mediante el predicado *winPoints*, y los obtiene para luego mostrarlos por pantalla y dando así finalizada la partida.

La función *handleSuccess(response)* es llamada cuando el servidor Penguin retorna true. Dicha función se encargará de actualizar el tablero en base a un movimiento de un jugador o mostrará el puntaje obtenido por ambos jugadores dando por finalizada la partida, qué se realiza dependerá del parámetro response recibido.

La función *handleFailure()* se encargará de mostrar un mensaje de alerta cuando el servidor Penguin retorne false.

## Conclusión

El desarrollo de este proyecto nos ha permitido incorporar nuevos conceptos sobre el ámbito lógico y cómo este se relaciona con el campo de las ciencias de la computación, además nos introdujo a un nuevo paradigma de programación, el paradigma lógico. También se aprendió de una manera general, el funcionamiento entre un servidor, la internet y un cliente, y los lenguajes involucrados en estos procedimientos.

Finalmente, queremos agradecer a todo el equipo de ayudantes de la cátedra por su excelente predisposición a la hora de responder a nuestras preguntas y por guiarnos tan eficientemente en la producción de este trabajo.