TEC | Tecnológico de Costa Rica

Área académica de ingeniería en computadores

Lenguajes, Compiladores e Intérpretes | CE3104

Grupo 1

Tarea 1: Programación funcional con Racket

TicTacToe

Profesor: Marco Rivera Meneses

Estudiantes:

- Jose Antonio Espinoza Chaves - 2019083698
- Eduardo Zumbado Granados – 2019003973

I Semestre 2022

Índice

Algoritmos y arquitectura desarrollada

Se utilizó un algoritmo goloso para elegir la mejor opción de movimiento en cada turno de la máquina, para que de esta forma se tenga una solución óptima al final de la partida y se logre ganar. Primeramente, se debe encontrar la lista de los posibles candidatos para la solución local, esto se realiza mediante el método *findCandidates*, el cual solicita al método *findPlacedTiles* la lista de entradas de la matriz donde se encuentran las fichas del jugador y de la máquina. Esta lista de candidatos está constantemente actualizándose en cada método de entrada del jugador o de la máquina y se pasa entre funciones mediante parámetros, por lo que siempre está actualizado.

Inmediatamente después de que el usuario termina su turno se ejecuta la función runGreedyAlgorithm, la cual inicia la ejecución del algoritmo voraz. Esta llama a otras funciones para determinar cuál sería la posición más idónea para colocar la ficha, tomando en cuenta sus fichas y las del jugador. Para esto llama a la función de viabilidad utilizando la matriz, los candidatos de la matriz, la lista de fichas colocadas por el jugador y la lista de fichas colocadas por la máquina.

Para la segunda parte del algoritmo se determina la viabilidad de los candidatos, para así elegir el mejor movimiento en el turno. En este caso se revisa si se puede ganar con una combinación horizontal, vertical o en línea recta, sin embargo, se analizan también los casos en los que la máquina pueda perder en cualquiera de las direcciones anteriores, de esta manera, revisa si puede ganar, si es así revisa si la celda en la que puede ganar está disponible, de ser así pone la ficha para ganar la partida. En caso de que en ningún lado pueda ganar, se pone a revisar la lista del jugador a ver si el jugador está a punto de ganar en el siguiente turno, de ser así, pone la ficha en la posición donde el jugador iba a ganar protegiéndose así de perder. De no aplicarse ninguno de estos casos, pone la ficha en una de las posiciones de las celdas disponibles, la cual está dada por la lista de candidatos.

Funciones implementadas

Para las funciones implementadas se mostrarán solo las funciones de la sección lógica del programa para evidenciar el uso de la programación funcional, ya que el programa de la interfaz utiliza expresiones no nativas del lenguaje funcional.

(GenerateLogicMatrix n m)

Función que valida que los valores introducidos de n y m están entre 3 y 10, si es así llama a la función createMatrix.

(createMatrix n m matrix)

Crea la matriz de manera recursiva, para cada columna especificada por el usuario, añade una fila de tamaño n llamando a la función createRow.

(createRow n row)

Crea las filas de la matriz, es llamada por cada columna de la matriz, y recursivamente añade ceros en casa fila de la matriz, de manera que se obtiene al final una matriz m*n llena de ceros (valor de vacío).

(executePlayerSelec matrix m n mMax nMax)

Encargada del movimiento hecho por el jugador. Revisa que ambos valores, m y n no sobrepasen el valor máximo (dimensiones especificadas), si lo hace llama a errorNotification, que despliega una ventana emergente con el error. Si no sobrepasa el valor, verifica que la ficha está disponible y llama a la función placePlayerSymbol para colocar la X.

(placePlayerSymbol m n counter mMax oldMatrix matrix newMatrix element)

Encargada de colocar la ficha del jugador de manera lógica. Primero verifica si la posición de m es igual al contador, de ser así, significa que la llamada recursiva actual es igual a la posición de columna correcta, por lo que coloca el símbolo en esa columna y en la fila especificada. Para esto, se llama recursivamente aplicando cdr a oldMatrix y agrega a newMatrix las columnas con el símbolo ya puesta en la posición correcta de n. Si el contador es diferente de m, significa que no esta en la columna correcta aún, por lo que se llama recursivamente sumando 1 al contador y

aplicando cdr a la matriz vieja, y de igual manera agrega la columna actual a la nueva matriz. Para el caso base, se tiene que cuando el contador es mayora mMax, significa que la matriz entera ha sido recorrida, reconstruida y actualizada a partir de oldMatrix. Cuando esto ocurre, se llama a updateMatrix para actualizarla.

(placeTileInRow nPos counter nMax row newRow element)

Coloca la ficha en la columna especificada, y obtenida de placePlayerSymbol. Trabaja de manera parecida a placePlayerSymbol, ya que reconstruye la nueva fila a partir de la vieja recursivamente hasta que el contador sea igual que el valor n especificado. Cuando esto pasa, el valor es cambiado por el elemento. El caso base se da cuando el contador es mayor que nMax, cuando esto pasa, sabemos que la fila fue correctamente recorrida y reconstruida, y retorna la fila actualizada a placePlayerSymbol, para que pueda seguir con la reconstrucción de la matriz.

(printCheck message)

Función de pruebas, que imprime el valor que se le dé, aclarando el turno, es utilizado para llevar un control del flujo de la matriz.

(solution message)

Función que hace Display a la solución encontrada por el programa, muestra en texto el car del mensaje (el cual es una matriz), y de esa manera obtiene la respuesta a la que llegó el sistema.

(updateMatrix matrix newMatrix mPos nPos element)

Muestra al usuario la matriz actualizada. Funciona como el ciclo de juego, donde devuelve el programa al ciclo una vez el usuario termine su turno.

(errorNotification errType matrix mMax nMax)

Notifica al usuario si hay algún error, si los valores de n o m son menores o mayores que el mínimo o el máximo (3-10), mostrará el error en concreto. Error tipo 1, es cuando m se excede de los límites, el error tipo 2 es cuando n se excede de los límites, y el error final es si se intenta poner una ficha en una posición ya tomada.

(findPlacedTiles matrix currentm mCounter nCounter element positions)

Encuentra todas las posiciones (m n) que contienen 1 (valor del jugador), 2 (valor de bot) y 0 (valor vacío). Su caso base es cuando ambos, currentm y la matriz están nulos (ya todos los elementos han sido recorridos). En este caso, llama a updatePositions para notificar al usuario. Si solamente currentm es el que está vacío, significa que todas las posiciones de esa columna han sido revisadas, entonces se llama recursivamente aplicando cdr a la matriz, sumando 1 al contador de columnas y reiniciando a 1 el contador de n. Ahora, si currentm no está vacío significa que todavía hay elementos por revisar en esa columna, por lo que se llama recursivamente cambiando el currentm al cdr del currentm y sumando en 1 al contador de n.

(findTile matrix mPos nPos currentm)

Encuentra el valor de la celda en una posición especificada. Se llama recursivamente aplicando recorridos por columnas y filas de manera que encuentra la posición que buscamos, al llegar al caso base donde mPos y nPos son iguales a 1. Cuando esto se da, retorna el car del currentm, el cual posee el elemento dentro de esa celda.

(updatePositions positions element)

Notifica al usuario de la posición de sus fichas y las del sistema. Si el elemento es 1, es una ficha del usuario, si es 2, es una ficha del robot y si es 0 es una celda vacía. Retorna una representación gráfica del estado actual de la matriz en la consola.

(listLastElement lista)

Retorna el ultimo valor en la lista.

(placeBotSymbol matrix element)

La función inicial del algoritmo goloso. Se usa para un control organizado del flujo de turnos.

(checkHorizontalWin botTiles botTilesAux mToWin tileToWin candidates)

Función inicial de revisión si hay gane horizontalmente, ya sea por parte del jugador o de la máquina. Valida si botTilesAux es nula (caso base), de ser así, significa que no había una solución

válida en todas las fichas puestas por el usuario o la máquina, entonces retorna un tileToWin vacío. Si botTilesAux no es nulo, revisa si la ficha para ganar es nula, si es así, significa que no hay una solución correcta dentro de las posibles filas horizontales, por lo tanto, se llama recursivamente aplicando cdr a la lista de fichas y actualizando la ficha para ganar llamando a tileToWinHorizontaly. Por último, si tileToWin no es nula, significa que encontró una solución horizontal con la cual se puede ganar y devuelve esa ficha o celda, de manera que, si la encuentra, el programa no se sigue corriendo, sino que la retorna y para.

(tileToWinHorizontaly botTiles tileToCheck mToWin positions candidates)

Segunda función de revisión si hay gane horizontal. Se encarga de revisar recursivamente que en la fila en cuestión falte solo una ficha más para ganar, ya sea ficha de jugador o de la máquina.

(determinateTileToWinHorizontaly positions tileToCheck counter candidates)

Tercera función de revisión si hay gane horizontal. Solamente se ejecuta si el programa sabe que solo falta 1 celda para ganar, a través de tileToWinHorizontaly. De manera recursiva recorre la fila en cuestión buscando cual es la celda que falta con símbolo para ganar, una vez lo hace retorna la posición de la celda ganadora (posición n de tileToCheck, contador).

(checkVerticalWin botTiles botTilesAux nToWin tileToWin candidates)

Funciona igual que checkHorizontalWin, de igual manera se compone de tres funciones. Recorre botTilesAux recursivamente hasta que se hace nula, y recursivamente llama a tileToWinVerticaly para encontrar si falta o no una celda para ganar verticalmente.

(tileToWinVerticaly botTiles tileToCheck nToWin positions candidates)

De igual manera que tileToWinHorizontaly, recorre la lista de celdas hasta que encuentra una columna que está a una celda de ganar, si lo hace llama a determinateTileToWinVerticaly para obtenerla.

(determinateTileToWinVerticaly positions tileToCheck counter candidates)

Determina la celda que se necesita para que el usuario o el robot ganen verticalmente. Si se llama esta función, es que ya sea el usuario o el robot están a una ficha de ganar. Revisa si esa celda está dentro de la lista candidatos, si lo está es disponible para ganar.

(validDiagonals matrix)

Función que se encarga de retornar todas las diagonales válidas que podrían generar un gane dentro de la matriz de juego. Se compone de cuatro subfunciones que encuentran las diagonales de tipo descendentes y ascendentes, en dos variantes denominadas “diagonales verticales” y “diagonales horizontales”.

(upwardVerticalDiagonals mMax nMax mActual nActual start_n currentDiagonal diagonals)

Función encargada de identificar las diagonales de tipo ascendente vertical que se encuentran dentro de la matriz. Esta función va recorriendo, por cada start_n, el diagonal que se puede formar hasta llegar a un punto donde no queden posiciones que agregar al currentDiagonal. Lleva un recuento del nActual y mActual, y en el momento en que ambas superen a nMax y mMax respectivamente, se sabe que no quedan posiciones en diagonal que agregar al currentDiagonal, por lo que se procede a repetir el proceso, pero reduciendo la fila donde empieza el diagonal start_n, con el fin de considerar todas las diagonales de tipo ascendente vertical.

(upwardHorizontalDiagonals mMax nMax mActual nActual start_m currentDiagonal diagonals)

Se encarga de identificar las diagonales de tipo descendente vertical dentro de la matriz. Analiza cada diagonal hasta que se llega a que start_m es mayor a mMax, lo que significa que no quedan diagonales de tipo descendente vertical.

(topDownVerticalDiagonals mMax nMax mActual nActual start_n currentDiagonal diagonals)

Se encarga de identificar las diagonales de tipo ascendente vertical dentro de la matriz, analiza cada diagonal, igualmente, hasta que start_m exceda el límite, lo que provoca que ya la matriz está revisada completamente y no queden diagonales ascendentes verticales por analizar.

(topDownHorizontalDiagonals mMax nMax mActual nActual start_m currentDiagonal diagonals)

Se encarga de identificar las diagonales de tipo ascendente horizontal dentro de la matriz, analiza cada diagonal, igualmente, hasta que start_m exceda el límite, lo que provoca que ya la matriz está revisada completamente y no queden diagonales ascendentes horizontales por analizar.

(discardDiagonals diagonal_list validDiagonals)

Por medio de llamadas recursivas y comparaciones de cada diagonal, se determina cuales tienen una longitud mayor o igual a tres, por lo que pueden ser consideradas válidas para ganar. Luego de analizar las diagonales, retorna la lista de diagonales válidas.

(checkDiagonalWin matrix diagonals_list flag_win player code)

Encargada de analizar si existe una posición en la cual, en caso de colocar una ficha, se produzca un gane para el jugador. Se llama esta función de forma recursiva hasta que checkDiagonal retorne una posición que no sea nula, o se acaben los diagonales posibles.

(checkDiagonal matrix player diagonal auxDiagonal amountSpaces code)

Encargada de analizar cada diagonal para determinar si existe un posible gane dentro de la misma. Por cada posición dentro de la diagonal, y en caso de que la función checkDiagonalPos retorne un true, se suma un 1 a amountSpaces, por lo que, al terminarse las posiciones de diagonal, y si amountSpaces está a una unidad de la longitud de auxDiagonal, se retorna el auxDiagonal o la posición para ganar proveniente de findWinTileDiagonal, en función de si el código es “Interfaz” o “Lógica”. El motivo por el cual se utiliza el código es porque, para la interfaz se tienen que enviar solo la posición inicial y final de la diagonal (con el fin de dibujar la línea del gane), y para la lógica se tiene que retornar únicamente la posición que permita el gane.

(checkDiagonalPos matrix player position)

Utiliza findTile para determinar si en esa posición existe una celda o ficha de valor de jugador (1 o 2). Si es así, retorna un 1 que se suma a amountSpaces.

(findWinTileDiagonal matrix diagonal auxDiagonal code)

Determina cual es la posición donde hace falta una ficha dentro de la diagonal. Para cada elemento dentro de la diagonal, se verifica si la posición actual de la diagonal es vacía, si es así, devuelve

esa celda o posición. Si no es así, se mueve a la siguiente posición de la diagonal hasta que se encuentre una nula.

(formLines matrix player_tiles player possibles candidates)

Se encarga de que la máquina haga sus líneas, en caso de no poder ganar o no tener que colocar fichas para no perder. Por cada ficha del jugador, agrega nuevas posiciones posibles por medio de la función updatePositions, y al ser la lista de player_tiles nula, se determinará en cual de las posiciones posibles se debe crear la línea.

(formLines_aux possibles m_toForm n_toForm mMax nMax candidates up right down left)

Se encarga de determinar en cuales posiciones posibles podría la máquina continuar creando sus líneas. Utiliza flags para no repetir validaciones, para revisar cada vecino de las fichas previamente colocadas. Para ser un vecino válido, y ser colocado dentro de la lista de posibles posiciones, se debe cumplir que la posición de ese vecino no exceda mMax o nMax, además que su m y n deben ser mayor a 0, una vez evaluadas las coordenadas, se retorna la lista de posibles.

(placeTilesFormLines matrix tileToPlace)

Se encarga de colocar la ficha correspondiente al momento de conformar la línea. Se encarga de crear el mensaje para la GUI con la forma indicada en la función para poner la ficha deseada.

(checkCandidates candidates position)

Revisa la lista de candidatos y determina si una posición es válida para colocar una ficha en la matriz. Recursivamente se comparan las posiciones de la tupla de la primera posición de candidatos con las posiciones de la tupla position, si coinciden, se colocará la ficha en un candidato correcto, el cual es un espacio vacío en la matriz. Si no coinciden, se recorta la lista candidatos y se llama recursivamente.

(runGreedyAlgorithm matrix)

Inicia la ejecución del algoritmo goloso, es llamada cada vez que la máquina le toca colocar una ficha, y a través de sus subfunciones, se determina cual es la posición idónea para colocar la ficha, analizando las fichas propias de la máquina y las del jugador.

(findCandidates matrix)

Es la primera parte del algoritmo voraz, se encarga de encontrar el conjunto de candidatos posibles que pueden contribuir a la solución, o, en otras palabras, las posiciones que pueden ser ocupadas por la nueva ficha a poner por la máquina, para esto, llama a findPlacedTiles que retorna la lista con todas las posiciones con valor 0 (vacía) en ese momento.

(viabilityFunction matrix candidatesToPlace player_tiles botTiles)

Es la segunda parte del algoritmo goloso, se encarga de determinar la viabilidad del conjunto de candidatos. Para esto recibe a la matriz, la lista de candidatos, la lista de fichas colocadas por jugador y la lista de fichas colocadas por la máquina. Para esto utiliza las funciones de chequeo de gane horizontal, vertical y diagonal, primero revisando horizontalmente, luego vertical y finalmente diagonal. Una vez revisado si puede ganar, revisa si puede perder, con el mismo orden, revisa horizontal, vertical y diagonalmente. Si al revisar si puede ganar es así, pondrá la ficha para ganar, sino puede ganar en ese turno revisará si puede perder, si es así, se correrá como si fuera a ganar como el jugador, de modo que evita que gane. Si ninguno de estos aplica, pone la ficha en un lugar aleatorio.

Ejemplificación de las estructuras de datos desarrolladas

En el proyecto se utilizó una matrix m x n para manejar el funcionamiento del tablero, el usuario indica estos valores al iniciar la partida, esta a su vez fue implementada mediante una lista de listas. Mediante la función ***createRow*** se generan las filas de la matriz, todas con un valor de 0, luego estas sublistas son agregadas a una lista, mediante la función ***createMatrix.***

```
#|
Function that creates the matrix recursively, for every column specified by the user,
it adds a row with size n using createRow.
Receives: n (rows), m (columns)
Returns: Call for another function, message if failed.
Restrictions: Must be integers.
|#
(define (createMatrix n m matrix)
  (if (> m 0)(createMatrix n (- m 1) (append matrix (list(createRow n '()))))matrix))
#|
Function that creates the rows for the matrix, called for every column in the matrix,
and recursively adds a 0 for every row in the matrix, so that it ends up creating a
n*m matrix full of 0 (value set as empty tile)
|#
(define (createRow n row)
  (if (> n 0)(createRow (- n 1) (append row '(0)))row))
```

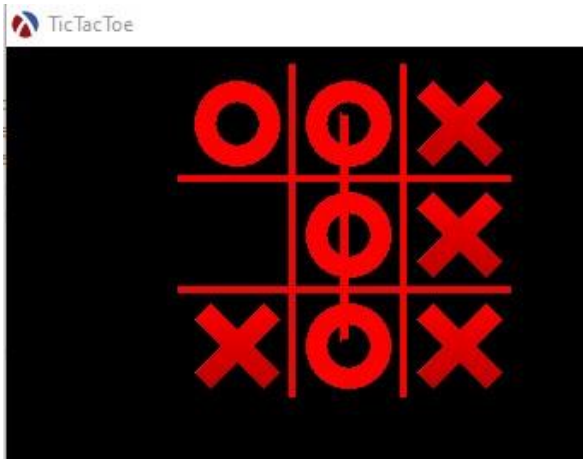
La matriz se implementa como la matriz mxn donde inicialmente todas las entradas son 0, a medida que avanza el juego se colocan las “fichas” de los jugadores, siendo “1” el jugador y “2” la máquina. Para el recorrido de la matriz se utilizan las funciones cdr y car y contadores, para poder recorrer las listas y a su vez la matriz.

Mediante el método ***replace_value_matrix*** se reemplazan valores en la matriz, mediante recursión se va generando una nueva matriz y cuando se llega al elemento de posición (m, n) se realiza el cambio. También se utilizan funciones de recorrido de la matriz en los casos de revisión y validación de gane, pues mediante cdr y car el algoritmo se mueve a través de las columnas y las filas, de manera que se pueden obtener y validar datos moviéndose vertical, horizontal y diagonalmente. Esto implica que además de la estructura de la matriz también se usó la “list” nativa de Racket, pues se definió la matriz como una matriz de matrices.

```
(define (replace_value_matrix newValue line column matrix)
  (if (and (<= line (length matrix)) (<= column (length(car matrix))))

      (replace_m_aux newValue line column matrix '() 1)

      #f

      )
  )

(define (replace_m_aux newValue line column matrix newMatrix current_line)
  (if (equal? current_line line)
      (replace_m_aux newValue
                     line
                     column
                     (cdr matrix)
                     (append newMatrix (list(replace_value_matrix newValue (car matrix) column)))
                     (+ 1 current_line))

      (if (not (null? matrix))
          (replace_m_aux newValue
                         line
                         column
                         (cdr matrix)
                         (append newMatrix (list(car matrix)))
                         (+ 1 current_line))
      newMatrix
      )
  )
  )
```

Problemas sin solución

En el proyecto, se llegó a un problema al cual no se le pudo encontrar solución. Este consiste en que, si el jugador está a punto de ganar verticalmente y la máquina también, al usuario clickear sobre la celda que lo haría ganar, termina su turno y la línea no se dibuja, solamente para la máquina y la máquina termina ganando. Se intentó resolver este problema, sin embargo, implicaba cambiar completamente el sistema de turnos y la función de candidatos, por lo que no se le pudo encontrar solución. El error se puede ver a continuación:



Plan de Actividades realizadas por estudiante

Tarea	Tiempo estimado	Responsable a cargo	Fecha de entrega
Crear funciones de creación y manejo de matrices.	6 horas	Eduardo Zumbado	30 de abril
Creación de ventanas necesarias para la GUI (Investigación incluida)	6 horas	Jose Antonio	30 de abril
Manejo de matriz inicial, con recorridos y chequeos de valores, dar valor a fichas jugador, fichas máquina y celda vacía	5 horas	Eduardo Zumbado	3 de mayo
Colocación gráfica de las líneas, dependiendo de los valores de m y n, al igual que la colocación de las fichas X y O con funcionalidad de clics	10 horas	Jose Antonio	3 de mayo
Hacer las funciones necesarias para alterar la matriz con los valores de input del jugador.	6 horas	Eduardo Zumbado	7 de mayo
Comenzar la investigación y funciones necesarias para la lógica de la máquina para el algoritmo voraz	10 horas	Jose Antonio	7 de mayo
Terminar el algoritmo voraz, y unir la GUI con la lógica.	10 horas	Jose Antonio Eduardo Zumbado	10 de mayo
Completar la documentación interna y externa del proyecto, además del manual de usuario.	5 horas	Jose Antonio Eduardo Zumbado	11 de mayo

Problemas encontrados

Se encontró un problema que con la matriz en 3x3, al usuario ganar por medio de una diagonal, el programa se cae debido a que recién se sube el contador a 3 (mínimo para ganar) y al llamar a la función de chequeo de gane diagonal este busca el campo vacío para ganar, pero ya está completo, por lo que no puede avanzar en el chequeo y termina cayéndose. Esto se intentó arreglar cambiando el orden de llamada de las verificaciones diagonales, pero solamente se logró que se arreglara ese lado, pero el mismo error apareció, pero para la diagonal de derecha a izquierda descendente, por lo que no se logró arreglar este problema.

Conclusiones del Proyecto

A partir de la experiencia recolectada al hacer la tarea, se confirmó que las soluciones obtenidas por medio de Scheme (Racket), son más directas y se obtienen en menor cantidad de líneas ya que los problemas se solucionan dividiendo el problema en varias funciones de manera que una complementa la otra. Por lo que este “divide and conquer” aplicando funciones auxiliares se utilizó en todo el proyecto y permitió su desarrollo de manera más eficiente.

Por otro lado, se concluyó que la recursividad aplicada en el paradigma funcional, si es aplicada correctamente, tiene grandes beneficios, donde una función completa se resume en varios condicionales, con llamadas recursivas y un caso base, y de esta manera las funciones, por ejemplo, las de recorrido de la matriz, salieron de manera muy fácil y relativamente corta. Además, se llegó a la conclusión que las funciones cdr y car son métodos de Racket que brindan mayor control y manejo con las listas y matrices, procesos que los lenguajes modernos no tienen implementado y el desarrollador debe hacer, por lo que se puede concluir que con estos métodos Scheme promueve ventajas en el manejo de estructuras de datos.

Por último, se puede llegar a la conclusión que el lenguaje de Scheme o Racket, aparte de ser funcional, promueve un ambiente de interfaz gráfica cómodo y muy sencillo comparado con lenguajes como C, que sus librerías gráficas dejan que desear. Por lo que Scheme o Racket, es un lenguaje y entorno que no presenta mucho problema para implementación de programas como el de esta tarea.

Recomendaciones del Proyecto

Se recomienda aplicar alguna medida para que el algoritmo voraz no sea tan eficiente, ya que el implementado en esta tarea es muy difícil de ganarle, ya que revisa simultáneamente todos los casos donde puede ganar o perder y los aplica de la mejor manera, por lo tanto, se recomienda que cuando la máquina no puede ganar ni perder en ese turno, simplemente ponga la ficha en un lugar aleatorio, para que así el usuario tenga un poco más de opciones de poder ganar.

También, al ser un trabajo de gato o tres en línea, este es básicamente una matriz, por lo que se recomienda utilizar la estructura de datos de lista (incluido en el lenguaje) y por ende matrices, que son listas de listas, y de esa manera, el manejo de datos para los valores se hace muchísimo más sencillo.

Bibliografía consultada en todo el Proyecto

[1] JavaTPoint, “Greedy Algorithm”, s.f [Online]. Disponible: https://www.javatpoint.com/greedy-algorithms

[2] Beautiful Racket, “Data structures”, s.f [Online]. Disponible: https://beautifulracket.com/explainer/data-structures.html

[3] Beautiful Racket, “Recursion”, s.f [Online]. Disponible: https://beautifulracket.com/explainer/recursion.html

[4] Beautiful Racket, “Lang line”, s.f [Online]. Disponible: https://beautifulracket.com/explainer/lang-line.html

[5] Geeks for geeks, “Traverse a given Matrix using Recursion”, s.f [Online]. Disponible: https://www.geeksforgeeks.org/traverse-a-given-matrix-using-recursion/

Enlace del repositorio

https://github.com/Eduardo2108/TicTacToe