Trabajo Práctico 1
Captura la Bandera

Sistemas Operativos

Integrante	LU	Correo Electrónico
Teo Kohan	385/20	teo.kohan@gmail.com
Joaquin Morales Pessacq	430/20	joacomp489@gmail.com
Santiago Pla	146/20	santiago_pla@hotmail.com
Martín Santesteban	397/20	martin.p.santesteban@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

1 Introducción

El juego de captura la bandera es un juego que simula una contienda entre dos armadas.

Dos equipos de igual cantidad de jugadores forman cada uno una base de operaciones donde resguardan su bandera. El objetivo del juego es obtener la bandera contraria y llevarla hasta la propia base. El primer equipo en poseer ambas banderas es declarado el ganador.

En la variante que nos compete basta capturar la bandera contraria sin necesidad de retornar hasta nuestra base para ser victoriosos. En particular vamos a analizar una simulación del juego, desarrollado sobre una grilla como se ve en la figura 1.

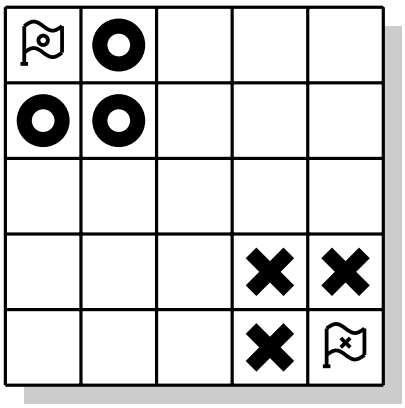

Figura 1: *Ejemplo de un tablero de* 5×5 *y equipos de 3 integrantes.*

1.1 Desarrollo de una partida

Dos equipos de N jugadores compiten por obtener la bandera contraria en una grilla de $n \times m$ casillas. La posición inicial de ambas banderas y los jugadores es dada. El juego se desarrolla en rondas, alternando el equipo que juega. El equipo que esté jugando mueve a sus integrantes de acuerdo a una estrategia preseleccionada. Esta estructura de rondas se repite hasta que algún integrante de algún equipo alcance la bandera contraria, ese equipo es declarado el ganador.

1.1.1 Reglas del juego

R1. Comienza jugando el equipo rojo.

R2. Un jugador puede moverse a una casilla ortogonal a la propia mientras esta no esté ocupada por otro jugador ó la bandera de su propio color.

R3. Un jugador debe moverse si le es posible.

R4. Si un jugador obtiene la bandera contraria el juego termina inmediatamente y su equipo es declarado el ganador.

R5. Una vez que no queden integrantes por mover es el turno del otro equipo.

1.1.2 Estructura

El juego se lleva a cabo en los objetos Belcebú y Equipo. El primero dirige el juego, realizando las transiciones de estado en el tablero, sincronizando las rondas entre los equipos y terminado el juego cuando se haya coronado un ganador. El segundo instancia los jugadores, tiene acceso a la información que es pertinente a los mismos, como las posiciones de las banderas y de los jugadores. El método Jugador es la función a ejecutar por cada jugador, implementada con *threads* de la biblioteca estándar de C++.

2 Sincronización

El mayor desafío es sincronizar a los Jugadores entre sí y con Belcebú.

Para sincronizar a los jugadores la herramienta principal son dos arreglos de semáforos, uno para cada equipo. Donde $A[\,i\,]$ contiene el semáforo correspondiente al i-ésimo jugador del equipo A y su valor indica cuantos movimientos puede realizar el jugador i-ésimo del equipo A.

Ambos equipos pueden comunicarse con Belcebú y con el método Comenzar, inicializan los N *threads* con la función objetivo Jugador. La estrategia a seguir de cada equipo está definida en Jugador y TerminóRonda, donde se lleva a cabo la mayor parte de la sincronización entre ellos. Cuando termina una ronda, TerminóRonda se encarga de iniciar una nueva ronda haciendo uso de las estructuras de sincronización relevantes a la estrategia elegida.

2.1 Estrategias

Una estrategia determina *cuando* y *cuanto* se mueven los jugadores en el tablero. Trayendo distintos problemas de sincronización.

2.1.1 Secuencial

Los jugadores de un equipo se mueven una vez por ronda, en cualquier orden, una única casilla.

Al comenzar una nueva ronda se envía una señal a todos los jugadores del equipo. Estos realizan sus turnos y luego incrementan un contador atómico, si el valor del contador es igual a la cantidad de jugadores se señaliza que terminó la ronda.

Para evitar condiciones de carrera al momento de realizar movimientos se utiliza un *mutex* que evita que más de un jugador pueda moverse al mismo tiempo, garantizando la exclusión mútua al moverse.

2.1.2 Round robin

Se mueven los jugadores de un equipo en orden ascendente, mientras reste *quantum* de una casilla a la vez.

Al comenzar una nueva ronda se envía una señal al jugador número 1, este realiza un movimiento, disminuye el *quantum* restante de la ronda en 1 y envía una señal al siguiente jugador (con *wraparound*, e.g. en un equipo de n jugadores el "siguiente" al jugador número n es el jugador número 1). Si al finalizar un movimiento no queda *quantum* entonces se señaliza que terminó la ronda.

Como cada jugador señaliza al siguiente tras haber completado su movimiento, no es necesaria ninguna estructura de sincronización adicional ya que se garantiza la exclusión mútua.

2.1.3 Shortest

Se mueve una única vez por ronda al jugador más cercano a la bandera contraria.

Al comenzar una nueva ronda se determina el jugador más cercano a la bandera contraria, y se le envía una señal para que éste realice un único movimiento. Luego de moverse se señaliza que terminó la ronda.

Por lo tanto, como siempre hay a lo sumo un único jugador en la sección critica, no es necesaria ninguna estructura de sincronización adicional.

Además notar que al mover al jugador en la posición más cercana a la bandera, en la mayoría de los casos este seguirá siendo el más cercano y consecuentemente será el único en moverse.

2.1.4 Proximidad

La estrategia beneficia a jugadores que estén cerca de aliados, y perjudica aquellos cerca de enemigos.

Dado un radio constante R se calcula la cantidad de aliados y enemigos a distancia menor o igual a R en distancia Manhattan. Sea

$$F^{(+)}(n, R) = \text{cantidad de aliados a distancia menor o igual a } R \text{ del jugador } n.$$
$$F^{(-)}(n, R) = \text{cantidad de enemigos a distancia menor o igual a } R \text{ del jugador } n.$$

Luego sea Q una constante que define una cantidad de movimientos base para cada jugador, la cantidad de movimientos que puede realizar cada jugador está dada por

$$W(n, R) \quad = \max\{1,\, Q + F^{(+)} - F^{(-)}\}.$$

Cada jugador j puede realizar una cantidad de movimientos $W(\#j, R)$ en una ronda (no necesariamente la misma cantidad para todos los integrantes del equipo).

Al comenzar una nueva ronda se calcula $W(\#j, R)$ para todo jugador j del equipo actual y se envían $W(\#j, R)$ señales a cada jugador, que está aguardando que se abra una barrera. Una vez que se envían todas las señales se abre la barrera para permitir a los jugadores realizar sus movimientos. Si al finalizar un movimiento este es el último que le corresponde a este jugador se incrementa un contador atómico, si el valor del contador es igual a la cantidad de jugadores se señaliza que terminó la ronda.

Se utiliza una barrera para restringir el movimiento de los jugadores hasta que se hayan cargado todos los permisos de la ronda, i.e. cargado los semáforos de todo jugador.

Si la barrera no estuviese, al cargar los semáforos estos pueden ser consumidos por los jugadores antes de tiempo, validando que es el último movimiento cuando no lo era e incrementando al contador atómico inadecuadamente. Como consecuencia terminando la ronda precozmente.

Además, de la misma forma que en la estrategia secuencial se utiliza un *mutex* para evitar condiciones de carrera al momento de realizar movimientos. Esto evita que más de un jugador pueda moverse al mismo tiempo, garantizando la exclusión mútua al moverse.

2.2 Guardián del ciclo principal

Es de especial interés analizar el guardián del ciclo principal, que controla cada movimiento de cada jugador. Este ciclo se encuentra en la función `jugador` del archivo `equipo.cpp`.

```
sem_wait(&(belcebu->equipos[equipo][nro_jugador])) == 0  &&
(strat != PROXIMIDAD || sem_wait(&belcebu->barrera) == 0)&&
!belcebu->termino_juego()
```

Analicemos cada parte para ver que no se generan *race conditions:*

- `sem_wait(&(belcebu->equipos[equipo][nro_jugador])) == 0`

  Cada jugador aguarda que se señalice su semáforo personal. Esta variable solo es modificada por Belcebú cuando ocurre el cambio de ronda ó por otros jugadores en la estrategia *round robin*, en cuyo caso esta solo se modifica cuando terminó de moverse el jugador anterior. Como cada jugador evalúa únicamente su semáforo personal, no se generan *race conditions* entre los jugadores.

- `(strat != PROXIMIDAD || sem_wait(&belcebu->barrera) == 0)`

  Esta parte solo aplica en caso de que estemos utilizando la estrategia *proximidad*. Si este es el caso esperamos a que Belcebú nos indique que es momento de jugar al abrir la barrera. Esta se abre únicamente una vez el semáforo de cada jugador haya sido inicializado. Luego, todos los jugadores del equipo podrán moverse en el tablero. Como solo Belcebú puede cambiar la barrera, no se pueden generar *race conditions* al evaluarla entre los jugadores.

- `!belcebu->termino_juego()`

  No terminó el juego, es importante que esta condición sea la última para minimizar la posibilidad de que un jugador juegue un turno cuando la partida ya terminó. En el improbable caso de que esto suceda el movimiento del jugador no será aceptado por Belcebú, quien le devolverá un código de error marcándole que su movimiento no fue computado.

Es por esto que evitamos que se genere una *race condition* al evaluar la guarda del ciclo. La única situación donde una condición podría cambiar luego de haber entrado en el ciclo es la de que el juego no haya terminado. A pesar de esto, como ya se mencionó, Belcebú se encarga de evitar que un jugador se mueva cuando se encuentra en esta situación.

3 Búsqueda de la bandera

Buscar la bandera contraria implica revisar una matriz $M \in \mathbb{N}^{n \times m}$ en búsqueda de una única casilla, $(i, j) \in \mathbb{N}^2$ tal que M_{ij} contenga la bandera contraria.

Para este problema primero vectorizamos la matriz, i.e. la tratamos como un objeto unidimensional.

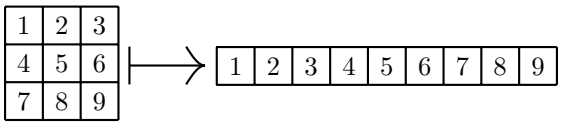

Figura 2: *Vectorización de un tablero de* 3×3.

Luego la trozamos en N trozos iguales del mayor tamaño posible, tantos como la cantidad de jugadores, y un resto que se repartirá de a una casilla.

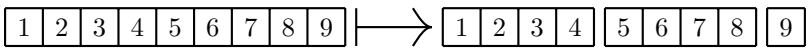

Figura 3: *Trozamiento del tablero de* 3×3 *para* 2 *jugadores.*

Finalmente repartimos una casilla del resto r a cada jugador j tal que $\#j \leq r$.

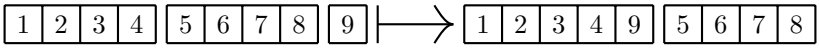

Figura 4: *Asignación del resto.*

De esta forma no existen jugadores i, j tales que sus vectores asignados difieran en cantidad de casillas en más de una casilla. Luego cada jugador revisa sus casillas y cuando encuentra la bandera retorna sus coordenadas.

4 Testing

Confirmar la corrección de un programa paralelo es más complicado que uno secuencial. Además de la complejidad añadida por tener varios *threads* ejecutándose en paralelo surge la posibilidad de encontrarnos con *race conditions*, donde dependiendo de la traza del programa, el comportamiento difiere incluso en formas que no fueron contempladas.

Para asegurarnos mientras realizábamos el proyecto de no introducir *bugs* desarrollamos un *suite* de *tests* que constataba que las diferentes estrategias siguieran funcionando aún luego de introducir funcionalidad nueva. Para cada estrategia se ejecutaba una configuración y aseguraba que terminaba la ejecución y adicionalmente se constataba que ocurriese *deadlock* en ciertas configuraciones problemáticas.

A lo largo del código se introdujeron varias sentencias `assert` que verificaban propiedades inmutables del estado del programa. Estas fueron de gran utilidad en el desarrollo temprano y para darnos seguridades fundamentales luego.

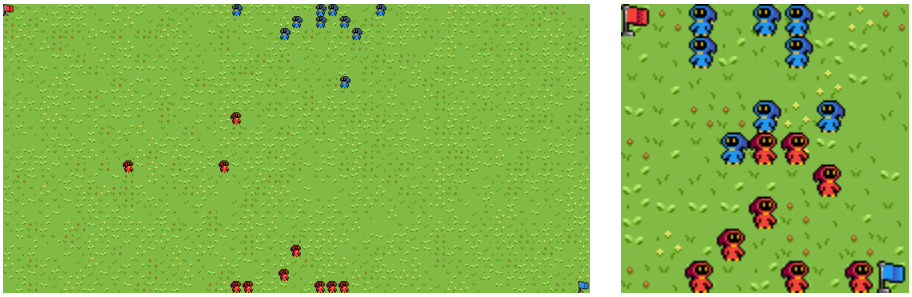

Figura 5: *Cuadros de los `.gif` animados donde observamos el comprotamiento de los agentes.*

Como además es difícil entender qué es exactamente lo que falla cuando ocurre un error desarrollamos un sistema donde cada movimiento es registrado en un archivo, que luego se compila y renderiza en un archivo `.gif` utilizando alguno sprites. Luego observando el `.gif` (como se ve en la figura 5) se pueden notar comportamientos indeseados (jugadores atravesando otros jugadores, turnos fuera de orden, más movimientos de los que deberían).

5 Experimentación

A continuación realizaremos distintos experimentos tanto respecto a las estrategias en sí, como en cuanto a la función de búsqueda de la bandera. El enfoque principal será el análisis del tiempo de espera y de cómputo.

5.1 Cantidad de movimientos y tiempo total vs. cantidad de jugadores

Este experimento tratará de mostrar cómo se comportan las distintas estrategias al variar la cantidad de jugadores, en cuanto al tiempo total en llegar a la bandera contraria por un lado, y la cantidad de movimientos necesarios por otro.

Se fijó un tablero de tamaño 40×40, y un *quantum* de 10 para todas las estrategias. Optamos por no variar estos elementos ya que los resultados deberían ser análogos, sin importar su valor. La cantidad de jugadores variará entre $[1, 30]$, con *steps* de 5.

5.1.1 Hipótesis

Para comenzar, creemos que la cantidad de movimientos de la estrategia *shortest* se verá constante, ya que siempre se mueve un solo jugador de a una casilla, y las dimensiones del tablero no cambian. En cuanto al tiempo total, tampoco debería crecer, por el mismo argumento.

La estrategia *secuencial* debería por naturaleza crecer en cantidad de movimientos de manera lineal, ya que la cantidad de rondas r necesarias para llegar a la bandera se mantiene constante (se requieren $r \cdot n$ movimientos para terminar la partida).

Creemos que la estrategia *round robin* se verá igual a la secuencial, hasta llegar a los 10 jugadores, ya que es el valor del *quantum* fijado. A partir de este punto, la cantidad de movimientos se verá estancada, ya que sin importar la cantidad de jugadores, sólo se moveran los primeros 10. En cuanto al tiempo, éste también será menor respecto a *secuencial*, puesto que al mover menos jugadores, será más rápida la llegada a la bandera contraria.

Por último, la estrategia de *proximidad* tiene el potencial de ser de las que menos tiempo le tome llegar a la bandera contraria. Esto sucedería ya que en cada ronda, los jugadores suelen realizar mucho más que un sólo movimiento (a diferencia de *shortest*, o *secuencial*). Esto también implicaría que la cantidad de movimientos totales sea menor.

5.1.2 Resultados

Comencemos analizando los resultados de la estrategia *shortest* de la figura 6. Al principio nos sorprendió ver en el gráfico de la derecha que la cantidad de movimientos disminuye a medida que se aumentan los jugadores. Sin embargo, luego de analizarlo llegamos a la conclusión que eso se debe a la posición inicial de los jugadores nuevos: éstos aparecen cada vez más cerca de la bandera contraria, por lo que son luego elegidos por la estrategia para recibir el movimiento en cada ronda, llegando a la bandera de forma más rápida (consecuentemente realizando una menor cantidad de movimientos). También nos sorprendió el hecho de que el tiempo total aumente, ya que no se correlaciona con que la cantidad de movimientos sea menor. La justificación que encontramos para esto es el *overhead* que aparece al crear más *threads*. De todas formas, el tiempo va desde 0.00 a 0.005, lo cual es prácticamente despreciable.

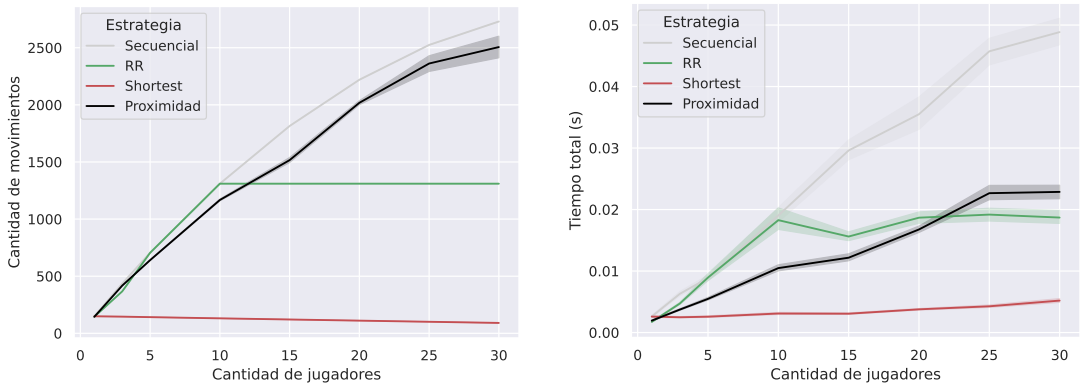

Figura 6: *izquierda: tiempo total (s) vs. cantidad de jugadores - derecha: cantidad de movimientos vs. cantidad de jugadores (graficado con intervalos de error)*

Respecto a *secuencial* y *round robin*, se cumple lo que creíamos que sucedería: mientras que el *quantum* sea menor o igual a la cantidad de jugadores, ambas estrategias se comportan igual.

Luego, efectivamente *round robin* hace la misma cantidad de movimientos, ignorando la cantidad de jugadores, mientras que *secuencial* sigue aumentando, ya que en cada ronda todos deben moverse una vez. Esto mismo lleva a que el tiempo de *secuencial* siga aumentando, a diferencia de *round robin*, que encuentra cierta estabilidad, ya que le toma casi siempre el mismo tiempo llegar a la bandera.

La reducción en cantidad de movimientos y tiempo en la estrategia *secuencial* tiene una justificación similar a lo que sucede con *shortest*. La posición inicial de los nuevos jugadores más cerca de la bandera contraria lleva a que la cantidad de rondas necesarias disminuya.

Finalmente, es interesante la comparación que se puede realizar entre *proximidad* y *secuencial*. Si bien la cantidad de movimientos es casi la misma, los tiempos son considerablemente distintos. Esto se debe a que los jugadores que van por delante se mantienen juntos, contrarrestando la aparición de enemigos y manteniendo una cantidad alta y estable de movimientos, reduciendo así la cantidad de rondas por partida, lo cual disminuye el tiempo total. Por otro lado, en *secuencial*, el jugador que se encuentra adelante no sólo tiene que esperar a que todo el equipo completo se mueva (esto sucede en *proximidad* también) sino que cuando le toca, solo se mueve una vez, por lo que la cantidad de rondas será mucho mayor y por ende el tiempo total será más alto.

5.2 Waiting time vs. tamaño de tablero

Este experimento se concentrará en comparar el *waiting time* total de las distintas estrategias, para tableros de tamaño $n \times n$, con $n \in [50, 300]$ (con *steps* de 50). La cantidad de jugadores será 10, y el *quantum* también será de 10. Llamamos *waiting time* al tiempo que están los *threads* en los semáforos, esperando moverse (únicamente de los que logran moverse en algún punto). Esto implica que no se cuenta el tiempo de los jugadores que nunca se mueven, como en la estrategia *shortest*, o en *round robin* cuando se excede la cantidad de jugadores necesarios para terminar la partida.

5.2.1 Hipótesis

El *waiting time* de *shortest* debería ser prácticamente nulo ya que no hay concurrencia al moverse un único jugador a la vez. El *waiting time* de las estrategias *secuencial* y *round robin* debería ser parecido ya que en ambos se esperan todos entre sí hasta completar una ronda. Sin embargo en *secuencial*, ya que no hay un orden de turnos preestablecido, también se debe conformar una sección crítica al mover, lo cual incurrirá en un pequeño extra de *waiting time* (justamente para ver esta diferencia fijamos el *quantum* igual a la cantidad de jugadores).

En proximidad nos encontramos en una situación similar a la de *secuencial* ya que no hay un orden preestablecido, pero adicionalmente cada jugador se deberá mover más de una vez por lo tanto el waiting time deberá ser mayor.

5.2.2 Resultados

Comenzamos viendo que efectivamente *shortest* tiene un *waiting time* cercano a 0. Además, con el aumento de las dimensiones del tablero, el tiempo de espera se mantiene constante.

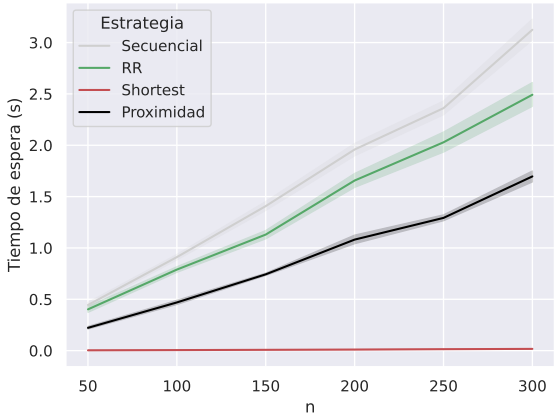

Figura 7: *tiempo de espera (s) vs. tamaño de tablero* $(n \times n)$ *(graficado con intervalos de error)*

Al ver los gráficos de *secuencial* y *round robin*, se observa, como se esperaba, un comportamiento similar entre ellas. La diferencia se debe al tiempo de espera de los *mutex* al entrar a la sección crítica en *secuencial*.

En cuanto a los resultados de *proximidad*, si bien al principio tuvimos problemas asimilando el gráfico, ya que no encontrabamos razón aparente para que tarde menos que *secuencial*, llegamos a una respuesta luego de discutirlo y ver el experimento 5.1. La razón por la cual *proximidad* tiene un *waiting time* total menor simplemente se debe a que ésta estrategia termina más rápido que *secuencial* y *round robin* (figura 6).

5.3 Waiting time vs. cantidad de jugadores

En este experimento se compara el *waiting time* total de las distintas estrategias, para un tablero de tamaño de 40×40, la cantidad de jugadores será n con $n \in [5, 30]$ (con *steps* de 5), y el *quantum* será 10.

5.3.1 Hipótesis

El *waiting time* de *shortest* debería ser constante respecto a la cantidad de jugadores y además debería ser relativamente pequeño ya que el tiempo de ejecución total es corto. El *waiting time* de *secuencial* esperamos que sea comparable al de *round robin*, en los dos se deben esperar todos los jugadores entre sí. *Secuencial*, ya que no hay un orden de turnos preestablecido, contiene una sección crítica, lo cual suma un extra de *waiting time* (justamente para ver esta diferencia fijamos el *quantum* igual a la cantidad de jugadores).

Proximidad será similar a *secuencial* ya que no hay un orden preestablecido y también tiene una sección crítica. Como adicionalmente cada jugador se mueve más de una vez, el *waiting time* deberá ser mayor.

5.3.2 Resultados

shortest tiene un *waiting time* cercano a 0 y no aumenta en función a la cantidad de jugadores.

round robin se comporta como *secuencial*, hasta la marca de 10 jugadores, luego *round robin* llega a un estrepitoso párate ya que los nuevos jugadores nunca reciben su turno y por lo tanto no son considerados para el calculo del *waiting time* (si consideraramos a esto como *waiting time* entonces sería el mayor al *waiting time* de *secuencial*).

secuencial se comporta de forma cuadrática, cada jugador debe esperar el turno del que se agrega $(n-1 \times n)$, esto sigue escalando a medida que aumenta la cantidad de jugadores.

proximidad se comporta de forma cuadrática pero con un coeficiente menor, si bien cada jugador debe esperar a los demás como en *secuencial* y *round robin* al terminar la ejecución antes como se ve en la figura 6 el *waiting time* total será menor.

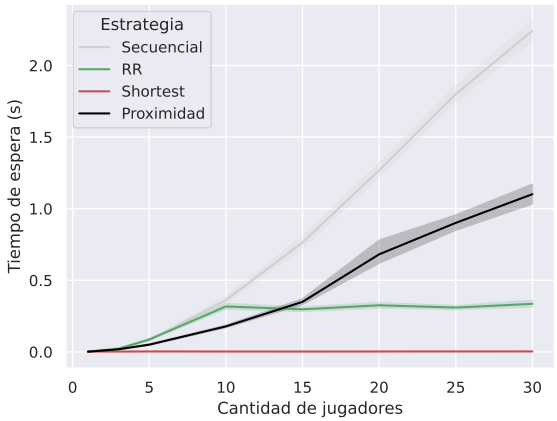

Figura 8: *tiempo de espera (s) vs. cantidad de jugadores (graficado con intervalos de error)*

5.4 Búsqueda de bandera - Tablero cuadrado

Vamos a experimentar cómo la cantidad de *threads* en la función de búsqueda de la bandera contraria, mencionada en la sección 3, impacta en el tiempo de ejecución de la misma.

Consideraremos dos tableros cuadrados, uno de 1000×1000 y otro de 10000×10000 casillas. En cuanto a la cantidad de *threads* que se utilizarán para dividir el trabajo, tomaremos valores en el rango $[1, 16]$.

Para el tablero de 1000×1000, se graficará el promedio de tomar 512 muestras por equipo (en total, 1024 muestras). Por otro lado, el tablero de 10000×10000 tendrá un respaldo de 256 muestras por equipo (en total, 512 muestras). La gran cantidad de muestras intenta reducir la varianza.

5.4.1 Hipótesis

A medida que incrementa la cantidad de threads, se espera que los tiempos disminuyan. Esperamos que exista cierta cantidad de threads para la cual la mejoría se vuelva despreciable debido a que a mayor cantidad de *threads*, menor es la diferencia de trabajo entre ellos y se vuelve más relevante el *overhead* de sincronizar cada thread en ejecución.

Se espera observar el mismo comportamiento para ambos casos, donde los tiempos aumentan linealmente con la cantidad de casillas.

5.4.2 Resultados

Los gráficos de la figura 9 muestran resultados similares, relativos al tamaño de cada tablero. Se observa una mejora en cuanto al tiempo, con únicamente dos *threads* se reduce el tiempo de ejecución a la mitad en ambos casos.

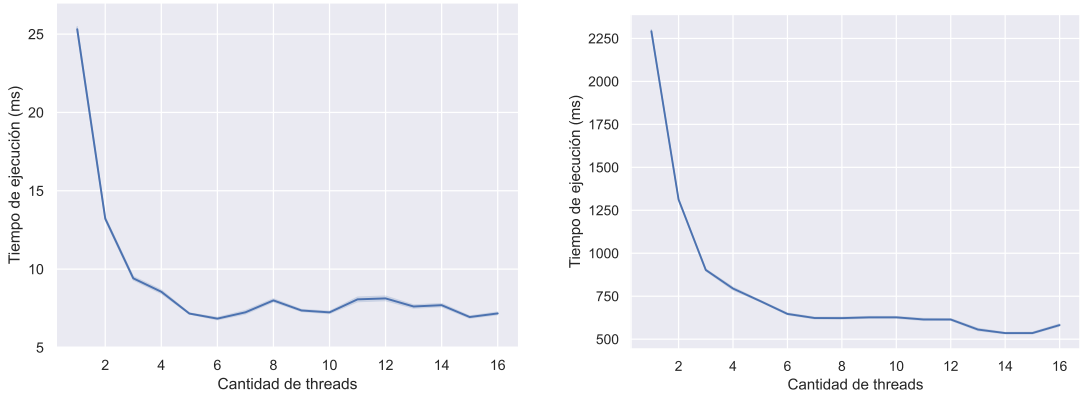

Figura 9: *izquierda:* 1000×1000*, 1024 muestras. derecha:* 10000×10000*, 512 muestras.*

Dicha mejora continúa en ambos gráficos hasta llegar a los 6 *threads*. El gráfico de la izquierda es menos "estable" que el de la derecha debido a que los resultados son tiempos muy pequeños (entre 5 y 10 milisegundos). Con 6 *threads* se alcanza el valor mínimo. Por otro lado, el gráfico de la derecha, que tiene menos muestras a comparación del anterior es más estable, puesto que trabaja con tiempos mayores. No es notable la diferencia entre utilizar de 7 a 12 *threads*. El mínimo se alcanza a los 15 *threads*.

La razón de dicho fenómeno se debe a que al aumentar la cantidad de *threads virtuales*, estos pueden ser asignados a *threads reales* ociosos. Eventualmente la cantidad de *threads* virtuales excede a los reales, además de "pisarse" con *threads* utilizados para trabajos del sistema operativo, en este punto cada *thread* agregado no aporta rendimiento adicional.

Por lo visto, podemos además concluir que a partir de los 7 *threads* el rendimiento es muy bueno, tanto en el tablero de tamaño 1000×1000 como en el de 10000×10000, por lo que seleccionaríamos dicha cantidad en caso de tener que generalizar el método para distintos tableros.

5.5 Búsqueda de bandera - Tablero rectangular

Es de interés analizar el tiempo de ejecución en función de la cantidad de *threads* dependiendo de las dimensiones del tablero en sí. Consideraremos dos tableros, uno de 10000×1000 y otro de 1000×10000 casillas. La cantidad de *threads* en el rango $[1, 16]$, y se tomarán 256 muestras por equipo (512 en total) para cada caso.

5.5.1 Hipótesis

Mientras que la cantidad total de casillas del tablero sea la misma, los threads en ambos casos se repartirán la misma cantidad de casillas y como consecuencia se observarán los mismos tiempos de ejecución.

5.5.2 Resultados

La figura 10 demuestra que no impactan las dimensiones del tablero al tiempo que le toma a los jugadores buscar la bandera del equipo contrario. Los resultados son idénticos a lo largo de toda la curva, ya que siempre se recorre todo el tablero.

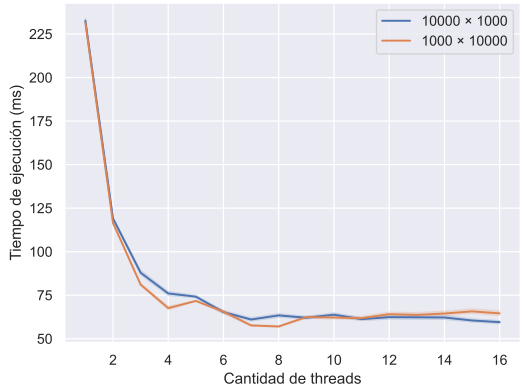

Figura 10: *Tablero* 10000×1000 *y tablero* 1000×10000*, 512 muestras.*

Notar la diferencia entre usar uno o más *threads*. Dos *threads* reducen el tiempo de cómputo a la mitad, con mayor cantidad de threads reduciendo aún más el tiempo de cómputo. A partir de cierta cantidad de *threads*, $8\sim$, la mejoría se ve "estancada". Este resultado es muy similar al visto en el experimento anterior.

Entendemos que el comportamiento de los experimentos es similar debido a que como el tablero se vectoriza, es indistinta su configuración bidimensional ya que se transforma en un vector unidimensional. Luego el único factor que influye en el tiempo de ejecución es la cantidad de casillas del tablero.

6 Comentarios y decisiones menores

6.1 Movimiento

Al mover un jugador, se determina una dirección para acercarse a la bandera contraria. Puede suceder que se encuentre en una posición que lo obligue a retroceder, como en la figura 11.

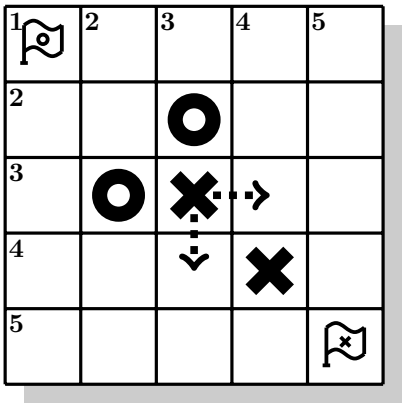

Figura 11: *El jugador en la fila 3 columna 3 quiere dirigirse a la bandera contraria ubicada en la fila 1 columna 1 pero no existe un movimiento válido que reduzca su distancia al objetivo.*

Cuando deba moverse de nuevo, realizará el mismo movimiento, bloqueándose permanentemente. Para evitar estos bloqueos, se introdujo un elemento de azar. Al evaluar los posibles movimientos, se selecciona aleatoriamente entre aquellos que acercan al jugador a la bandera como en la figura 12.

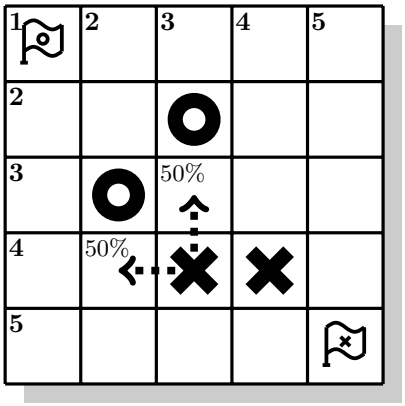

Figura 12: *Selección al azar de un movimiento.*

De esta forma evitando en ciertos casos que un jugador quede atascado permanentemente. Notar

que a veces hay una única dirección óptima, en cuyo caso el azar no entra en juego.

7 Conclusión

A lo largo de este trabajo, pudimos resolver diferentes problemáticas que surgieron en la implementación de programas que se ejecutan de forma paralela (como por ejemplo las condiciones de carrera que se presentaron al tener variables compartidas) mediante el uso de herramientas de sincronización como los semáforos y variables atómicas. Notar además que la herramienta de semáforos fue utilizada en distintos formatos, tales como "mutex" o *barreras*, dependiendo de la situación en particular. Estas problemáticas se presentaron en la implementación del juego "capturar la bandera".

Experimentamos con distintas estrategias a seguir por los equipos, realizando un análisis tanto de los tiempos de espera como de ejecución y cantidad de movimientos totales. Para esto, se observó la relación entre los tiempos con la cantidad de jugadores y las dimensiones del tablero. Pudimos concluir que la estrategia *shortest* es la más rápida (ejecuta la menor cantidad de movimientos y en el menor tiempo), ya que el problema no pide necesariamente que todos los jugadores se muevan y como consecuencia mover uno solo es la mejor forma de ganar. Sin embargo, en cuanto a las estrategias que usan a todos los jugadores concluimos que la más eficiente es *round robin*, ya que es la primera en terminar y es la que genera menores tiempos de espera una vez que se tienen más jugadores que *quantum*. Si este no es el caso, la estrategia más rápida es *proximidad*.

También pudimos observar que los tiempos de espera aumentan de forma lineal con las dimensiones del tablero.

Por otro lado, también utilizamos *threads* para agilizar la ejecución de la búsqueda de la bandera contraria, comparando su rendimiento al paralelizarla en mayor o menor medida. Vimos que cuantos más *threads* se utilizan, más rápidos son los tiempos de ejecución. Sin embargo llega un punto en el que la mejoría es despreciable. En el proceso descubrimos que los *threads* son una herramienta muy útil, aunque es necesario tener algunas consideraciones en cuenta a la hora de implementar una solución que haga uso de los mismos.