

Proyecto 1

LCC -

Informe

Comisión 33

Índice

- Trabajo realizado
- Arquitectura del proyecto
- Funcionalidades del proyecto
- Trabajo en React
 - Llamadas a Prolog desde React
 - Consideraciones u observaciones
- Trabajo en Prolog
 - Estrategia de resolución
 - Predicados principales utilizados
 - Predicados predefinidos de Prolog
 - Consideraciones u observaciones
- Manual de uso
- Ejemplo de uso

Trabajo realizado

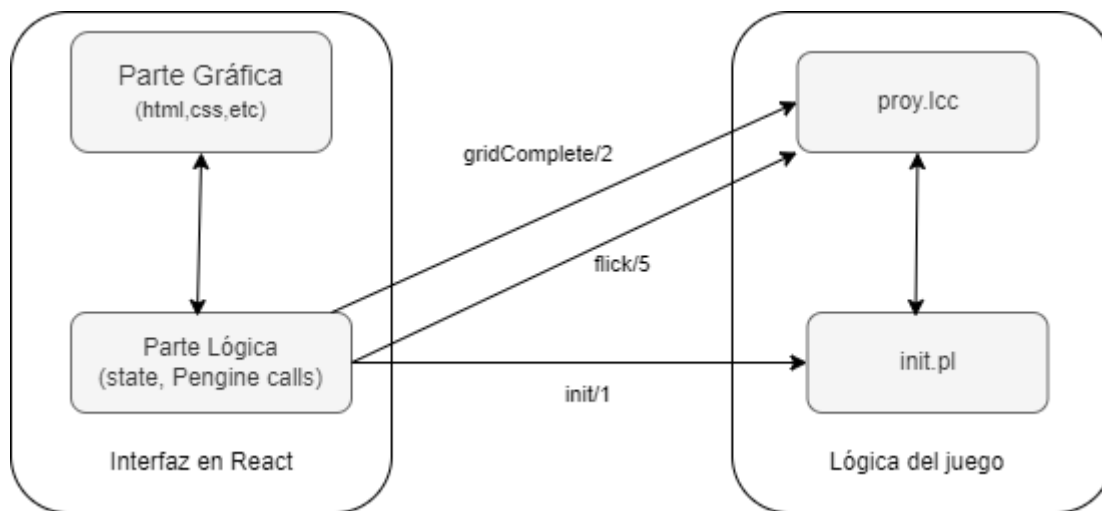
El proyecto consta de crear un juego con diversas funcionalidades, la premisa del juego es tener una grilla de 14 x 14 donde cada celda puede estar pintada de 6 colores distintos, además proveer botones que permiten seleccionar estos 6 colores. Luego de presionar un botón una de las celdas, llamada origen que es seleccionada por jugador o por defecto se asigna una, y esta cambia al color seleccionado dejando así una marca en el historial del color seleccionado y además contar con un nuevo número de turno actual y de celdas capturadas.

Las celdas cambian de color dependiendo si existe una relación entre la celda origen, esta relación se llama *adyacenteC* y esta se cumple cuando una celda es adyacente a otra y son del mismo color o existe una serie de celdas que sean *adyacentesC* de ambas. El juego finaliza cuando todas las celdas están pintadas del mismo color.

Fue provisto un repositorio en github con una implementación parcial del proyecto, lo que nosotros implementamos fue un cambio de estilización de algunos componentes, creación de nuevos métodos y contenedores (detallado en la parte de Trabajo en React) con la finalidad de poder brindar información para el usuario requerida por el enunciado de este proyecto. Además se trabajó en Prolog para poder establecer la lógica del cambio de la grilla luego de seleccionar un color y poder saber si el juego acabó o no.

La creación de este proyecto se realizó con Visual Studio Code versión 1.66.1 ,
SWI-Prolog 64 bits, versión 8.4 y Node.js versión 16.13.0.

Arquitectura del proyecto



Funcionalidades del proyecto

El proyecto brinda diversas funcionalidad requeridas por el enunciado enunciadas a continuación:

- Permite la selección de una celda para comenzar el juego, de no asignarse alguna se asume por defecto la celda superior izquierda de la grilla
- Se muestra al usuario a través de la interfaz el número de celdas capturadas actualmente , es decir, son adyacentesC* de la celda origen
- Se mantiene un historial de jugadas donde se almacenan los colores que involucran jugadas válidas durante una sesión de juego.
- Ofrece una breve explicación en pantalla de la funcionalidad del juego
- Detecta cuando el juego culmina y el jugador gana

Si bien el proyecto no permite explícitamente crear grillas de otro tamaño , lógicamente el proyecto es capaz de funcionar normalmente con cualquier dimensión de grilla, el único impedimento es la interfaz gráfica.

Trabajo en React

Se alteraron algunos archivos desde la clonación del repositorio provisto por el enunciado, uno de ellos es el *Game.js*, la modificación fue con al intención de que en su *state* almacenará nuevos valores que nos servirán para realizar nuestros objetivos entre ellos se encuentran los siguientes

- **history:** es un arreglo que guarda la letra principal de cada color utilizado, se usa cuando se quiere representar el historial.
- **origen:** guarda una *String* que representa la celda que se utilizara de origen durante la partida, por defecto se declara en [0,0] a menos que se declare lo contrario.
- **started:** condición booleana que comienza en false, nos permite identificar si el juego ya está en curso, se utiliza para saber si puede cambiarse o no la celda de origen.
- **capturados:** almacena el valor entero de las celdas capturadas por el jugador, lo utilizamos para saber si el juego se acabó.

Además de estos nuevos valores introducidos a *state* también se creó un método *assignOrigen* responsable de cambiar la celda de origen del juego, es decir el valor de *origen* en *state*, realiza un control de ver si la partida ya comenzó o no y avisa la celda elegida.

Se realiza una consulta a Prolog preguntando, *gridComplete/2* si el juego se acabó, es decir, la cantidad de celdas capturadas es igual a la dimensión de la grilla, o de otra manera, que toda la grilla tiene el mismo color.

Por último se agregaron nuevos contenedores, *div*, con sus respectivas clases con el fin de modularizar la creación del historial, matriz de botones que permite elegir la celda de origen y demás. Cada una de ellas se encuentra estilizada en el archivo de *index.css* pero eso escapa la implementación de **React**.

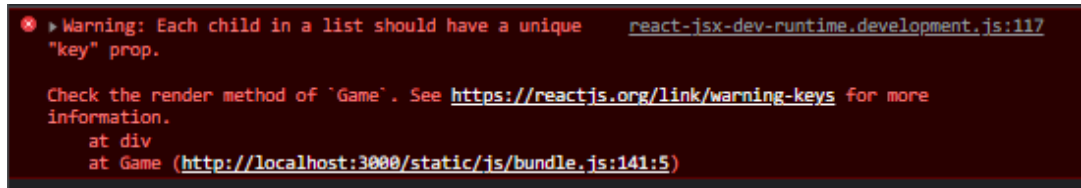
Llamadas a prolog desde React

flick(Grid,Origen,Color,FGrid,Capturados) : nosotros proveemos los valores de *Grid*, *Origen*, explicado previamente, y *Color*. Mientras que Prolog nos retorna la nueva grilla tras el Flick, llamada *FGrid*, la cual mostraremos al jugador y el nuevo valor de *Capturados*, que lo seteamos en el *state* y lo utilizamos en el chequeo de finalización del juego.

gridComplete(Grid,Capturados) : le pasamos la grilla que resultó del flick y luego la cantidad de capturados, esto nos informa si la grilla está completa con un solo color y determinamos si el juego acabó o no.

Consideraciones u observaciones

- Se realiza un mapeo sin key para generar el historial y la matriz de botones, si bien esto da un warning, no interrumpe la ejecución del programa. Probablemente existan alternativas más preferibles pero se tomó la decisión de implementarlo de esa manera.



```
Warning: Each child in a list should have a unique "key" prop.
    Check the render method of `Game`. See https://reactjs.org/link/warning-keys for more
    information.
    at div
    at Game (http://localhost:3000/static/js/bundle.js:141:5)
```

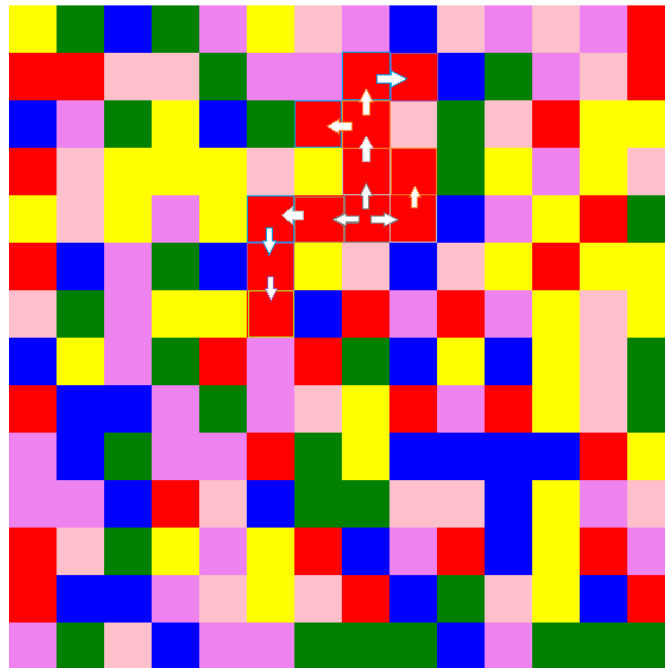
Warning generado por hacer un mapeo sin key

- La grilla de botones nunca se eliminó del programa, únicamente dejó de funcionar, probablemente existan maneras de eliminar componentes luego de ciertas acciones de React, pero a nuestro desconocimiento decidimos tomar la decisión de prevalecer con un fondo invisible e inutilizándola. Se puede ver porque el estilizado del botón se mantiene, **ejemplo** : `.origenBtn:hover`.
- Durante la instalación de *Node.js* se generaron conflictos que fueron resueltos utilizando `npm audit fix --force` especulamos que se debe a una diferencia de versiones.

Trabajo en prolog

Estrategia de resolución

La estrategia principal que se utilizó en el proyecto es evitar los llamados recursivos en vano, intentando crear un recorrido similar al de un grafo donde se descubren y se hacen llamados recursivos que se propagan descubriendo más celdas que cumplen con la condición deseada. Mientras se tuvo en cuenta la modularización y la generalización de los predicados, resultando esto en que la parte lógica del programa pueda funcionar para grillas de cualquier dimensión.



Ejemplo del recorrido del predicado que recorre en búsqueda de la cláusula transitiva adyacentesC, cada color hace referencia a una ejecución del predicado y la flecha que ejecución del predicado lo llamo, comenzando en la imagen en el gris [4,7].*

Predicados principales utilizados

Los predicados principales utilizados, su objetivo y una idea a alto nivel de su funcionamiento son :

- ***adyacentes([X,Y],LimiteX,LimiteY,L)***

Objetivo : dado una lista con un par de coordenadas , el límite de la grilla con respecto al eje X y al eje Y, se calculan dentro de los límites una lista con pares ordenados adyacentes y se los retorna en la lista L.

Funcionamiento: hace un leve cheque que la posición pasada este dentro del parámetros, delega el chequeo dentro de límites a otro predicado *getCoords/5* luego justa todas las coordenadas en una lista, si alguna coordenada se escapaba de los límites se le asigna esa coordenada como out, al finalizar antes de retornar la lista se limian las coordenadas que contienen “out”.

- ***getCoord(X,Y,LimiteX,LimiteY,E)***

Objetivo: dado dos coordenadas, X e Y, y además limites para esos valores, LimiteX y LimiteY, chequea si la coordenada esta o no dentro de lo limites, de estar la coordenada queda igual, de no estarlo asigna out. Devuelve E una lista con un par ordenado.

Funcionamiento: chequea que los valores X e Y no sean negativos ni que alcancen el limite de la grilla, si lo alcanzan entonces cuando se crea la lista E aquella/s posiciones que hayan estado lejos de los limites se asignan como “out” en la lista, sino se asigna su componente. Una componente en la lista retornaría E como [X,Y].

- ***adyacentesC(Grid,X,Y,LimiteX,LimiteY)***

Objetivo: calcula dado una Grid, grilla, pares de coordenadas X e Y, y limites de grilla LimiteX y LimiteY, si la celda en la posicion X de Grid es adyacenteC de la que se encuentra en la posicion Y, es decir, son del mismo color y son adyacentes.

Funcionamiento: delega o otro metodo probar si X e Y son adyacentes, de serlo se evalua si son del mismo color, de serlo se cumple.

- ***generateAdyacentesCTransitiva(Grid,X,L,LimiteX,LimiteY)***

Objetivo: prepara el programa para generar una lista con las posiciones de la clausla transitiva de adyacentesC

Funcionamiento: delega todo al metodo *adyacentesCTransitiva* pero antes hace un retractall(visitado(_)) con el fin de limpiar el recorrido antes de iniciarlo, manda como inicio del recorrido una lista con un unica lista con el par ordenado que representa las coordenadas de la celda a evaluar, X, en nuestro caso se utiliza como la celda origen.

- ***adyacentesCTransitiva(Grid,[X]Xs,L,LimiteX,LimiteY)***

Objetivo: dado una grilla , una lista de posiciones y los límites de esta grilla calcula todos los *adyacentesCTransitivos* a partir de todas las posiciones de la lista.

Funcionamiento: hace un *assert(visitado(X))* por la cabeza de la lista de posiciones X que se está evaluando, luego se pide por sus adyacentes que no se hayan visitado y se propaga de la siguiente manera:

- Si la lista de posiciones sin la cabeza no está vacía y la lista de adyacentes de X tampoco lo está entonces llamados *adyacentesCTransitivo* a la lista de adyacentes y el resto de posiciones, luego las juntamos.
- Si la lista de posiciones sin la cabeza está vacía y la lista de adyacentes no lo está entonces propagamos con *adyacentesCTransitivo* a la lista de adyacentes únicamente y retornamos la cabeza de lista con la lista que nos dio el metodo previo.
- Si la lista de posición sin la cabeza está vacía y la lista de adyacentes también entonces retornamos una lista con la posición que estamos evaluando, esto es porque ya no se puede propagar.
- Último, si la lista de posición sin la cabeza no esta vacía y la lista de adyacentes si lo esta entonces propagamos *adyacentesCTransitiva* a las posiciones restantes y luego retornamos la lista generada previamente con cabeza de la posición que estamos evaluando.

Si en algún caso se quiere evaluar alguna que ya estuvo visitada retorna una lista vacía.

- ***esAdyacente(X,Y,LimiteX,LimiteY)***

Objetivo: evaluar si los pares de coordenadas X e Y pasados por parámetro que se hallan dentro de los límites , LimiteX y LimiteY, son adyacentes.

Funcionamiento: genera la lista de adyacentes de X y luego ve si dentro de esta lista se encuentra Y, de estarlo entonces son adyacentes.

- ***flick(Grid, Origen, Color, FGrid,Capturados)***

Objetivo: dado una grilla,Grid, una posición de Origen, par ordenado [X,Y] y un Color representado con una letra , calcula una nueva grilla, FGrid, que es el resultado de cambiar todas las celdas *adyacentesC** de la celda Origen al color C, y también retorna el número de celdas Capturadas por *adyacentesC**.

Funcionamiento: primero evalúa que la celda de origen no se quiera cambiar al mismo color, luego descompone la grilla para conocer sus LimiteX y LimiteY con el fin de no calcularlos múltiples veces durante el juego, luego genera la cláusula transitiva de *adyacentesC* de la grilla y delega al método *flickColor/4* que se cambien las celdas en las posiciones que se encuentran en *adyacentesC**, luego esta nueva grilla se le vuelve a calcular los *adyacentesC** Para saber su dimensión, esa dimensión

de la lista será la cantidad de celdas capturadas, se hace después porque puede haber nuevas celdas capturadas desde el flick.

- ***flickColor(Grid, ListaPos, C, GridNew)***

Objetivo: dado una grilla, Grid, una lista de posiciones, ListaPos, y un color C, recorre toda la lista de posiciones cambiando el color de las celdas en esas posiciones al color C.

Funcionamiento: descompone las posiciones de la lista y las delega al método *setColor/4* para hacer el cambio y luego la grilla que resulta de ese predicado se hace llamado otra vez a *flickColor/4*, esto se repite hasta que no hay más posiciones.

- ***setColor(Grid, Coord, C, GridNew)***

Objetivo: dado una grilla, Grid, un par de coordenadas, Coords, y un color C, retorna una grilla nueva, GridNew, donde es idéntica a la grilla pasada por parámetro por excepción de la que se encuentra en la posición del par de coordenadas, que fue cambiada por el color C.

Funcionamiento: utilizando métodos auxiliares busca la lista que se desea que represente la primera coordenada y luego en esa lista busca la posición que representa la segunda coordenada, y luego recursivamente se devuelve la matriz con esa celda cambiada.

- ***gridComplete(Grid, Capturados)***

Objetivo: dado una grilla, Grid, y un número de celdas capturadas en esa grilla, Capturados, consulta si la grilla es toda del mismo color o no.

Funcionamiento: si bien no evalúa el color de la grilla, capturadas significa que fueron alcanzadas y se convirtieron en adyacentesC entonces son del mismo color, el metodo lo unico que hace es evaluar la dimensión de la grilla y calcular que la cantidad de celdas de esta grilla sea igual al número de celdas capturadas, de serlo se completó la grilla.

Predicados predefinidos de prolog

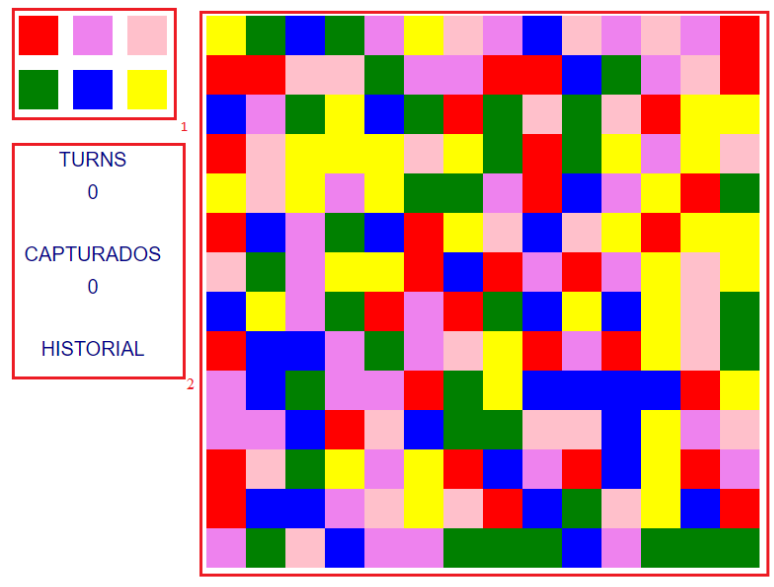
Se utiliza el predicado predefinido *length/2* para conocer la cantidad de elementos en una lista, se utiliza para calcular los **capturados** y la **dimensión** de la grilla, además se utiliza el predicado *nth0/3* para descomponer la lista de pares ordenados que representan las posiciones.

Consideraciones u observaciones

- Se calcula una única vez la dimensión de la grilla una única vez, por eso se pasan en ciertos predicados el **LimiteX** y **LimiteY** que denotan los límites de la grilla. La otra alternativa es calcularlas cada vez que se necesiten pero pensamos que es un desperdicio ya que predicados como *adyacentes([X,Y],L)* se utilizan múltiples veces y tendría que calcular los límites cada vez, igual que *getCoords(X,Y,E)*.
- Mientras se trabajó con el predicado de *adyacentesCTransitivo* se había llegado a otra implementación pero que no recorría bien los arcos por definir en *adyacentesC* su transitividad, con la que decidimos quedarnos no realiza eso sino que es mas optimo.
- Existe un caso en el que *adyacentesCTransitivo* visita un arco que cuando lo agrega a su lista de adyacentes por visitar no estaba marcado pero al volver recursivamente si lo está, se contempla dejando una lista vacía pero existe una alternativa que es marcar los arcos como visitados al pasarlos a la lista de adyacentes por visitar.

Manual de uso

Primera vista del juego:



Color flick

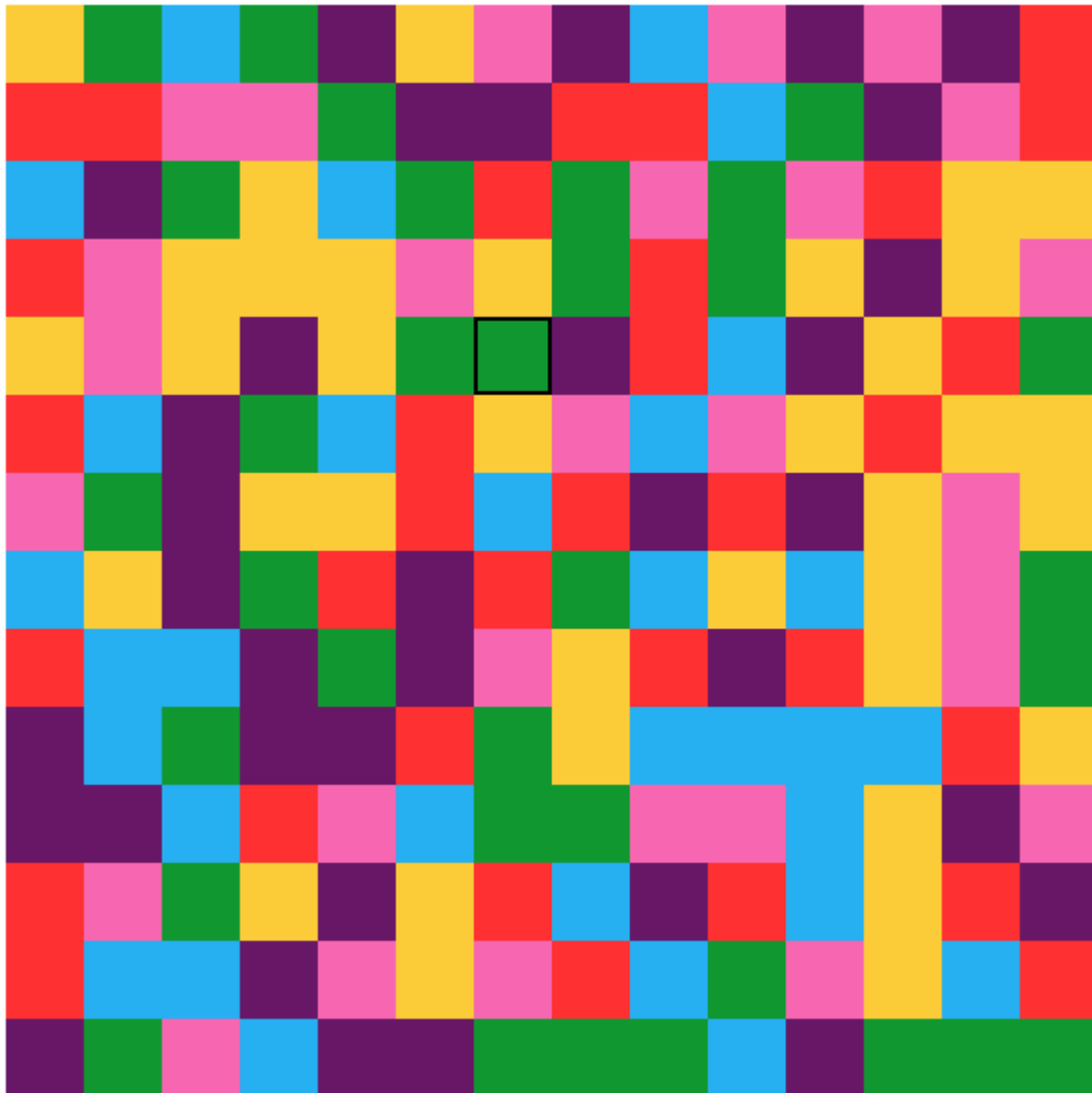
Para jugar primero debe seleccionar la celda donde desea originar, una vez seleccione una celda esta no puede ser cambiada, de seleccionar una el juego le asigna la celda en la parte super izquierda de la grilla.

El juego consta de ir seleccionado colores, estos colores cambian las celdas que son adjacentes y del mismo color a la celda origen, el juego se acaba cuando el jugador logra hacer que toda la grilla tenga el mismo color. A jugar.

1. Botones seleccionar color. Al clickear se cambian todas las celdas capturadas hasta el momento al color seleccionado. Clickear dos veces seguidas el mismo botón no va a hacer efecto, ya que solo se puede cambiar de color si el botón del turno actual es distinto al del turno anterior.
2. Información de la partida.
Se ven tres datos en pantalla: Turnos, Capturados e Historial. El primero se refiere a la cantidad de veces que cliqueamos para cambiar de color, el segundo lleva la cuenta de cuantas celdas llevamos capturadas, y el tercero nos va a ir mostrando con imágenes todos los colores que fuimos presionando
3. El tablero con las celdas. En este tablero tenemos las celdas que se van a capturar, primero se debe seleccionar con un click la celda inicial desde dónde se empieza el juego. De no hacerlo se iniciará por defecto en la celda de la esquina superior izquierda.

Ejemplo de como jugar:

Primero seleccionamos una celda:



En este caso vamos a elegir la celda en la posición (1,1)



Luego de eso podemos seleccionar en la tabla de botones algún color al que quiera cambiar la celda seleccionada, en este caso presionaremos el botón rojo:



Cambia de color amarillo a rojo



Como teníamos una celda roja abajo la capturamos, y como esta última tiene una celda roja a su derecha también la capturamos. Ninguna de las capturadas tiene al lado otra celda roja, por lo que necesitamos clickear en otro color para poder seguir capturando celdas. Hasta ahora llevamos un turno, 3 celdas capturadas y una movida en el historial.

TURNOS

1

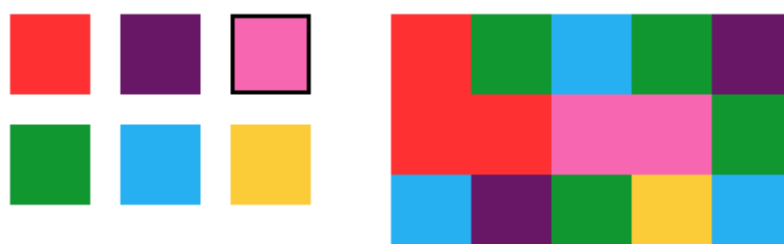
CAPTURADOS

3

HISTORIAL



Ahora movemos con otro color, en este caso el rosa:



Presionamos para cambiar



Al lado de una de las que capturamos hay una rosa, por lo que la capturamos, y como esta celda recién capturada tiene otra celda rosa adyacente también la capturamos. Los turnos, capturadas y puntajes quedan así:

TURNOS

2

CAPTURADOS

5

HISTORIAL



Luego seguimos moviendo con otros colores. Así quedaría si presionamos el verde:



TURNOS

3

CAPTURADOS

9

HISTORIAL



El juego termina en el momento en el que todas las celdas son del mismo color, lo cual se ve de la siguiente manera:



Obs: puede quedar de cualquier color dependiendo del orden en el que el jugador los haya clickeado.

Fuentes

El proyecto es una extensión del repositorio provisto por la cátedra que se puede encontrar en <https://github.com/LCC-DCIC-UNS/tic-tac-toe/tree/flick> .