Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Algoritmos y Estructuras de Datos III

Trabajo Práctico 3

Autores:
Nicolás Chehebar, mail: *nicocheh@hotmail.com*, LU: 308/16
Matías Duran, mail: *mato_ fede@live.com.ar*, LU: 400/16
Lucas Somacal, mail: *lsomacal@gmail.com*, LU: 249/16

Índice

1. El Juego **2**
1.1. Descripción . . . 2

2. Jugador Óptimo **2**
2.1. El algoritmo . . . 2
2.1.1. El Pseudocódigo . . . 3
2.2. Poda α - β . . . 4
2.2.1. Pseudocódigo . . . 4
2.3. Complejidad . . . 5
2.4. Experimentación . . . 6
2.4.1. Contexto . . . 6
2.4.2. Sin poda . . . 6
2.4.3. Con poda . . . 8

3. Jugador parametrizable **9**
3.1. El algoritmo . . . 9
3.1.1. El Pseudocódigo . . . 9
3.2. Complejidad . . . 11
3.3. Experimentación . . . 11

4. Grid Search **12**
4.1. El algoritmo . . . 12
4.1.1. El Pseudocódigo . . . 12
4.2. Experimentacion . . . 12
4.2.1. Primera exploración . . . 12
4.2.2. Modificaciones . . . 13
4.2.3. Grid recomendado . . . 14
4.2.4. Redefinición de mejorar . . . 14

5. Algoritmo Genético **16**
5.0.1. El Pseudocódigo . . . 16
5.1. El algoritmo . . . 18
5.2. Experimentacion . . . 18
5.2.1. Funciones del Algoritmo Genético . . . 18
5.2.2. Jugadores . . . 23

6. Análisis comparativo con paper **26**

7. Conclusiones **27**

1. El Juego

1.1. Descripción

El juego es una generalización del popular 4 en línea [1]. Consiste en una grilla de M filas y N columnas en la cual dos jugadores colocan alternadamente una ficha propia (identificada con rojo las de un jugador y azul las del otro). Las fichas se pueden colocar en cualquier columna de la grilla y una vez elegida la columna, esta determina el movimiento, ya que irá a la fila de "más abajo" (la de numeración más baja) que esté desocupada. El objetivo de un jugador será lograr tener un línea recta (diagonal, vertical u horizontal) de C fichas propias. Cuando esto suceda, el jugador ganará el partido. Además, cada jugador dispone de P fichas. En caso de que ambos se queden sin fichas y ninguno haya ganado, la partida finaliza en empate. También se da un empate si la grilla queda llena (y ninguno había ganado). Se trata de una generalización del 4 en línea ya que si tomamos parámetros $M = 6, N = 7, C = 4, P = 21$ se replicarían las condiciones iniciales del juego.

En los ejemplos de la Figura 1 se pueden observar distintas configuraciones posibles del juego. La Figura 1 (a) muestra un tradicional 4 en línea donde el jugador azul ha conseguido la victoria porque abajo a la izquierda ha conseguido formar una línea horizontal de 4 fichas propias. La Figura 1 (b) es un ejemplo de 3 en línea (no confundir con el nombre español del tatetí) en un tablero de 4 filas y 3 columnas. En este caso se ha dado un empate pues ninguno de los dos jugadores ha podido formar una línea de 3 y ya se ha llenado el tablero.

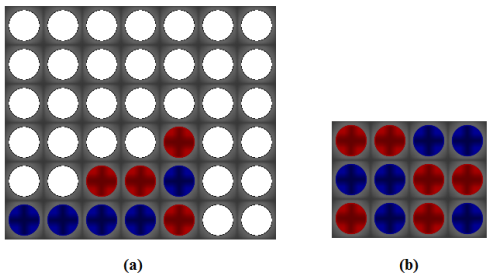


Figura 1: (a) Ejemplo de victoria del azul en un 4 en línea tradicional. (b) Ejemplo de empate con $c = 3$ y un tablero de 4×3

2. Jugador Óptimo

2.1. El algoritmo

El algoritmo del jugador del punto 1.a brinda un jugador óptimo. Nos asegura que este jugador hará la estrategia ganadora si hubiera. En caso de que esta no exista, realizará una de empate. Y si tampoco existiera esa, jugará indistintamente sabiendo que perderá. Este análisis se realiza jugada a jugada. Para ejemplificar esto, podría suceder que el oponente tenga la estrategia ganadora y en ese caso nuestro jugador hará cualquier movida indistintamente (pues sabe que perderá), pero si en la próxima jugada el oponente no realiza la correspondiente a su estrategia ganadora y da un nuevo estado del tablero en el que esta vez nuestro jugador tiene estrategia ganadora, nuestro jugador jugará y ganará ya que ahora sí tiene estrategia ganadora.

Para lograr esto, utilizamos una técnica algorítmica similar al Backtracking en el sentido de que exploramos todas las soluciones posibles y nos quedamos con la óptima. Pero esta vez tenemos dos jugadores interviniendo en la situación donde lo que uno busca es todo lo contrario a lo que busca el otro. Podemos decir que un tablero finalizado tiene 3 puntajes posibles, 1 si ganamos nosotros, 0 si es empate, -1 si ganó el otro (podríamos sino generalizarlo para todo tablero y que haya un cuarto valor que sea inválido si aún no hemos calculado el valor de dicho tablero). Así, lo que sabemos es que turno a turno, uno quiere maximizar el puntaje y el otro minimizarlo. Es por esto que dicha técnica algorítmica se llama Minimax.

De esta manera, igual que en Backtracking tenemos un árbol de ejecución donde cada nodo es un estado del tablero y la raíz es el tablero vacío. Cada nodo (que no sea hoja) tendrá N hijos donde cada uno representará que la próxima jugada fue en alguna de las N columnas. De esta forma, recorremos todos los tableros posibles. Según quién comience, en el primer nivel trataremos de maximizar o minimizar, en el siguiente lo contrario y así sucesivamente. Todos los niveles impares minimizarán si empieza el contrincante

[1] *https : //es.wikipedia.org/wiki/Conecta_4*

y maximizarán si empieza nuestro jugador. El que maximiza le asignará a su nodo un puntaje que será el máximo de los puntajes de todos sus hijos. Análogamente el que minimiza le asignará a su nodo un puntaje que será el mínimo de los puntajes de todos sus hijos.

Así, ejecutando dicho algoritmo la raíz tendrá la información de quién tiene la estrategia ganadora, o que ambos pueden asegurar el empate según quién empiece y haya un 1, 0 o -1. Esta es la idea general del algoritmo, lo veremos más claro en pseudocódigo.

2.1.1. El Pseudocódigo

Comenzaremos con el `main` que es donde se hace el intercambio con el juego en sí. Para ello, primero se le ingresan los parámetros del tablero y las condiciones de juego y luego se desarrolla la partida. Esta termina cuando se lee el mensaje ”ganaste” ”perdiste” o ”empataste”. En cada iteración del ciclo (es decir, cada turno) se llama a la función `minimax` (que se detallará luego y es el algoritmo en sí que resuelve el problema).

Antes del ciclo hay un `If` que es la ejecución del primer turno si empezamos nosotros a jugar ya que el ciclo comienza con la jugada del otro o la finalización del juego. Esto está determinado por un mensaje que se lee al iniciar el ciclo; si es un número, indica en qué columna jugó el adversario, si es el *string* ”ganaste”, ”perdiste” o ”empataste” termina la ejecución.

Algorithm 1: Main

```
leer parámetros: color, oponent_color, columns, rows, c, p
inicializar tablero
leer quién empieza
if empiezo yo then
    movida = minimax(rows, columns, c, p, tablero, true, -1).first
    AgregarAtras(tablero[movida], 1)
    resto 1 a p
    jugar(movida)
end
while true do
    leer mensaje
    if el mensaje es de finalización then
        break
    end
    AgregarAtras(tablero[movida del otro jugador], 2)
    resto 1 a p
    movida = minimax(rows, columns, c, p, tablero, true, -1).first
    AgregarAtras(tablero[movida], 1)
    resto 1 a p
    jugar(movida)
end
```

El algoritmo del `minimax` devuelve una tupla de enteros donde el primero es la columna donde se consigue el mejor valor posible, es decir, la jugada óptima desde el tablero de entrada, y el segundo es el mejor valor que se puede conseguir al hacer esa jugada. Se usaron los valores 1, 0 y -1 para asignar los valores respectivos a ganar, empatar o perder. Además, se usó ∞ como número de relleno cuando todavía no se había inicializado.

REVISARRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR

Las funciones `gane` y `perdi` devuelven **`true`** si con la jugada se gana la partida o se pierde, respectivamente. Ambas consisten en llamar a la funcion `ganoJugador` pasándole como jugador al 1 o al 2

Algorithm 2: Minimax

Input: **int** rows, **int** columns, **int** c, **int** p, *matriz*(**int int**) tablero, **bool** maximizo, **int** ultimajugada

```
if gane then
    return (∞,1)
end
if perdi then
    return (∞,-1)
end
if empate porque no hay más fichas then
    return (∞,0)
end
if empate porque se llenó el tablero then
    return (∞,0)
end
recursión llamando al siguiente en el árbol de minimax
if Maximizo then
    return (columna del máximo, máximo)
else
    return (columna del máximo, máximo)
end
```

respectivamente. Esta función va verificando las distintas condiciones de victoria para el jugador que se le pasó por parámetro (es decir, si con la última jugada se forma c en línea vertical, horizontal o diagonal) hasta que encuentre alguna devoliendo **true**. Si no encuentra, devuelve **false**.

2.2. Poda α - β

La poda α - β consiste en disminuir la cantidad de accesos a nodos del árbol de minimax a partir de información obtenida por accesos previos a otros nodos. La idea general es que, si ya se sabe que por un camino se gana, no hace falta recorrer otra pues, en el mejor de los casos, igualamos el resultado anterior.

El algoritmo se denomina α - β por el nombre de las variables donde se guardan el máximo y el mínimo. Más precisamente, α es el valor de la mejor opción hasta el momento del camino del máximo. Análogamente, β es el valor de la mejor opción hasta el momento del camino del mínimo. De este modo, cuando se analice un nodo para determinar el camino máximo, se compara con el valor de α y, si es menor, no se sigue buscando. Del mismo modo sucede con el mínimo: si es mayor a β, se deja de buscar.

Cuando esto no sucede, es decir, se examina el camino y se encuentra un valor mejor (ya sea, mayor a α o menor a β) que el que se tenía hasta entonces, se actualizan las variables.

En nuestro problema, el número del nodo es 1, 0 o -1 para ganar, empatar o perder, por lo que la poda se puede simplificar diciendo que, si ya se conoce un camino que asegura la victoria a partir de cierto tablero, no hace falta mirar el resto de las opciones que se desprenden de ese tablero.

2.2.1. Pseudocódigo

Para la poda se reutilizó el mismo código y se hizo un cambio al hacer la recursión. Para no copiar todo el pseudocódigo, a continuación se lo presenta con la inicialización previa de las variables *posibles, mejorPos, mejorValorMaximizo* y *mejorValorMinimizo*. *posibles* es un vector de tuplas de enteros donde cada elemento representa una columna del tablero. El primer elemento de la tupla es la columna en sí y el segundo es ...**COMPLETAR**... Las otras variables se inicializan con los valores del primer elemento de la primera posición de *posibles*, $-\infty$ y ∞ respectivamente.

Algorithm 3: minimaxPoda

Input: **int** rows, **int** columns, **int** c, **int** p, matriz(**int**, **int**) tablero, **bool** maximizo, **int** α, **int** β, **int** ultimajugada

vector(pair(**int**,**int**)) *posibles* = [(1, ∞),..,(*columns*, ∞)]
int *mejorPos* = *posibles*[0].first
int *mejorValorMaximizo* = $-\infty$
int *mejorValorMinimizo* = ∞
for *todos los posibles* **do**
    matriz(**int**, **int**) *tablero*2 = *tablero*
    actualizar *tablero*2 según quién juega
    posibles[*i*].second = recursión con *tablero*2, $p-1$ e invirtiendo el valor de verdad de *maximizo*
    if $maximizo \wedge posibles[i].second == mejorValorMaximizo$ **then**
        actualizo *mejorPos* y $mejorValorMaximizo$
    end
    if $\neg maximizo \wedge posibles[i].second < mejorValorMinimizo$ **then**
        actualizo *mejorPos* y $mejorValorMinimizo$
    end
    α = max(α, mejorValorMaximizo)
    β = min(β, mejorValorMinimizo)
    if $\alpha \geq \beta$ **then**
        break
    end
end
if *maximizo* **then**
    return ($mejorPos$, $mejorValorMaximizo$)
else
    return ($mejorPos$, $mejorValorMinimizo$)
end

2.3. Complejidad

Veremos que la complejidad es $\mathcal{O}(a^b)$ donde $a = |ColumnasDondeEsPosibleJugar|$ y $b = \text{mín}\{|FichasRestantes|, |CasillasLibresEnElTablero|\}$ que es en particular acotable por $\mathcal{O}(N^{\text{NM}})$ ya que a lo sumo se puede jugar en N columnas (no hay más en el tablero) y a lo sumo hay $N * M$ casillas libres (o sea, el tablero esta vacío). Cabe aclarar que *fichas restantes* son la fichas restantes totales (las mías + las suyas). Esto es porque el algoritmo de `minimax` es como un árbol de `Backtracking` en el sentido de que checkea todas las posibles sucesiones de jugadas que se pueden dar, considerando tanto lo que yo juego como lo que él juega.

Si bien en total hay menos de 3^{NM}) tableros posibles (cada casilla puede estar vacía, o tener una ficha de alguno de los dos jugadores, y acota por arriba pues hay casos que no son posibles, como que haya una ficha encima de una vacía) esto no cuenta la cantidad de posibles partidas que se pueden dar, ya que un mismo tablero podría ser alcanzado de varias formas. Pero a cada paso, cada jugador tiene N opciones de jugada y a lo sumo se juega $N * M$ veces (se llena el tablero) y por ende tenemos N^{NM} estados posibles del tablero. Lo que hace el algoritmo es para cada uno de estos tableros, evaluar si hay o no un ganador y en base a eso establecer un puntaje al tablero que es el que transmite `minimax`. Este checkeo tiene un costo de $\mathcal{O}(c)$ ya que la función, por cómo fue implementada, recibe cuál fue la última jugada y sabe cómo se empata o si con la última jugada se logra que algún jugador gane. Además, checkear las primeras dos condiciones es $\mathcal{O}(1)$ sabiendo la cantidad de fichas que quedan y de filas y columnas (que es un dato que se pasa por parámetro). Pero si gano con esa ficha solo hay que verificar las c fichas para todos los sentidos y direcciones (8 en total, 4 direcciones con sus dos sentidos cada una), lo que es $8\mathcal{O}(c) = \mathcal{O}(c)$.

Luego, dado un tablero, conociendo el valor de minimax de sus hijos, queda tomar máximo o mínimo según corresponda. Pero como tiene N hijos a lo sumo (si no es hoja) ya que son las N posibles jugadas inmediatas, esto es $\mathcal{O}(N)$. Esta operación la hacemos para todo nodo (como también checkear si hay un ganador). Por lo tanto, la complejidad total resulta $|Nodos|\mathcal{O}(N)\mathcal{O}(c)$, y ya se ha visto que $|Nodos| = a^b$ por lo que se obtiene una complejidad $\mathcal{O}(Nca^b)$ lo que es acotable, como se ha visto por $\mathcal{O}(NcN^{\text{NM}}) = \mathcal{O}(cN^{(\text{N}+1)\text{M}})$. Pero si recordamos que $c < \text{mín}\{N, M\}$ podemos deducir que la complejidad es menor que $\mathcal{O}(N^{((\text{N}+1)\text{M})+1})$.

En cuanto a la complejidad del algoritmo con poda, no hay modificaciones ya que la misma cota es válida puesto que la única diferencia que tiene la poda es que guarda registro de dos valores (α y β) y los compara a cada paso, lo que es $\mathcal{O}(1)$ y por ende no afecta la complejidad. Si bien la poda corta varias ramas del árbol de minimax, no podemos asegurar una cota de cuántas ramas cortará y por ende, la complejidad teórica seguirá siendo la misma. Sin embargo uno espera que la poda se ejecute y varias ramas no se checkeen, reduciendo así el tiempo de ejecución en comparación con el algoritmo sin poda.

2.4. Experimentación

2.4.1. Contexto

La experimentación se realizó toda en la misma computadora, cuyo procesador era Intel Atom$^{\text{TM}}$CPU N2600 @ 1.60GHz, de 36 bits physical, 48 bits virtual, con una memoria RAM de 2048 MB. Para experimentar, se calculó el tiempo que tardaba el algoritmo sin considerar el tiempo de lectura y escritura ni el tiempo que llevaba armar la matriz (ya que se leía un dato, se escribía la matriz y luego se leía el siguiente).

El tiempo se medía no como tiempo global sino como tiempo de proceso, calculando la cantidad de ticks del reloj (con el tipo `clock_t` de C++) y luego se dividía el delta de ticks sobre `CLOCKS_PER_SEC`. En todos los experimentos el tiempo se mide en segundos. El contexto de experimentación será el mismo a lo largo de todo el trabajo.

2.4.2. Sin poda

En principio, para verificar experimentalmente que el jugador era óptimo se jugó en tableros pequeños (en particular, de $2 \times 2, 3 \times 3, 2 \times 3, 3 \times 2, 2 \times 4, 4 \times 2, 3 \times 4, 4 \times 3$) con un $c = 2, 3, 4$ siempre que cumpla

la condición del juego ($c \leq min\{N, M\}$) contra el jugador random. En todos los casos la cantidad de iteraciones fue de $2 * 3^{\mathrm{NM}}$ por lo que todo tablero posible tenía una probabilidad considerable de ser jugado (notar que la cantidad de juegos posibles es menor que N^{NM} -de hecho es menor que 3^{NM} ya que cada casilla tiene o bien nada o ficha de un jugador o del otro-). Tomamos esta cantidad de repeticiones ya que a cada paso lo que se elige es una de N -o menos si hay columnas llenas- opciones donde jugar y esto se repite hasta que termine el juego -a lo sumo se lena la grilla $N * M$ veces-. Así, tenemos tantas iteraciones como secuencias posibles de jugadas, o sea partidas en general. Como el jugador es random equiprobable, la distribución respecto de qué jugada de las posibles realizar es uniforme y lo multiplicamos por 2 para que sea más probable que salga cualquier partida posible (estamos jugando toda partida posible con una probabilidad considerable). Más aún, es una probabilidad bastante alta ya que se puede notar que solo queremos contar todas las opciones de secuencias de jugadas que puede dar el segundo jugador (en este caso el aleatorio), lo que nos provee una cota menor de la cantidad total de jugadas distintas que se le pueden hacer al jugador optimo, acotándolo por $N^{\frac{NM}{2}}$.

Estos experimentos se repitieron en dos casos cada uno, cuando comenzaba el `minimax` y cuando comenzaba el random. Además para fijado el jugador que empieza, se repitió con valores de fichas que fueron $p = \frac{NM}{2}, \frac{NM}{3}, \frac{NM}{4}$ donde el primer valor de p aseguraba que se pueda jugar sin límite de fichas (siempre se podría llegar a llenar el tablero) y los otros dos sí imponían un limite de fichas. Luego de cada experimento, nos fijamos en el archivo `.log` devuelto por dichas iteraciones y en todos se observó el mismo comportamiento:

- O bien siempre ganaba (lo que nos indica que había estrategia ganadora para el que empieza). Al invertir la situación (cambiando el jugador que empieza), se daba el ítem 3.
- O bien siempre ganaba o empataba (lo que nos indicaría que la mejor estrategia para ambos resulta en un empate, pero como el jugador random no siempre juega lo mejor, le daba la posibilidad a nuestro jugador de ganar). Al invertir la situación (cambiando el jugador que empieza) se obtenía este mismo ítem.**CHEHEBAR: QUÉ CAZZO QUISISTE DECIR CON ESTA ÚLTIMA ORACIÓN**
- O bien siempre ganaba o empataba o perdía (lo que nos indicaría que el que no empieza tiene estrategia ganadora, pero como el jugador random no siempre juega lo mejor, le daba la posibilidad a veces a nuestro jugador de ganar). Al invertir la situación (cambiando el jugador que empieza), se daba el ítem 1.

Como se explicó, estas situaciones, en todos estos tableros, reforzaron fuertemente la idea de que se trataba de un jugador óptimo. Cabe aclarar que se realizó solo con tableros pequeños debido a que, para tableros muy grandes el jugador demoraba demasiado tiempo en decidir que jugar (como se vio en la complejidad teórica exponencial, por lo que crecía brutalmente al crecer el tamaño del tablero y la cantidad de columnas) y por ende resultaba inviable realizar una alta cantidad de iteraciones para recorrer una gran cantidad de tableros posibles lo que nos permita reforzar la idea de optimalidad del jugador.

Pero en tableros pequeños hemos podido comprobar que en todos los tableros que se dieron, cumplieron que el jugador era óptimo ya que si tenía la posibilidad de ganar lo hacía, en caso de que esta no existiera, si tenía la posibilidad de empatar lo hacía y recién en caso de no existir, jugaba cualquier cosa sabiendo que perdería. Pero como hemos visto, esto se comprobaba paso a paso, por lo que el jugador podía jugar creyendo que perdería y, como el otro no jugó optimamente luego, pasar a poder ganar o empatar (y efectivamente hacerlo pues es óptimo); o sea, asume optimalidad del rival.

Esta experimentación nos permitió reforzar nuestra idea (al menos para estos tableros pequeños y con una probabilidad muy alta) de que el jugador era efectivamente óptimo.

Para tratar de reforzar experimentalmente la complejidad teórica deducida previamente, se realizó una experimentación en la que se registró para varios tableros de tamaño pequeño (de $2 \times 2, 3 \times 3, 4 \times 4, 2 \times 3, 3 \times 2, 2 \times 4, 4 \times 2, 5 \times 2, 2 \times 5, 2 \times 6, 2 \times 7, 2 \times 8, 6 \times 2, 7 \times 2, 8 \times 2, 3 \times 4, 4 \times 3, 3 \times 5, 5 \times 3$) ya que para tamaños mayores los tiempos de ejecución fueron inabordables (lo que reforzó en parte la complejidad

teórica esperada). Para cada uno de esos tableros, se jugó con fichas de sobra (más que la cantidad que entraba en el tablero, 88 particularmente) y el c se movió entre $2, 3$ y 4 siempre y cuando se cumplieran las condiciones del juego. Además, siempre se jugó contra el jugador random provisto por la cátedra (que con distribución uniforme decidía aleatoriamente en cuál de las N posibles columnas jugar). Para asegurar distintas configuraciones, se repitió cada partido 10 veces.

Para todos estos juegos se registró a cada jugada del algoritmo el tiempo de ejecución que este tomó en decidir qué jugada realizar y se registró una variable que reflejaba un refinamiento de la complejidad esperada. Esta variable se definió como Nca^b donde $a = |ColumnasDondeEsPosibleJugar|$ y $b = \text{mín}\{|FichasRestantes|, |CasillasLibresEnElTablero|\}$ que es en particular acotable por $cN^{(\mathrm{N}+1)\mathrm{M}}$. Luego, se graficó el $\frac{tiempo}{Nca^b}$ en función de Nca^b. Se esperaría obtener un gráfico acotable por una constante que reforzaría la idea de que el aloritmo cumple la cota propuesta. Se graficó, pues, esto en la Figura 2, (junto con su análogo con el algoritmo con poda) y efectivamente se cumplió con lo esperado ya que se ve que el gráfico es prácticamente constante salvo en los primeros valores, lo que tiene sentido y refuerza aún más la hipótesis, porque para esos valores toman mucha más importancia los términos constantes que no dependen de la variable propuesta como complejidad. Al aumentar la variable propuesta, la importancia del término constante se reduce y pasa a ser constante. Más aún, esa constante es 0, lo que nos indica que es incluso menor a la complejidad propuesta (o sea que cumple lo pedido, ya que en particular es menor, que es a lo que apunta la notación $\mathcal{O}$) ya que este método de dividir por la complejidad esperada es muy sensible, ya que da una constante si dividimos por algo que cumpla ser θ, y sino ante una pequeña alteración en la complejidad se refleja claramente en el gráfico (o bien pasa a ser creciente o decreciente). Tiene sentido que sea menor ya que muchas veces el partido termina antes de que completemos efectivamente el tablero (que es la cota que propusimos) y además no siempre tenemos tantas opciones como columnas posibles de jugar ahora (cota que propusimos), sino que con el desarrollo del juego las opciones de columnas posibles van disminuyendo. Igualmente, esto nos permite reforzar claramente nuestra hipótesis de que cumple la complejidad teórica propuesta.

2.4.3. Con poda

En el caso con poda se realizó la misma experimentación para comprobar la optimalidad. Como el algoritmo tardaba un poco menos (por la poda realizada), se realizó (además de con los ya mencionados) con tableros de tamaño $4 \times 4, 3 \times 5, 5 \times 3, 4 \times 5, 5 \times 4, 3 \times 6, 3 \times 7$, con los mismos valores de c que antes (siempre y cuando cumpliera las condiciones del juego). También se varió la cantidad de fichas de la misma forma.

Nuevamente, los resultados y conclusiones sobre estos fueron los mismos, lo que nos permitió reforzar nuestra idea (al menos para estos tableros pequeños y con una probabilidad muy alta) de que el jugador era efectivamente óptimo.

Para verificar que la poda efectivamente se ejecutaba y permitía podar varios niveles del árbol de ejecución del minimax, decidimos analizar el tiempo de ejecución ya que si no se analizaran estas partes podadas, el tiempo de ejecución sería menor. Por eso se ejecutó para exactamente las mismas partidas con las que se experimentó sin poda, la misma cantidad de veces cada una, contra el mismo jugador (que era el jugador random) que es importante aclarar que como no era determinístico medimos en la misma partida (o sea, ambos algoritmos -con y sin poda- recibían el mismo tablero a cada jugada) el tiempo de ejecución de ambos (y finalmente jugaba el "con poda" la jugada -lo que nos resultaba poco relevante ya que queríamos medir tiempos de ejecución para distintos tableros y verificar nuestra complejidad que no dependía de cómo jugaban y tampoco era el objetivo analizar como jugaban-). Luego se compararon los dos tiempos medidos para las 688 jugadas en todos los partidos. Así, solamente fueron 9 jugadas en las que el algoritmo con poda tardó más que el sin poda y estas diferencias fueron realmente despreciables (tanto en cantidad de jugadas diferentes, como en diferencia de tiempo). Listamos a continuación los valores en los que difirieron (5.8e-05 contra 5.6e-05, 0.000383 contra 0.000352, 0.000212 contra 0.000202, 0.000535 contra 0.000251, 0.000282 contra 0.00027, 0.00028 contra 0.000264, 0.000469 contra 0.000459, 0.000417 contra 0.000355, 0.000341 contra 0.000323 -del algoritmo con poda contra el sin poda respectivamente-).

SI HAY TIEMPO, HACER UNA TABLITA, QUEDA MÁS FACHERO

Esta experimentación nos permitió no solo reforzar nuestra hipótesis (justificada teoricamente en el algoritmo) de que el tiempo de ejecución era efectivamente menor, por lo que se ejecutaba la poda, sino también comprobar que se seguía manteniendo la complejidad teórica propuesta para el algoritmo con poda (ya que era la misma que la del algoritmo sin poda y hemos visto que el tiempo de ejecución era menor). Más aún esto se puede ver en la Figura 2 donde se ve que el algoritmo con poda nunca supera al algoritmo sin poda en tiempo de ejecución.

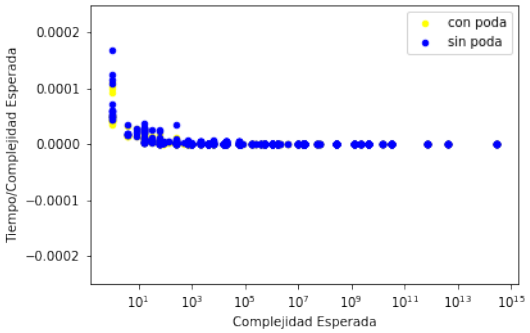


Figura 2: Gráfico de segundos de ejecución (de cada jugada) dividido a^b en función de a^b para los partidos jugados por el algoritmo con y sin poda en tableros pequeños contra el jugador random.

3. Jugador parametrizable

3.1. El algoritmo

El algoritmo consiste básicamente en establecer un sistema de puntos para cada tablero, donde se miden solamente ciertas características que decidimos nosotros. A cada una de esas características se le asignará una importancia. El jugador es goloso en el próximo paso, o sea, evalúa todas las opciones que tiene para jugar (todas las columnas donde se puede jugar) y se queda con la que tenga un tablero que le dé el mayor puntaje. La libertad que tiene el jugador es la importancia de cada característica, que es lo que luego se buscará optimizar. Las características que decidimos considerar para ambos jugadores fueron:

- cantidad de fichas en la columna central, en algun borde, en alguna esquina;
- cantidad de fichas consecutivas de tamaño $1, 2, .., c-1$;
- cantidad de fichas consecutivas de tamaño $1, 2, .., c-1$ que se pueden extender (es decir, que tengan al lado casilleros vacíos);
- cantidad de fichas consecutivas de tamaño $1, 2, .., c-1$ que se pueden extender en el próximo turno (extensiblesprox);
- cantidad de fichas consecutivas de tamaño $1, 2, .., c-1$ que se pueden extender en ambos sentidos (biextensibles).

3.1.1. El Pseudocódigo

En el `main` a la variable *movida* se le asigna el resultado de la función `parametrizable` que es el algoritmo goloso propuesto para esta parte del problema. Esta toma los mismos parámetros que `minimax` con el agregado del conjunto de **parametros** *params* y devuelve el número de columna que se debe jugar.

La idea es que se asigne un puntaje a cada tablero y de las ocho opciones posibles se tome la de mayor puntaje (esta sería la parte golosa).

En la función se llama a la función `puntaje` que se verá luego.

La variable *tablero* se pasa por referencia.

La función `puntaje` devuelve un entero asociado a cierta jugada que se pasa como input (es decir, no solo se pasa la columna donde se se juega, sino también, el tablero con las fichas). La función en sí consiste simplemente en sumar ciertos parámetros asociados a características del estado actual del juego. Por ejemplo, como no es lo mismo tener una ficha en el centro que en un borde, la variable asociada a

Algorithm 4: parametrizable

Input: **int** rows, **int** columns, **int** c, **int** p, matriz(**int**, **int**) tablero, **int** ultimajugada, cjto(**parametro**) params

```
1  posibles = [0, .., columnas − 1]− las columnas llenas del tablero
2  for i ← [0, .., |posibles|) do
3  |  if jugando posibles[i] se evita perder en la siguiente jugada then
4  |  |  return posibles[i]
5  |  end
6  end
7  float maximoPuntaje = puntaje(rows, columns, c, p, tablero, posibles[0], params)
8  int maxpos = posibles[0]
9  for q ← [1, .., |posibles|) do
10 |  AgregarAtrás(tablero[posibles[q]], 1)
11 |  if puntaje(rows, columns, c, p, tablero, posibles[q], params)> maximopuntaje then
12 |  |  actualizar maximopuntaje y maxpos
13 |  end
14 |  Sacarle el último a tablero[posibles[q]]
15 end
16 return maxpos
```

la cantidad de fichas del centro tendrá más peso que la del borde. De este modo, se cuentan los distintos parámetros asociados al tablero actual tanto para el jugador 1 como para el 2. Finalmente, entonces, se devuelven las cantidades del jugador 1 y se le restan las del jugador 2 (pues son las características del estado asociado al oponente).

Previamente se verifica si con la jugada se gana o no. Si es así, el puntaje es ∞ así claramente es el máximo y la función anterior lo elige como jugada.

La variable *tablero*, como en `parametrizable` también se pasa por referencia pero como `const`.

Algorithm 5: puntaje

Input: **int** rows, **int** columns, **int** c, **int** p, matriz(**int**, **int**) tablero, **int** ultimajugada, conj(**parametro**) params

```
1  if gane then
2  |  return ∞
3  end
4  contar los que estan en el borde de cada jugador
5  contar la cantidad de fichas que estan en una esquina de cada jugador
6  contar la cantidad de fichas en el centro de cada jugador
7  contar la cantidad de casillas libres alrededor de cada ficha de cada jugador
8  contar la cantidad de fichas consecutivas de cada jugador
9  contar cantidad de inmediatemente extensibles en la próxima jugada de cada jugador
10 contar cantidad de extensibles en cualquier momento de cada jugador
11 contar cantidad de bi-extensibles (extensibles en ambos sentidos) en cualquier momento de cada
    jugador
12 return ∑ cantidad_jugador1*(puntaje del parámetro)-cantidad_jugador2*(puntaje del parámetro)
```

Cabe aclarar que contar la cantidad de fichas en determinadas posiciones es simplemente recorrer el tablero en búsqueda de esas posiciones. De forma similar, contar las consecutivas es mirar todas las filas, columnas y diagonales que hay en el tablero y recorrerlas, contando las consecutivas de un jugador (que

es recorrer todo el tablero 4 veces, una horizontalmente, otra vertical, una en diagonal desde arriba a la izquierda, hacia abajo a la derecha y la última en diagonal desde arriba a la derecha, hacia abajo a la izquierda). Teniendo esto en cuenta es simple chequear si cada una de estas tiras es extensible o bi-extensible, viendo simplemente si sus casillas por donde podrían seguir están vacías o no (si existen, cosa que podría no suceder en caso de estar en el borde del tablero) y para ver si son extensibles en el proximo turno es cuestión de analizar si estas casillas tienen debajo alguna ficha o son la primera de su columna (o sea, si se puede jugar ahí en el próximo turno).

3.2. Complejidad

Veamos que el jugador que presentamos decide que jugada realizar en tiempo $\mathcal{O}(N^2M)$ que es menor que $\mathcal{O}(N^2M^2)$ y por ende, cumple la complejidad pedida.

En primer término, veamos que calcular el puntaje de un tablero es $\mathcal{O}(NM)$ puesto que lo único que hacemos es recorrer todas las filas, columnas y diagonales (que en total son 4, aunque no importa verdaderamente cuánto sino que es una cantidad constante) y en cada una de ellas contamos cuántas fichas son nuestras y cuántas de él. Pero esto lo repetimos una vez por cada parámetro que tenemos. Como tenemos finitos (y constantes) parámetros, pasamos a recorrer una cantidad constante de veces el tablero $\mathcal{O}(c)$. Además, chequeamos si ganamos, que como ya hemos visto en la complejidad del jugador óptimo es ...**COMPLETAR** $\mathcal{O}(NM)$... Esto es porque recorrer el tablero es $\mathcal{O}(NM)$ ya que para cada sentido de recorrida, pasamos solo una vez por cada casilla, o sea recorremos $N*M$ casillas; repitiendo esto una cantidad constante de veces (la cantidad de parámetros) y para una cantidad constante de sentidos (cuatro) nos da que recorrimos una cantidad constante de veces $N*M$ casillas, por lo que la complejidad de calcular el puntaje es efectivamente $\mathcal{O}(NM) + \mathcal{O}(c) = \mathcal{O}(NM)$ puesto que $c < \text{mín}\{N, M\}$.

Cabe aclarar aquí que consideramos como un parámetro a los vectores de parámetros (como es el caso de consecutivos) ya que para ellos recorremos solo cuatro veces el tablero (en todos los sentidos) y lo que cambia es dónde alojamos cada valor del vector según cuántas fichas consecutivas encontremos del mismo color. Cabe aclarar también que la operación para saber qué ficha está en una casilla es $\mathcal{O}(1)$ porque guardamos al tablero en una matriz. Esto es fundamental para poder asegurar la complejidad pedida.

Pero nuestro jugador no hace solo eso, sino que analiza cómo quedaría el puntaje del tablero para cada una de sus N posibles jugadas y luego se queda el máximo de ellas (esta es la parte golosa). Repite entonces N veces algo $\mathcal{O}(NM)$ por lo que da una complejidad de $\mathcal{O}(N^2M)$ y luego calcula el máximo sobre N elementos que suma $\mathcal{O}(N)$ (es recorrer linealmente un vector) y da una complejidad total de $\mathcal{O}(N^2M) + \mathcal{O}(N) = \mathcal{O}(N^2M)$.

3.3. Experimentación

Si bien el grueso del análisis y experimentación sobre este jugador consistirá en buscar buenos parámetros que mejoren al jugador, consideramos relevante reforzar experimentalmente nuestra hipótesis de complejidad teórica esperada. Para esto tomamos mediciones análogas a las del jugador óptimo solo que jugando en más tableros. Se jugó en tableros que comprendían todas las combinaciones factibles con las columnas y filas moviéndose entre 2 y 10, c moviendose entre 2 y $\text{mín}\{columnas, filas\}$ y p moviéndose entre 2 y $filas * columnas/2$. En cada jugada, los parámetros que determinaban la próxima jugada a realizar se sorteaban aleatoriamente. Todo esto fue para ver experimentalmente que la dependencia que proponíamos en la complejidad dependía efectivamente solo de n y m. Así, igual que antes, realizamos un gráfico de $\frac{tiempo}{filas*columnas^2}$) en función de $filas*columnas^2$ donde esperábamos obtener un gráfico constante que reforzaría nuestra hipótesis de que la complejidad (como se justificó teóricamente) era de $\mathcal{O}(N^2M)$. Este gráfico se puede ver en la Figura 3.

Efectivamente, como se puede ver en la Figura 3, pareciera cumplirse la complejidad esperada ya que el gráfico es practicamente constante. Más aún, al variar por todas sus posibilidades c, p y el parámetro de la función, esto refuerza la hipótesis (que se justificó en la explicación del algoritmo) de que esta cota de dependencia no se ve afectada por el parámetro en particular, o la cantidad de fichas que queden, o el c.

Sabiendo esto, tenemos la posibilidad de experimentar para tableros de mayor tamaño (ya que no tenemos que repetirlos para distintos c y p). Por esto, se experimentó con un c aleatorio (con distribución uniforme, usando `rand()` de C++) entre 2 y mín$\{columns, rows\}$, y $p = N\frac{M}{2}$ para que no se interrumpa el partido por falta de fichas y se varió nuevamente las filas y columnas pero con el límite superior igual a 20 en vez de 10. Nuevamente, se realizó el mismo gráfico que antes con los nuevos , que se puede ver en la Figura 4.

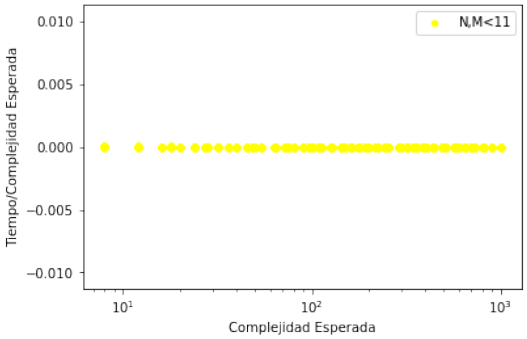


Figura 3: Gráfico de segundos de ejecución (de cada jugada) dividido N^2M en función de N^2M para los partidos jugados por el jugador que utiliza la función parametrizable a cada paso. Varian c y p. Columnas y filas menores a 11.

Podemos ver en el gráfico de la Figura 4, que los resultados siguen siendo los mismos incluso para tableros de tamaño mayor. Nuevamente el gráfico es prácticamente una constante. El análisis y su justificación es totalmente análogo al realizado en la Figura 3 y al ser con tamaños de entrada mayores, refuerza más aún experimentalmente la complejidad que fue propuesta teóricamente.

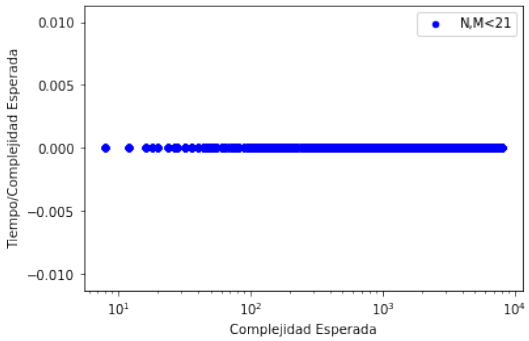


Figura 4: Gráfico de segundos de ejecución (de cada jugada) dividido N^2M en función de N^2M para los partidos jugados por el jugador que utiliza la función parametrizable a cada paso. Columnas y filas menores a 51.

4. Grid Search

4.1. El algoritmo

Para mejorar el rendimiento del jugador se implementó, en primer lugar, un `grid search`. Esta técnica consiste en tomar el conjunto de parámetros que definen una instancia del jugador y buscar la combinación óptima de valores de dichos parámetros. Para ello, se define, en primer lugar, un intervalo para cada parámetro por el cuál se buscará el resultado. En nuestro caso, se eligió usar el mismo para todos, el intervalo $[-1, 1]$. Ahora bien, como este espacio es continuo, para buscar individualmente cuál es mejor, se lo discretiza tomando como diferencial a 0,1. De este modo, para cada parámetro, se recorrieron los valores $-1, -0{,}9, -0{,}8, ..., 1$.

Una forma más precisa de verlo es con una grilla (de ahí el nombre del algoritmo) de n dimensiones (con n la cantidad de parámetros) donde cada fila o columna n-dimensional es un valor fijo de un parámetro. Pensándolo solo con 2 parámetros, las filas corresponderían a los valores $-1, -0{,}9...$ de un parámetro y las columnas a sendos valores del otro. Cada casilla representaría el resultado obtenido de jugar con el

conjunto de parámetros dado por esa fila y columna. De este modo, si se recorre la grilla completa se puede encontrar el valor óptimo y, por consiguiente, la mejor combinación de parámetros.

Para determinar cuándo una combinación es mejor que otra simplemente se enfrentan los dos jugadores y, quien resulta ganador es porque presenta una mejor combinación de parámetros. Esta combinación ganadora se guarda en la variable *mejor* que se irá actualizando cada vez que un nuevo conjunto de parámetros juegue mejor que él. Cabe aclarar que, como los jugadores son deterministas, con solo dos enfrentamientos alcanzan (invirtiendo quién empieza a jugar), pues, para idénticos parámetros se desarrollarán idénticos juegos. Un jugador se considera mejor que otro si y solo si cuando fue primero le ganó y cuando fue segundo, por lo menos empató.

4.1.1. El Pseudocódigo

A continuación se presenta el pseudocódigo del algoritmo de `grid search`.

Algorithm 6: Grid Search

Input: int rows, **int** columns, **int** c, **int** p, cjto(**parametro**) params

```
1 cjto(parametro) mejor
2 for i ← [1..|params|] do
3 |   for j ← [−1, −0,9, −0,8, .., 1] do
4 |   |   if jugador(p_i + j) > jugador(mejor_i) then
5 |   |   |   mejor_i = p_i + j
6 |   |   end
7 |   end
8 end
9 return mejor
```

4.2. Experimentacion

4.2.1. Primera exploración

La primer idea que surge al comenzar la experimentación computacional del método heurístico con `grid search` es que a causa de la cantidad de parámteros (28 para el 4 en línea) que tiene en cuenta la función paramétrica detallada en la sección 3, resulta demasiado costoso computacionalmente explorar todo el espacio de posibilidades con un `grid search` clásico. Incluso, con una granularidad muy baja, comparando entre tan poco como 20 valores posibles para cada parámetro (cada uno en [-1,1] con saltos de 0,1) se deben analizar 20^{28} combinaciones de parámetros distintas. Por esto, en lugar de utilizar el algoritmo detallado en la sección 4.1, como era la idea original, se recurrirá a una modificación del mismo para acelerar el tiempo de ejecución.

A causa del tamaño inabarcable del espacio de búsqueda se decidió entonces utilizar un método que reduce de forma drástica el tiempo necesario para obtener resultados del `grid search`, con el conocimiento de que no se podrá explorar todo el espacio de esta forma. Empezando con un conjunto de parámetros aleatorios se buscaba con el método de `grid search` la configuración óptima únicamente para los primeros dos parámetros. Una vez que se encontraron los valores óptimos para los primeros dos parámetros en el contexto del conjunto aleatorio inicial, se procede a optimizar el próximo par de parámetros pero manteniendo los mejores valores encontrados para el par ya explorado. Esto se repite hasta optimizar todos los parámetros a tener en cuenta. La desventaja de este método es que a medida que se van modificando los valores de pares, las optimizaciones iniciales van perdiendo relevancia, ya que pueden no ser las óptimas para esta nueva configuración.

Este primer intento no dio buenos resultados. El jugador resultante no realizaba las jugadas esperadas y no podía reconocer ni siquiera instancias en las que estaba a un turno de perder y podía salvarse (no

reconocía que -si jugaban al 4 en línea- cuando el oponente tenía 3 en línea sólo extensibles por un lado le convenía tapar eso porque sino pierde seguro).

4.2.2. Modificaciones

En busca de mejores resultados se realizaron los siguientes cambios para mejorar el conjunto de parámetros resultante de ejecutar el `grid search` con un jugador inicial aleatorio al cual se le van optimizando un par de parámetros a la vez. El primer cambio fue simplemente iterar la versión anterior varias veces, aprovechando que recorrer matrices de solo 2 dimensiones hace al algoritmo más rápido. Como siempre se guarda el mejor jugador hasta el momento y se lo suplanta únicamente si se encontró otro conjunto de parámetros mejor, realizar más iteraciones en el peor caso mantendría la misma solución. El objetivo de este cambio sería darle otra oprtunidad a los parámetros a optimizarse en este nuevo contexto en el cual las otras preferencias del jugador no son las iniciales aleatorias, sino unas calculadas en base a este mismo algoritmo.

Otra modificación fue poner los parámetros que consideramos más importantes (la cantidad de filas biextensibles grandes, por ejemplo) más adelante en el orden en que se chequean, es decir que se optimicen primero, antes de tocar los otros parámetros aleatorios. De esta forma, se intenta dejarlos buscar su configuración óptima sin estar condicionados por los valores convenientes para otros parámetros menos importantes.

Puede verse que esta última modificación es importante si se tiene en cuenta lo siguiente: los valores de los parametros en nuestra implementación están acotados en el conjunto [-1,1], por lo tanto si al valor que determina la preferencia del jugador por poner fichas en las esquinas se lo optimiza primero, como todos los otros valores son aleatorios podría ser que el resultado óptimo para esta preferencia de jugar en las esquinas sea muy alto, cercano a 1. Si esto ocurriera, cuando se llegara a intentar optimizar otros valores en principio más importantes, como la importancia de cadenas biextensibles de tamaño $c-1$ no tendría la posibilidad de darle una importancia mayor a la de jugar en las esquinas, ya que no podría pasarse de 1. Esto presenta un problema ya que sabemos que un jugador bueno seguramente debe tener ciertos valores en grados de magnitud mayores que otros. Es importante aclarar que aunque no se acotaran los valores entre [-1, 1] no se evitaría el problema ya que los límites a la hora de implementar el algoritmo fuerzan inevitablemente una cota superior sobre los valores elegidos.

4.2.3. Grid recomendado

Incluso con las modificaciones a la idea original, la forma de recorrer la grilla de a pares de parametros resultaba muy acotada, y no llegaba a encontrar buenas combinaciones de valores que resultaran en un jugador bueno. Esto es de esperar ya que si recordamos el tamaño gigante de 20^{28} posibilidades no hay manera de explorarlo bien en un tiempo razonable. A causa de esto se recurrió a la posibilidad de comenzar el `grid search` no con un conjunto de parámetros aleatorios, si no con un jugador hecho "a mano", con parámetros medianamente buenos. Con esto se buscaba que el `grid search` explore opciones cercanas a un conjunto que ya se sabe que no es muy malo. De este modo se intentó evitar la gran cantidad de combinaciones inútiles que el algoritmo estaba recorriendo previamente, y en su lugar mejorar una solución inicial.

Con esta recomendación de parámetros iniciales se esperaban observar resultados interesantes del `grid search`. El jugador resultante surge de haber vencido a los valores de entrada, por lo cual no sólo debería tener un nivel de juego muy superior a los intentos previos del algoritmo, además mejoraría por sobre los parámetros de entrada. El problema surge en este punto, a causa de la naturaleza compleja del juego, resulta difícil especificar exactamente *qué* define a un mejor jugador. ¿El `output` del `grid search` debe necesariamente ganarle al `input`?

La forma en la que se iba mejorando al mejor jugador encontrado hasta el momento era tomar el inicial, y solo declarar que el conjunto de valores que se está explorando en este momento supera al mejor que se tiene si puede jugarle 2 partidos cambiando quién comienza sin perder. El problema de esto es que

los partidos son completamente deterministas, y este nuevo campeón vence seguramente al jugador que "destronó"pero si juegan exactamente esa apertura. Por todo lo que se sabe es posible que si le juega a un jugador enteramente aleatorio pierda ampliamente.

A pesar de esto, los resultados de este método recomendado eran mejores, pero no porque el algoritmo mejorara el `input` sino porque al comenzar ya con un jugador bueno y explorar a partir de él es más probable llegar a buenos resultados.

El problema radica en la dificultad de decidir cuándo la opción que se está explorando "juega mejor"que la que ya se tiene guardada. A causa de la naturaleza determinística de nuestra función parametrizable, al hacer jugar a estos dos conjuntos de valores solo se explora una línea de juego, cuando en realidad un buen jugador debe poder responder a una amplia variedad de situaciones. Para poder cuantificar la variedad de respuestas de los distintos jugadores se los hizo jugar múltiples partidos contra algoritmos distintos a la función parametrizable. Estos algoritmos se detallarán en la próxima sección.

4.2.4. Redefinición de mejorar

Con todas las variaciones anteriores del algoritmo, se veía que al ejecutarse se iban encontrando nuevas combinaciones de valores que se consideraban mejores. En ese sentido se veía el resultado buscado. El problema radicaba en que al examinar estos parámetros optimizados (que habían resultado de ir viendo si el que examino le gana al actual) no presentaban ni una lógica apreciable al examinar sus juegos con razonamiento humano, ni conseguía buenos resultados al enfrentarse a nadie. Como lo único que garantizaba era ganarle al campeón anterior, ni siquiera le ganaba al jugador construido a mano con el cual habíamos comenzado la exploración, es decir que si definimos la habilidad de un jugador por ganarle a la versión anterior, este algoritmo de búsqueda empeoraba los parametros de entrada.

Para contrarrestar esto se buscaron jugadores no determinísticos contra los cuales enfrentar cada conjunto de valores para los parámetros numerosas veces. Como ahora cada partido explora lineas de juego distintas, se expone a la función parametrizable a situaciones variadas en las cuales se refleja mejor la habilidad general de un "buen jugador". Ahora se considera que un nuevo conjunto de valores es mejor que el anterior, y se lo guarda como el nuevo mejor solo si de cientos de partidos, su cantidad de *partidos ganados − partidos perdidos* es mayor que el otro.

El primer intento de jugadores externos fue el de un jugador completamente aleatorio, sencillo de implementar. En cada turno este jugador elige dónde jugar entre todas las columnas con distribución uniforme (se utilizó la función `std::rand()` considerándola suficientemente aleatoria para este experimento). Este jugador es muy rápido por lo que permitió realizar chequeos con muchas combinaciones de valores para los parámetros. Pero si bien permite probar muchos casos, no es bueno para calificar la habilidad de los distintos jugadores parametrizables. Cuando se intentó obtener gráficos que mostraran cómo a lo largo de la ejecución del `grid search` cambiaba el porcentaje de partidos ganados y perdidos, se observó que muy rápidamente la función parametrizable ganaba los partidos casi en su totalidad. Esto parecería un buen resultado, pero a la hora de verificar la calidad de optimización de los parámetros obtenidos se observaba que éstos nuevamente no presentaban el nivel de juego esperado. Nuevamente la función con la cual se calificaba a los jugadores no era lo suficientemente general, exigente y tampoco garantizaba un crecimiento continuo de habilidad del jugador.

chebar este párrafo de acá arriba es el que digo que chocaría con lo de que en el genético ganar al random nos confirma que es bueno

A pesar de la utilidad nula del jugador aleatorio como criterio de habilidad de un jugador, el entrenamiento de éstos, el hecho de que los jugadores del `grid search` lo venciera regularmente se refleja positivamente sobre nuestra función parametrizable **QUE CAZZO QUISIERON DECIR ACA? Sugerencia?: A pesar de la utilidad nula del jugador aleatorio como criterio de habilidad de un jugador, el hecho de que los jugadores del `grid search` lo vencieran regularmente refleja que nuestra función parametrizable y el entrenamiento del jugador sirve/no es malo/da resultados**. Más allá de los valores que se les asignen, los parametros elegidos (detallados en la sección 3) parecerían capturar valores esenciales del *C en linea*, ya que permiten responder a las muy variadas

situaciones que el jugador aleatorio presenta, lo que nos vuelve a llevar al problema de cómo decidir si un jugador es mejor a otro. Con eso parecería que se podrían conseguir jugadores que sepan responder a situaciones variadas, como los resultados contra los jugadores aleatorios sugieren.

Como última variación sobre el algoritmo de `grid search`, para decidir si un nuevo jugador superaba en habilidad al mejor registrado se los hacía competir contra una versión menos costosa computacionalmente del jugador óptimo detallado en la sección 2. Recordemos que originalmente buscábamos recorrer un espacio de soluciones de tamaño muy amplio, por lo que resulta necesario que cada chequeo de la habilidad de una posible solución debe realizarse en un tiempo rápido. Si además se recuerda que se buscaba explorar múltiples lineas de juego para poder definir de forma más amplia qué tan bueno era un jugador, se verá que debemos correr numerosos partidos para cada jugador que se desea calificar. Esto presenta una situación en la que cada partido debe jugarse de forma muy rápida para explorarse en un tiempo razonable el espacio de soluciones, por lo que se decidió que el jugador óptimo no llegaría a ver todas las opciones a futuro, si no solo algunos turnos más adelante.

A cada jugador se lo enfrentaba, entonces -25 veces siendo primero y 25 segundo- contra un algoritmo que podía reconocer un tablero en el cual gana, pierde o empata. También ve todos los tableros posibles 4 jugadas más adelante. Es decir, hace el `backtracking` para las próximas 4 jugadas.

La decisión de que el algoritmo viera exactamente 4 turnos a futuro se concretó luego de unas rápidas pruebas de distintos valores. A valores más bajos se veía un efecto similar al del jugador aleatorio, en el que los jugadores parametrizables rápidamente dominaban la competencia. A valores más altos, cada partido resultaba muy costoso computacionalmente y no se hubieran llegado a recorrer tantas potenciales soluciones en la grilla de búsqueda.

completar con los resultados que de el grid search

5. Algoritmo Genético

El algoritmo genético se basa en la teoría genética darwiniana y busca básicamente recrear la evolución de una población. Los individuos más aptos tendrán mayor probabilidad de perdurar a lo largo de las generaciones, transmitiendo así los genes que hacen que sean más aptos. Se recrea también la mutación que permite que haya cambios en los genes (que pueden ser beneficiosos para la aptitud como no) de forma aleatoria. También se da el *crossover*, donde los genes de uno de los individuos se cruza con los de otro, generando así nuevos individuos que no son copia de uno ni de otro sino una mezcla de ambos. Para mayor precisión, a continuación se encuentra el pseudocódigo. Solo hay que hacer una salvedad previa: para medir qué individuo es mejor que otro, se usaron dos funciones distintas de *fitness*.

5.0.1. El Pseudocódigo

Como muestra el pseudocódigo, el algoritmo lo que hace es establecer un tamaño de población e inicializa de alguna manera (en principio de forma aleatoria) a todos estos individuos. En nuestro caso, nuestros individuos son una estructura que contiene un valor particular para cada uno de los parámetros que tomaba la función del jugador parametrizable.

Hasta que se cumpla una condición de corte (en principio una cantidad fija de generaciones, pero esta variará **también siendo un tiempo NO COMPRENDOU ESTO**, o podría ser también cierta condición de convergencia), se generarán nuevas poblaciones, que serán la próxima generación. Cada generación sufre un proceso de mutación y para dar lugar a una nueva, se da un proceso de selección que determina pares de individuos con los cuales se realizará el `crossover` dando lugar asi a un nuevo individuo que será parte de la próxima generación.

Para decidir a qué individuos seleccionar, se utiliza una función de `fitness` que basicamente decide (de alguna forma) cuán apto es un individuo. A partir de estos valores, el proceso de selección decidirá quiénes serán los que darán lugar a la nueva generación (en general, los más aptos son los que tienen más probabilidad de dar descendencia, ya que sus genes dan individuos más aptos).

Algorithm 7: genetico

Input: **int** rows, **int** columns, **int** c, **int** p

```
1  int tamanopoblacion, totalgeneraciones
2  float min, max, pmutar, pcrossover
3  vector(conj(parametro)) poblacion = vector de longitud tamanopoblacion donde cada elemento
     está dado por sendos llamados a paramrandom(c)
4  for generacion = [1..totalgeneraciones) do
5  end
6  poblacion = seleccion
7  for i ∈ poblacion do
8  |  mutar(i) con probabilidad pmutar
9  end
10 float maximo = fitness(poblacion[0], poblacion, rows, columns, c, p)
11 int maxpos = 0
12 for int i = [1..poblacion) do
13 |  float actual = fitness(poblacion[i], poblacion, rows, columns, c, p)
14 |  if actual > maximo then
15 |  |  actualizar maxpos y maximo
16 |  end
17 end
18 return poblacion[maxpos]
```

A lo largo de toda la experimentación utilizamos diversas funciones de estas que se detallan a continuación.

- **Crossover:** Consiste en, dados dos individuos (padres, dos estructuras de parámetros), generar un nuevo individuo que cada gen (cada parámetro particular) sea o bien de un padre o bien del otro. En nuestro caso toma un valor *pcrossover* que indica la probabilidad de que se cambie el padre del cual se están copiando los genes. El primer gen lo copia del primer padre pasado por parámetro, y cada vez que va al siguiente gen, sortea un numero (con una función random de distribución uniforme, `rand()` de C++) en $[0, 1]$. Si el resultado cae en $[0, pcrossover]$ copia el siguiente gen del otro padre. Sino sigue copiando del mismo. En principio, como los vectores que son parámetros (extensibles, extensiblesprox, biextensibles, consecutivos) presentan una clara relación entre ellos, decidimos que sean tomados como un gen entero (o bien se copia el vector de un padre o bien del otro)

- **Mutación:** Consiste en dado un individuo, cambiar cada uno de sus genes en base a alguna distribución de probabilidad. En nuestro caso, la función tomaba el valor *pmutar*. De forma análoga al `crossover`, para cada gen se sorteaba un numero y si caía en $[0, pmutar]$ ese gen mutaba y pasaba a tomar un nuevo valor aleatorio entre un valor *min* y *max* que eran los valores mínimos y máximos que podía tomar el puntaje.

- **Selección 1:** Consiste en calcular el `fitness` de todos los individuos de la población y para los que estén en una proporción de los que tienen más `fitness` (se fijo inicialmente en la mitad con más `fitness`) tomar dos aleatoriamente) y a esos aplicarles `crossover` obteniendo así un individuo de la nueva población. Repetimos este proceso de tomar dos aleatoriamente de ese grupo hasta que la nueva población tenga el mismo tamaño que la anterior. Siempre se incluye tal cual estaba en la nueva generación al mejor de la anterior (esto se denomina elitismo, propiciamos que el mejor de todos se mantenga). Notar que si bien siempre se deja como estaba, este individuo (si no lo evitamos) podría sufrir alguna pequeña mutación.

- **Selección 2:** Consiste en generar grupos de un cierto tamaño (que se fijó inicialmente en el 20 % de la población) tomando a sus integrantes de toda la población (con distribución uniforme). Luego, dado un grupo se elegía al más apto dentro de ese grupo (al que tuviera mayor `fitness`), se repetía lo mismo obteniendo un segundo individuo (obtenido desde un nuevo grupo) y luego se realizaba `crossover` con estos dos obteniendo un individuo de la nueva generación. El proceso se repetía hasta que la nueva generación tuviera el mismo tamaño que la anterior. Nuevamente se utilizó elitismo, dejando en la nueva población tal cual estaba al mejor de la anterior.

- **Fitness 1:** Se hace jugar a cada individuo contra todos los de su población dos partidos, donde en uno empieza y en el otro va segundo. Se cuenta la cantidad de partidos en los que no perdió y se los divide por la cantidad de partidos jugados para normalizarlos (queda el `fitness` en [0,1] y además permite que quede en el mismo rango para disintos tamaños de población).

- **Fitness 2:** La idea de esta función de `fitness` es establecer un sistema de puntos. Nuevamente se hace jugar a cada individuo contra todos los de su población dos partidos, donde en uno empieza y en otro va segundo. Se da un puntaje según pierda, gane o empate (pierde= 0; empata= 5; gana= 10) y se da un bonus si se empieza segundo ya que suponemos que en tableros medianamente grandes es una ventaja comenzar (BonusPierde= 1; BonusEmpata= 2; BonusGana= 4). Luego, se divide al puntaje sobre la cantidad de partidos jugados (para poder tener los mismos valores de `fitness` en poblaciones de distintos tamaños).

5.1. El algoritmo

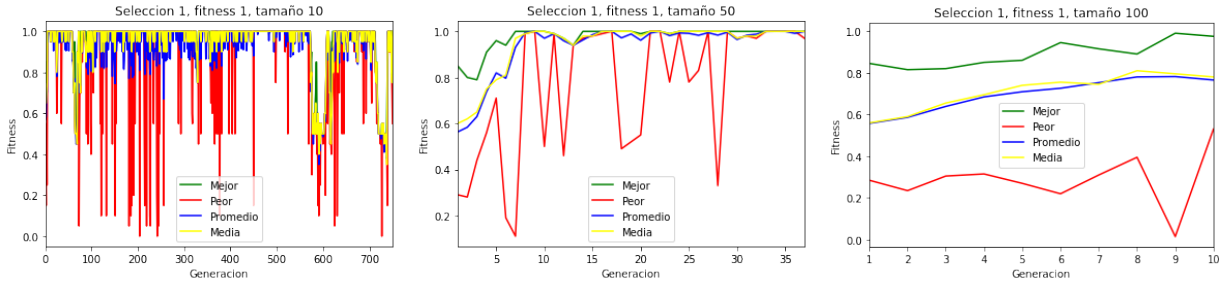
5.2. Experimentacion

Para comenzar con la experimentación, tratamos de comprender cómo influían los métodos de seleccion, funciones de *fitness, crossover* y *mutación* para así luego buscar la mejor forma de utilizar el algrotimo genético en nuestro caso. Para esto comenzamos estudiando estas funciones del algoritmo genético en el 4 en línea por ser una instancia particular que no es pequeña (tiene un tamaño considerable para ver diversas estrategias), ni es muy grande (los tiempos para experimentar son abarcables) pero podríamos haberlo hecho con cualquier otra sin problema.

5.2.1. Funciones del Algoritmo Genético

Primero, tratamos de comprender cómo afectaría el tamaño de las poblaciones. Si bien una población grande tiene como ventaja aportar mucha más variedad genética, los tiempos de ejecución son mucho mayores. Por su parte, una población pequeña si bien tiene poca variedad genética, es más eficiente en tiempo, dando muchísismas más generaciones en la misma cantidad de tiempo. Se buscó entender qué tipo de poblaciones brindaban mejores jugadores en el mismo tiempo. También se buscó encontrar qué *fitness* brindaba mejores jugadores y cómo eran las características de los métodos de selección.

Es por esto que se corrió el algoritmo genético para ambos métodos de selección y ambas funciones de *fitness* durante dos horas y media con poblaciones grandes, intermedias y chicas (de 100, 50 y 10 individuos cada una). Las poblaciones se inicializaron de forma aleatoria, donde cada parámetro se inicializaba aleatoriamente como un float entre -1 y 1 (con distribucion uniforme, se utilizo la funcion `rand()` de C++). Las poblaciones de 100 individuos dieron 10 generaciones, las de 50 individuos dieron 37 y las de 10 individuos dieron 750 generaciones en el tiempo indicado. En todos estos casos, la probabilidad de mutar se fijo en 0.005 y la de *crossover* en 0.4.



(a) tamaño 10 (b) tamaño 50 (c) tamaño 100

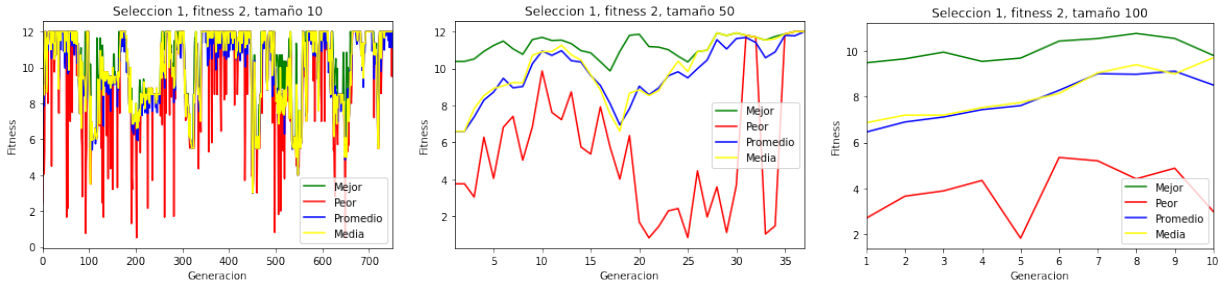
Figura 5: Gráfico de *fitness* en función de generación. Se grafica el mejor, peor, la media y el promedio. Selección 1, *fitness* 1, $pmutar = 0{,}005$, $pcrossover = 0{,}4$

Como podemos observar en la figura 5, efectivamente a mayor tamaño de población, menor es la cantidad total de generaciones que se generan en el mismo tiempo (2.5 horas). Tanto en la Figura 5.a como la 5.b podemos ver que hay momentos en los que la población pareciera converger. O sea, el mejor, peor, promedio y media valen 1, que es el máximo valor de *fitness* que pueden tomar (recordamos que estamos utilizando *fitness* uno que es la proporción de partidos no perdidos sobre el total). En ambas, esta situación no se mantiene por el resto de las generaciones ya que hay mutaciones que perturban esta situación. Sin embargo es notable que en todos los casos la media sigue manteniéndose muy alta (casi 1, que es el valor máximo) a pesar de haber mutaciones (a excepción de la mutación que se ve en la Figura 5.a alrededor de la generación 100, 600 y 700). Además de la media, el mejor sigue también manteniéndose en valores máximos (a excepción de las mutaciones mencionadas). Esto puede suceder gracias al elitismo, puesto que aseguramos que el mejor individuo de la generación pasada se incluya en la nueva generación sin ser modificado. Si observamos los descensos de la media, no tienen un descenso del mejor (esto indicaría que se generan mutaciones que son peores individuos y desaparecen rápidamente, volviendo a converger) salvo en los casos señalados. Estas situaciones en las que cambia el mejor y la media, podrían ser en las que el mejor individuo muta. La otra opción sería que aparezca un nuevo individuo que es no es ni mejor ni peor que el que era el mejor hasta ahora. Es decir, a veces le gana y a veces pierde, por lo que ambos no tienen *fitness* máximo. Esto último pareciera ser menos factible ya que no debería descender tanto la media por solamente un individuo así, a menos que sean varios, lo que es verdaderamente muy poco probable (ya es poco probable que varios muten considerablemente).

La convergencia de la media y el promedio a un valor cercano al máximo de *fitness* pareciera ser rápida. En el caso de una población de tamaño 50 se da alrededor de la generación 10. En el caso de la de tamaño 10 se da tan solo en la quinta generación. Esto tiene sentido ya que es menor el tamaño de la población. Por último, en las 2.5 horas transcurridas, se ve un claro aumento de la media en la población de tamaño 100, pero no llegó aún a converger en dicho tiempo.

Más aún, cabe notar que una mutación en una población pequeña la afecta mucho más que en una población grande (porque es una cantidad de genes mucho mas significativa respecto del total). Estos gráficos ayudan claramente a reforzar dicha hipótesis. Esto se puede ver ya que las variaciones repentinas, a mayor tamaño de población son más suaves, ya que en el gráfico de la Figura 5.a hay ver picos muy acentuados en los que desciende hasta por debajo de 0.8, situación que en la Figura 5.b no desciende por debajo de 0.9 siquiera -una vez que ya convergió-. Más aún, si observamos la figura 5.c vemos que la media siempre aumenta, a excepción de la generación 7 donde disminuye casi despreciablemente.

Cabe destacar que esta poca variabilidad que hay en la función de *fitness* y la rápida convergencia se puede deber no solo a que se tratan por igual las victorias y empates, tanto se empiece o no, sino también a que el método de selección solamente toma los genes de los mejores individuos (de la mitad superior), por lo que la variabilidad es menor. Para tratar de comprender mejor esto, se realizó la misma experiencia con la funcion de `fitness2`, que distinguía empates de victorias y quién comenzaba.



(a) tamaño 10 (b) tamaño 50 (c) tamaño 100

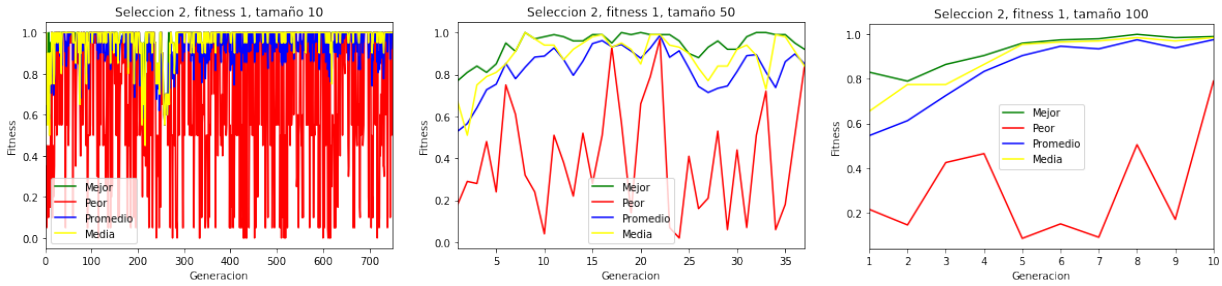
Figura 6: Gráfico de *fitness* en funcion de generación. Se gráfica el mejor, peor, la media y el promedio. Seleccion 1, fitness 2, $pmutar = 0{,}005$, $pcrossover = 0{,}4$

Efectivamente como podemos ver en la Figura 6, la variabilidad aumenta mucho más al cambiar la función de *fitness* como esperabamos ya que esta distingue muchas más situaciones (distingue empates de victorias y distingue si uno empieza o no). Para todos los tamaños de población la media y promedio sufren alteraciones mayores en comparacion con la media y promedio de su gráfico análogo con la otra funcion de *fitness*.

Algo destacable de esta función de *fitness* es que las mutaciones y cambios en generaciones modifican el *fitness* del mejor, que disminuye varias veces bastante (cosa que prácticamente no sucedia con la anterior función de *fitness* -ver Figura 5-). Esto último sucede porque el *fitness* depende mucho más del resto de los jugadores ya que considera más situaciones. Además, la convergencia sucede varias generaciones después que con la primer función de *fitness* (en el caso de poblacion 50 se puede ver una convergencia recién en las últimas generaciones, las mayores a 30, mientras que con el otro *fitness* era a la décima generación).

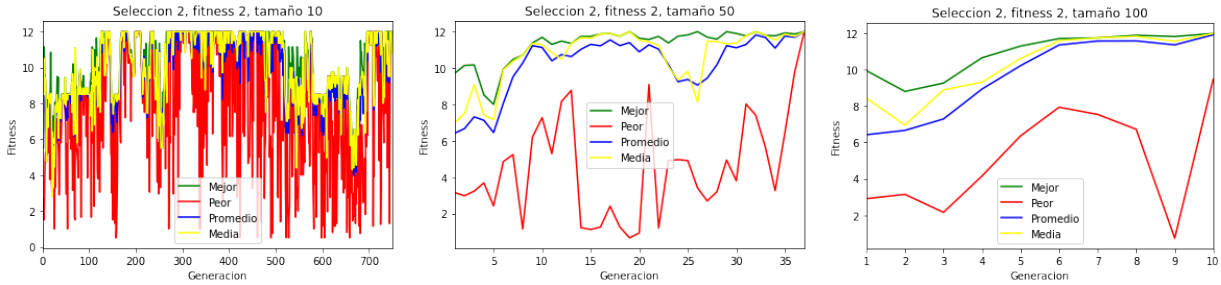
Analizando el gráfico 6.a podemos ver que las mutaciones afectan mucho más al *fitness* ya que la media desciende considerablemente, llegando a la mitad de su valor máximo en reiteradas ocasiones. Nuevamente podemos observar que las mutaciones afectan mucho más en poblaciones pequeñas, como se observaba en la Figura 5.

Teniendo entonces un análisis de la interacción entre la selección 1 con los dos *fitness*, repetimos el análisis pero con la selección 2.



(a) tamaño 10 (b) tamaño 50 (c) tamaño 100

Figura 7: Gráfico de *fitness* en función de generación. Se grafica el mejor, peor, la media y el promedio. Selección 2, *fitness* 1, $pmutar = 0{,}005$, $pcrossover = 0{,}4$



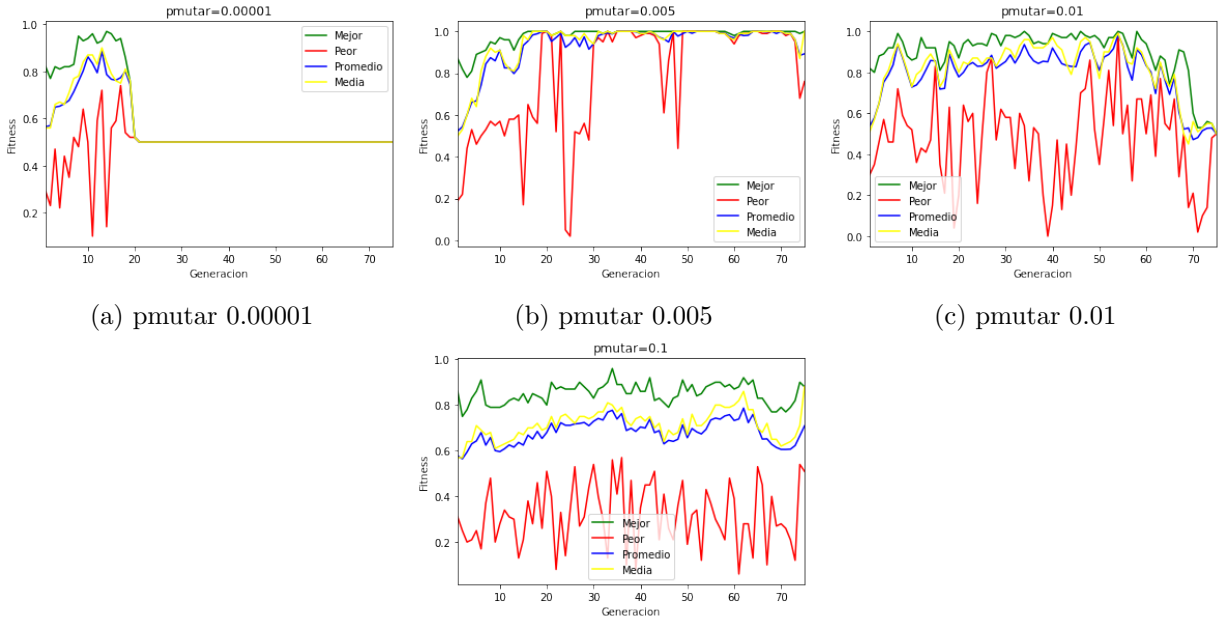
(a) tamaño 10 (b) tamaño 50 (c) tamaño 100

Figura 8: Gráfico de *fitness* en función de generación. Se grafica el mejor, peor, la media y el promedio. Seleccion 2, *fitness* 2, $pmutar = 0{,}005$, $pcrossover = 0{,}4$

En las Figuras 7 y 8 podemos ver características que refuerzan que ciertas situaciones se daban según lo conjeturado. Por ejemplo, nuevamente vemos que el *fitness* 2 da mayor variabilidad y reduce más veces el *fitness* del mejor. Nuevamente podemos ver también que la convergencia es más rápida con la función de *fitness* 1, reforzando la hipótesis de que es porque esta considera menos situaciones (no distingue victoria de empate ni si empiezo o no).

Por último, una importante diferencia entre los gráficos con distintos métodos de selección es que los que usan el segundo método presentan peor caso con mucho menor *fitness* y mucha más variabilidad (el desvío estandar es mucho mayor, lo que nos indica una convergencia que es mas lenta). Esto tiene sentido, ya que en el método de selección 1 solo se reproducían los que eran del 50 % con mejor fitness, mientras que en el método de seleccion 2 se puede reproducir cualquiera de los que estén en el 80 % de mejor *fitness* (ya que el torneo era de tamaño del 20 % de la población y el 20 % peor nunca podría ganar alguno). En este sentido, la selección 2 si bien aporta más variabilidad, tiene peores jugadores.

Tras este análisis, tratamos de entender cómo afectaban las dos variables que quedaban por ver, que eran el porcentaje de mutación y el de *crossover*. Para ver cómo afectaba la probabilidad de mutación, ejecutamos con el primer método de selección, ya que es el que proveía menor variabilidad genética y perimtiría visualizar mejor cómo afecta el porcentaje de mutación. También se utilizo la primer función de *fitness* por el mismo motivo. Por esto se ejecuto durante 5 horas, la población de tamaño 50 (intermedia, para tener una cantidad considerable tanto de generaciones como de individuos en la población) con diferentes probabilidades de mutar, con $pmutar = 0{,}00001$, 0,005, 0,01, 0,05 y 0,1. El *crossover* se fijó en 0,4.



(a) pmutar 0.00001

(b) pmutar 0.005

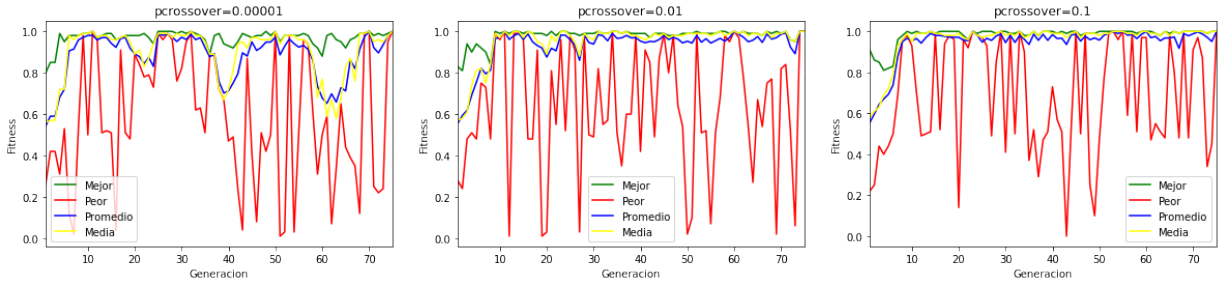
(c) pmutar 0.01

(d) pmutar 0.1

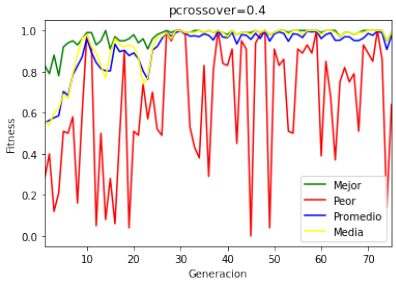
Figura 9: Gráfico de *fitness* en función de generación. Se grafica el mejor, peor, la media y el promedio. Selección 1, *fitness* 1, $pcrossover = 0{,}4$

La Figura 9 refleja claramente lo esperado. En nuestro algoritmo genético hay operaciones que favorecen la convergencia, como es el proceso de selección y *crossover*, y hay en oposición operaciones que favorecen la divergencia, como es el caso de la mutación, lo que se puede ver claramente en esta Figura. Con menores probabilidades de mutación, la convergencia no solo es más rápida, sino que los cambios en el *fitness* una vez que convergió la población (momentos donde cambia la media) son cada vez menos. En la Figura 9.d podemos ver que hay bastantes picos, tanto en la media como en peores, mejores y promedio. Esto es porque la probabilidad de mutación es altísima (es un 10 % para cada gen y hay 28 genes por individuo en este tablero) por ende es muy probable que todo individuo mute. En la Figura 9.c) la situación es la misma ya que un 1 % para cada uno de los 28 genes sigue siendo bastante alto (cada individuo tiene un 25 % de mutar que es es $1 - (0{,}99)^2 8$ o sea, la probabilidad del complemento, que es que no haya mutación en ninguno de los 28 genes). En estas dos figuras no se llega a ver una convergencia en las generaciones que se dieron (podemos conjeturar que un porcentaje de mutación tan alto no va a permitir esta convergencia). Ya en la Figura 9.b podemos ver que hay efectivamente convergencia, pero como sigue habiendo un porcentaje considerable de mutación, hay generaciones donde se sale de este pico un poco (en algunas desciende un poco la media y el promedio, mientras que en otras solo desciende el peor). Finalmente en la Figura 9.a vemos una convergencia mucho mas rápida y una vez que esta se dio, todos los *fitness* se mantienen así durante más de 50 generaciones. Esto es porque al poner una probabilidad tan baja, la mutación practicamente no sucede y no hay otras operaciones que permitan divergencia.

Por otro lado, la selección y el *crossover* son operaciones que favorecen y buscan la convergencia. Para ver la influencia de la variable restante, el *pcrossover*, se ejecutó durante 5 horas, para la población de tamaño 50 (intermedia, para tener una cantidad considerable tanto de generaciones como de individuos en la población) con diferentes probabilidades de mutar, con $pcrossover = 0{,}4, 0{,}1, 0{,}01, 0{,}00001$. El *pmutar* se fijó en 0,005.



(a) pcrossover 0.00001 (b) pcrossover 0.01 (c) pcrossover 0.1

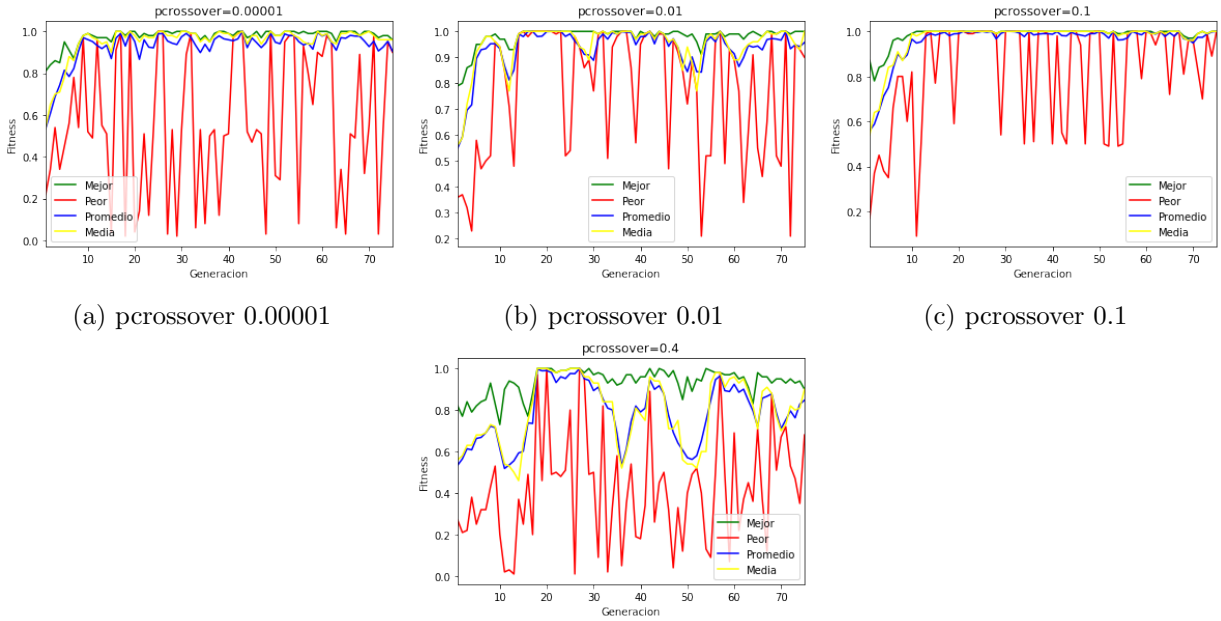


(d) pcrossover 0.4

Figura 10: Gráfico de fitness en funcion de generación. Se gráfica el mejor, peor, la media y el promedio. Seleccion 1, fitness 1, pmutar=0.005

Como podemos ver en la Figura 10, en contra de lo esperado, pareciera no haber una convergencia más rápida si aumentamos *pcrossover*. Sin embargo, considerando que la probabilidad es para cada uno de los genes, la probabilidad de *crossover* de 0,4 resulta altísima (considerando que para esta instancia de tablero particular hay 28 parametros), por lo que los individuos mezclan demasiado sus genes, tardando mucho más en converger que en el resto. Por otra parte, reforzando la hipótesis de que favorece la convergencia, si vemos algo con *pcrossover* realmente bajo como es el caso de 0,00001 en comparación con 0,01 la convergencia es más lenta una vez que se sale de esta (la media desciende mucho más y tarda más en tomar nuevamente el valor de convergencia).

Quizás algo que pueda estar influyendo es que primeramente decidimos dejar a los vectores como una unidad, como todo un gen que se transmitía o bien de un padre o del otro. Por ende, realizamos la misma experiencia solo que ahora se decidía de qué padre copiar cada uno de los elementos del vector según *pcrossover*, o sea, con este cambio el hijo puede tener una parte de un vector de uno y la otra parte del otro.



(a) pcrossover 0.00001 (b) pcrossover 0.01 (c) pcrossover 0.1

(d) pcrossover 0.4

Figura 11: Gráfico de *fitness* en funcion de generación. Se grafica el mejor, peor, la media y el promedio. Selección 1, *fitness* 1, $pmutar = 0{,}005$

Este cambio no pareciera sin embargo reflejar algo muy distinto, la única diferencia es que aporta un poco más de variabilidad en todos los casos, lo que se puede notar claramente en el gráfico 11.d ya que la media y promedio presentan mayores cambios en comparación con el gráfico 10.d, no presentan una convergencia sostenida en varias generaciones (ni siquiera en 11). Es esperable que aporte más variabilidad debido a que los individuos mezclan sus genes mucho más, dando lugares a nuevas posibilidades de valores de los vectores que son parámetro, cosa que antes solo se daba con mutaciones eventuales.

Sin embargo, con este cambio también, podemos notar que no hay influencia en la velocidad de convergencia por parte de la probabilidad de *crossover*, contrariamente a como esperabamos que sea al principio. Esto podría ser debido a que los parámetros, al ser la importancia de cada cosa, son bastante independientes entre sí y no tenemos bloques grandes que estén correlacionados entre sí en cuanto al juego (a lo sumo los pares que determinan mi importancia respecto de la del otro), por lo que no es tan relevante si copiamos toda una tira de parámetros de un mismo padre. Esta idea surge al ver que no pareciera haber cambios en la velocidad de convergencia cuando rompimos esas tiras (lo que graficamos en la Figura 11).

Sí, en cambio, resulta importante para dar más variabilidad y evitar que la media se quede en valores cercanos al de convergencia como claramente se puede ver que sucedió en el gráfico de la Figura 11.d, donde además de ser alta la *pcrossover*, usamos la segunda forma donde miramos como gen heredable de algún padre a cada elemento del vector y no a este en su totalidad.

5.2.2. Jugadores

Habiendo estudiado entonces la influencia de los distintos parámetros, tratamos de buscar una combinación de estos que dé un buen jugador. Todos los jugadores que fueron generados al final (el mejor de la última generación para cada experimento) compitieron entre ellos.

En los primeros experimentos con 2.5 horas, para cada función de *fitness* y cada método de selección, hicimos competir a los tres jugadores todos contra todos dos partidos, donde se alternaba quien comenzaba. Para el método de selección 1, con ambas funciones de *fitness* resultaron ganadores siempre los segundos jugadores y después se vió que jugaban prácticamente partidos similares donde jugaban en las casillas de más cercanas al borde. Como reflejaban los parámetros, se le daba lo que pareciera ser poca importancia

a los biextensibles y extensibles en general. Al jugarle a estos jugadores, todos perdieron rápidamente ya que ninguno tomaba el centro, y al armar rápidamente una linea de 2 que sea biextensible, como no era cubierta, pasaba a ser una de tres biextensible, lo que sentenciaba la derrota del programa. Aquí no se pudo ver diferencia entre las tres poblaciones a simple vista ya que eran malos jugadores los tres. Pero con el método de selección 2 el panorama fue distinto...

Primero, con la función de *fitness* 1, el jugador de población inicial de tamaño 100 resultó ser el peor (perdía o empataba, según empezara o no). Luego, los de tamaño 50 y 10 quedaron empatados (ganaba el segundo). Esto nos refuerza nuestra hipótesis de que la poca evolución que presentó el de población de tamaño 100 (tan solo 10 generaciones) no permitió que se desarrollen buenos individuos. Luego le jugamos a cada uno de estos dos y resulto que el de tamaño 10 no detectaba biextensibles (al igual que cuando usabamos selección 1) pero el de tamaño 50 sí logró detectar biextensibles, jugando mucho mejor que el de 10 y perdiendo porque no pude detectar una linea que era extensible en el próximo turno de tamaño $c-1$. Esto refuerza la hipótesis de que tener una población de tamaño 10, si bien evolucionaba mucho (dando 750 generaciones) tenía tan poca variabilidad que daba malos jugadores. Más aún, tiene sentido que sean peores porque en la función de *fitness* (en ambas) evaluó los partidos contra todos los de su generación, por lo que al ser pequeña la población, se jugaban menos partidos y con mas fácilidad se obtenía un *fitness* alto (esta es una clara diferencia respecto del paper, en la que se ahondará más adelante).

Más aún, con el método de selección 2 y la función de *fitness* 2, el jugador de la población de tamaño 10 fue el peor del torneo, perdiendo todos los partidos, lo que refuerza nuestra idea. Como vimos, este método y este *fitness* eran los que presentaban mayor variabilidad, pero al tener una población esto se desaprovechaba, lo que indicaría el motivo de por qué la de tamaño 10 perdió todos los partidos. Luego, la de tamaño 50 ganó todos los partidos, siendo la mejor de las tres. Esto refuerza más aún nuestra hipótesis de que no conviene una población muy grande ya que permite poca evolución (pocas generaciones) ni una tan pequeña ya que da poca variabilidad; la de tamaño intermedio (50) permite tener tanto una variabilidad como una evolución considerables. Nuevamente, para ver las estrategias y cuán bueno era cada uno, les jugamos nosotros y el de 100 no tomó el centro ni detectó biextensibles por lo que perdió rápidamente. En el caso del de 10, sucedió lo mismo. Sin embargo, en el caso del de 50, tomo el centro, cubrió extensibles nuestros, no permitió que generáramos biextensibles (quizás no los detecta, pero por lo menos no lo permitió) y terminó ganando ya que hubo un momento que ambos jugadores llenamos el tablero por no poder jugar en una columna ya que ganaría el otro. Repetimos varios juegos contra este jugador, empezando nostros y notamos que cubría biextensibles al principio del juego, pero avanzado el juego dejaba de hacerlo y perdía por eso. Efectivamente, observando los parámetros podemos ver que le da importancia biextensibles (más a generarlos que evitar que el otro lo haga) y a los extensibles en la próxima (más a taparlos que generarlos), pero no es mucho mayor que otras comparativamente por lo que al haber más características en juego, podría dejarlos de lado. Le daba importancia a jugar en el centro también.

Esta experimentación nos permitió ver que resultaba más efectivo, en la misma cantidad de tiempo, una población de tamaño intermedio (50) que una chica (10) o grande (100) ya que proveía tanto evolución como variabilidad genética. Más aún nos mostró que los métodos de *fitness* cobraban más importancia al tener una población mayor (ya que se completan más partidos). También nos mostró que dio mejores resultados el proceso de selección 2, lo que tenía sentido ya que aportaba muchísima más variabilidad genética. Dentro del proceso de selección 2 se noto claramente que daba mejores resultados una población intermedia y también que la función de *fitness* 2 era más efectiva (ya que diferenciaba más situaciones, que al jugar partidos solo contra los de nuestra generacion, se volvían importantes estas diferencias) puesto que aportaba más variabilidad.

Por estos resultados y todo lo analizado, decidimos a la hora de buscar jugadores buenos utilizar el proceso de selección 2, la funcion de *fitness* 2 ya que aportan más variabilidad y distinguen más situaciones. Tomamos una población de tamaño 50 que daba una variabilidad considerable y también permitía realizar varias generaciones sin tomar tanto tiempo, además porque este tamaño daba más fidelidad a la función de *fitness* ya que para determinar cuan bueno era un jugador lo hacíamos competir contra más rivales.

La probabilidad de *crossover* la fijamos en 0,4, y tomamos cada elemento de los vectores como un gen, para asegurar que haya variaciones dentro de los parámetros que utilizan ya los jugadores seleccionados. La probabilidad de mutar se fijó en 0,005 para asegurar que haya variaciones pero no sean demasiadas como para evitar la convergencia.

Se ejecutó esto durante 10 horas (en un total de 150 generaciones) y se obtuvo así un jugador que parecía ser bastante bueno y superador (le ganaba al obtenido anteriormente con una población de tamaño 50 y dos horas, lo que tenía sentido pues se entrenó durante más generaciones). Para checkear la validez de este método y que efectivamente se trataba de un jugador con una estrategia definida buena, se lo hizo competir 1000 veces contra el jugador random siendo segundo y perdió 49 veces, empataron 2 veces y siendo primero perdió 21 veces y empataron 2 veces. Si consideramos los 2000 partidos jugados, **no ganó tan solo en 74 WAS IST DAS? SI GANÓ SOLO 74 VECES, NO ES EL 96,3 % ...**, o sea que ganó en un 96,3 % de los partidos. Esto valida que nuestro jugador era bueno, considerando que salió de una población incial en que todos los valores eran aleatorios. Más aún, creamos un jugador nosotros con parámetros puestos a mano que creíamos que serían los mejores y ese jugador perdió contra el que la máquina había dado, tanto siendo primero como segundo, aunque obtuvo mejores resultados contra el random (ganó en un 100 %).

EUUUUUU, OJO. En el grid me gustaría decir que comparar con random es malisimo porque le ganas siempre,asi justificamos por qué usamos el backtracking. Caga mucho la estructura de la experimentación esta decir eso?? Nada, te escribo estos para que lo pienses y domingo a la tarde lo hablamos

Sin embargo, algo de mucha importancia no analizado aún es que como solo competimos dentro de los jugadores de nuestra generaciones, sería bueno tener jugadores bastante variados, con diversas estrategias y que le den mas importancia a algunas características que a otras. Si bien, como ya se dijo, obtuvimos un jugador bueno en la experimentación hasta ahora realizada, no era lo suficientemente bueno (no superaba en principio al que nosotros le elegimos los parametros). Por esto, decidimos para ver si obteníamos mejores jugadores, brindar una población con alguna estrategia inicial buena. Utilizamos entonces el jugador que propusimos que priorizara lo que creíamos era bueno (dar más importancia a biextensibles, extensibles-proximo, extensibles, jugar en el centro al principio) y al resto de la población la pusimos como random. Esto era factible de funcionar porque nuestra selección tiene elitismo, o sea que el mejor individuo anterior siempre es copiado tal cual, por lo que si nuestro jugador fuese el mejor, nunca debería ser reemplazado si nadie lo supera. Sin embargo, el mejor individuo seguía siendo factible de sufrir mutaciones, por lo que corrimos un caso donde el mejor individuo de la generación pasada no podía mutar. Se esperaba entonces que el jugador obtenido sea mejor que el inicial. Se ejecutó durante 10 horas (4 veces más de lo que se había dejado anteriormente), o sea un total de 150 generaciones.

Tras esto, obtuvimos un jugador que resultó ser peor que nuestro jugador inicial (aunque no considerablemente peor, ganaba en un 98.6 % al random mientras que el inicial en un 100 % -sobre una muestra de 1000 partidos como primero y 1000 como segundo-). Si bien esto contraría lo esperado, podría suceder algo similar a lo que sucede en `grid search` cuando dimos una solución inicial a partir de la cual explorar el espacio, ya que al jugar solo partidos contra los de nuestra población (que además son todos determinísticos en cuanto a sus jugadas, dos jugadores -una vez fijado quién empieza- siempre jugarán el mismo partido) es bastante probable encontrar jugadores que sean "malos" en general pero que en particular le ganen a nuestro jugador ya que la rama de partido que se da entre ellos dos hace que gane el "malo". De esta forma, no es raro que se genere un individuo parecido al inicial que le gane a este y pase a ser el nuevo de mejor *fitness*. PODEMOS PONER ALGO QUE DA IGUAL MUCHO MENOS PEOR QUE EN EL GRID, ACA DIO UN 98.6 % DE GANABA, SE ALEJO UN 1.4, FIJATE CUANTO EN EL TUYO. (PODES MIRARLO FACIL SI BORRAS LOS ARCHIVOS LOG Y LO HACES JUGAR VIA EL JUEZ MIL VECES Y TE IMPRIME EN EL LOG Y TOCAS CTRL F PERDISTE O EMPATASTE Y LOS CONTAS A MANO Y HACES LA PROPORCION.

NO SE, EN EL GRID ME DA QUE CUALQUIER COSA LE GANA CASI SIEMPRE A RANDOM... CREO QUE JUSTO TUVISTE MALA SUERTE CON LO QUE TE QUEDO DEL GENETICO

ACA INVENTE ESTO DEL PARRAFO DE ABAJO, PERO SI LES PARECE BIEN, LO DEJAMOS,

YA FUE.

Se repitió la misma experimentación en otro tablero con $N = 9$, $M = 8$, $C = 5$ y $p = 100$ (fichas de sobra). Donde, en cuanto a la distribución que determinaban los métodos de selección *fitness* y *pmutar* fue análoga, sin aportar mucha diferencia (como era esperable). Lo único que destacamos es que el pocentaje que le ganaba al random creció considerablemente (llegó a un 98.4 % contra el 96.3 %) en el obtenido por el algoritmo genético con una población inicial aleatoria. Esto lo atribuimos a que al ser mayor el tablero, es más difícil que el random gane ya que la probabilidad de caer justo en un juego en el que pierda (o que gane el random análogamente) es mucho menor porque hay más columnas donde jugar y es menos probable que el random haga algo bueno (en principio logre formar líneas). Además el c es mayor por lo que para que el random gane debe alinear 5 fichas que es menos probable que 4. Efectivamente los resultados de la técnica pueden variar para distintos tableros, donde hay que tener en cuenta también que utilizar al jugador random para validar cada vez se vuelve menos viable al aumentar el tamaño del tablero. Para reforzar esto se repitió lo mismo con un tablero de $N = 14$, $M = 12$, $C = 10$ y $p = 500$ (fichas de sobra) y el jugador obtenido ganó en un 99.8 % de las veces.

6. Análisis comparativo con paper

En comparación con el paper, antes que nada cabe destacar que hay grandes diferencias en los juegos. Si bien ambos son de dos jugadores, en el Tatetí la cantidad de configuraciones es abarcable (de hecho, sacando los no factibles y vía isomorfismos, los autores muestran que hay tan solo 827 configuraciones posibles) mientras que en el c en línea esto no sucede por varios motivos. Primero que nada, como podemos variar el tamaño del tablero, la cantidad de configuraciones puede aumentar muchísimo y como la grilla (salvo contados casos) es mucho mayor, la cantidad de partidas que se pueden dar es mucho mayor ya que si miramos los tableros -factibles o no- el tatetí presenta 3^9 y el c en linea 3^{NM} donde $N * M$ suele ser mayor que 9 salvo contados casos. Como esta cantidad crece exponencialmente, al aumentar un poco N y M, esta cantidad se hace inabarcable por lo que resulta imposible realizar una estrategia similar a la del paper (que mira la configuracion actual y para esa tiene definida que jugada hacer previamente en un arreglo).

Sin embargo, podemos adaptar la estrategia a nuestro juego particular y es por esto que se estableció un sistema de puntos que permitía caracterizar los posibles tableros que se pueden dar tras nuestra jugada y así elegir el que sea el mejor. Y es aquí donde entra la libertad que le damos al jugador para que luego trabaje el algoritmo genético. Esta libertad también existía en el paper y era mucho más flexible aún. De hecho era lo más flexible posible porque dejaba elegir para cada situación que jugar al algoritmo. En cambio, por la dimensión del problema, nosotros no podíamos dar tanta flexibilidad ya que resultaría inabarcable; por esto surge el sistema de puntos, nosotros tomamos ciertas características del tablero y permitimos que luego el algoritmo genético decida la importancia de cada una de esas características y juegue en base a dicha importancia. La función que establecía el puntaje de un tablero es la que en realidad toma los parámetros que el genético busco modificar y de hecho, estos parámetros son los que determinan la importancia de cada una de las características que nosotros decidimos. Aquí está la flexibilidad que tiene nuestro jugador (en algún punto, no es tan flexible ya que solo puede variar dentro de la importancia de las características que nosotros determinamos, todas las que no hayamos elegido no las podra distinguir).

Más aún, hay una notable diferencia que es que al ser menos flexible y tener que elegir qué características uno quiere que distinga el jugador para que luego el genético decida su importancia, pareciera ser muy importante la elección de estas características. Efectivamente lo es, pero nos permite ver algo que no sucedía en el paper, que es que poner características de más que no sean relevantes no es algo tan problemático ya que el algoritmo genético luego determinará que son poco o nada importantes.

Además, nosotros tomamos dos funciones de *fitness* que en esencia eran las mismas que en el paper para poder analizar cómo esto variaba según el juego. Efectivamente en nuestro juego, el c en línea, pudimos ver que el *fitness* de puntaje fue más fructífero que el de proporción de partidos no perdidos. Nuestra hipótesis es que esto sucedió porque distinguía más situaciones (empate de victoria y empezar o ir último). Sin embargo, el hecho de que en el paper el sistema de punteos también distinguía más situaciones y haya

resultado peor no contradice esto ya que hay una diferencia esencial en el cálculo de *fitness* en ambos casos: nuevamente al ser una cantidad abarcable todos los tableros del ta-te-ti en el paper a un jugador se lo hace jugar todos los juegos posibles y se evalúa ahí cuántos no perdió y al ver todas las situaciones es claro que cobra mucha más importancia minimizar las perdidas. Mientras que en el c en línea resulta inabarcable hacer jugar a cada jugador de cada generación todos los partidos posibles (ya hacer jugar a un jugador todos los partidos posibles es caro como la primer jugada de `minimax`, que en tableros grandes puede tardar muchísimo tiempo en jugar por el crecimiento exponencial que presenta esta cantidad), por lo que importa mucho más tener en cuenta estas diversas situaciones ya que la cantidad de partidos que podemos hacer jugar a un jugador es mucho más acotada.

Incluso esto nos permitió entender por qué no tuvieron éxito las poblaciones de 10 jugadores ya que ademas de presentar una poca variabilidad, las funciones de *fitness* se basaban en partidos contra muy pocos jugadores. En este caso, a diferencia del paper, variar los tamaños de las poblaciones sí mostró mejoras en los jugadores (no lo hizo en el *fitness* final del mejor jugador de la última generación). Por el tamaño del problema, nuevamente, resulta imposible encontrar un jugador óptimo, cosa que fue posible por medio del algoritmo genético en un problema más pequeño como es el del tatetí. Lo que sin embargo se busco hacer es encontrar un buen jugador de c en línea a través de encontrar los parámetros adecuados para el jugador parametrizable por medio del algoritmo genético.

Esto último va de la mano con lo que miden las funciones de *fitness*, ya que en el c en línea no fue posible definir una función que midiera realmente cuán bueno era un jugador ya que eso implicaría jugar todos los partidos posibles lo que no era abarcable. Sí permitían ver cuán bueno era un jugador comparativamente en relación con otro/s (lo que se determinaba simplemente haciendolos competir). Por esto, nuestro algoritmo genético con estas funciones de *fitness* podía dar buenos jugadores pero no óptimos, ya que requeriría en principio alguna medida de cuán bueno es un jugador en todas las situaciones.

7. Conclusiones

Concluimos primeramente que presentamos un jugador óptimo mediante la técnica de *minimax*, que salvo para tableros muy pequeños, era inviable ya que consideraba todas las posibilidades de juegos que se podían dar y tomaba en las que se maximizaba el resultado (considerando que el otro jugador quería minimizar dicho resultado -el resultado es gano yo, empatamos o perdemos, con ese orden por lo que maximizar es buscar ganar, cosa que si no se puede empatar y si tampoco se puede, perder; recíprocamente es minizar-). Se mostró que este algoritmo hacía cada jugada en $\mathcal{O}(N^{(\mathrm{N}+1)\mathrm{M})+1})$ (se mostró una mejor cota que implicaba las casillas libres en el momento y las columnas donde se podía jugar, lo que se acotó por esta complejidad). Por ser de un gran tiempo de ejecución, se dio el mismo algoritmo con la poda $\alpha - \beta$ que concluimos permitió ejecutar el algoritmo en menor tiempo, pero para tableros que no sean pequeños seguía siendo inviable ya que el tiempo de ejecución seguía siendo altísimo porque si bien no se analizaba todo el árbol de minimax, si se chequeaba una gran parte del mismo. La poda si bien se ejecutaba, no nos permitía asegurar una complejidad teórica menor, aunque sí un menor tiempo de ejecución que se comprobó experimentalmente.

Teniendo esto en cuenta, se busco generar un jugador que tenga un tiempo de ejecución considerable jugada a jugada y que analice sólo una jugada adelante (no más, ya que sino sería como hacer -al menos por unas jugadas- minimax). Este jugador era goloso en el sentido de que decidía la mejor de las siguientes jugadas (en nuestro caso la que de un tablero de mayor puntaje). La idea era que este jugador tenga parámetros que se dejen libres, para despues buscar qué valores tenían que tomar para dar un mejor jugador. Se mostró experimentalmente que este jugador tardaba $\mathcal{O}(N^2M)$ en realizar cada jugada. Estos parámetros que se dejaron libres fueron características del juego decidida por nosotros y luego se mejoraría al jugador decidiendo qué características eran más importantes. Este trabajo se hizo con dos técnicas distintas: `Grid Search` y un Algoritmo Genético.

CONCLUSIONES GRID

CONCLUSIONES GENÉTICO (HAGANLAS USTEDES, LEAN LO MIO Y RESUMAN, YO DESPUES CUANDO LO VEA EL DOMINGO A LA TARDE-NOCHE LO COMPLETO Y CORRIJO)