UNIVERSIDAD PERUANA DE CIENCIAS APLICADAS

Trabajo Final
2020-2

Complejidad Algorítmica
CC-184

Integrantes:

- Sebastian Wilder Contreras - **U20171D516**
- Marc Oliva Eslava - **U201421830**
- Víctor Fajardo Rojas - **U201612009**

Sección: WS6A

2020 – 02

Contenido

Introducción

Este trabajo consiste en la implementación del juego de mesa “Quoridor” como un programa desarrollado en Python.

El juego “Quoridor” consta de un tablero de 9x9 recuadros, una ficha por cada jugador (con un mínimo de 2 y un máximo de 4) y un total de 20 paredes las cuales son distribuidas de forma equitativa entre todos los jugadores.
El objetivo de este es mover la ficha de jugador correspondiente a la casilla del extremo opuesto a donde esta empieza.

El juego consta con un sistema de turnos, en el cual, el jugador correspondiente deberá mover su ficha de forma vertical u horizontal (salvo excepciones), así mismo el jugador puede optar por colocar una barrera en el tablero, esto sirve para obstaculizar la ficha del resto de jugadores y por ende que demoren más en llegar al extremo opuesto. Ha de mencionarse que no está permitido encerrar una ficha con barreras ni tampoco hacer una casilla inaccesible con las mismas.

Ante la situación en la cual un jugador se encontrase frente a otro, existe la opción de saltar por encima de la otra ficha de ser posible, en caso contrario se permite moverse una casilla de forma diagonal.
Lo explicado anteriormente será implementado en este trabajo utilizando Python como lenguaje de programación, de forma que pueda ser jugado en contra de un Bot con dificultad variable.

ESTADO DEL ARTE

En esta sección, se mostrará algunos conceptos y algoritmos que nos ayudaran a resolver los problemas del juego “Quoridor”.

Backtracking

Es una técnica algorítmica que se basa en el enfoque de fuerza bruta, el cual considera todas las posibles combinaciones para resolver un problema. Con esta técnica podemos resolver el problema: *Encontrar un camino hacia el otro lado del tablero para ganar la partida.*

Teoría de grafos

BFS (breadth-first-search)

Un algoritmo de teoría de grafos que nos ayuda a recorrer todo el grafo por “niveles”, teniendo esta propiedad de recorrido, nos puede servir para el siguiente problema: *Encontrar el camino **más corto** desde la posición del jugador (I.A.) hasta el otro lado del tablero.*

Este algoritmo es muy eficiente, ya que su complejidad es de O(V + E), donde V es la cantidad de vértices del grafo G y E es la cantidad de aristas en el grafo G.

Componentes conexas

Es un concepto en teoría de grafos, para todo grafo no dirigido, podemos dividir el conjunto de vértices en varios subconjuntos, de tal forma que para dos vértices cualesquiera que pertenecen al mismo subconjunto, exista un **camino** entre ellos.

Este concepto nos ayudará a entender el siguiente concepto y algoritmo.

Puente, borde de corte o arco de corte

En teoría de grafos, un puente es todo nodo que cumple lo siguiente: *Si eliminamos al nodo **v** entonces el grafo **G** tendrá más componentes conexas.* Es decir, si eliminamos dicho nodo del grafo, la cantidad de componentes conexas que tiene el grafo aumentará.

Este concepto y algoritmo nos puede ayudar para resolver el problema sobre: *Como saber y mantener el tablero con posibles caminos para que cualquier jugador pueda llegar al otro lado.*

Es decir, podemos saber si al poner una pared en el juego “*Quoridor”*, esta elimina un **puente** y de esta forma, cumplir la regla de que todo jugador tenga **al menos** un camino para ganar el juego.

Algoritmo MiniMax

En el campo de inteligencia artificial y en teoría de juegos, es un algoritmo de decisión el cual, tomando como referencia el estado del juego (entiéndase a estado de juego como las posiciones de las fichas en ajedrez o como las casillas marcadas en Tic-Tac-Toe), elige el siguiente movimiento de tal manera que sea el mejor para ti suponiendo que el contrincante escoja la peor opción para ti.

Es un algoritmo recursivo, y se puede entender como un árbol donde cada nodo o vértice representa un estado del juego.

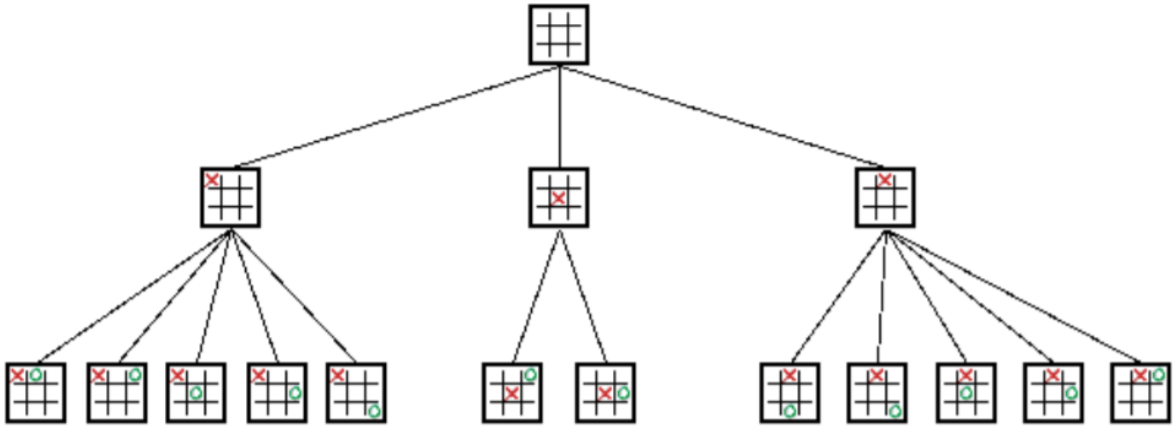


Figura 1, el árbol de juego de un Tic-Tac-Toe(Borovska, 2007)

La decisión de tomar el mejor camino se toma utilizando la información de los hijos, donde se calcula un número positivo cuando ese estado de juego te beneficia y un número negativo cuando no (cuando el contrincante se beneficia). En cada nivel del árbol se van intercambiando los turnos (en un nivel, es tu turno en todos los estados de juego y en otro nivel, el turno de mover es el de tu contrincante) y en base a esto se decide si entre los hijos se escoge el mayor de todos (el que da mayor beneficio) o el menor de todos (menor beneficio).

Este algoritmo nos puede ayudar a la construcción de la inteligencia artificial del juego “Quoridor”.

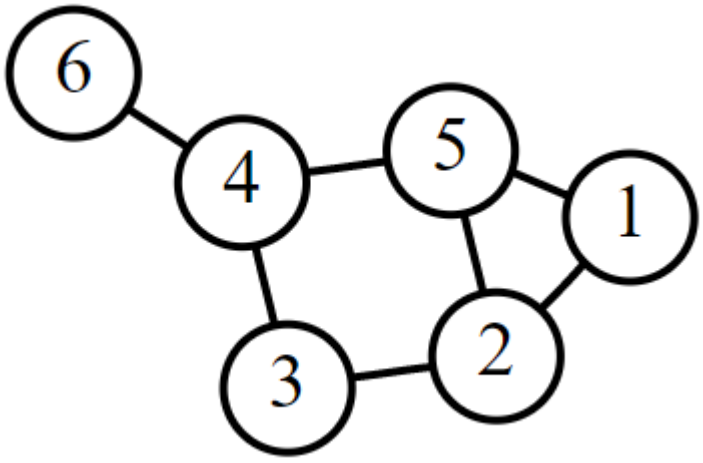
Alpha-Beta pruning

El algoritmo MiniMax puede ser optimizado utilizando el Apha-Beta pruning. Debido a que el algoritmo MiniMax considera todos los estados de juegos para elegir el siguiente movimiento, se puede evitar el cálculo de algunos estados de juego cuyo valor no influya en el resultado final, esta decisión de *podar* algunas hojas del **árbol de juego (Figura 1)** se realiza tomando en cuenta los cálculos de otras hojas que ya fueron calculadas.

Metodología

A continuación, se detalla el proceso a seguir para cumplir con el objetivo del proyecto.

- **Análisis:** La situación que se planteó para este proyecto fue encontrar una solución viable para el juego quoridor.
- **Objetivos:**
    - Desarrollar la competencia general de razonamiento cuantitativo y la competencia específica de uso de técnicas y herramientas acorde a los objetivos del curso.
    - Desarrollar un algoritmo que permita resolver completa o parcialmente el problema del juego quoridor.
    - Determinar la importancia de la aplicación de algoritmos eficientes a la hora de resolver un problema.
    - Analizar la eficiencia y complejidad de los algoritmos planteados.
- **Marco teórico:**
    - **Grafos**: En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Son objeto de estudio de la teoría de grafos. Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas). Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).



    - **Algoritmo Bellman Ford**: El algoritmo de Bellman-Ford es un algoritmo que calcula las rutas más cortas desde un único vértice de origen a todos los demás vértices en un dígrafo ponderado, este también trabaja con pesos. Su funcionamiento consiste en partir de un vértice origen. Seguidamente, Bellman-Ford relaja todas las aristas, esto lo hace $|V| - 1$ veces, siendo $|V|$ el número de vértices del grafo. Las repeticiones permiten a las distancias mínimas recorrer el árbol, ya que, en la ausencia de ciclos negativos, el camino más corto solo visita cada vértice una vez. A diferencia de la solución voraz, la cual depende de la suposición de que los pesos sean positivos, esta solución se aproxima más al caso general.

1. *Complejidad del algoritmo*: La idea fuerte del algoritmo es que, si el grafo tiene V nodos, alcanza con recorrer las aristas $|V| - 1$ veces. Se ve entonces que la complejidad del algoritmo es $O(V.E)$ donde V es la cantidad de vértices y E la de aristas.

2. *Espacio de búsqueda del algoritmo:* El espacio de busqueda de Bellman Ford debido a su naturaleza voraz encuentra el camino en un tiempo menor a comparación de los otros. Debido a que al seleccionar vorazmente el nodo de peso mínimo aun sin procesar simplemente relaja todas las aristas, y lo hace |V|-1 veces, siendo |V| el número de vértices en el grafo.

3. *Código de Bellmand-Ford:*

```
n, m = map(int, input().split())
ed = [] # Lista de aristas
for i in range(m):
    x, y, w = map(int, input().split())
    ed.append((x-1, y-1, w))

inf = 10**18
d = [inf] * n
d[0] = 0

for i in range(n-1):
    for u, v, w:
        if d[u] < inf and d[u] + w < d[v]:
            d[v] = d[u] + w
```

- **Herramientas de desarrollo:** El programa se implementará en Python. Para manejo de matrices usaremos la librería **“N**umpy**”** y para el manejo de representación en grafos se utilizará **“NetworkX”**. Por último, se utilizará la librería "**pygame**" para la interfaz gráfica.

Espacio de búsqueda

En esta sección se explicará, el espacio de búsqueda para encontrar un camino al “goal” en Quoridor.

Estado Inicial

Como estado inicial es un tablero 9x9, donde se tendrá entre 2 a 4 peones (representan a los jugadores), los cuales estarán en sus posiciones iniciales de la siguiente manera:

- Primer peón: quinta fila y primera columna.
- Segundo peón: quinta fila y novena columna.
- Tercer peón: primera fila y quinta columna.
- Cuarto peón: novena fila y quinta columna.

Para la representación gráfica solo se utilizará a un peón, ya que se trata de graficar el espacio de búsqueda para encontrar un camino.

Estado Final

Como estado final, un peón debe llegar a cualquier casilla del extremo opuesto de su posición inicial, es decir:

- Para el primer peón: Cualquier casilla de la columna 9.
- Para el segundo peón: Cualquier casilla de la columna 1.
- Para el tercer peón: Cualquier casilla de la fila 9.
- Para el cuarto peón: Cualquier casilla de la fila 1.

De esta forma, el juego acaba y el peón que cumpla lo anterior, gana la partida.
En la representación gráfica, se utilizará la condición del primer peón para considerarlo un estado final.

Transiciones

Como en el juego, en un turno se puede realizar dos tipos de movimientos: **mover al peón o poner un muro**), entonces estos movimientos se consideran como transiciones.

Sin embargo, en la representación gráfica solo se considerará como transición al **movimiento del peón** de la siguiente manera:

- hacia arriba
- hacia abajo
- hacia la derecha
- hacia la izquierda

Siempre y cuando, al realizar el movimiento, la casilla de llegada este dentro de los límites del tablero, por ejemplo: si estoy en la casilla fila 1, columna 1, no puedo ir hacia arriba ni hacia la izquierda, ya que me saldría de los límites del tablero. Además, la casilla de llegada no debe de haber sido “visitada”. Hacemos esta restricción para que no se generen bucles.

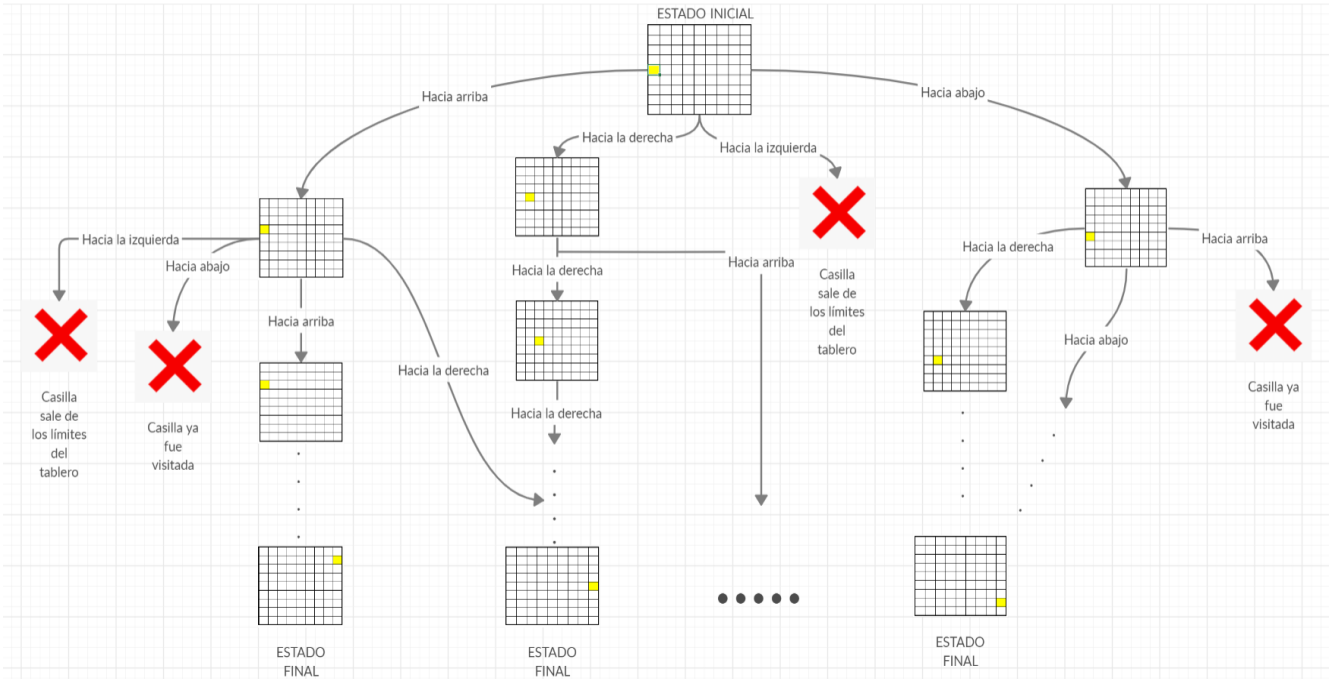
Estructuras de datos a utilizar

Para representar el camino, se utilizará una lista de nodos, donde cada nodo se representa con dos números (fila y columna). La lista de nodos estará en orden. Es decir, si busco un camino desde la fila 1, columna 1 y quiero llegar a mi “goal”, fila 1 columna 9. Entonces, la lista de nodos debe tener como mínimo al punto inicial y al punto de llegada, y entre ellos puede haber otros nodos.

Asimismo, la lista de nodos debe representar un camino válido, es decir, dado 2 nodos consecutivos cualesquiera de la lista de nodos, debe existir una arista entre ellos o, en otras palabras, no debe haber un muro entre ellos.

Por otro lado, para representar el tablero en cada estado del espacio de búsqueda se utilizará un grafo no dirigido en el que cada casilla del tablero es un vértice (simbolizado por dos números, los cuales son: la fila y columna) y un posible movimiento será representado por las aristas de los nodos. Por ejemplo: Si estoy en la casilla fila 1, columna 1 (vértice [1, 1]) entonces estará "conectado" por aristas a las casillas fila 2, columna 1 (vértice [2, 1]) y fila 1 columna 2 (vértice [1, 2], pero no tendrá una arista a la casilla fila 6 columna 2 (vértice [6, 2]). De esta forma, también es fácil darse cuenta si hay un muro entre dos casillas adyacentes, ya que, si estoy en la casilla fila 1, columna 1 (vértice [1, 1]) entonces debería haber una arista que me conecte con la casilla fila 2, columna 1 (vértice [2, 1]), si no la hay, entonces si o si debe haber un muro entre las dos casillas.

Representación gráfica



Entrada de datos

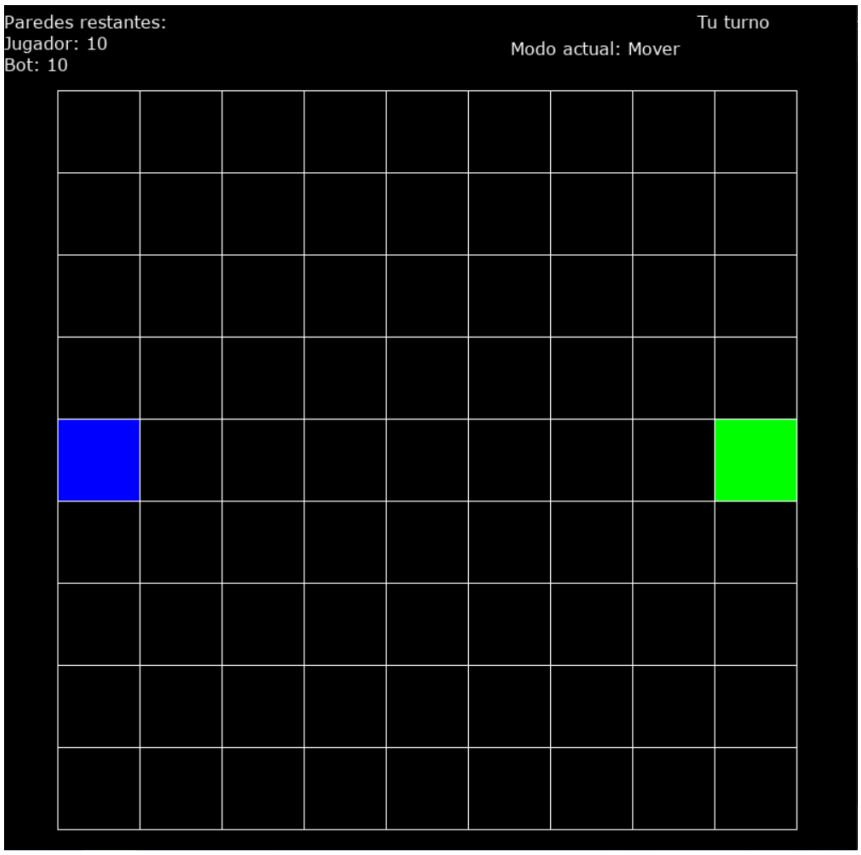
Algoritmos de Búsqueda

En los algoritmos de búsqueda de camino, hay dos parámetros que me indican la casilla inicial, la posición donde se encuentra actualmente y la casilla final, me indica a donde quiero llegar.

Interfaz Gráfica

La interfaz gráfica hecha con la librería **PyGame**, recibe los siguientes datos:

- Si es el turno del jugador:
  - Tecla ‘q’: El jugador mueve su ficha. El modo actual cambia a “Mover”.
  - Tecla ‘w’: El jugador coloca un muro de forma vertical. El modo actual cambia a “Colocar muro vertical”.
  - Tecla ‘e’: El jugador coloca un muro de forma horizontal. El modo actual cambia a “Colocar muro horizontal”.
  - Clic izquierdo:
    - Si el jugador se encuentra en modo “Mover”, el jugador indica la casilla que desee moverse, mientras se cumpla las restricciones del juego (posición y movimiento válido), si no el juego no moverá la ficha.
    - Si el jugador se encuentra en modo “Colocar muro vertical” o “Colocar muro horizontal”. El jugador indica el lugar a poner el muro. El juego validará si colocar el muro se sigue manteniendo las reglas del juego.



Inicio del juego Quoridor. Cuadrado color azul (Jugador). Cuadrado color verde (Bot).

Métricas

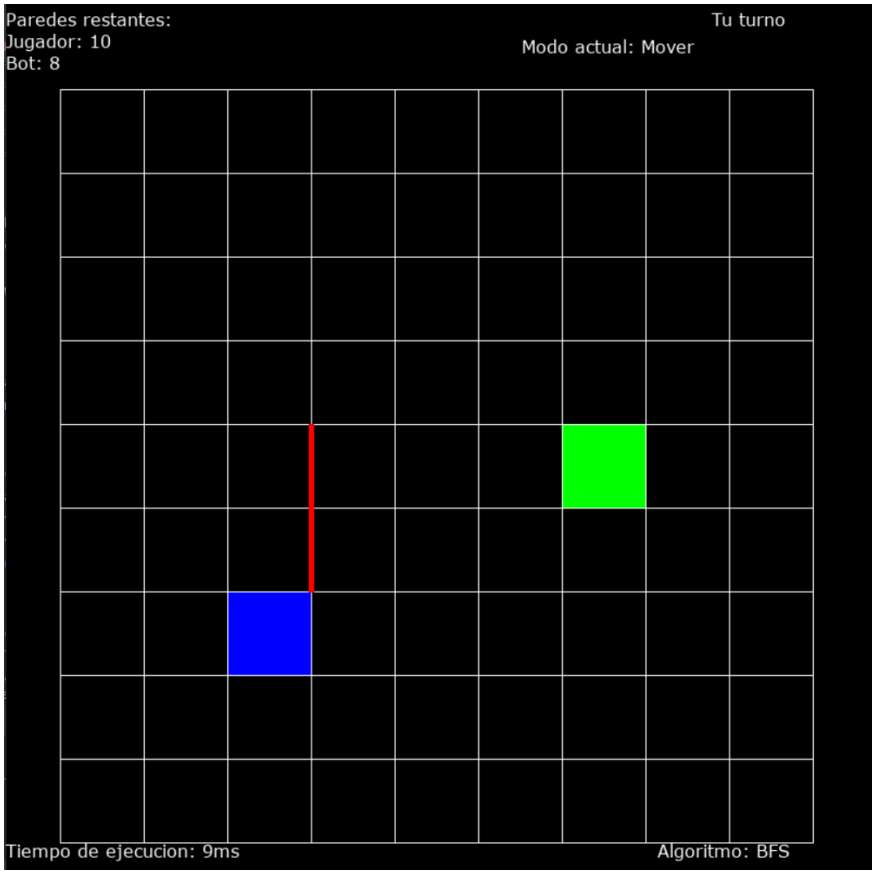
Como métricas para la evaluación de los algoritmos de búsqueda serán:

- Tiempo,
- Complejidad Algorítmica (Big O) y
- Memoria.

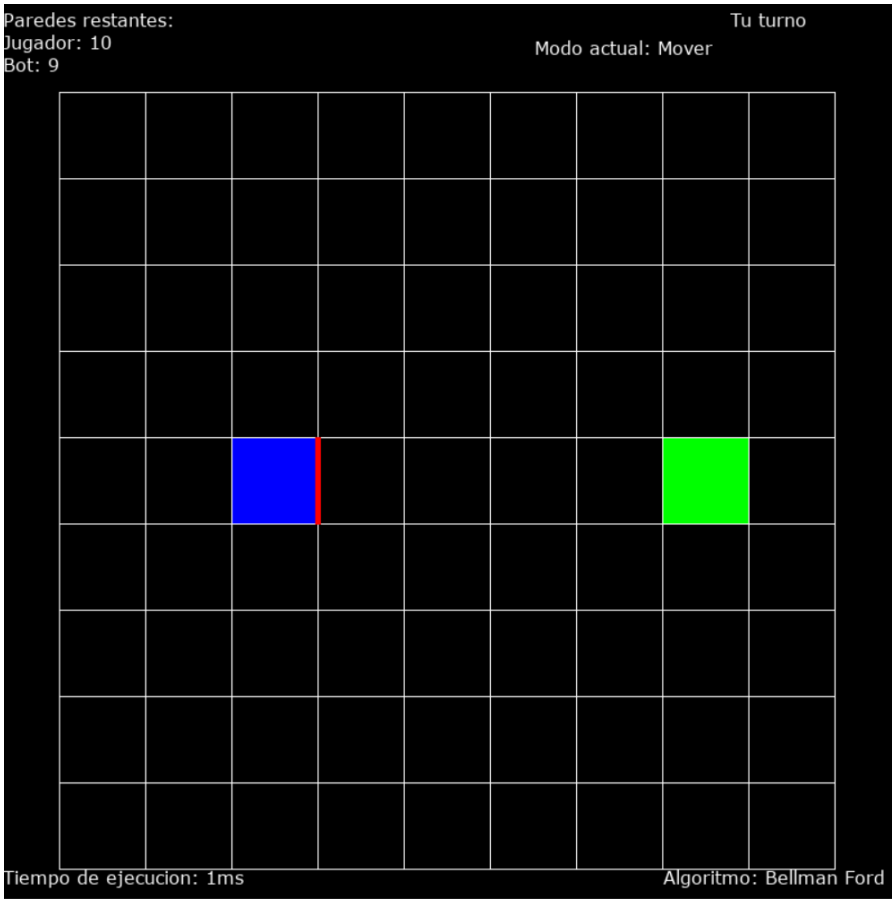
En la siguiente tabla se observa la comparación de las métricas de los algoritmos

	ALGORITMO		
	BFS	DFS	Bellmand-Ford
TIEMPO (ms)	9	1	1
BIG O	$O(V+E)$*	$O(V+E)$ *	O$(V \times E)$ *
MEMORIA	Los 3 algoritmos utilizan la representación del tablero como grafos. Por lo que el uso de memoria para los tres es el mismo. $O(V+E)$ *en memoria*		

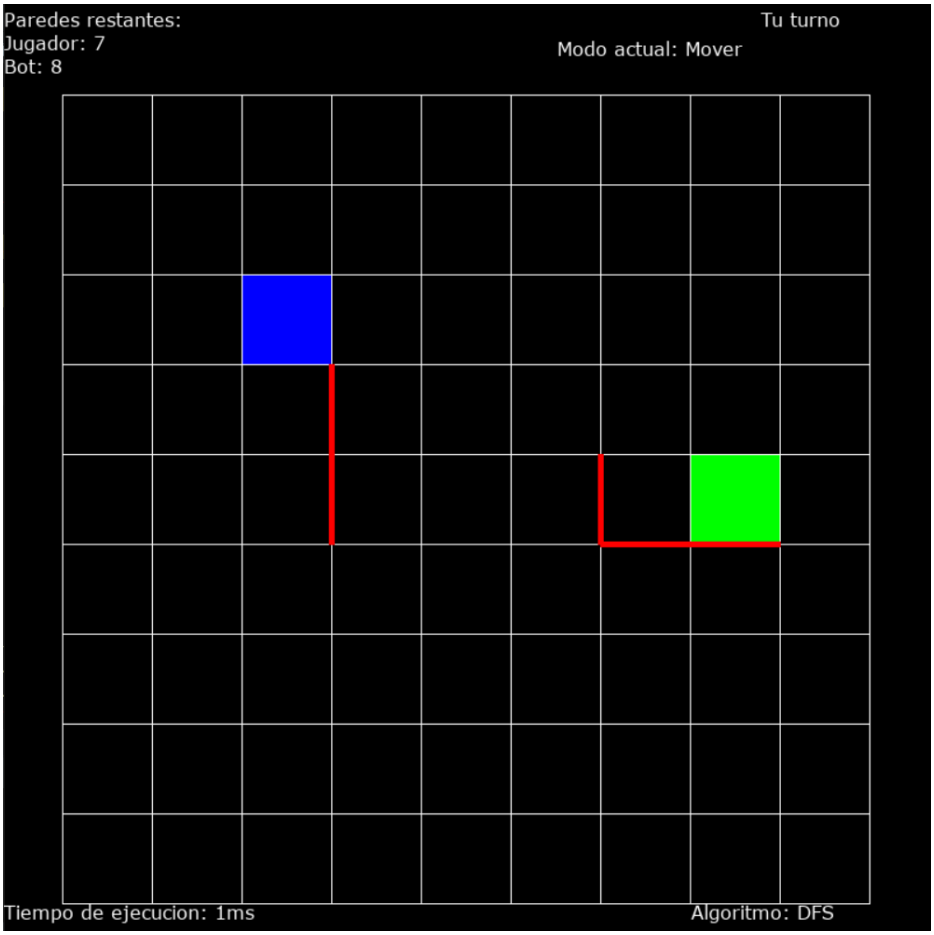
(*) V: *cantidad de vértices, en el caso del Quoridor, es el número de casillas, en total* 81 *casillas*, 81 *vértices*.
(*) E: *cantidad de aristas*.



Métrica de tiempo para algoritmo BFS



Métrica de tiempo para algoritmo Bellman Ford



Métrica de tiempo para algoritmo DFS

Conclusiones

A partir de la sección anterior podemos observar que:

1. El algoritmo BFS es el que encuentra el camino más corto junto con el de Bellmand-Ford, en situaciones con muros y sin muros.
2. A pesar de tener la misma complejidad el BFS y DFS, el que encuentra el camino más corto es el BFS.
3. El tiempo de ejecución del BFS es mucho mayor al del DFS. A pesar de que tienen la misma complejidad algorítmica $O(V + E)$.
4. Cuando hay muros, el BFS sigue encontrando el camino más corto junto con Bellman Ford.
5. El aumentar la profundidad al árbol de decisión proveía un mejor rendimiento, pero, al mismo tiempo, aumentaba el tiempo de ejecución del juego. Probablemente, con mejores funciones de evaluación se hubiera reducido el tiempo de ejecución, eso se haría en futura investigación.
6. Al enfrentar los algoritmos de Búsqueda de camino, el que mayor rendimiento mostró fue: BFS. Como segundo quedo Bellmand-Ford, ya que, a pesar de prácticamente lograr el mismo rendimiento del BFS, su complejidad es la mayor de todas. Y como tercer lugar quedo el algoritmo DFS.
7. El movimiento que da ventaja en el juego es el de "doble salto".

Referencias

Borovska, P., Lazarova, M. (2007). *Efficiency of parallel minimax algorithm for game tree search*. Association for Computing Machinery, New York, NY, USA, Article 14, 1–6. DOI: https://doi.org/10.1145/1330598.1330615

MIT OpenCourseWare. (2014, 10 enero). 6. *Search: Games, Minimax, and Alpha-Beta* [Vídeo]. YouTube. https://www.youtube.com/watch?v=STjW3eH0Cik

Sebastian Lague. (2018, 20 abril). *Algorithms Explained – minimax and alpha-beta pruning* [Vídeo]. YouTube. https://www.youtube.com/watch?v=l-hh51ncgDI

The Coding Train. (2019, 11 diciembre). Coding Challenge 154: Tic Tac Toe AI with Minimax Algorithm [Vídeo]. YouTube. https://www.youtube.com/watch?v=trKjYdBASyQ