DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: Threads

Sistemas Operativos

Integrante	LU	Correo electrónico
Lisandro Diaz Di Meo	795/18	ddmlisandro22@gmail.com
Andrés Felder	245/19	andyfelder16@gmail.com

Resumen : El objetivo del presente trabajo práctico es implementar el conocido juego "Capturar la bandera" mediante la utilización de threads. En dicho juego participan dos equipos el **rojo** y el **azul** cuya cantidad de jugadores esta predeterminada. El juego consiste en hacer una serie de movimientos en cada turno de equipo mediante distintos tipos de estrategias, para alcanzar la bandera del equipo contrario.

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Reglas del juego

- Comienza moviendo el quipo rojo.
- Los equipos se mueven por turno.
- Un jugador no puede pasar sobre otro jugador.
- Cada jugador puede moverse hacia arriba, abajo, izquierda o derecha.
- Una vez que un jugador llegue a la posición de la bandera contraria, se termina el juego.

2. Estrategias implementadas

Las siguientes estrategias son utilizadas como criterio para decidir qué jugador mover, no hacia donde.

- **Secuencial:** Sin determinar orden, mueve cada jugador un paso hacia algún objetivo. Al mover todos los jugadores de un equipo, le tocará al otro. Si un jugador está rodeado y no puede moverse, pierde su turno.
- **Round Robin:** Moverá *Quantum* pasos cada equipo. De no alcanzar la cantidad de jugadores, se moverá 1 vez cada jugador. De ser *Quantum* > N , con N la cantidad de jugadores, volverán a jugar los jugadores hasta terminar el quantum. Dada la situación en la que ya todos los jugadores se movieron una vez y me sobra *Quantum* se repetirá el mismo orden. Si se acaba antes de terminar la segunda pasada, igualmente la próxima ronda iniciará con el primer jugador.
- **Shortest distance first:** Se deberá mover el jugador que está más cerca de la bandera un jugador de cada equipo por vez. Nótese que una vez encontrado el que más cerca está, es ese el que se continuará moviendo todo el resto de las rondas.
- **Quantum random:** Igual a Round Robin pero cada jugador tiene una cantidad independiente de Quantum Random.

3. Representación del sistema

3.1. Procesamiento de los datos de entrada

Los datos iniciales necesarios para la implementación son tomados por la clase ***config*** provista por la cátedra desde un archivo de tipo *.csv* en el cual se definen las variables de entrada. De esta manera nuestros datos iniciales quedan inicializados en dos tuplas **banderaRoja** y **banderaAzul** ambas con las coordenadas de la bandera de cada equipo y un vector de coordenadas iniciales de cada jugador para cada equipo **posRojo** y **posAzul**.

3.2. Orquestación del juego

Toda la información necesaria para una partida está organizada por el **belcebu** de la clase **gameMaster**. Su objetivo es administrar los datos para poder orquestar el funcionamiento del juego. Belcebu sabe el número de ronda, las administra, sabe la cantidad de jugadores por equipo, conoce el tablero, las posiciones de los jugadores, de las banderas, el turno actual y las estrategias de los equipos. Como orquestador de la partida y en favor de poder administrar las rondas también se definen los semáforos encargados de la sincronización de los jugadores.

3.3. Equipos y jugadores

El funcionamiento de los equipos está implementado en la clase Equipo. Cada jugador esta representado por un thread, por lo tanto un equipo es un conjunto de threads que deben ser sincronizados para poder jugar respetando las reglas. Al momento de iniciar una partida, los equipos deben crear sus jugadores con la función **comenzar**, esta inicializa los threads en la función **jugador**, que dado un número de jugador setea la estrategia bajo la cual se desarrollará la partida para dicho equipo. Luego una vez finalizada la última ronda, se llama a la funcion **terminar** que hace los joins correspondientes para cada thread.

4. Ejercicios

4.1. Ejercicio 1: implementaciones del belcebu

1. Completar el constructor de game master: Como ya fue mencionado anteriormente, el propósito de la clase game master es encargarse de la coordinación del desarrollo del juego. En el constructor, belcebu toma los datos de config para inicializar sus variables internas y crear el tablero con las dimensiones determinadas. Pone a los jugadores y las banderas en las posiciones de entrada y finalmente envía un mensaje que nos comunica que ha sido inicializado exitosamente.
   En función de las posteriores implementaciones, se decidió que en el constructor de game master inicializar un semáforo llamado **ronda anterior finalizada** con el valor cero. Este semáforo es utilizado luego en la implementación de las estrategias de juego, para avisar que ya termino una ronda (definida como Rojo → Azul) de manera que el equipo Rojo pueda iniciar una nueva ronda una vez que todos hayan terminado.

2. Implementación de la función **moverJugador(direccion dir, int nroJugador)** : Esta función toma como parámetros una dirección (ARRIBA, ABAJO, IZQUIERDA, DERECHA) y el número de jugador que se quiere mover hacia dicho lado.

Algorithm 1 moverJugador(**direccion** dir, **int** nroJugador)

```
1: posicionJugador ← posicion(turno, nroJugador)
2: proximaPosicion ← proxPosicion(posicionJugador, dir)
3: if !posicionValida(proximaPosicion) then
4:     return nroRonda
5: if !estaVacia(proximaPosicion) & !proxPosicionEsBanderaContraria then
6:     vector<coordenadas> next ← movimientosAlternativos(posicionJugador, dir, proximaPosicion)
7:     i ← 0
8:     while i < ||posiblesMovimientos|| do
9:         proximaPosicion ← posiblesMovimientos[i]
10:        if !estaVacia(proximaPosicion) & !proxPosEsBanderaContraria then
11:            i ← i + 1
12:        else
13:            break
14: else
15:    return nroRonda
16: if !terminoJuego() then
17:    actualizarPosiciones(turno)
18:    if proxPosBanderaContraria then
19:        ganador ← turno
20:    else
21:        moverJugadorTablero(posicionJugador, proximaPosicion, turno)
22: return nroRonda
```

Inicialmente, almacenamos en la variable **posicionJugador** la posición del jugador cuyo número es el que nos pasaron por parámetro. Luego en la variable **proximaPosicion** guardamos la coordenada de la posición a la que nos queremos mover. Si la próxima posición no es válida, es decir, no es una posición dentro del tablero, entonces no hacemos nada y se devuelve el numero de ronda. Luego, identificamos si no hay ningun jugador en dicha posición y si esta es la posicion de la bandera del equipo contrario. Si no pasan ninguna de estas dos cosas entonces debemos buscar movimientos alternativos, pues no se termina el juego ya que no hemos encontrado la bandera contraria y tampoco es posible moverse en la dirección que calculada. En caso de que no hubiera movimientos alternativos válidos devolvemos el numero de ronda.
Para buscar movimientos alternativos se implementó la función **movimientosAlternativos** que devuelve un vector de coordenadas con los posibles movimientos mas sensatos. Esto implica que sea un movimiento que me acerque a la bandera contraria y que sea válido. Luego, recorremos dicho vector fijándonos la coordenada que me da como opción si está libre y si es la posición de la bandera contraria. Si no se cumplen ninguna de esas cosas entonces pasamos al siguiente elemento, si se cumple alguna significa que o bien me puedo mover en esa dirección o que el jugador está por ganar el juego, en cualquier caso paramos de buscar.

Por último, si aún no ha terminado el juego entonces efectuamos el movimiento modificando el valor en el vector de posiciones del jugador con el número que nos pasaron por parámetro. Luego, debemos chequear si habiendo hecho el movimiento se ganó el juego. Si es así, entonces seteamos la variable ganador con el color de equipo correspondiente y sino llamamos a la funcion
moverJugadorTablero(posicionJugador, proximaPosicion, turno) de game Master. Esta función se encarga de modificar el color de la coordenada a la que nos movimos en el tablero y de dejar libre la que abandonamos.

3. **terminoRonda()**: El propósito de esta funcion es ser llamada por cada equipo al momento de informar que ya terminaron de jugar todos los participantes. Lo unico que hace es cambiar la variable "turno"de gameMaster para que ahora contenga al equipo contario y aumentar el número de ronda de la partida. Su mayor utilidad se da en las implementaciones de las estrategias de la clase **Equipo**

4.2. Ejercicios 2 y 3: Desarrollo de los métodos de los equipos

1. **Constructor de equipos:**
Esta función se encarga en principio de inicializar las estructuras internas de la clase. Como se detalló previamente, la clase Equipo se ocupa de implementar y coordinar el funcionamiento de las estrategias mencionadas. Por esta razón, al construir un equipo se inicializan las estructuras de sincronización requeridas para esto:

   - **mutexes de RR:** En la implementación de la estrategia Round Robin, se utilizó un semaforo inicializado en cero para cada jugador. Para lograr esto, al construir un equipo iteramos en un array de semáforos cuya posición corresponde al jugador al que pertenece.
   Como las reglas indican que la primera ronda la inicia el equipo rojo, se inicializa el primer jugador de dicho conjunto en 1.
   - **barrier** El semáforo barrier es utilizado para coordinar que una ronda de jugadas sea terminada por todos los integrantes antes de dar pie a que se inicie la del contrincante.
   - **reiniciarQuantums()** Esta funcion setea por primera vez los *quantums* aleatorios de nuestra estrategia.
   - **Semáforos del belcebu** Están encargados de anunciar al equipo contrario que la ronda del actual terminó y darle paso. Al igual que los semáforos de Round Robin, el del azul está inicializado en cero y el del rojo en la cantidad de jugadores por ser el equipo que juega la primera ronda.

2. **Comenzar(), jugador(int nroJugador) y estrategias (punto 3 incluido)**
La función comenzar es llamada para dar inicio a la partida, en ella iteramos para crear n threads (siendo n la cantidad de jugadores) y cada uno es inicializado en la función **jugador(nroJugador)**. Desde el momento en el que son creados los threads, inician la partida buscando cada uno por un sector la bandera del equipo contrario. Una vez localizada, el equipo rojo empieza a jugar su ronda. La estrategia elegida para la partida está predefinida para ambos equipos, por lo tanto, cada thread sigue las reglas de la misma entrando en un switch que le permite seguir la implementación correspondiente.

   - **Secuencial:** Esta estrategia se limita a elegir quien juega segun su número de jugador hasta que se complete la ronda. Como la implementación está hecha para ambos equipos, lo primero que se verifica es que juegue el del color correspondiente. Para lograr esto, utilizamos los semáforos **turnoRojo)** o **turnoAzul** de belcebu. Estos mantienen a los threads del equipo contrario esperando hasta que el equipo actual complete su turno.
   Un caso que acá podía suceder es que cuando terminaba de jugar el rojo y empezaba el azul, si bien los rojos estaban en su semáforo esperando el turno, si un azul terminaba, y liberaba dicho semáforo para un rojo, esto podía generar trazas donde algunos azules seguían jugando pero ya empezaban a jugar algunos rojos. Para ello mantenemos un atómico que determina si todos los del azul terminaron (solo para el azul, ya que asumimos que la finalización del azul marca el fin de una ronda, y en la que sigue vuelve a empezar el Rojo). De ser así la situación (todos los azules ya terminaron) liberamos el semáforo de rondaAnteriorFinalizada.
   Posteriormente, los jugadores proceden a obtener la posición de la bandera contraria y a setear la posición en la que se deben mover para alcanzarla. Hasta ahora los jugadores habian realizado estas

tareas simultáneamente, sin embargo como las reglas dicen, a la hora de moverse deben hacerlo uno a la vez, de modo que es aquí el lugar en el que pasan por su primer semáforo mutex **moverse**. Una vez lockeado en la sección crítica, se llama a la función **moverJugador(dis, nroJugador)** que describimos en el apartado anterior y aumenta en uno la cantidad de jugadores que ya jugaron. Si esta es igual a la cantidad de jugadores totales por equipo, entonces este jugador es el último en moverse, por lo tanto es quien debe activar el semáforo **rondaAnteriorFinalizada** para darle paso al inicio de la ronda del contrincante, terminando su labor en la sección crítica.
Luego, mantenemos a todos los jugadores en el semáforo **barrier** hasta que el último los libere para poder hacer la señal y así poder dar paso a la ronda del equipo contrario.

Algorithm 2 Estrategia Secuencial:

```
1: turnoColor.wait()
2: if nroRonda = 0 then
3:     rondaAnteriorFinalizada.wait()
4:     rondaAnteriorFinalizada.signal()
5: proxDireccion ← apuntarA(posicionBanderaContraria())
6: */ Inicio seccion critica */
7: moverse.lock()
8: if !terminoJuego() then
9:     moverJugador(proxDireccion, nroJugador)
10: cantidadDeJugadoresQueYaJugaron ← cantidadDeJugadoresQueYaJugaron + 1
11: if yaJugaronTodos then
12:     ultimo ← nroJugador
13:     rondaAnteriorFinalizada.signal()
14: moverse.unlock()
15: */ Final seccion critica */
16: if yaJugaronTodos & ultimo = nroJugador then
17:     cantJugadoresQueYaJugaron ← 0
18:     terminoRonda(Color)
19:     barrier.signal()
20: barrier.wait()
21: rondaAnteriorFinalizada.signal()
22: turnoColor.signal()
```

- **Round Robin:** Como ya se describió anteriormente, el componente principal de esta estrategia es el valor *Quantum*; este nos dice cuantos jugadores harán su movimiento antes de que se termine la ronda.
  Para esta implementación se utilizaron dos conjuntos de semáforos (**mutexesRR**), cada uno de estos corresponde a un número de jugador de manera que el semáforo i da paso al número de jugador i. Esto nos da la ventaja de no tener que tener secciones críticas en el código, ya que cada thread habilita al otro cuando termina de jugar. Tampoco fue necesario utilizar los semáforos de turno de belcebu.
  Inicialmente, ponemos en espera todos los semáforos del equipo que vaya a jugar la ronda actual. Luego una vez que un jugador entra al código de la estrategia, se le pregunta si aún hay *quantum*. Si la respuesta es si, entonces se hace el procedimiento para realizar el movimiento y se decrementa en uno dicho valor. Una vez hecho esto, el jugador actual da paso a que juegue el siguiente haciendo una señal al número de su compañero.
  Si contrariamente, el *quantumRestante* es igual a cero, entonces se procede a reiniciar las variables y dar por finalizada la ronda.
  Por último, si el jugador actual era el último que quedaba por jugar, entonces en lugar de darle la señal a su siguiente compañero, se finaliza el turno del equipo para que juegue el otro.

Algorithm 3 Round Robin:

```
1: mutexesRR[nroJugador].lock()
2: if quantumRestante > 0 then
3:     proximaDireccion ← apuntarA(posicionBandera)
4:     moverJugador(proximaDireccion, nroJugador)
5:     quantumRestante ← quantumRestante - 1
6:     mutexesRR[nroJugador + 1].unlock()
7: else
8:     ultimo ← nroJugador
9:     quantumRestante ← quantum
10:    terminoRonda()
11: if nroJugador = ultimo then
12:    mutexesRRcontrario[0].unlock()
```

- **Shortest distance first:** Al iniciar una partida, los threads se encargan simultaneamente de recorrer distintas secciones del tablero en busca de la posición de la bandera contraria. Luego, antes de que cada jugador tome el camino de la estrategia elegida para la partida, debera pasar por una sección de la funcion en la que se determinará si éste es el que está a menos distancia de la misma. Si el jugador actual es el más cercano entonces seguirá su camino por la implementación de la estrategia y si no, no participará.
  Como se mencionó en la descripción de la estrategia, el criterio de la misma hace que siempre haya un único jugador cuya posición sea la mas ventajosa, por lo tanto las rondas de esta constarán únicamente de un movimiento para cada equipo. Esto hace que al igual que en *Round Robin* no sea necesario tener secciones críticas

Algorithm 4 Shortest distance first:

```
1: turnoColor.wait()
2: proximaDireccion ← apuntarA(bandera)
3: moverJugador(proximaDireccion, nroJugador)
4: terminoRonda()
5: turnoColor.signal()
```

- **Quantum random:** La estrategia propuesta se basa en *Round Robin*, la idea principal es que cada jugador tenga una cantidad de quantum aleatoria y que el mismo se mueva esa cantidad de veces, de manera que en una ronda jueguen todos repetidas veces.
  Para su implementación se inicializa en el constructor de Equipo un vector **quantumsPorJugador** que contiene en la posición i, el quantum asignado para el jugador i.
  Inicialmente usamos al igual que en *Round Robin* los **mutexesRR** para cada jugador; ponemos en espera todos los semáforos de los equipos que vayan a jugar la ronda actual. Luego, dado el número de jugador que está jugando iteramos la cantidad de veces que su quantum nos permita y realizamos esa cantidad de movimientos.
  Por último, si nuestro jugador es el último, hacemos que de por terminada la ronda actual se reinician los quantums para la siguiente y da señal el primer jugador del equipo contrario para iniciar su jugada.
  Si por el contrario, el jugador en cuestión no era el último, le da señal al mutex de su próximo compañero.

Algorithm 5 Quantums random:

```
1: mutexRR[nroJugador].lock()
2: while quantumsPorJugador[nroJugador] > 0 do
3:     proximaDireccion ← apuntarA(posicionBandera)
4:     moverJugador(proximaDireccion, nroJugador)
5:     quantumsPorJugador[nroJugador] = quantumsPorJugador[nroJugador] - 1
6: if nroJugador = ultimo then
7:     terminoRonda()
8:     reiniciarQuantums()
9:     mutexRRcontrario[0].unlock()
10: else
11:    mutexRR[nroJugador + 1].unlock()
```

4.3. Ejercicio 4: Búsqueda de bandera

1. Observación:

   Si bien la implementación provista por la catedra restringe las banderas a coordenadas (1, y) y (x - 1, y) para el equipo rojo y azul respectivamente, este análisis y los algoritmos están hechos pensando en una búsqueda general sobre todo el tablero.

2. Teóricamente:

   Sean un tablero de m x n, y cantidad de jugadores por equipo igual a k:

   El método de búsqueda de bandera 'naive' consiste en iterar sobre el tablero completo, en búsqueda de la bandera del equipo contrario.

   En peor caso la complejidad de este algoritmo es de $O(mn)$

   Para aprovechar el uso de threads, nuestra idea de búsqueda consiste en que cada thread (numeremoslos 1, 2, ..., k) comience iterando por la fila del tablero indicada por su número en búsqueda de la bandera.

   En caso de que la bandera no haya sido encontada (por el mismo u otro thread que haya terminado antes), y que la cantidad de jugadores sea menor a la cantidad de filas del tablero, seguimos iterando por la fila dada por su número + k

   Complejidad del i-ésimo thread:

   Este método de búsqueda hace que la complejidad se mida por thread, donde la iteracion por fila es de $O(n)$, y suponiendo que recorremos todas las filas posibles, es de $O(\frac{m}{k}n)$

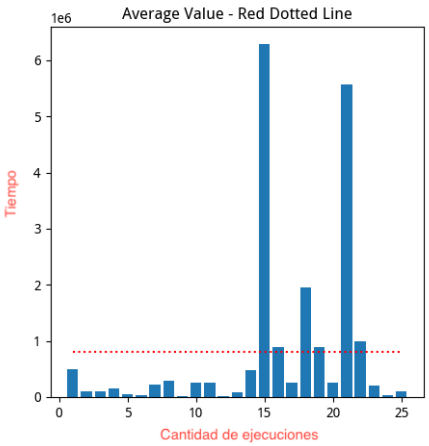
   Dado que el orden y duración de estas busquedas, realizadas por threads, está determinado por el scheduler. En una "mala"disposición del scheduler tenemos k-1 threads que terminaron sus iteraciones sin éxito, y un último thread que encuentra la bandera en su última iteración, siendo así la complejidad total de $k * O(\frac{m}{k}n) \subseteq O(mn)$

   En la práctica, esperamos que la búsqueda paralela de banderas sea mucho más rápida que su implementación single thread naive.

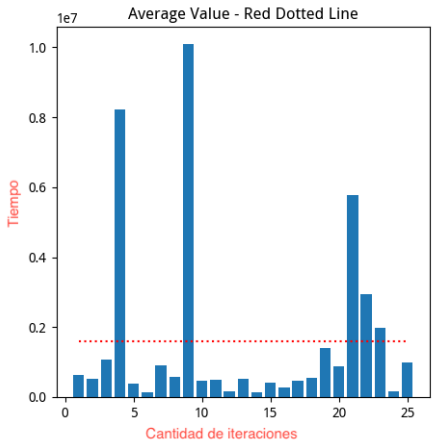
3. Experimentación:

   Dado un tablero de 8x8 posiciones, donde la bandera Roja se encuentra en la posición mas cercana (1,1) y la bandera Azul se encuentra en la posición más lejana (8,8). Medidas las tardanzas de ambos algoritmos en nanosegundos para 25 ejecuciones independientes obtenemos los siguientes datos:

Con algoritmo paralelo:



(a) Búsqueda bandera Roja

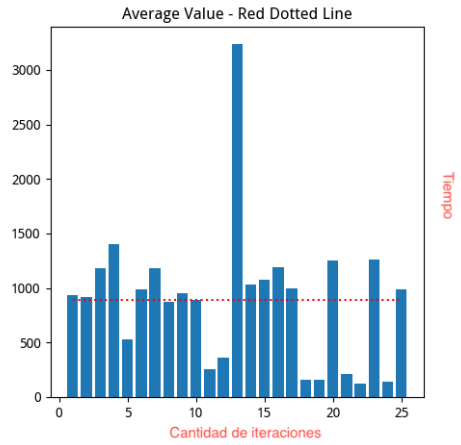


(b) Búsqueda bandera Azul

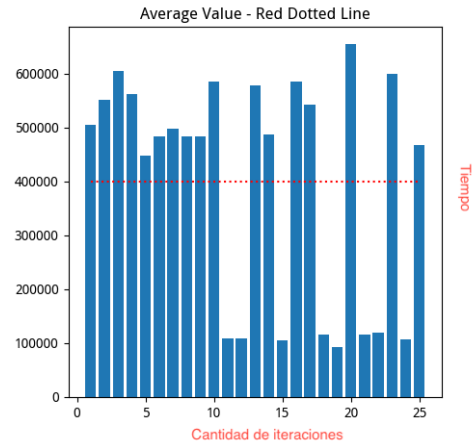
Figura 1: Tardanzas para encontrar las respectivas banderas con el algoritmo paralelo

Obteniendo un resultado de la media para encontrar la bandera Roja: 799533.76 (nanosec).
Obteniendo un resultado de la media para encontrar la bandera Azul: 1606089.84 (nanosec).

Con algoritmo naive:



(a) Búsqueda bandera Roja



(b) Búsqueda bandera Azul

Figura 2: Tardanzas para encontrar las respectivas banderas con el algoritmo naive

Obteniendo un resultado de la media para encontrar la bandera Roja: 891.64 (nanosec)
Obteniendo un resultado de la media para encontrar la bandera Azul: 399527.52 (nanosec)

4. Observaciones sobre los resultados:

Algoritmo y bandera	Tiempo promedio (nanosec)
Paralelo, bandera roja	799533.76
Paralelo, bandera azul	1606089.84
Naive, bandera roja	891.64
Naive, bandera azul	399527.52

Cuadro 1: Comparación de resultados promedios.

Podemos ver que en el mejor caso del algoritmo naive (bandera Roja en posicion (1,1)) donde el algoritmo hace una sola iteración, obtenemos el resultado en tiempo muchísimo mejor que era lo esperado.

Sin embargo, aunque esperabamos resultados similares para el peor caso (bandera en última posición), el algoritmo naive sigue teniendo una media menor.

A pesar de esto, es apreciable que el algoritmo paralelo es muy variante con sus datos, y que en la mayoría de sus ejecuciones tiende a encontrar la bandera rápidamente.

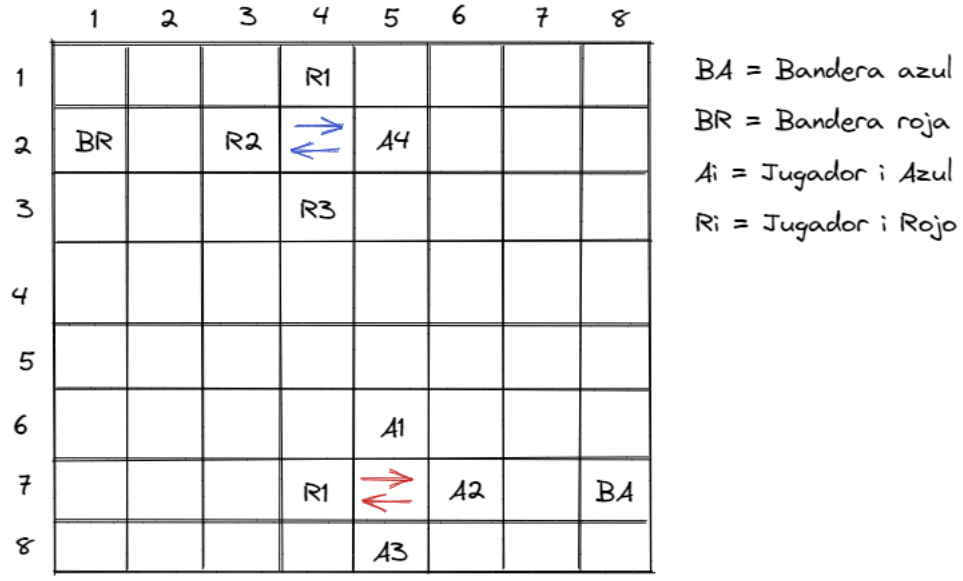
En cambio, el algoritmo naive tiene resultados mucho más estables al correr en 1 solo thread.

4.4. Aclaraciones finales

Toda la experimentación se realizó en una computadora con la siguientes características:

- Sistema operativo: Ubuntu 22.04lts.
- Procesador: Intel Core i5 2.50GHz (8 cores).
- Memoria: 8192MB RAM.

Hay algunas situaciones que se pueden llegar a presentar en el tablero donde el juego no termina.
Un ejemplo de esto es, con la estrategia 'shortest' donde solo se mueve el jugador más cercano, la siguiente situación:



Donde los jugadores no tienen la inteligencia para rodear a sus contrincantes. Si bien nuestra implementación tiene limitaciones, pusimos el foco de este trabajo en utilizar las herramientas correctas de organización para el movimiento de los jugadores.

5. Testing

Para el testeo de nuestra implementación, como el enunciado no aclaraba requerimientos para los mismos, optamos por crear distintos archivos csv con sus respectivas configuraciones para el tablero, cantidad de jugadores, posiciones, etc. De esta manera pudimos evaluar en diversos escenarios que nuestro código y estrategias funcionen como esperabamos.