Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Grupo n, con $n = 12$

Integrante	LU	Correo electrónico
Federico Lebrón	313/09	fede_lebron@hotmail.com
Ivan David Postolski	216/09	eveepostolski@hotmail.com
Martín Alejandro Miguel	181/09	m2.march@gmail.com
Paula Chocrón	190/09	paulachocron@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Sistema de archivos

```
|-- doc
|   `-- informe
|       `-- tp.pdf
|-- src
|   |-- evaluacion.cpp
|   |-- ia.cpp
|   |-- ia.h
|   |-- isle_utils.cpp
|   |-- juego.cpp
|   |-- juego.h
|   |-- jugador.cpp
|   |-- Makefile
|   |-- prelude.h
|   |-- profiling.cpp
|   |-- profiling.h
|   |-- tablero.cpp
|   |-- tablero.h
`-- test
    |-- battles
    |   |-- dats
    |   |   |-- processors
    |   |   |   |-- ev_proc.py
    |   |   |   |-- masterss_proc.py
    |   |   |   |-- tout_proc.py
    |   |   |   `-- tx_proc.py
    |   |   |-- ev_h.dat
    |   |   |-- ev_h.full
    |   |   |-- ev_i.dat
    |   |   |-- ev_i.full
    |   |   |-- masters_r.dat
    |   |   |-- th_full201011241125
    |   |   |-- th_pout201011241125
    |   |   |-- th_tout201011241125
    |   |   |-- ti_full201011241342
    |   |   |-- ti_pout201011241342
    |   |   |-- ti_tout201011241342
    |   |   |-- tx.dat
    |   |   `-- tx-t.dat
    |   |-- plots
    |   |   |-- ev_h.p
    |   |   |-- ev_h.ps
    |   |   |-- ev_i.p
    |   |   |-- ev_i.ps
    |   |   |-- heurb_c.p
    |   |   |-- heurb_c.ps
    |   |   |-- Makefile
    |   |   |-- masterss.p
    |   |   |-- svg2pdf
    |   |   |-- th.ps
    |   |   |-- ti.ps
    |   |   |-- torneo_heurb.p
    |   |   |-- torneo_inter.p
    |   |   `-- torneo_x.p
    |   |-- heurb_challenge.py
    |   |-- masterss.py
    |   |-- battle.py
    |   |-- torneo_heurb.py
    |   |-- torneo_inter.py
    |   |-- torneo.py
    |   |-- torneo_x.py
    |   `-- utils.py
    |-- lib
    |   |-- progressGui.py
    |   |-- pyColors.py
    |   `-- sysUtils.py
    |-- measurements
    |   |-- outs
    |   |   |-- ab_2x2
    |   |   |-- ab_3x3
    |   |   |-- ab_4x4
    |   |   |-- ab_5x5
    |   |   |-- mm_2x2
    |   |   |-- mm_3x3
    |   |   |-- mm_4x4
    |   |   `-- mm_5x5
    |   `-- tables
    |       |-- ab_2x2
    |       |-- ab_3x3
    |       |-- ab_4x4
    |       |-- comp_ab_mm_4x4-ab.svg
    |       |-- comp_ab_mm_4x4.dat
    |       |-- comp_ab_mm_4x4-mm.svg
    |       |-- comp_ab_mm_4x4.p
    |       |-- comp_ab_mm_4x4.pdf
    |       |-- comp_ab_mm_4x4.svg
    |       |-- mm_2x2
    |       |-- mm_3x3
    |       `-- mm_4x4
    `-- playground
        |-- judge.c
        |-- jugadores_cat
        |   |-- jug_labi.cpp
        |   |-- jugs_varios.cpp
        |   `-- jug_wabi.cpp
        |-- jugadores_ours
        |   `-- Makefile
        |-- jug_gtk.py
        |-- ladrillos.py
        `-- Makefile
```

1.1. Descripción de Directorios

1.1.1. doc/informe

En este directorio se encuentra el presente informe.

1.1.2. src

En este directorio se encuentran todos los archivos fuentes desarrollados para las distintas implementaciones de jugadores. Todo lo explicado y desarrollado en el trabajo se condensa en el ejecutable jugador, que se compila mediante el Makefile presente en el directorio. A continuación presentamos los parámetros que debe recibir jugador para efectuar cada una de las implemetnaciones requeridas por el enunciado:

- **Minimax** ./jugador De esta forma el jugador juega de la forma más primitiva, que para nosotros conforma el *minimax*
- **Alphabeta** ./jugador –alphabeta Activa la poda $\alpha - \beta$. Con esta extensión agrega la poda la búsqueda de jugadas efectuada por el algoritmo.
- **Goloso** ./jugador –height 1 –funcion 1 Le exige al algoritmo que baje en la recursión un máximo de 1 nivel y evalúe los tableros con la función 1, que es nuestra función heurística. De esta forma el jugador juega golosamente acorde al resultado de la función golosa.
- **Goloso parametrizado** ./jugador –height 1 –funcion 1 –params a b c d e Equivalente al goloso, pero ahora parametriza a la función heurística con los valores pasados por parámetro.
- **Adaptativo** ./jugador –params a b c d e –adaptive x k Esto activa el mecanismo adaptativo en relación a los valores de x y k como es descrito en la sección correspondiente del informe.

1.1.3. test/measurements

En este directorio se encuentran las mediciones realizadas sobre el árbol generado por *minimax* y $\alpha - \beta$ *pruning*. El subdirectorio outs tiene los outputs obtenidos de la versión modificada de los algoritmos de juego que devolvía información sobre el árbol. tables posee los archivos prolijos para la confección de tablas y gráficos. mm_*nxn* representa las mediciones de *minimax* para un tablero de tamaño n. ab_*nxn* es el análogo para $\alpha - \beta$ *pruning*. comp_ab_mm_4x4.* son los archivos de preparación para el grafo de comparación entre ambos árboles.

1.1.4. test/lib

En este directorio se encuentran distintos archivos fuente de python que son utilizados como utilitarios para otros fuentes.

1.1.5. test/playground

Este directorio fue armado para utilizar como terreno de juego. Finalmente terminó tomando la funcionalidad de ubicación de todos los ejecutables relacionados con las partidas de Ladrillos™. El subdirectorio jugadores_cat contiene los fuentes y el Makefile para compilar los jugadores de la cátedra en ese mismo directorio. El subdirectorio jugadores_ours posee un Makefile que compila los fuentes en src y copia los ejecutables pertinentes . Es necesario aclarar que el ejecutable jugadores se copia a este directorio como alphabeta por cuestiones de compatibilidad con todo el otro software desarrollado que llamaba al binario por ese nombre.

1.1.6. test/battles/

En este directorio se encuentran, entre otras cosas, todos los scripts en python que corren lo distintos torneos mencionados a lo largo del trabajo práctico. Para correr cualquiera de estos es necesario haber hecho make en test/playground

- **battles.py:** este script corre los dos tipos de algoritmos evolutivos. Para la versión de competencia interna los parámetros son "0 -i", para la competencia contra heurb son "0 -h". Devuelve en el stderr el output completo de la evolución y en stdout la versión resumida.
- **masterss.py:** este script corre el **Masters Series**. Recibe como parámetros archivos de jugadores de los cuales obtendrá los jugadores para el torneo. Los archivos de jugadores se formatean como líneas de números separados por espacios donde cada línea es un jugador y cada número un parámetro.

- **heurb_challenge.py:** este script no es mencionado en el trabajo. Recibe como parámetro un archivo de jugadores (ver item anterior) que hace competir contra heurb, dejando en el stdout una tabla con los resultados.
- **torneo_heurb.py:** este script corre el **torneo suizo** contra heurb. Devuelve los resultados en archivos de tres tipos th_full, th_pout, th_tout (ver más a adelante).
- **heurb_inter.py:** este script corre el **torneo suizo** con competencia interna. Análogo a torneo_heurb.py devuelve archivos de tipo ti_full, ti_pout, ti_tout.
- **torneo_x.py:** corre el torneo utilizado para evaluar el algoritmo adaptivo. Devuelve en el stdout la tabla de resultados.

1.1.7. test/battles/dats

Este es el directorio que contiene los datos resultado de las distintas simulaciones.

- **ev_*.* :** estos son los datos resultado de los algoritmos evolutivos. Las variantes ev_h.* están relacionadas con la corrida con competencia contra heurb y las de tipo ev_i.* con la competencia interna. Luego la extensión ev_*.full son archivos de output completo mientras que ev_*.dat son la versión resumida.
- **masters_r.dat :** este archivo contiene la tabla resultado del **Masters Series**.
- **t*_*n :** estos archivos constituyen los distintos outputs de los torneos. Las variantes th_*n corresponden a torneo_heurb.py y ti_*n a torneo_inter.py. t*_full$n$ contiene el output completo del algoritmo, t*_tout$n$ contiene los resultados con la evolución de la tabla de los mejores 15. Finalmente t*_pout$n$ contiene las estadísticas de los mejores 15 finales. n es la fecha en que se generó el archivo.
- **tx* :** tx.dat es la tabla resultado de torneo_x.py. tx-t.dat son las tablas utilizada para los ploteos de tiempo. Esta fue calculada a partir de torneo_x.py

1.1.8. test/battles/dats/processors

En este directorio se encuentran algunas utilidades creadas para parsear las tablas en test/battles/dats/ ya sea para obtener otras tablas para plotear o para pasarlas a un formato cómodo en latex.

1.1.9. test/battles/plots

En este directorio se encuentran los scripts de GNUPLOT para la realización de gráficos. Estos terminan en .p. Los archivos .ps son archivos de jugadores. El nombre sigue la nomenclatura utilizada previamente para relacionarlos con el torneo del cual fueron originados. El Makefile presente ejecuta los scripts de GNUPLOT y los convierte a formato pdf utilizando la utilidad svg2pdf cuyo binario se encuentra en el mismo directorio.

2. Introducción

El siguiente trabajo práctico hace uso de algoritmos heurísticos y meta-heurísticos para el desarrollo de porgramas de inteligencia artificial. Estos programas juegan un juego denominado *Ladrillos*. En una primera etapa se trabaja con algoritmos óptimos para luego pasar a investigar el potencial de distintos algoritmos heurísticos. Finalmente se busca una combinación de las ventajas de ambos para lograr un jugador final cuyo tiempo de ejecución sea acceptable y mantenga una calidad de juego suficientemente buena.

En la primera parte del desarrollo del trabajo se implementa una inteligencia artificial que juega de forma óptima mediante la técnica *minimax*. Luego se la optimiza mediante α-β *pruning*. Viendo como la complejidad temporal y espacial del algoritmo hace impracticable su uso, se pasan a implementar algoritmos de juego utilizando distintas heurísticas de evaluación estática del tablero. Estas heurísticas son luego aprovechadas junto con la técnica de cálculo de jugadas óptimas para obtener mejores jugadores.

El trabajo práctico se enfoca en la investigación que aparece alrededor del desarrollo de los métodos heurísticos, lo que da lugar a la evaluación de distintas ideas y variantes dentro de las mismas. Esto es principalmente dado que las heurísticas suelen estar parametrizadas y la búsqueda de parámetros óptimos representa un conflicto en si mismo.

3. Algoritmo óptimo

3.1. Implementación: minimax

minimax es un algoritmo utilizado para resolución de problemas de decisión. Para el caso particular de los juegos de suma-cero, constituye un algoritmo que devuelve la movida óptima a cada paso.

Supongamos que se está jugando un juego de suma-cero de dos jugadores. Tenemos entonces que todo lo que gana un jugador lo pierde el otro. En un determinado turno un jugador tiene una serie de movidas posibles. Cada una de ellas establece un tablero, en primer lugar distinto del resto. El jugador va a intentar jugar la movida que deje el tablero en la posición más favorable para él. La cuestión está en como calcular cual es el tablero más favorable. Consideremos ahora que el otro jugador posee un oráculo que le dice con qué jugada obtiene el mayor provecho. En particular, cuanto más obtenga él, menos obtenemos nosotros, por ser el juego de suma-cero. Si el primer jugador conociera el resultado del oráculo, tendría que elegir la movida que le diera al oponente (o sea, nosotros) la menor ganancia, de forma de maximizar la suya. El cálculo que hace el jugador es

$$\max_{j \in \text{Jugadas mías}} \{ \min_{j' \in \text{Jugadas oponente para } j} \{ganancia(j')\}\}$$

donde *ganancia* sería la función oráculo. No obstante, esta función se hace innecesaria, ya que así como el primer jugador calcula su movida en base a la del oponente, este puede hacer lo mismo. Empezaría así a construirse un árbol de jugadas donde se evalúa para cada movida posible las movidas del oponente en respuesta, luego las respuestas posibles a ellas y así sucesivamente. La construcción termina cuando se llega a una movida terminal, que da por terminado el juego. En esa instancia la función *ganancia* está correctamente definida y establece cuanto gana un jugador y pierde el otro.

Formalizando estas ideas, consideremos un juego J de suma-cero, de dos jugadores I y II. Llamemos $E(J)$ al conjunto de todos los posibles estados del juego, y dado un $t \in E(J)$, llamamos $j(t, I)$ a las jugadas de I para t, y $j(t, II)$ las de II en el estado t. La función solo está definida si en el estado t es el turno del jugador del que se preguntan las jugadas. Para un estado terminal devuelve el conjunto vacío. Para un $t \in E(J)$ y un j_0 en $j(t, I) \cup j(t, II)$, definimos la operación $t + j_0 \to t'$ como la forma de obtener el estado del juego luego de una jugada en un estado anterior.

También definimos $\neg : \{I, II\} \to \{I, II\}$ como $\neg(k) = \begin{cases} I & \text{si } k = II \\ II & \text{si } n = I \end{cases}$

Con esto dicho podemos definir apropiadamente la función:

$$ganancia : E(J) \times \{I, II\} \longrightarrow \mathbb{R}$$

$$ganancia(t, x) = \begin{cases} \max_{a \in j(t,x)} -ganancia(t + a, \neg(x)) & \text{si t no es terminal} \\ valor(t, x) & \text{si t es terminal} \end{cases}$$

Esto es equivalente a la definición con min/max, dado que, al ser un juego de suma cero, maximizar el negativo de la ganancia del otro jugador es equivalente a maximizar mi propia ganancia.

Veamos ahora la implementación propiamente dicha y algunos detalles de esta. La implementación de la función *ganancia*, que llamaremos *minimax*, recibe los mismos tipo de valores. En este caso vamos a diferenciar los jugadores, siendo YO el jugador del cuál queremos maximizar la ganancia y ÉL de quién queremos minimizarla. Luego dividimos el mín máx en dos llamadas a la función utilizando estas definiciones. La función también debe devolver la jugada óptima.

```
MINIMAX(E(J) t, Jugador x) → (int, jugada)
 1  if j(t, x) ≠ ∅
 2     then
 3          jugada max_j ← None
 4          int max_r
 5          if x ==YO
 6            then max_r ← ∞
 7            else  max_r ← −∞
 8          for a ∈ j(t, x)
 9               do
10                  (m′, j′) ← MINIMAX(t + a, ¬x)
11                  if x ==YO
12                    then
13                         if m′ > max_r
14                           then
15                                max_r ← m′
16                                max_j ← a
17                    else
18                         if m′ < max_r
19                           then max_r ← m′
20                                max_j ← a
21
22          return (max_r, max_j)
23     else
24          return (VALOR(t, x), None)
```

Código 1: Implementación del código para MINIMAX

3.2. Análisis de complejidad

Vamos a acotar por arriba la complejidad del algoritmo *minimax*. El algoritmo recorre todos los nodos del árbol del juego al que se aplica. Por lo tanto, podemos analizar nuestro juego y ver cuantos nodos va a tener nuestro árbol. Debemos considerar que en cada hoja del árbol (i.e. tablero terminal), hacemos la evaluación de puntaje del tablero al que llegamos, tomando tiempo $\mathcal{O}(n^2)$, y que hacemos $\mathcal{O}(n^2)$ trabajo en los demás nodos, los internos, para saber donde poner las fichas. Encontremos, entonces, una cota superior para el tiempo que tarda nuestro algoritmo.

Primero veamos algo sobre la forma del árbol.

Lema 1. *Un nodo del árbol a distancia k de la raíz tiene como máximo $2n(n-1) - k$ hijos.*

Demostración. Cada vez que ponemos una pieza, estamos sacando posibilidades de juego. A veces sacamos varias posibles jugadas al poner una pieza (hasta 7), otras solo 1 jugada. Vamos a acotar por el peor caso: el que solo sacamos una jugada posible. Este es el peor caso pues mientras más jugadas saquemos por movida, más angosto se hace nuestro árbol, dado que los hijos de una movida son las posibles movidas próximas.
Vamos a acotar el árbol por el árbol que obtenemos si nosotros jugamos primero: si jugamos segundo, nuestro árbol es simplemente un sub-árbol de éste.

Inicialmente, podemos poner nuestra ficha en $I = 2n(n-1)$ posiciones. Esto es porque podemos poner la ficha de 2 maneras (horizontal y vertical) en cualquier casilla (i, j) con $i, j < n - 1$, un cuadrado de $(n-1)^2$ casillas. Luego, podemos poner en la última fila del tablero $n - 1$ fichas horizontalmente, y otras $n - 1$ posibilidades

surgen de poder poner fichas verticalmente del lado derecho ($j = n - 1$). Esto nos da:

$$\begin{aligned} I &= 2(n-1)^2 + 2(n-1) && (1)\\ &= 2(n^2 - 2n + 1) + 2n - 2 && (2)\\ &= 2n^2 - 4n + 2 + 2n - 2 && (3)\\ &= 2n^2 - 2n && (4)\\ &= 2(n^2 - n) && (5)\\ I &= 2n(n-1) && (6) \end{aligned}$$

Sabemos que en cada movida que hagamos, vamos a sacar por lo menos 1 posibilidad de juego (en particular, la posibilidad que estamos usando al jugar). Luego por inducción, a un nodo en el k-ésimo nivel del árbol (se toma a la raíz como nivel 0), le voy a haber sacado k posibilidades de juego, pues habrán jugado k veces los jugadores, sacando 1 posibilidad cada vez. Por tanto va a tener como máximo $I - k = 2n(n-1) - k$ hijos, al menos k jugadas no siendo posibles si el nodo está a distancia k de la raíz. □

Ahora veamos cuantos nodos habrá en cada nivel.

Lema 2. *En el árbol hay como máximo $n_k = \frac{(2n(n-1))!}{(2n(n-1)-k)!}$ nodos a distancia exactamente k de la raíz.*

Demostración. Por inducción.
Queremos ver que $n_k = \frac{(2n(n-1))!}{(2n(n-1)-k)!}$ es una cota de la cantidad de nodos a distancia exactamente k de la raíz.
Para facilitar las cosas, llamemos m_k a la cantidad de nodos a distancia exactamente k de la raíz.
Solo hay un nodo a distancia $k = 0$ de la raíz: la raíz misma. Entonces veamos:

$$\begin{aligned} m_0 &= 1 && (7)\\ &= \frac{I!}{I!} && (8)\\ &= \frac{(2n(n-1))!}{(2n(n-1)-0)!} && (9)\\ &= n_0 && (10) \end{aligned}$$

Queda entonces probado el caso base $k = 0$, y en particular vale la igualdad.

A continuación, asumimos que $m_k \leq n_k$ y queremos probar $m_{k+1} \leq n_{k+1}$.

Ahora, sabemos por (1) que un nodo a distancia k de la raíz tiene, como máximo, $2n(n-1) - k$ hijos. Entonces, sabiendo que un nodo a distancia $k+1$ de la raíz debe tener a su padre a distancia k de la raíz, que hay como mucho n_k de estos por hipótesis inductiva, y estos a su vez pueden tener como máximo $2n(n-1) - k$ hijos, vemos que la cantidad de nodos en el nivel $k+1$ es, como mucho, $(2n(n-1) - k) \cdot m_k = (I - k) \cdot m_k$.

$$\begin{aligned} m_{k+1} &\leq m_k \cdot (I - k) && (11)\\ &\overset{\text{H.I.}}{\leq} n_k \cdot (I - k) && (12)\\ &= \frac{I!}{(I-k)!} \cdot (I - k)) && (13)\\ &= \frac{I!}{(I-k) \cdot (I-(k+1))!} \cdot (I - k) && (14)\\ &= \frac{I!}{(I-(k+1))!} && (15)\\ &= n_{k+1} && (16) \end{aligned}$$

Esto prueba el caso inductivo, y entonces queda demostrado que $m_k \leq n_k \ \forall\ k$, que es lo que queríamos demostrar. □

Lema 3. *Sean $I \in \mathbb{N}$ y $F \in \mathbb{N}$, tal que $I \geq 1$ y $F \leq I - 2$. Entonces*

$$\sum_{k=0}^{F-1} \frac{I!}{(I-k)!} \leq \frac{I!}{(I-F)!} \tag{17}$$

Demostración. Inducción en F:

- **F = 1**:

$$1 = \frac{I!}{(I-0)!} = \sum_{k=0}^{0} \frac{I!}{(I-k)!} \leq \frac{I!}{(I-1)!} = I \tag{18}$$

  Por hipótesis, esto es cierto.

- **F ⇒ F + 1**:

  Quiero ver que: $\sum_{k=0}^{F} \frac{I!}{(I-k)!} \leq \frac{I!}{(I-F-1)!}$ Analizo la primera parte:

$$\sum_{k=0}^{F} \frac{I!}{(I-k)!} = \sum_{k=0}^{F-1} \frac{I!}{(I-k)!} + \frac{I!}{(I-F)!} \overset{H.I.}{\leq} 2 \times \frac{I!}{(I-F)!} = \tag{19}$$

$$= \frac{2I!}{(I-F)(I-F-1)!} \tag{20}$$

  Ahora comparamos:

$$\sum_{k=0}^{F} \frac{I!}{(I-k)!} \leq \frac{2}{(I-F)} \times \frac{I!}{(I-F-1)!} \leq \frac{I!}{(I-F-1)!} \tag{21}$$

$$\frac{2}{(I-F)} \leq 1 \tag{22}$$

$$2 \leq I - F \tag{23}$$

$$F \leq I - 2 \tag{24}$$

  Por hipótesis, esto es cierto.

□

Lema 4. $\frac{n^2}{2} \leq 2n^2 - 2n - 2 \quad (\forall n \geq 2)$

Demostración.

$$\frac{n^2}{2} \leq 2n^2 - 2n - 2 \iff 0 \leq 3n^2 - 4n - 4 \iff n \geq 2 \tag{25}$$

□

Vistas estas propiedades sobre el árbol, pasemos a demostrar el teorema.

Teorema 1. *El algoritmo es* $\mathcal{O}((2n^2 - 2n)! \times n^2)$ *en tiempo.*

Demostración. Nuestro algoritmo hace $\mathcal{O}(n^2)$ trabajo en cada llamada que no son caso base (i.e. posición terminal), pues usa dos bucles anidados de n pasos para calcular $j(t, x)$ (línea 8). Asimismo, cuando la llamada es un caso base, se usa $\mathcal{O}(n^2)$ trabajo para saber cuanto vale el tablero al que se llegó, para el jugador que somos. La conclusión es que el algoritmo hace $\mathcal{O}(n^2)$ trabajo en cada nodo del árbol. La complejidad temporal del algoritmo es $\mathcal{O}(T \times n^2)$, con T el tamaño del árbol.

Acotemos la cantidad de nodos del árbol. Para ello utilizamos

$$T = \sum_{k=0}^{n^2/2} m_k \leq \sum_{k=0}^{n^2/2} m_k \leq \sum_{k=0}^{n^2/2} \frac{I!}{(I-k)!} \leq 2 \times \frac{I!}{(I-\frac{n^2}{2})!} \in \mathcal{O}((2n^2 - 2n)!) \tag{26}$$

Finalmente la complejidad del algoritmo es $\mathcal{O}((2n^2 - 2n)! \times n^2)$ □

3.3. Implementación: $\alpha - \beta$ pruning

Sobre este minimax, usamos una optimización deterministica llamada $\alpha - \beta$ *pruning*. Esta poda toma en cuenta que, durante el algoritmo minimax, hay ramas del árbol de juego que no necesitamos en realidad explorar, porque ya sabemos que no van a afectar a la salida del algoritmo. Por ejemplo, imaginemos que estamos maximizando nodos. Si ya tengo un nodo hijo que tiene valor de juego 6, y estoy viendo otro hijo P que minimiza, que tiene un hijo Q de valor 2, puedo dejar de ver a P, pues - como él está minimizando - su valor nunca será mayor que 2, dado que tiene a Q de hijo. Al estar yo maximizando, y sabiendo que ya tengo un hijo

de valor 6, no me interesa ver una rama cuyo valor es a lo sumo 2; por eso dejo de ver a P apenas encuentro el valor de Q. El caso en el que yo estoy minimizando es análogo.

Esto lo implementamos manteniendo contadores de mínimo/máximo cada vez que recorremos los hijos de un nodo. Si estamos maximizando (porque estamos jugando jugadas del jugador I), entonces el contador es un contador de máximos, mientras que, en caso contrario, es un contador de mínimos. Con esto, podemos hacer una restricción a la función minimax: le pasamos 2 parametros, α Y β, y exigimos que el resultado devuelto por la función pertenezca al intervalo (α, β). Estas actúan como las cotas superiores e inferiores anteriormente discutidas; cuando estamos maximizando, pasamos una cota inferior (β) a la recursión, cuando estamos minimizando, pasamos una cota superior (α) a la recursión.

Esta técnica depende fuertemente de en qué orden yo visite los nodos del árbol de juego. En particular, si los visito en orden creciente de puntuación al estar maximizando, no voy a poder podar nada, mientras que si los visito en orden inverso, voy a podar la mayoría del árbol. Es análogo el caso en el que yo estoy minimizando.

Entonces, lo que vemos es que puede no reducir el comportamiento asintótico si no lo aprovecho bien, pero que, en la práctica, generalmente se espera que dé un buen resultado, porque es poco probable analizar los nodos en exactamente orden creciente o decreciente de puntaje.

3.4. Análisis empírico

Como fue explicado anteriormente, el algoritmo *minimax* genera el árbol entero de todos los posibles partidos que pueden suceder dado un tablero y un jugador inicial. El algoritmo *alpha-beta prunning* es una optimización sobre el mismo que se encarga de recortar subárboles que no tiene sentido explorar. A continuación nos dedicamos a investigar el tamaño empírico del árbol *minimax* y como este se reduce mediante la *poda* $\alpha - \beta$.

Por lo que fue dicho anteriormente, la complejidad del algoritmo depende principalmente del tamaño del árbol de jugadas. Este está directamente relacionado con el tamaño del tablero. En las siguientes tablas y gráficos vemos como crece el árbol en relación al tablero.

	Total Nodos		
Nivel	**2x2**	**3x3**	**4x4**
0	1	1	1
1	4	16	24
2	4	88	448
3	-	336	6264
4	-	432	62232
5	-	-	406560
6	-	-	1548000
7	-	-	2782080
8	-	-	1451520
Total	9	869	6257129

Cuadro 1: Tabla comparativa del tamaño del árbol según tablero

A partir de estos números puede verse la velocidad de crecimiento del árbol. Pasar de 3 filas a 4 implica multiplicar por 7200 la cantidad de nodos del árbol. Esto impacta directamente en el tiempo que tarda el algoritmo en jugar una partida.

	Duración de la partida		
Jugador	**2x2**	**3x3**	**4x4**
Primero	0.009s	0.009s	4.059s
Segundo	0.008s	0.008s	0.243s

Cuadro 2: Tabla comparativa de tiempos según tablero y posición del juego

Esta última tabla nos muestra que el crecimiento del tablero también produce un aumento estrepitoso en el tiempo de ejecución. Para los valores de esa tabla el contrincante fue fixed, de forma que el tiempo tuviera la mayor relación posible con la corrida de nuestro algoritmo. Aquí se nota también la diferencia que produce ser el primer o el segundo jugador. Al ser el segundo jugador, cuando le toca jugar al algoritmo ya hay una pieza puesta en el tablero, lo que equivale a tener un nivel menos en el árbol. Esto hace que el algoritmo deba calcular solo uno de los subárboles de la raíz del original. No fue posible hacer mediciones para tableros de tamaño mayor dado el tiempo necesario para hacer los cómputos. Para un tablero de 5x5 casillas, el algoritmo minimax

se dejó correr por 6 horas, luego de las cuales fue detenido a la fuerza. Estas 6 horas no fueron suficientes para que se empezase a jugar el partido. Para ese momento los contadores `int` de nodos por nivel en los niveles 10 y 12 habían dado overflow.

El tablero tiene n filas y columnas, por lo tanto n^2 casillas y cada movida ocupa dos de las casillas. Luego, puede haber lo sumo $\frac{n^2}{2}$ movidas. Esto significa que el árbol tiene a lo sumo esa cantidad de niveles. Siendo que este número es muchísimo menor que el número total de nodos, el árbol debe extenderse horizontalmente. Para ver esto medimos la cantidad de hijos que tienen los nodos en un determinado nivel del árbol, según el tamaño del tablero. En particular medimos el máximo, el mínimo y el promedio de estos valores. Los resultados se presentan en las siguientes tablas.

	Cant.	Hijos por nivel		
Nivel	**Nodos**	**Max**	**Min**	**Prom**
0	1	4	4	4
1	4	1	1	1
2	4	0	0	0

Cuadro 3: Datos del árbol de minimax para 2x2

	Cant.	Hijos por nivel		
Nivel	**Nodos**	**Max**	**Min**	**Prom**
0	1	12	12	12
1	12	8	6	7.333
2	88	5	3	3.8181
3	336	2	0	1.28571
4	432	0	0	0

Cuadro 4: Datos del árbol de minimax para 3x3

	Cant.	Hijos por nivel		
Nivel	**Nodos**	**Max**	**Min**	**Prom**
0	1	24	24	24
1	24	20	17	18.6667
2	448	17	12	13.9821
3	6264	13	8	9.93487
4	62232	10	4	6.53297
5	406560	7	1	3.80756
6	1548000	4	0	1.79721
7	2782080	1	0	0.521739
8	1451520	0	0	0

Cuadro 5: Datos del árbol de minimax para 4x4

En estas tablas se presenta la información sobre la amplitud de los árboles. Vemos que la cantidad de nodos por nivel va aumentando, incluso cuando disminuye la cantidad de hijos por nodo de ese nivel. Es en realidad por esto mismo que, para 4x4, el último nivel tiene menos nodos que el anterior, ya que empiezan a aparecer nodos terminales que no producen nuevos nodos y el resto produce a lo sumo 1. La distancia entre el máximo y el mínimo no varía mucho con el aumento de nivel.

Más tarde optimizamos el proceso de generación y recorrida del árbol de juego del minimax utilizando la denominada *poda alpha-beta*. Lo que hace esta poda es, estando en un nodo, reconocer la situaciación donde el valor de este está definido por los subárboles ya vistos, siendo entonces que ningún otro subárbol sin ver puede modificarlo. El resultado es que algunos subárboles de juego son directamente no tomados en cuenta. Cuales y cuanto subárboles son omitidos depende del juego jugado y del órden en que se recorren las posibles movidas dado un estado de juego. Esto hace sumamente dificil establecer la verdadera calidad de la poda. Las siguientes tablas muestran mediciones comparativas en el tamaño del árbol al usar y no usar la *poda alpha-beta*.

Tamaño	**minimax**	**alpha-beta**
2x2	9	9
3x3	869	306
4x4	6257129	365249

Cuadro 6: Nodos totales minimax vs alpha-beta por tamaño de tablero

El primer cuadro compara la cantidad total de nodos. Vemos que la *poda alpha-beta* disminuye el tamaño del árbol notoriamente para tableros de 3x3 y 4x4. El segundo cuadro y su representación gráfica siguiente compara otras estadísticas entre los dos árboles para el caso 4x4. En particular, vemos que la *poda alpha-beta* no mejora la máxima cantidad de hijos en los nodos de cada nivel, pero si hace que el mínimo sea 1 a partir del nivel 1. Por otro lado, vemos que el promedio de hijos por nodos disminue aproximadamente a la mitad. La *poda alpha-beta*

	Minimax				Alpha-beta			
	Cant.	Hijos por nivel			Cant.	Hijos por nivel		
Nivel	**Nodos**	**Max**	**Min**	**Prom**	**Nodos**	**Max**	**Min**	**Prom**
0	1	24	24	24	1	24	24	24
1	24	20	17	18.6667	24	20	1	9.16667
2	448	17	12	13.9821	220	17	1	7.55455
3	6264	13	8	9.93487	1662	13	1	5.24128
4	62232	10	4	6.53297	8711	10	1	3.90793
5	406560	7	1	3.80756	34042	7	1	2.6935
6	1548000	4	0	1.79721	91692	4	0	1.59986
7	2782080	1	0	0.521739	146694	1	0	0.560371
8	1451520	0	0	0	82203	0	0	0

Cuadro 7: Comparación de estadísticas del árbol para 4x4

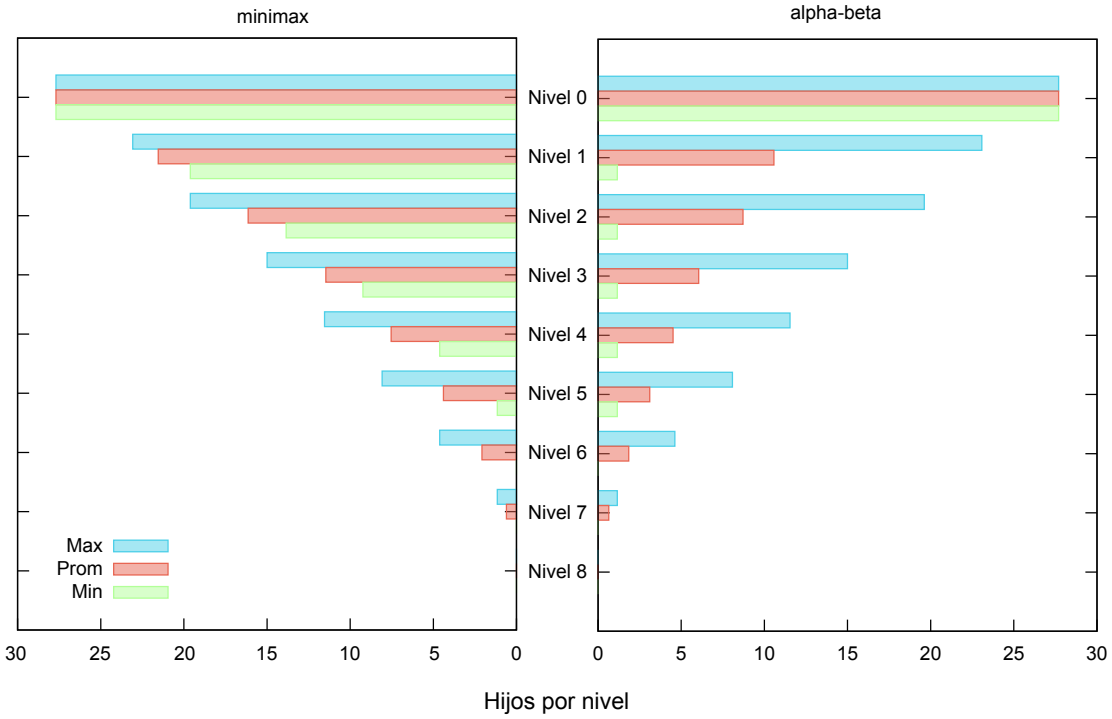

Figura 1: Gráfico comparativo de estadísticas para el árbol 4x4

no logra ningún efecto particular en el nivel 0. Esta diferencia en el tamaño del árbol hicieron que algoritmo *alpha-beta* lograra resolver el juego para un tablero 5x5 en 2 horas y 10 minutos.

4. Programas aproximados

4.1. Puntuador de tableros

Para crear el puntuador de tableros parametrizable, consideramos una serie de aspectos que hacen de un tablero dado, que sea favorable para determinado jugador. Estos aspectos los ideamos jugando a Ladrillos y viendo qué considerábamos al elegir una jugada por sobre otra, y tienen que ver principalmente con la posibilidad futura de expansión propia y enemiga que existe en cada tablero. Teniendo estas ideas, comenzamos a programar funciones que analizaran, con complejidad lineal, qué tan bueno es un determinado tablero en cierto aspecto y los parametrizamos. A continuación detallamos cada uno:

4.1.1. Puntaje

En principio, es claro que cuanto más puntaje propio provea un determinado tablero, mejor. Esto es lo que nos hace elegir poner una ficha adyacente a una de nuestras islas grandes en vez de dividirnos por el tablero, esta idea también tiene su contra pues dificilmente nuestra heurística decida separarse de su isla inicial (o de mayor tamaño)y seguramente avance muy lentamente por el tablero permitiendo así que otro jugador la encierre llevandola a una posición desfavorable del juego y haciendola perder. Esto se observa fácilmente jugando en un tablero lo suficientemente amplio. Programamos esto en la función `puntaje_lineal`, que tiene complejidad $\mathcal{O}(n^2)$. Además, esta función arma un mapa de a qué isla pertenece cada casillero y cuantas unidades poseé dicha isla, muy útil para los siguientes puntos.

4.1.2. Expansión

Algo importante es la capacidad de expansión que tienen las islas propias y las enemigas; intuitivamente es mejor que una isla avance en el tablero que cubrir con gran densidad una porción pequeña; porque de esta manera será más difícil dejar la isla encerrada y además habra más oportunidades para colocar fichas futuras. En la función `expansion(jugador x)` calculamos la cantidad de casilleros libres adyacentes a islas existentes del jugador x. Sin embargo, resulta más importante la posibilidad de crecimiento de una isla grande que el de una isla pequeña, por lo que para tener un resultado más significativo, lo que en realidad calculamos es la suma de, para cada isla de x, la cantidad de casilleros libres adyacentes a esa isla multiplicados por el tamaño de la isla. Como es lógico, se quiere maximizar la expansión propia y minimizar la enemiga; por lo tanto este valor se calcula para ambos jugadores: el propio suma mientras que el del otro resta.

4.1.3. Distancia a los ejes

Considerando los aspectos anteriores teníamos un jugador medianamente aceptable para cuando jugaba segundo, pero muy malo cuando empezaba. Esto sucedía porque al empezar todos los tableros posibles le daban lo mismo; entonces generalmente elegía alguna posición lateral, cuando naturalmente conviene comenzar jugando por el medio, Esto fue solucionado con otra función, pensando que probablemente siempre sería mejor jugar en el medio de un lugar vacío que en un costado. `dist` calcula la suma de la distancia mínima a una pared para cada casillero ocupado con una ficha propia. Minimizando este valor, obtenemos jugadores que prefieren jugar en el medio.

4.1.4. Encerrados

También nos dimos cuenta de que resulta muy importante encerrar al oponente, de manera que no pueda seguir un camino dado hacia ningún lado. Para esto implementamos la función `encerrados`, que recorre cada casillero y cuenta todos los que están ocupados por el enemigo, están encerrados (no tienen ningún casillero adyancente libre) y tienen algún vecino ocupado por mí, para asegurarme que sean casilleros que yo encerré y no parte interior de una isla. Esta función tiene similitud con la que calcula la expansión ajena, pero aquí se suman las fichas que de alguna manera yo encerré, entonces querremos maximizar este valor.

4.1.5. División del tablero

Notamos que es muy conveniente dividir el tablero con líneas que toquen la pared, porque así se limita la expansión del otro jugador a sólo una parte del tablero. Nuestros jugadores que consideraban las heurísticas anteriores ya tendían a expandirse hacia los costados, por `Expansión`; sin embargo notamos que muchas veces no ponían la última ficha que cerraría la línea con la pared. Para considerar esto, ya que suelen ser muy buenas jugadas, creamos la función `divisiones`, que mira los $\sqrt{n}$ casilleros del medio de los cuatro costados y se fija si

alguno está en la misma isla que algún casillero del medio (consideramos el medio como los $\sqrt{n} \times \sqrt{n}$ casilleros del medio). Si es así, suma 1 por cada costado que lo cumpla. Esta función devuelve números entre el 0 y el 4; como todas las demás son de orden n multiplicamos el resultado obtenido por esto, para que no perdiera relevancia al final.

Versión 2: Un cambio hecho en una segunda versión de esta heurística fue no considerar los $\sqrt{(}n)$ casilleros del medio de cada pared, sino todos, dividiéndola en dos mitades, y sumando uno por cada una de ellas si tienen algún nodo que comparta casillero con los del medio (que siguen siendo $\sqrt{n} \times \sqrt{n}$).

4.1.6. Futuras Heurísticas

Aquellas heurísticas que pensamos serian interesantes pero por diversos motivos no implementamos:

- `Espejo:` una heurística que copie algunos movimientos del rival, siempre y cuando sea posible, y en condiciones que defina como favorables por alguna heurística auxiliar como las que ya nombramos decida dejar de imitar. Pensamos que sería una buena alternativa para evitar el problema de atarse a una determinada área del tablero pues tiene el mismo beneficio/perdida que para mi rival tiene su jugada, entonces no puede ser mala.
- `Cerrarse:` durante los partidos del jugador heurístico es notable como por momentos podría limitar la cantidad máxima de fichas del rival en una determinada área y no lo hace. Una posible implementacion para a la misma saldría de tomar algún ejemplar perteneciente a la isla más significativa del rival actualmente y hacer un DFS ó BFS marcando como ejes ya visitados inicialmente todas las posiciones de mis fichas, y en el mismo contar aquellos espacios vacios o ya ocupados. Luego tableros con mayores espacios vacíos serán mejor puntuados.

4.2. Jugadores Golosos

Para generar nuestro jugador goloso utilizamos como base lo desarrollado en el punto 1, asignándole al alphabeta prunning profundidad de un solo nivel. De esta manera, sólo abre el árbol de jugadas una vez en cada posición del juego, viendo en cada turno todas las jugadas posibles para la movida próxima. El jugador goloso luego evalúa todos estos tableros utilizando el puntuador parametrizable desarrollado, y decide que tablero jugar de acuerdo al puntaje obtenido.

La manera de llamar al jugador goloso parametrizado es:

..jugadores_ours/alphabeta –alphabeta –height 1 –funcion 1 –params a b c d e (27)

Donde después de `params` van los parámetros que se quieren utilizar (en general van en el siguiente orden: Puntaje, Distancia al centro, Expansión propia, Expansión enemiga, Divisiones).

4.2.1. Eliminación de "Encerrados"

Al parametrizar y probar las diferentes heurísticas que habíamos hecho (jugando entre ellas, contra la heurísticas existentes, y contra nosotros) nos dimos cuenta de que "Encerrados" no estaba aportando nada al buen juego. No obteníamos el efecto deseado, y en general poniéndole valores considerables los jugadores comenzaban a jugar cada vez peor. Atribuimos esto a que mide algo demasiado específico; en general ni siquiera es una referencia clara de qué tan encerrado está el enemigo, ya que esto depende de muchas cosas además de lo que la función observaba. Decidimos eliminarlo para focalizar la búsqueda de mejores parámetros en las heurísticas que más servían.

4.3. Complejidad

Primero demostraremos que cada una de las funciones explicadas anteriormente para evaluar distintos aspectos del tablero tienen complejidad lineal en los casilleros del tablero, como pide el punto a.

- `puntaje_lineal:` La función la podemos analizar por partes separadas: primero las llamadas a `dfs`, y `puntaje_lineal` en sí. Claramente los bucles anidados de `puntaje_lineal` son $\mathcal{O}(n^2)$ trabajo (el push_back del vector es $\mathcal{O}(1)$ amortizado, con lo cual es $\mathcal{O}(n^2)$ en el total de las veces, si se llama $\mathcal{O}(n^2)$ veces). En vez de analizar cada llamada de `dfs` y multiplicar, podemos analizar la función por si sola. Cualquier secuencia de llamadas a `dfs` va a ser $\mathcal{O}(n^2)$, pues lo que hace la función es hacer un depth-first-search en el tablero, marcando a los casilleros como visitados cuando los vé. Por este motivo, nunca visita una

casilla dos veces. Como lo estamos llamando n^2 veces, va a intentar empezar un depth-first-search desde cada casilla. Entonces, va a hacer $\mathcal{O}(n^2)$ trabajo en el total de las llamadas.
Sumando (porque analizamos las dos cosas por separado) esto a los $\mathcal{O}(n^2)$ de `puntaje_lineal`, tenemos que la complejidad del puntuador es $\mathcal{O}(n^2)$ trabajo.

- `expansión:` esta función tiene dos `for`, uno dentro de otro, para iterar las filas y las columnas, por lo que ejecuta una vez por casillero (n^2 veces) lo que se encuentra dentro de estos `for`. Esto último se reduce a una cantidad constante de comparaciones y asignaciones para ver si las casillas de abajo, arriba a la derecha y a la izquierda de la actual están libres. Por lo tanto, el interior de los `for` es $\mathcal{O}(1)$; y en total la función en $\mathcal{O}(n^2)$
- `dist:` esta función también tiene los mismos dos `for` que la anterior. Aquí, para cada casillero también se hace una comparación y se sacan tres mínimos entre pares. Todo esto último es constante; por lo tanto esta función también es lineal en n^2
- `encerrados:` Si bien al final no la usamos, la analizaremos también. Esta función, al igual que las anteriores, recorre todos los casilleros del tablero utilizando dos `for`. Para cada casillero hace una comparación y llama a `estaEncerradayMeToca`, que al hacer solamente una cantidad constante de comparaciones y asignaciones es $\mathcal{O}(1)$. Por lo tanto la función tiene complejidad lineal en la cantidad de casilleros, o sea $\mathcal{O}(n^2)$
- `divisiones:` en esta función vemos, para cada costado, si tiene un casillero que comparta isla con alguno del medio. Sólo verificamos los $\sqrt{n}$ elementos del medio del costado. Por cada uno de estos, miramos si pertenece a la misma isla que alguno del medio. "El medio" corresponde a $\sqrt{n} \times \sqrt{n} = n$ casilleros. Ver si pertenecen a la misma isla es constante (así como las verificaciones de a quién pertenece el casillero), y todo esto se hace 4 veces (una por pared). Por lo tanto, finalmente tenemos que se hace $4 \times \sqrt{n}$ veces algo que es $\mathcal{O}(n)$, por lo tanto la complejidad es $\mathcal{O}(\sqrt{n} \times n)$. En la segunda versión vemos n en vez de $\sqrt{n}$ nodos por cada pared, por lo tanto la complejidad es $\mathcal{O}(4 \times n^2)$, que sigue siendo $\mathcal{O}(n^2)$.

Sabemos que los jugadores siempre juegan la misma cantidad de veces en un partido con determinado tablero, pues juegan hasta completar el tablero, lo cual sucede cuando se han puesto $\frac{n^2}{2}$ fichas, ya que cada una ocupa dos casilleros. Puede ser que en algunas partidas se ubiquen menos fichas porque se han dejado casilleros encerrados de a uno, pero esto sirve como cota superior. Cada jugador pone la mitad de estas fichas ya que juegan alternados, por lo tanto, un jugador se enfrenta a la tarea de elegir una jugada $n^2/4$ veces durante un partido en un tablero de $n \times n$.

Lo que nos queda calcular es qué sucede en cada "decisión de jugada". Sabemos que se consideran todos los tableros posibles a partir del dado, y para cada uno de ellos se calculan las funciones lineales de arriba. Por lo tanto, nos interesa cuántos tableros posibles hay en cada turno. Como la forma de generar tableros a partir del existente es poniendo una ficha, esta cantidad es igual a las fichas que se pueden poner. Al principio, se deben considerar todas las jugadas existentes, que son $(n-1) \times n \times 2$ (posibilidades en una fila por cantidad de filas, y lo mismo para una columna). Sin embargo, a medida que transcurren las jugadas cada vez es menor la cantidad de estas "jugadas posibles" que son realmente válidas, ya que cada vez más empiezan a estar ocupadas por fichas ya existentes. Por lo tanto es cada vez menor la cantidad de tableros para los que efectivamente se calculan las funciones lineares. Es interesante la pregunta de cuántas jugadas se inhabilitan con cada ficha que se pone. En general son varias, pero sabemos que siempre habrá por lo menos una jugada que ya no se puede hacer, por lo tanto tomaremos como cota que en cada turno hay que considerar dos tableros menos que en el anterior (mi ficha y la del adversario). Como la cantidad de turnos en todo el partido es $\mathcal{O}(n^2/2)$, podemos escribir la cantidad de partidos para los cuales hacemos la evaluación como:

$$\sum_{i=0}^{\frac{n^2}{2}} n^2 - 2i = \frac{n^2}{2} \times n^2 - \sum_{i=0}^{\frac{n^2}{2}} 2i = \frac{n^4}{2} - 2 \times \frac{\frac{n^2}{2} * \frac{n^2-1}{2}}{2} = \frac{n^4}{2} - \frac{n^4}{4} + \frac{n^2}{2} = \frac{n^4}{4} + \frac{n^2}{2} \tag{28}$$

Lo cual sigue siendo $\mathcal{O}(n^4)$. La complejidad total corresponde a los tableros evaluados por el costo de evaluarlos (no consideramos las visitas a tableros que finalmente no evaluaremos por imposibles, ya que estamos tomando como cota superior lo que sería evaluar todos los tableros en cada jugada) Por lo tanto, obtenemos que la complejidad total es $\mathcal{O}(n^4 \times n^2)$, o sea $\mathcal{O}(n^6)$

4.4. Búsqueda de parámetros y comparaciones

La búsqueda de parámetros para generar mejores jugadores puede ser considerado un problema de optimización global. En estos problemas se define una función objetivo $f : \mathbb{E} \to \mathbb{R}$ con $\mathbb{E}$ el espacio de búsqueda. Aquí,

$\mathbb{E} = \mathbb{R}^p$, vector que representa una parametrización particular de los jugadores. Luego lo que buscamos es:

$$\max_{x \in \mathbb{E}} \{f(x)\}$$

En nuestro caso, f es la competitividad del jugador resultado de los parámetros en x. Podemos considerar que cada x es un jugador distinto. Sin embargo, esta función no es analizable por los métodos matemáticos clásicos. Debido a esto, fue necesario tomar otro enfoque para encontrar los valores que maximizan la función. En particular, utilizamos distintos métodos para explorar de manera inteligente el espacio de búsqueda, así como distintas formas de evaluar la competitividad de un x particular.

Para buscar los mejores parámetros creamos dos programas que hacen competir de dos maneras a los jugadores diferentes. El primer programa está basado en algoritmos evolutivo. Obtiene los parámetros más adecuados a partir de un conjunto inicial, generando nuevos parámetros a partir de los mejores y eliminando los peores. El otro es un torneo suizo en el cual juegan todas las combinaciones posibles de jugadores en un cierto intervalo numérico para los parámetros. Nuestra decisión fue implementar las dos modalidades y utilizar ambas de dos maneras diferentes:

- Jugando entre ellas. Siempre se enfrentan dos jugadores desarrollados por nosotros, con parámetros distintos.
- Jugando contra `heurb`. De los jugadores provistos por la cátedra, este nos pareció el más difícil de vencer, por lo tanto lo elegimos como referencia para que nuestros jugadores se enfrentaran a él y se midieran entre sí según el resultado obtenido.

A continuación detallaremos cada modalidad de competencia

4.4.1. Torneo Suizo

El torneo más simple que se puede implementar es el *torneo round-robin*, en el que todos los jugadores se enfrentan con todos los demás. Sin embargo, este tipo de competencia es muy larga y costosa, por lo tanto decidimos hacer competir a nuestros jugadores mediante un *torneo suizo*. Aquí se elige una determinada cantidad de fechas y en cada una se desarrollan partidos, donde en cada partido compiten dos jugadores con puntajes aledaños, y todo jugador participa en exactamente un partido por fecha. Al iniciar el torneo se definen cuales son los jugadores originales y cuantas fechas totales se realizarán. De esta forma, no todo par de jugadores tiene la oportunidad de enfrentarse, pero si se asegura que ningún par se enfrente dos veces.

Para el caso particular del trabajo práctico, formamos el conjunto de jugadores originales definiendo un intervalo posible para los parámetros de la heurística y calculando todas las combinaciones existentes para estos valores, siempre enteros. Pensando al jugador como un vector de enteros, donde cada posición representa un parámetro de la heurística, tendríamos que:

$$\text{Jugadores Iniciales } = I^p \text{ con } I = [0; A] \subset \mathbb{N}_0 \text{ y } p = \text{cantidad de parámetros}$$

Después los ordenamos aleatoriamente, para que el orden inicial no influya en los resultados, y los hacemos competir de a pares (el primero contra el segundo, el tercero contra el cuarto, etc). Cuando terminan de enfrentarse, sumamos al puntaje de cada uno el obtenido en este juego. Una vez que terminaron de jugar todos, los ordenamos según el puntaje y juegan de vuelta de la misma manera, procurando jugar con el jugador de puntaje más cercano con él que no hayan jugado previamente. Esto se repite tantas veces como fechas fueron dispuestas inicialmente. Al terminar estas, se obtiene una lista de jugadores, ordenados por la suma de sus puntuaciones. Cuantas más fechas haya, más fino será este resultado.

Además consideramos hacer otro torneo del mismo tipo pero esta vez jugando frente a `heurb` esto quiere decir que hicimos que cada jugador, en vez de su vecino, jugara con la heurística sumando a su puntaje el obtenido en el partido. Fuimos alternando en cada fecha si jugaban primeros o segundos e hicimos esta pasada varias veces porque `heurb` no juega siempre de la misma manera. Para terminar, los ordenamos por puntaje final.

4.4.2. Evolutivo

Para poder hacer una búsqueda que no estuviera acotada a un subconjunto de los valores de $\mathbb{E}$, implementamos una metaheurística evolutiva. Esta metahurística se inspira en el fenómeno de la selección natural que se presencia en la naturaleza. La razón de esta elección es que la optimización que estamos haciendo, relacionada con la *competitividad* de jugadores, puede ser fácilmente pensada de esta forma. Es así que simulamos el proceso de evolución de especies de forma de encontrar aquellos espécimenes más apropiados:

1. El algoritmo comienza creando una población inicial $P_I = \{0,1\}^p$.

2. A continuación, ejecuta una serie de partidos entre los jugadores. Cada jugador pasa a tener como puntaje la suma de los puntos obtenido en las partidas jugadas. Esto nos da una evaluación de los integrantes de la población.

3. El siguiente paso es la reproducción: se elije un subconjunto de la población actual de manera aleatoria, pero dándole mayor oportunidad de ser elegidos a los jugadores de mayor puntaje. De este subconjunto se elijen pares aleatorios para que se reproduzcan, originando un nuevo jugador, cuyos parámetros son el promedio de los de sus padres. Estos valores son además modificados de manera aleatoria, para simular las mutaciones que suceden en la naturaleza. Los nuevos jugadores son considerados de la generación siguiente a la de sus padres.

4. La nueva generación y el subconjunto seleccionado constituyen una nueva población del mismo tamaño que la anterior. Finalmente se vuelve al paso 2 y se repite todo indefinidamente.

En el desarrollo de este algoritmo nos encargamos de los siguientes problemas:

- **Convergencia prematura:** un problema común en las metaheurísicas de optimización es la *convergencia prematura*, que sucede cuando la muestra del espacio de búsqueda se encuentra en un máximo local y está imposibilitado de salir en búsqueda de nuevos máximos. Para mitigar este problema es que le dimos a los jugadores con menores puntajes la posibilidad de persistir de una generación a otra y aparearse para producir nuevos jugadores con algunas de sus características. De esta forma permitimos que se siga un camino que sea peor comparado con los máximos obtenidos, pero que pueda llevarnos a un máximo global. Otra forma de evitar la convergencia prematura fue incluir una amplia posibilidad de mutaciones que pueda originar jugadores afuera del área del máximo local.

- **Ruido:** En este problema, la comparación de los jugadores al competir contra heurb es inexacta y produce ruido, ya que heurb hace las elecciones de las jugadas con cierto elemento de aleatoriedad. El efecto de esto es que un jugador puede no ser tan bueno y que en alguna generación particular la aleatoriedad en heurb le permita ganarle con muchos puntos. A la inversa, un jugador que en general es bueno puede terminar con un resultado muy negativo si al competir contra heurb este último tuvo la fortuna de hacer varias movidas óptimas. A nosotros nos preocupó más que nada el último caso, ya que en distintas corridas vimos como algunos jugadores que tenían buena historia desaparecían abruptamente. En respuesta a esto se agregó al algoritmo un *salón de la fama*, que mantendría el historial de los mejores 10 jugadores de la historia. La posibilidad de subsistencia de malos espécimenes en una generación particular también ayuda a mitigar este problema

El algoritmo se ajustó de forma aproximada modificando los parámetros que definen las distintas partes del comportamiento de la población (cantidad de partidos, porcentaje remanente de la generación anterior, probabilidad de mutaciones, etc...).

De esta metaheurística surgieron dos búsquedas de jugadores, dependiendo si estos se evaluaban compitiendo entre si o contra heurb.

4.5. Obtención de los mejores jugadores

Para obtener los mejores jugadores utilizamos todo lo desarrollado previamente para armar un gran torneo, inspirado en el Tennis profesional. Este torneo se conforma de cuatro torneos pequeños (*Grand slam*) y una liga final (*Masters Series*)[1]. Los 4 *Grand Slam* surgen del **torneo suizo** y el algoritmo **evolutivo** en sus dos versiones: competencia interna y contra heurb. De cada uno de estos algoritmos se obtiene la liga mayor de jugadores, siendo estos los 10 mejores de cada torneo. Finalmente los jugadores de la liga compiten entre sí para organizarse por su competitividad de juego en un **torneo round-robin**.

4.5.1. Torneo Suizo - Competencia Interna

A continuación presentamos tablas y gráficos con los resultados del **torneo suizo** con competencia interna. En este torneo el conjunto de jugadores fueron todas aquellas combinaciones de parámetros con estos enteros entre 0 y 5, lo que da un total de 7776 jugadores. Los datos fueron obtenidos luego de 6 fechas donde cada jugador compitió contra otro jugando tanto como primer jugador (local) como segundo (visitante). El **torneo suizo**

[1] http://en.wikipedia.org/wiki/Tennis

con competencia interna se corre mediante battles/torneo_inter.py. Este deja como resultado varios archivos de datos.

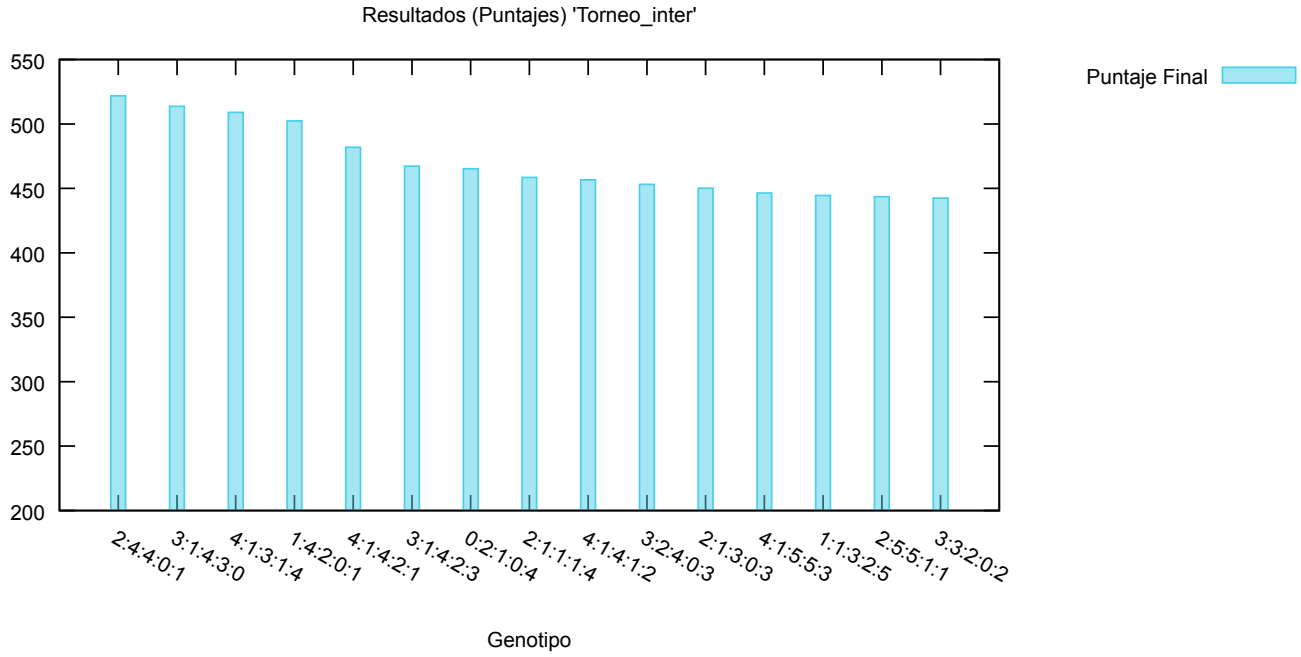

Figura 2: Puntaje final de los mejores jugadores 'torneo_inter'

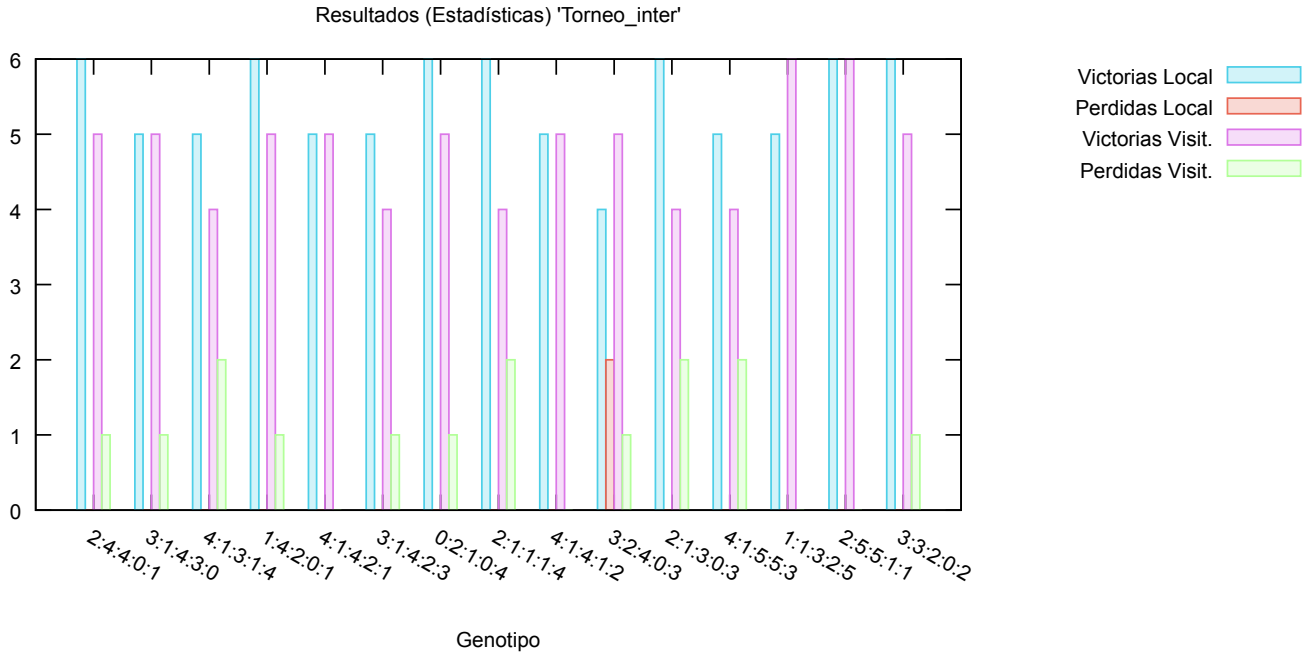

Figura 3: Estadísticas de los mejores jugadores 'torneo_inter'

Fecha	0	1	2	3	4	5
Tamaño	9	10	15	14	15	13
	3:1:5:0:2	2:1:4:4:2	2:1:4:4:2	2:1:5:0:4	2:4:4:0:1	2:4:4:0:1
	2:5:0:0:2	5:2:5:0:2	4:1:4:2:1	5:2:5:0:0	1:3:2:0:4	3:1:4:3:0
	5:1:0:0:2	0:3:5:4:1	3:2:5:0:0	2:4:4:0:1	4:1:5:5:3	4:1:3:1:4
	2:2:3:3:5	3:0:3:5:3	3:2:5:0:3	5:0:2:1:3	5:1:5:1:5	1:4:2:0:1
	1:4:5:5:0	3:5:5:5:3	3:4:3:0:1	2:1:3:0:3	1:3:2:0:5	4:1:4:2:1
	4:3:4:3:0	4:1:4:1:2	4:3:2:0:2	0:5:3:0:5	2:1:3:0:3	3:1:4:2:3
	1:2:5:5:4	5:3:5:3:0	4:1:2:4:3	1:3:2:0:5	2:1:4:5:2	0:2:1:0:4
Puestos	3:2:4:5:3	2:0:3:5:3	3:0:3:3:3	5:1:5:3:0	1:4:2:0:1	2:1:1:1:4
	3:2:4:1:1	3:4:2:0:1	2:2:0:0:3	2:5:0:0:2	3:1:4:0:5	4:1:4:1:2
	5:1:5:5:0	5:4:5:5:3	4:3:4:0:0	3:2:4:0:3	3:1:4:3:0	3:2:4:0:3
	0:3:3:4:1	3:3:5:0:1	2:1:5:3:5	2:1:4:5:2	2:1:5:0:4	2:1:3:0:3
	4:1:4:1:2	3:2:5:0:3	4:1:4:1:2	3:0:5:5:3	4:1:4:2:1	4:1:5:5:3
	4:2:3:5:3	1:3:3:1:0	2:1:4:2:1	3:0:3:5:2	4:3:4:0:5	1:1:3:2:5
	4:2:0:5:2	3:3:4:0:1	2:1:4:1:0	1:3:2:0:4	0:2:3:4:5	2:5:5:1:1
	0:1:1:4:4	3:4:3:2:1	4:1:5:2:2	2:5:4:0:4	2:0:1:2:5	3:3:2:0:2

Cuadro 8: Recorte de los mejores 15 en cada fecha (considerando historial) en 'torneo_inter'

El primer gráfico muestra el puntaje final de los mejores 15 jugadores al final del torneo. Los datos de este gráfico fueron obtenidos de ti_pout201011241342. Vemos que considerando los mejores 15 jugadores del torneo, sus puntajes varían pero en no más de 50 puntos. El promedio de puntos estuvo entre 43.5 y 36.9.

El segundo gráfico analiza la proporcion en victorias y derrotas de cada uno de los mejores jugadores. Se puede ver que los mejores jugadores no necesariamente son aquellos que más partidas ganaron porque por ejemplo los tres primeros genotipos tienen en su haber menos partidas ganadas que los últimos tres y aun así más puntos.

Podemos prestar atencion entonces al promedio de puntos obtenidos en las partidas ganadas por cada jugador y ver que el segundo 3:1:4:3:0 y el tercero 4:1:3:1:4 tienen una o dos victorias menos que el primero pero en cantidad de puntos están bastante cerca, por lo tanto su promedio de puntos por partidas ganadas resulta mayor.

Es interesante observar los parámetros que este torneo devuelve como los que más se deben puntuar. En general, los que tienen mayor valor en los jugadores ganadores son **Expansión propia** y, en menor medida **Puntaje**. Los demás van variando. Esto indica un juego que tiende a expandirse en las islas más grandes, tratando de conquistar todo el tablero.

Estas estadísticas surgen del archivo de datos ti_pout201011241342.

En la tabla se encuentra la evolución del TOP 15 por fecha. Además indica el tamaño del tablero en cada una. Estos datos se encuentran en ti_tout201011241342. En esta tabla puede notarse la variabilidad de los mejores 15 fecha a fecha. Por ejemplo, en la primer fecha, solo 4:1:4:1:2 y 5:1:0:0:2 aparecen después. 4:1:4:1:2 aparece en 4 de 6 fechas, consagrándose 4to. 5:1:0:0:2 vuelve recién en la 4ta, para luego volver a desaparecer. El ganador de la última fecha apareció en la 4ta en 3er lugar, subiendo al 1ero en la siguiente y manteniéndose ahí para el final. Del resto de los mejores 15 en la última fecha, solo 9 habían aparecido anteriormente en la tabla, y nunca más de 4 veces.

4.5.2. Torneo Suizo - Competencia Heurb

Ahora presentamos las tablas y gráficos con los resultados del **torneo suizo** compitiendo contra heurb. En este caso, en cada fecha los jugadores compitieron contra heurb, tanto de local como visitante (jugador 0 y 1, respectivamente). Al igual que en la competencia interna, el torneo constó de 7776 jugadores originados de la combinación exahustiva de parámetros y se jugaron 6 fechas. El **torneo suizo** con competencia contra heurb se corre mediante battles/torneo_heurb.py. Este deja como resultado varios archivos de datos.

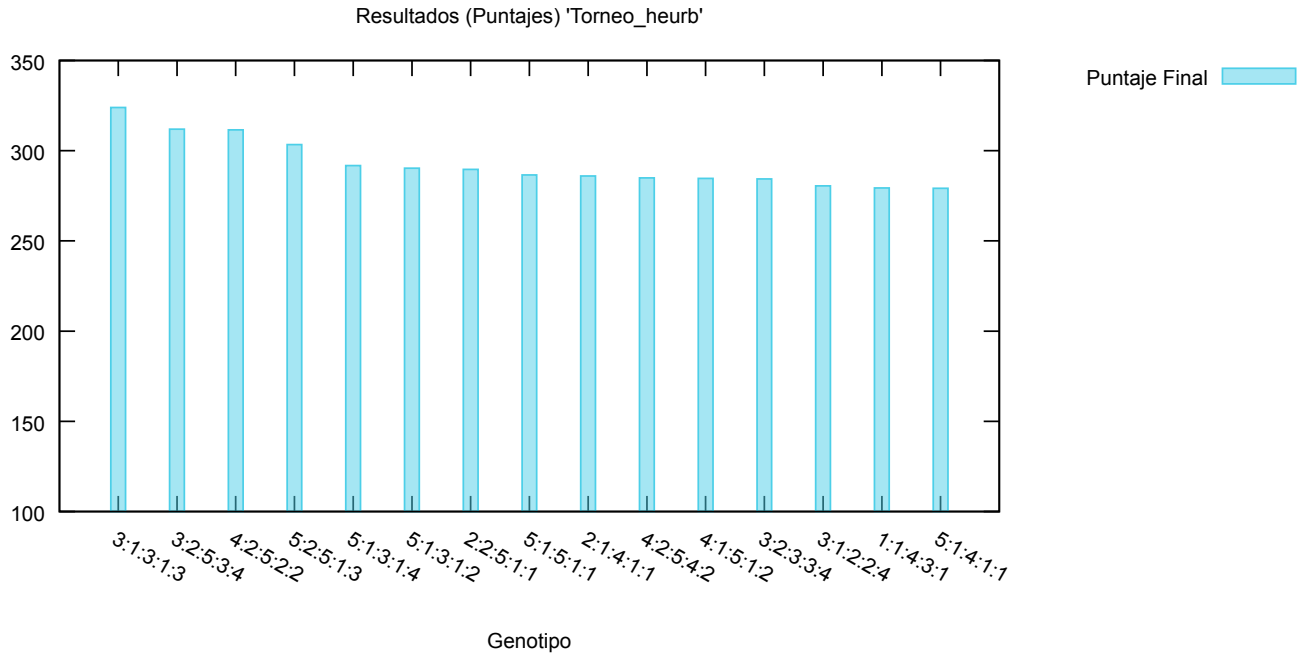

Figura 4: Puntaje final de los mejores jugadores 'torneo_heurb'

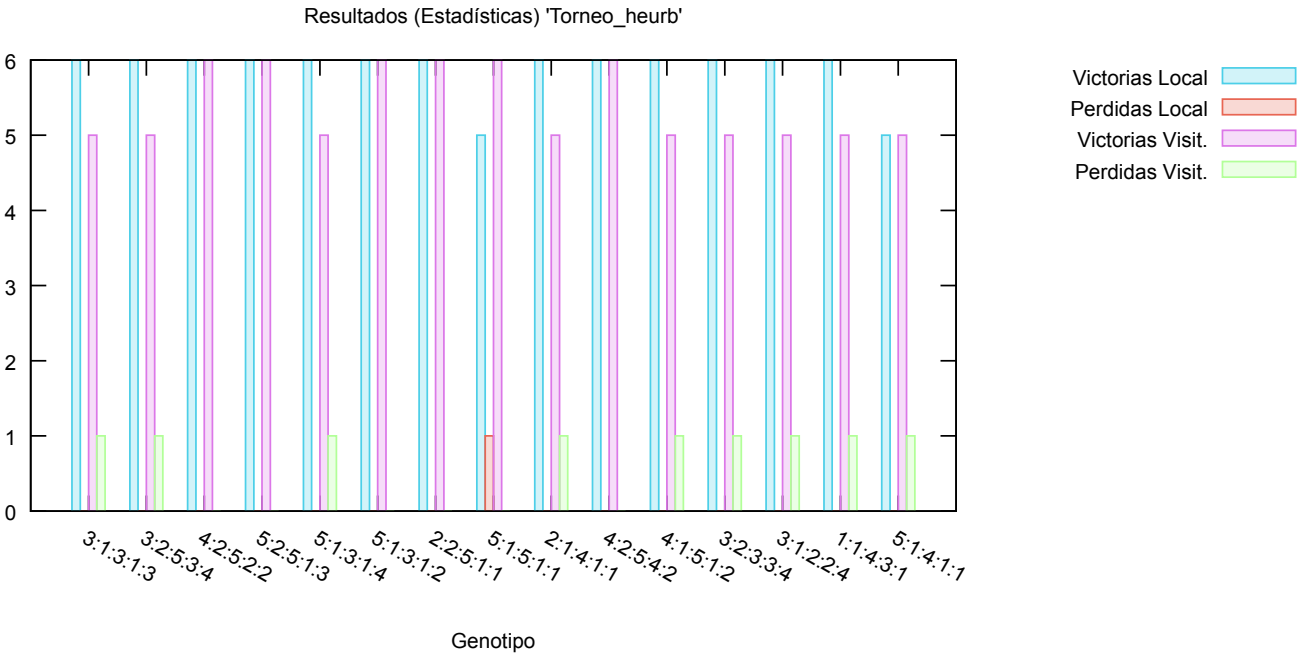

Figura 5: Estadísticas de los mejores jugadores 'torneo_heurb'

Fecha	0	1	2	3	4	5
Tamaño	9	9	7	13	8	10
	1:2:2:4:5	2:1:5:4:1	2:1:5:4:1	4:2:5:1:1	3:1:3:1:3	3:1:3:1:3
	2:2:5:4:3	1:4:2:1:2	3:2:5:3:4	2:1:4:1:1	5:1:4:1:0	3:2:5:3:4
	5:1:4:4:3	2:1:3:2:4	3:1:5:4:3	3:2:5:3:4	3:1:3:3:2	4:2:5:2:2
	5:1:2:2:5	5:1:3:1:4	5:1:4:1:0	3:2:5:1:1	5:2:5:2:4	5:2:5:1:3
	3:2:3:5:4	3:1:4:1:1	1:4:2:1:2	3:1:3:3:2	4:2:5:2:2	5:1:3:1:4
	0:2:3:4:5	5:2:5:2:4	4:1:3:2:1	5:1:4:1:0	3:2:5:1:1	5:1:3:1:2
	5:1:1:1:4	3:2:5:3:4	5:2:5:2:1	3:1:3:1:3	4:2:5:1:1	2:2:5:1:1
Puestos	5:2:5:1:0	4:4:4:1:3	3:2:5:3:2	5:1:4:2:3	2:1:4:1:1	5:1:5:1:1
	1:4:2:4:0	4:3:5:4:4	2:1:4:2:2	4:2:5:2:2	4:2:5:4:2	2:1:4:1:1
	4:1:4:0:4	2:1:5:2:1	2:1:4:2:1	5:2:5:2:4	4:1:5:1:2	4:2:5:4:2
	3:2:5:3:4	5:2:2:4:4	4:0:5:4:0	2:1:4:1:5	3:2:5:3:4	4:1:5:1:2
	3:2:3:3:5	0:3:1:0:1	3:2:5:3:1	1:2:5:3:1	5:2:5:1:3	3:2:3:3:4
	3:1:4:1:1	5:1:4:3:1	2:1:5:2:1	1:2:1:0:5	5:1:3:1:4	3:1:2:2:4
	3:1:1:2:4	1:2:5:3:3	1:2:3:1:2	4:1:5:1:1	3:1:2:1:2	1:1:4:3:1
	1:4:2:1:2	3:2:5:3:2	4:2:5:1:1	2:1:5:1:3	3:1:2:2:4	5:1:4:1:1

Cuadro 9: Recorte de los mejores 15 en cada fecha (considerando el historial de puntaje) en 'torneo_heurb'

Al igual que en la competencia interna, el primer gráfico muestra el puntaje final de los mejores 15 jugadores al final del torneo. Análogamente, fueron obtenidos de th_pout201011241125. Contra heurb los puntajes fueron menores en general y también se redujo la diferencia. El máximo y el mínimo fueron 323 y 273 respectivamente (redondeando los decimales). Los promedios máximo y mínimo de puntaje por partida fueron 26.9 y 23.3. Esto resulta razonable ya que en general los tableros por fecha durante el torneo contra heurb son más grandes que los del torneo local y a mayor tamaño de tablero la cantidad de puntos en juego es mayor. En este caso, el rango de puntos también es un poco mayor.

Observamos también que dentro de los mejores jugadores del torneo no hay ningún parámetro que quede sin utilizarse y en nueve de los quince el parámetro con índice 3 vale uno.

En este caso, al estar jugando contra otro jugador las cosas son un poco diferentes: principalmente cobra más importancia el último parámetro, correspondiente a **Divisiones**. Esto tiene sentido si observamos la manera de jugar de Heurb, donde la acción de dividirle el tablero es muy útil para arruinar la heurística con la que juega.

Estas estadísticas sobre los mejores jugadores fueron obtenidas de th_pout201011241125.

Nuevamente tenemos la evolución de los mejores 15, indicando el tamaño del tablero en cada fecha. Estos datos se encuentran en th_tout201011241125. En este caso también son 9 de los 15 mejores en la última fecha los que tuvieron presencia en fechas anteriores en la tabla. No obstante, hay una mayor cantidad de ellos que aparece 3 veces, y el sub-campeón de este torneo, 3:2:5:3:4, aparece en las 6 fechas, variando su posición en la tabla (11, 7, 2, 3, 11, 2, en casa fecha). Esto habla de la impredecibilidad de heurb, ya que incluso compitiendo contra el mismo oponente cada vez, los genotipos en las tablas se renuevan y un jugador puede bajar o subir abruptamente de una fecha a otra.

4.5.3. Evolutivo - Interno

Los siguientes son los gráficos y tablas obtenidos del algoritmo evolutivo compitiendo entre si. Para efectuar la simulación se utiliza el ejecutable battles/battle.py con parámetros "0 -i". El algoritmo se dejó correr una cantidad arbitraria de tiempo, para finalmente obtener los mejores 10 jugadores en tableros de tamaño 11.

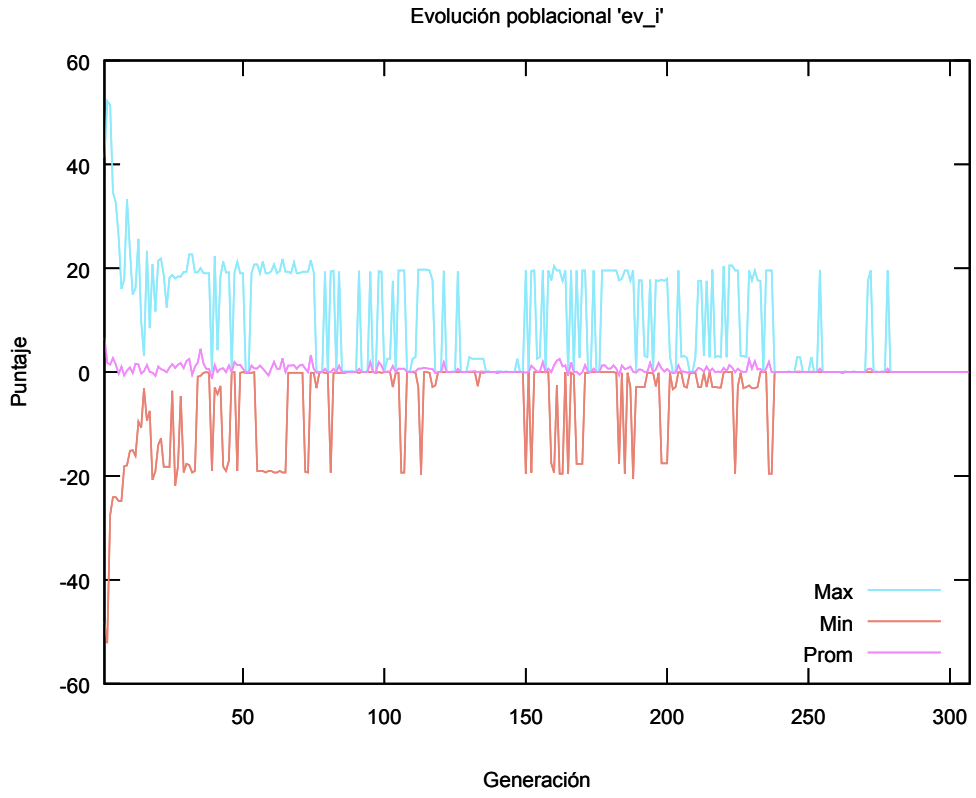

Figura 6: Puntaje de la población por generación en competencia evolutiva interna

Gen					Puntaje	Generación
1	**2**	**3**	**4**	**5**		
1.0	0.0	0.0	1.0	1.0	39.41	1
9.38	1.19	3.94	3.75	0.88	24.84	3
77.64	68.84	92.92	94.22	86.25	22.33	76
0.0	6.5	0.25	5.75	0.75	22.74	3
32.78	35.41	46.16	40.08	33.99	21.23	31
23.1	21.9	28.89	26.88	20.47	21.51	22
51.69	47.14	61.46	55.65	49.72	21.8	44
187.0	163.0	221.12	211.2	202.62	2.88	160
246.98	213.44	294.05	284.06	264.83	0.0	213
246.37	214.12	293.32	281.78	258.98	0.0	212

Cuadro 10: Tabla de campeones en la competencia evolutiva interna

El gráfico muestra la evolución del puntaje máximo, mínimo y el promedio dentro de cada generación. Estos valores se obtienen de la parte no comentada (#) de ev_i.dat.

Vemos que los números obtenidos son mucho más grandes que en los casos anteriores, en los que nosotros controlábamos la entrada. Sin embargo, sacando factor común se puede ver que en realidad la relación entre los parámetros no es tan distinta a la anterior. El parámetro obtenido como el ganador (1,0,0,1,1) es de la primera generación y muy diferente a los que obtenía el torneo suizo. Esto no deja de ser extraño, sin embargo pueden obtenerse casos como este por la manera en la que se generan y compiten los jugadores.

Otra cosa que resulta asombrosa es la presencia de jugadores con puntaje 0 entre los mejores. Viendo el segundo gráfico que muestra el desarrollo a través de las generaciones, notamos que es bastante común que el máximo y el mínimo se acerquen mucho cerca del 0. En particular pueden distinguirse períodos donde el máximo y mínimo se distinguen y otros donde no lo hacen. El gráfico muestra una simetría en el máximo y el mínimo sobre el 0 del eje x causa de que el juego sea de suma-cero. Esto es, todo lo que gana uno lo pierde otro. Luego, si un jugador obtiene una puntuación máxima p, otro habrá obtenido un puntaje $-p$, que será el mínimo.

Después de analizarlo, llegamos a la conclusión de que los jugadores enfrentados están muy balanceados entre sí por usar el mismo conjunto de heurísticas. Incluso podemos observar que los intervalos donde el máximo y el mínimo se separan son cada vez más breves. Esto puede entenderse como el hecho de que la población se está estabilizando y el nivel de todos los jugadores se emparejó.

En ev_i.full se encuentra el historial completo de la población (jugadores presentes en cada generación). Observando este archivo se ve como los valores de los parámetros van incrementándose en su conjunto. Una razón para este suceso es que el exista un camino neutral en el espacio de búsqueda y que dado el dinamismo que le imprimimos al sistema (por la fuerza de las mutaciones), la población deba estar en constante movimiento. De esta forma, a medida que se suceden las generaciones los parámetros se mueven, pero manteniéndose en relaciones cercanas, ya que en esa localidad estas parecerían óptimas.

La tabla muestra la información de los campeones de la competencia evolutiva. Esta información son las últimas 10 líneas de ev_i.dat.

4.5.4. Evolutivo - Heurb

Los siguientes son los gráficos y tablas obtenidos del algoritmo evolutivo compitiendo contra heurb. Para efectuar la simulación se utiliza el ejecutable battles/battle.py con parámetros 0 -h. El algoritmo se dejó correr una cantidad arbitraria de tiempo, para finalmente obtener los mejores 10 jugadores en tableros de tamaño 11.

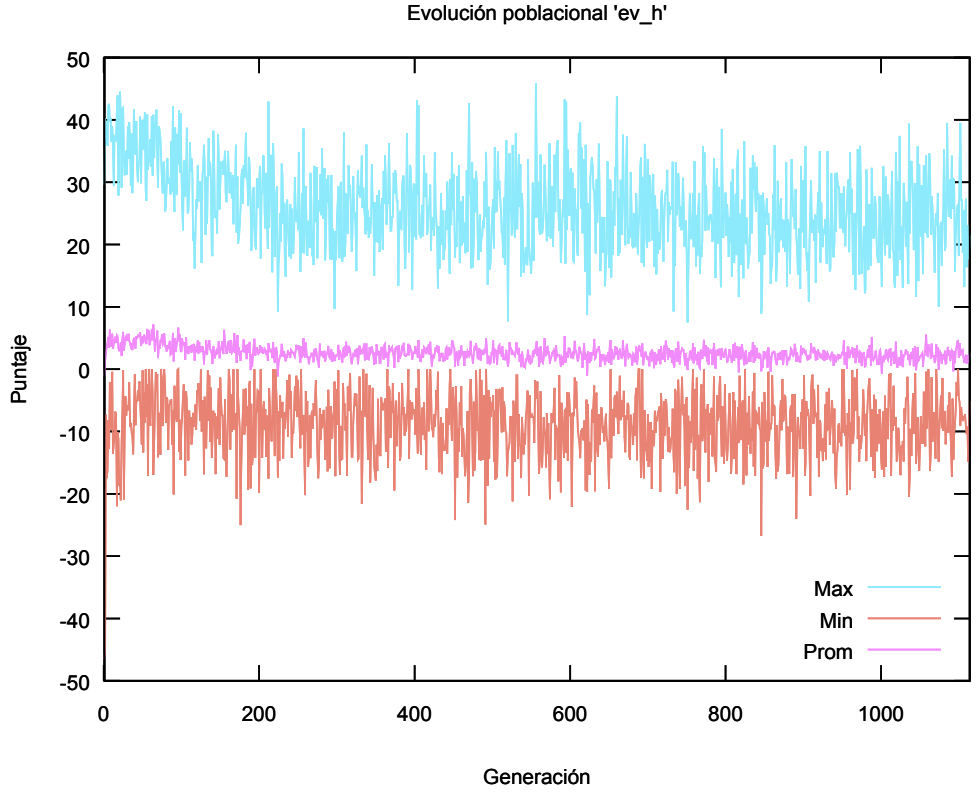

Figura 7: Puntaje de la población por generación en competencia evolutiva contra heurb

Gen					Puntaje	Generación
1	**2**	**3**	**4**	**5**		
484.88	461.63	479.02	482.29	476.4	42.91	383
24.76	17.57	27.2	23.76	25.78	40.0	21
10.43	6.68	13.98	7.43	10.77	39.47	11
10.43	6.68	13.98	7.43	10.77	39.47	11
303.09	282.62	319.78	314.03	305.65	37.88	252
32.05	23.26	37.06	31.1	29.42	38.25	24
28.34	19.16	32.15	21.57	21.02	38.25	21
49.66	38.73	55.12	52.01	52.93	36.48	42
915.58	899.12	922.65	917.67	903.71	15.59	747
916.6	897.89	920.18	914.36	906.84	7.68	754

Cuadro 11: Tabla de campeones en la competencia evolutiva contra heurb

Al igual que en la competencia interna, en el gráfico tenemos la evolución del puntaje por generación generación. Estos valores se obtienen de la parte no comentada (#) de ev_h.dat. El programa que efectúa estas simulaciones es battles/battles.py con parámetros "0 -h". Para el momento en que se realizaron estas mediciones, los jugadores recibían 6 parámetros de los que usaban todos menos `Encerrados`. Más tarde se modificaron los jugadores para que efectivamente usen solo 5 parámetros evitando este. Es por ello que tomamos en cuenta solo el subconjunto correcto de los parámetros presentes en el archivo de datos.

En la tabla tenemos la información de los campeones de la competencia evolutiva contra heurb. Esta información son las últimas 10 líneas de ev_h.dat.

En este caso, los valores más altos aparecieron recién en generaciones más avanzadas, lo cual implica que los jugadores de las primeras generaciones fueron desterradas de esta lista por las nuevas generaciones. Esto da a entender que el ajuste realizado por la meta-heurística fue beneficioso para conseguir genotipos que fueran más competitivos a la hora de enfrentar a heurb.

El gráfico de la evolución poblacional no nos muestra ninguna tendencia particular, salvo cierta estabilidad. Sin embargo sí refleja los picos presentes en la tabla y cómo estos son mayores que los presentes al inicio. Comparando con la búsqueda anterior, vemos que los valores máximos contra heurb son menores que en la competencia interna. Esto da la pauta de que, bien ajustados los parámetros, nuestra heurística se da más pelea a si misma.

4.5.5. Torneo Master Series

En este gráfico se pueden ver los resultados obtenidos en el *Masters Series*, el torneo de todos contra todos en el que hicimos enfrentarse a los 10 ganadores de cada uno de los torneos anteriores. Todos los partidos se llevaron a cabo en un tablero con $n = 11$. El programa que simula este torneo es battles/masterss.py que recibe como parámetro archivos de jugadores de los cuales obtiene MAX_PLAY jugadores. Un archivo de jugadores

Los resultados de este torneo son interesantes porque se observa principalmente una marcada victoria de los jugadores que salieron del torneo evolutivo por sobre los del torneo interno. Así vemos el éxito de la idea de crear nuevos parámetros a partir de los que ya resultaron buenos, para encontrar resultados más finos. Esto es algo que el torneo interno no podía hacer; al ser definidos los jugadores al principio, teníamos que elegir una cantidad razonable y acotada. De hecho, tal vez se necesitaba un alcance de los parámetros mayor a 5, pero ya se volvían demasiados jugadores como para hacerlos jugar.

Además, vemos que los jugadores ganadores del torneo evolutivo tienen parámetros muy grandes; al poder elegirlos así y no restringirse a enteros, es más probable encontrar una buena relación entre todos los parámetros.

A partir de estos resultados elegiremos como parámetros para nuestros jugadores futuros los que resultaron ganadores aquí, considerando que después de todos los enfrentamientos estamos en condiciones de decir que son, al menos, de los mejores.

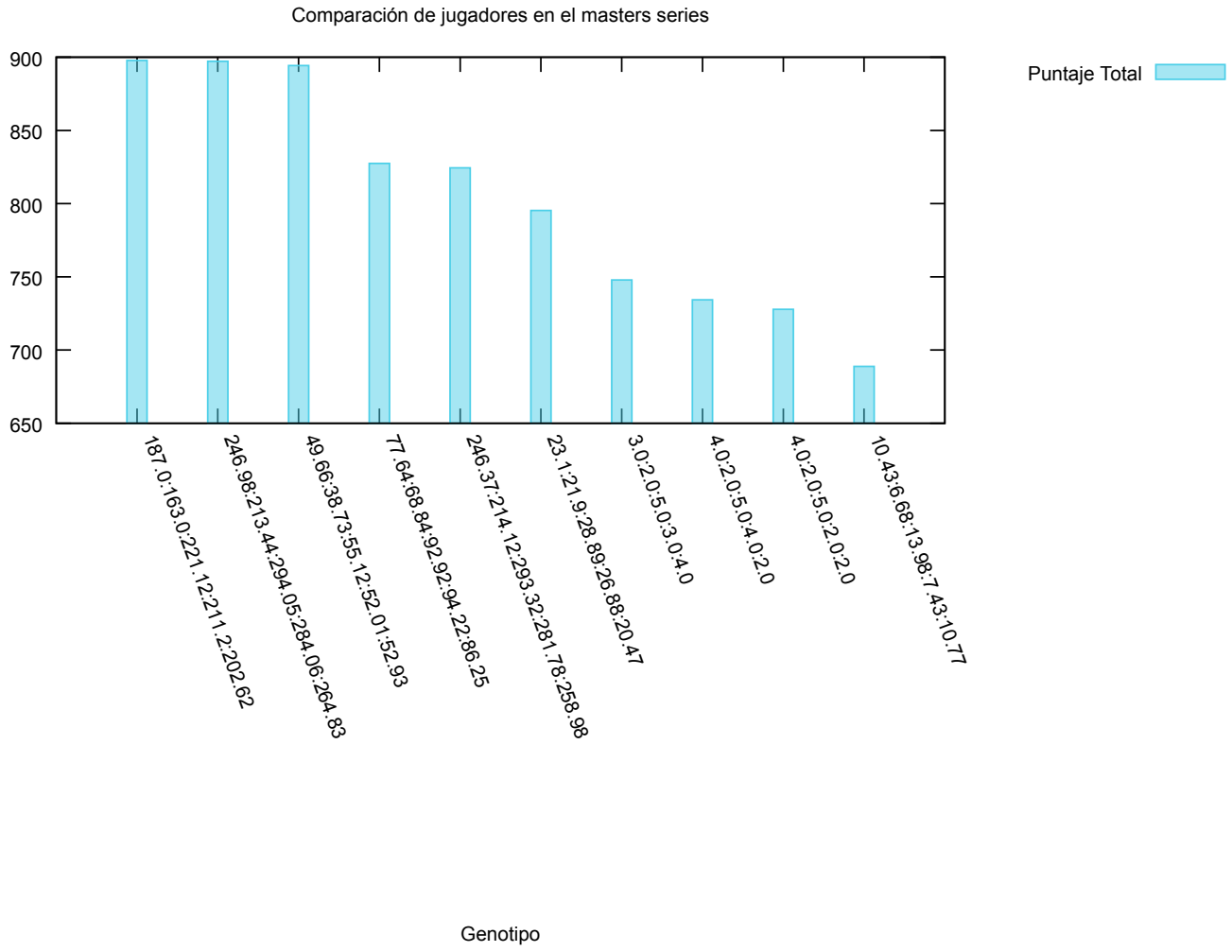

Figura 8: Puntaje de los primeros 10 jugadores en las **Master Series**

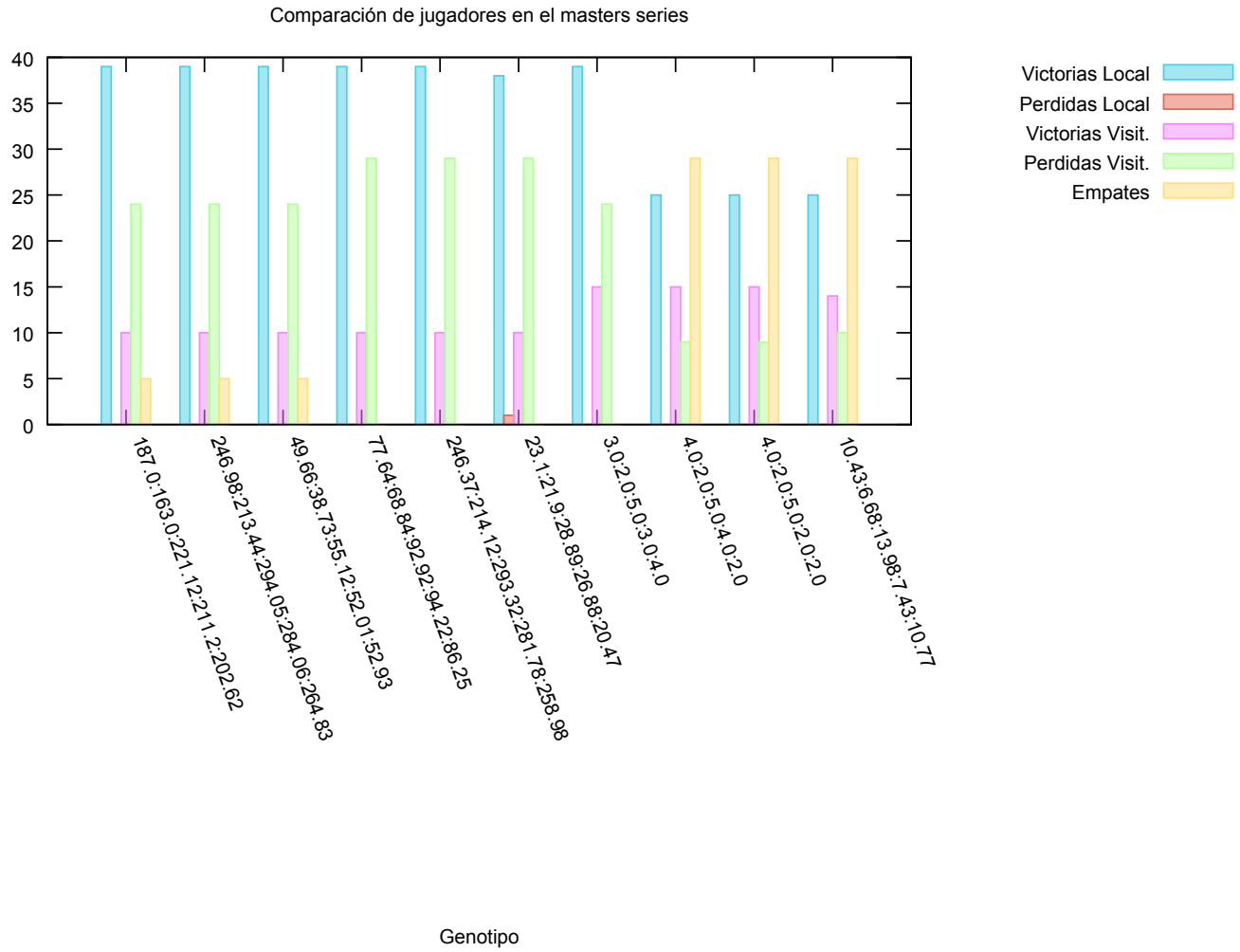

Figura 9: Estadísticas de los primeros 10 jugadores en las **Master Series**

Genotipo	Puntaje	Vic. Local	Perd. Local	Vic. Visit	Perd. Visit	Empates
187.0:163.0:221.12:211.2:202.62	897.805	39	0	10	24	5
246.98:213.44:294.05:284.06:264.83	897.216	39	0	10	24	5
49.66:38.73:55.12:52.01:52.93	894.303	39	0	10	24	5
77.64:68.84:92.92:94.22:86.25	827.435	39	0	10	29	0
246.37:214.12:293.32:281.78:258.98	824.416	39	0	10	29	0
23.1:21.9:28.89:26.88:20.47	795.365	38	1	10	29	0
3.0:2.0:5.0:3.0:4.0	747.894	39	0	15	24	0
4.0:2.0:5.0:4.0:2.0	734.285	25	0	15	9	29
4.0:2.0:5.0:2.0:2.0	727.862	25	0	15	9	29
10.43:6.68:13.98:7.43:10.77	688.819	25	0	14	10	29
10.43:6.68:13.98:7.43:10.77	688.819	25	0	14	10	29
5.0:2.0:5.0:1.0:3.0	679.521	24	1	15	9	29
4.0:1.0:4.0:2.0:1.0	640.418	25	0	15	9	29
3.0:1.0:4.0:2.0:3.0	640.418	25	0	15	9	29
2.0:1.0:4.0:1.0:1.0	599.775	36	3	15	24	0
9.38:1.19:3.94:3.75:0.88	587.405	25	0	12	12	29
4.0:1.0:3.0:1.0:4.0	555.841	23	2	15	9	29
3.0:1.0:3.0:1.0:3.0	555.841	23	2	15	9	29
5.0:1.0:3.0:1.0:4.0	555.841	23	2	15	9	29
5.0:1.0:3.0:1.0:2.0	555.841	23	2	15	9	29
4.0:1.0:4.0:1.0:2.0	554.274	22	3	15	9	29
5.0:1.0:5.0:1.0:1.0	551.05	22	3	15	9	29
2.0:2.0:5.0:1.0:1.0	476.72	38	1	15	24	0
3.0:1.0:3.0:4.0:0.0	440.814	23	2	12	12	29
3.0:2.0:4.0:0.0:3.0	267.109	23	1	12	25	17
303.09:282.62:319.78:314.03:305.65	-467.758	13	19	10	25	11
915.58:899.12:922.65:917.67:903.71	-541.72	13	19	9	30	7
916.6:897.89:920.18:914.36:906.84	-541.72	13	19	9	30	7
484.88:461.63:479.02:482.29:476.4	-546.448	13	19	9	30	7
24.76:17.57:27.2:23.76:25.78	-723.757	8	31	8	26	5
32.05:23.26:37.06:31.1:29.42	-745.471	8	31	8	26	5
28.34:19.16:32.15:21.57:21.02	-748.64	8	31	8	26	5
2.0:1.0:1.0:1.0:4.0	-780.39	13	19	4	34	8
51.69:47.14:61.46:55.65:49.72	-1008.544	6	32	9	30	1
0.0:6.5:0.25:5.75:0.75	-1036.355	7	32	9	30	0
32.78:35.41:46.16:40.08:33.99	-1082.765	5	33	5	33	2
1.0:4.0:2.0:0.0:1.0	-1420.31	3	36	12	27	0
2.0:4.0:4.0:0.0:1.0	-1539.152	9	29	2	37	1
0.0:2.0:1.0:0.0:4.0	-1842.069	3	36	12	27	0
1.0:0.0:0.0:1.0:1.0	-3359.988	0	39	0	39	0

Cuadro 12: Estadísticas de los jugadores en las **Master Series**

Lo que acabamos de ver las comparaciones y estadísticas de partidos jugados, ganados, perdidos y empatados de nuestros 40 mejores jugadores. Entre los análisis para hacer de estos resultados podemos decir que aquellos con mayor puntaje salen invictos jugando de local. Además, los primeros 5 son muy cercanos en puntaje. Algo interesante es ver que estos perdieron mucho más como visitantes que los jugadores del medio y sin embargo lograron tener una cantidad mayor de puntos, incluso 150 más puntos. Por otro lado, estos jugadores de mayor estabilidad (con pocos partidos perdidos), son en su mayoría originarios del **torneo suizo** o en su defecto poseen parámetros de valores bajos.

4.6. Conclusiones

La búsqueda de parámetros para nuestro jugador heurístico y, de forma más general, la optimización global, es un problema muy difícil de resolver de forma precisa y no menor para abordar de manera aproximada. La búsqueda heurística es un trabajo extenso y delicado que requiere conocimiento y un proceso de prueba y error. La investigación realizada conformó una búsqueda más que nada extensiva donde se utilizaron técnicas que apuntaban a poder distinguir los mejores candidatos dentro del universo de las muestras obtenidas. Todo esto manteniendo un nivel adecuado de eficiencia computacional para que el trabajo fuera realizable.

Siendo el **Master Series** la conclusión de la investigación y nuestra forma de filtrar los candidatos más adecuados, podemos distinguir los siguientes dos conjuntos de parámetros:

1. **187.0:163.0:221.12:211.2:202.62** como nuestro campeón final que obtuvo la mayor cantidad de puntos en la competencia que englobaba a los exponentes seleccionados por nuestros algoritmos.
2. **4.0:2.0:5.0:4.0:2.0** como el jugador que terminó el campeonato final con la menor cantidad de partidos perdidos y la mayor cantidad de puntos dentro de este subconjunto.

5. Jugadores Finales

En el tercer punto del trabajo práctico, la idea es combinar lo hecho anteriormente, para conseguir jugadores que aprovecharan ambas ideas (la búsqueda de jugadas avanzadas mediante *minimax* y el uso de jugadores aproximados) manteniendo una complejidad razonable; para finalmente encontrar un jugador representativo. Para esto se pedía armar jugadores parametrizados por k, tales que su complejidad fuese $\mathcal{O}(n^{2k})$. Como las heurísticas desarrolladas en el punto dos tienen un costo fijo para evaluar un tablero determinado, podemos decir que la complejidad de un jugador depende de:

- La profundidad de jugadas que se analizan en cada turno. O sea, a qué nivel llevamos nuestro árbol de jugadas en cada turno. Llamaremos a esto la **Profundidad**.
- La cantidad de tableros posibles que se desarrolla en cada nivel de profundidad. En el jugador exacto del primer punto, el árbol del tablero se abre siempre por completo, agregando todas las posibles jugadas a partir de cada tablero. Sin embargo, esto podría cambiar, eligiendo un subconjunto de todas las posibles. Llamaremos a esto el **Ancho**.

5.1. Error en la complejidad pedida

En el trabajo se pide que implementemos un jugador cuya complejidad sea $\mathcal{O}(n^{2k})$. Nosotros hacemos un análisis de nuestro jugador, pero en una parte debemos decir algo que es falso, dado que de otra manera, no es posible tener una complejidad de $\mathcal{O}(n^{2k})$. Esta falsedad es que $\prod_{i=0}^{k} O(n^2) \in O(n^{2k})$. En realidad, dadas constantes para las O $\alpha_1, ..., \alpha_k$, la multiplicación de estos órdenes de complejidad es $\mathcal{O}(\text{máx}\{\alpha_i\}^k \cdot n^{2k})$. O sea, hay una exponencial multiplicando. Esto es claro: $\prod_{i=0}^{k} O(1) \notin O(1)$, pero sí está en $O(\text{máx}\{\alpha_i\}^k)$, pues estamos tomando a k como variable, no como constante.

Si pretendemos que en realidad es $\mathcal{O}(n^{2k})$, se logra la complejidad pedida en el enunciado. Estamos al tanto de este error en el enunciado, y mencionaremos cuando cometamos este "error" a propósito para que dé la complejidad.

5.2. Parametrización de jugadores por altura (k)

La primera familia de jugadores en la que pensamos fue la que se obtiene abriendo siempre todo el ancho posible en cada nivel de profundidad, y cambiando la profundidad del árbol de jugadas analizada en cada turno. Esto se obtiene cambiando el valor del parámetro `height` en la llamada a función. A continuación explicaremos la relación entre k y `height`, y veremos que esto cumple con la complejidad pedida.

Podemos ignorar el efecto asintótico del $\alpha - \beta$ *pruning*, dado que, en el peor de los casos, es equivalente a `minimax` por si sólo.

Podemos, asimismo, acotar la cantidad de jugadas posibles en un determinado momento por $\mathcal{O}(n^2)$, pues empieza siendo $2n(n-1) \in \mathcal{O}(n^2)$ y disminuye en por lo menos 1 con cada jugada que se realiza (esa jugada no puede volver a realizarse).

Si exploraramos todas las jugadas hasta la altura h, podemos definir la recursión como:

$$T(h) = \begin{cases} O(n^2) \cdot T(h-1) + O(n^2) & \text{si } h \neq 0 \\ O(n^2) & \text{si } h = 0 \end{cases}$$

Esto es porque en cada llamada, vemos $\mathcal{O}(n^2)$ posibles jugadas, y por cada una de ella, vamos a llamar a nuestra función, viendo 1 altura menor. Para calcular las movidas, usaremos $\mathcal{O}(n^2)$ trabajo. Por último, cuando llegamos a $h = 0$, o sea, queremos ver 0 niveles más, tratamos al tablero como una "hoja" y usamos $\mathcal{O}(n^2)$ para calificarlo.

Podemos ver que esta recursión se resuelve con

$$T(h) = O(n^{2h+2}) = O(n^{2(h+1)}) \tag{29}$$

(Aquí usamos 5.1)

Por lo tanto, si vemos h jugadas para adelante, estamos usando complejidad $\mathcal{O}(n^{2(h+1)})$. Por este motivo, si queremos limitar la complejidad a $\mathcal{O}(n^{2k})$, podemos explorar todas las posibles jugadas, y ver $k-1$ jugadas en adelante.

5.3. Balanceo de profundidad y altura

En el jugador parametrizado por altura siempre se abre el árbol de jugadas todo lo que se puede. Lo siguiente que hicimos fue considerar si se podrían ver menos nodos en cada paso (eligiéndolos según alguna heurística), posibilitando así aumentar la profundidad manteniendo la complejidad. A continuación explicaremos en detalle las ideas que surgieron a partir de esto.

5.3.1. Idea básica

En el punto anterior mostramos que considerar un árbol n^2-ario en $k-1$ niveles de profundidad nos cuesta, con nuestra heurística, $\mathcal{O}(n^{2k})$.

Entonces lo que intentamos fue no abrirnos por todas las ramas horizontalmente, para en vez de construir un árbol que esté acotado por uno n^2-ario, considerar uno acotado por n^x con $0 \leq x < 2$, para así poder mantener la complejidad total $\mathcal{O}(n^{2k})$ pero mirando más niveles en profundidad.

El programa que hicimos toma un x, que representa el Ancho, o sea cuántas ramas (tableros) abrirá en cada nivel de profundidad. Este paso de abrir tableros consiste en calcular todos los "hijos" (los tableros posibles a partir de uno dado), obtener el valor correspondiente a cada uno utilizando el puntuador de tableros programado anteriormente y luego elegir los n^x mejores para seguir abriéndose por estos. A partir del x recibido, nuestro programa calcula h, que es el nivel de profundidad máximo que se puede alcanzar en cada turno sin exceder la complejidad pedida.

Para saber cuántos tableros podemos mantener en cada rama, tenemos que encontrar una relación entre h y x tal que se mantenga la complejidad al usar esta nueva profundidad. Sabemos que la complejidad de una jugada es $(n^x)^h n^4$. Aquí $(n^x)^h$ es el tamaño el árbol obtenido al finalizar, porque vemos siempre n^x nodos hijos, y descendemos h niveles. n^4 es una cota del trabajo hecho para cada tablero (obtener todos sus hijos y aplicarles la heurística final). En realidad no siempre este trabajo es n^4, ya que en los tableros finales, como no tienen hijos, solamente calculamos el puntaje del tablero, lo cual es lineal en n^2, pero esto sirve como cota superior.

(Aquí también usamos 5.1 para derivar esta cota de complejidad, la verdadera tiene un factor multiplicativo exponencial en h)

Recapitulando, la complejidad de una jugada es:

$$n^{x \times h+4} \tag{30}$$

Como queremos que esto sea menor a $n^{k \times 2}$, necesitamos encontrar una relación entre x y h tal que valga que

$$x \times h + 4 \leq 2 \times k \tag{31}$$

Para maximizar x y h tomaremos

$$x \times h + 4 = 2 \times k \tag{32}$$

Entonces

$$x = \frac{2 \times k - 4}{h} \tag{33}$$

Si en vez de eso queremos calcular la profundidad que podemos visitar, recibiendo el x como parámetro, simplemente usamos la ecuación como

$$h = \frac{2 \times k - 4}{x} \tag{34}$$

5.3.2. Mejora a esta idea

En el punto anterior, el x y el k están fijos desde el comienzo de una partida. A medida que se desarrolla el juego, los tableros posibles en un determinado momento son cada vez menos, porque hay cada vez más fichas. Por lo tanto, llega un momento en que hay menos tableros posibles que n^x, que es lo máximo que deberíamos mirar en cada nivel de profundidad; sin embargo mantenemos el mismo h que cuando sí abríamos todas esas ramas.

Nuestra idea fue no desperdiciar esto para, con la misma complejidad, mirar más niveles de profundidad en las últimas jugadas. La mejora que implementamos funciona de la siguiente manera:

En cada turno, comparamos las jugadas posibles que tenemos a partir del tablero como está con n^x. Si es mayor, utilizamos ese valor para hacer h niveles de profundidad, como en la versión anterior. Si es menor, calculamos cuánta profundidad podemos ver extendiendo el árbol completo, de manera que la complejidad se mantenga en n^{2k}. Considerando a m como el número de jugadas posibles, buscaremos el h' máximo que satisfaga esto. El cálculo que hacemos es:

$$m^{h'} * m^4 \leq n^{2k} \tag{35}$$

O sea

$$m^{h'+4} \leq n^{2k} \tag{36}$$

$$log_n(m^{h'+4}) \leq 2k \tag{37}$$

$$(h'+4)log_n(m) \leq 2k \tag{38}$$

$$h' \leq \frac{2k}{log_n(m)} - 4 \tag{39}$$

Con esto, trataremos de maximizar h'. La complejidad sigue siendo claramente la misma; en el comienzo por lo ya analizado en el punto anterior, y a partir de cuando empezamos a considerar la cantidad de jugadas, porque estamos calculando h', la nueva profundidad, especialmente para que ésta se mantenga.

5.3.3. Implementación

Esta idea fue implementada como un jugador que recibe como parámetros los valores de k y de x. A partir de estos, calcula mediante la cuenta ya mencionada la altura máxima que puede hundirse en cada turno, en las n^x posiciones que evalúe por turno. Esta altura es pasada como parámetro por default de altura a la función que calcula la mejor jugada. La misma va a filtrar las mejores n^x jugadas (con una heurística para evaluar todos sus hijos) y hará recursión en ellas. La lógica de la función es, por lo demás, similar a la implementación anterior de minimax con $\alpha - \beta$ pruning.

Algo que se debe mencionar es que, si bien en el alphabeta los tableros finales siempre se evalúan con la función de puntuación, nuestro programa también analiza las hojas utilizando las heurísticas. Esto no es lo mejor, porque en las hojas ya no importa el valor heurístico ya que se enfocan en la situación a futuro; lo único que es relevante ahora es el puntuaje actual. Sin embargo tuvimos que tomar esa decisión porque los nodos sí son evaluados con heurísticas, y hay una diferencia de magnitud entre los valores de éstas y del puntaje (de hecho las heurísticas siempre son el puntaje más otras cosas) por lo tanto en comparaciones entre nodos y hojas (que las hay) estas últimas saldrían siempre perdiendo, y la comparación no sería significativa.

5.3.4. Decisión final

Al programar la "mejora", notamos que necesitábamos hacer una recorrida del tablero aparte en cada jugada para contar antes de empezar los tableros posibles a partir del actual. Esto no se puede hacer en medio del juego porque la función que evalúa cada rama es recursiva, y sólo queríamos cambiar la altura en la primer llamada de cada turno.

La programamos de todas maneras, y el resultado fue que sólo empezaba a aumentar la altura cuando ya había muy pocas jugadas posibles, lo cual es lógico. En general, incluso, el aumento de profundidad no se utilizaba porque prácticamente no había tantos niveles en los tableros en los que se usaba. Los mejores resultados se obtenían con valores grandes de x y del tamaño del tablero. Finalmente decidimos que en general era preferible no recorrer el tablero una vez más, y perderse de extender la altura en las últimas jugadas, por lo tanto utilizamos la versión sin la mejora.

5.3.5. Búsqueda de la amplitud adecuada

Para encontrar el mejor balance entre x y h, armamos otro torneo en el cual comparamos, para cada k razonable, los resultados de partidos entre jugadores con diversos valores de x y un conjunto de otros jugadores fijo.

Para esta búsqueda usamos un jugador con los mejores parámetros obtenidos en el punto **2.d** fijos, y un x variante dentro de 20 valores diferentes (entre 0.1 y 2.0, incrementando en 0.1). A este jugador con ajuste en su búsqueda a lo ancho y profundo lo denominamos *adaptativo*. La manera de evaluar la competitividad

de las distintas estrategias fue definir un conjunto fijo de oponentes (heurb, el jugador goloso con el segundo mejor parámetro obtenido, también el primero), contra los cuales luego hacerlos competir. Dada la aleatoriedad presente en heurb, se lo agregó 3 veces a la lista, lo que nos da un total de 5 oponentes, luego 10 partidos por cada k. El puntaje final de un jugador con un x determinado es la suma de los puntajes obtenidos en los partidos contra la lista fija de oponentes. Contra cada oponente se compitió una vez de local y otra de visitante. Todos los partidos fueron en tablero tamaño 11. El torneo que efectúa estos partidos es battles/torneo_x.py. Devuelve la tabla de resultados presentada a continuación. De esta manera comparamos a cuál le había ido mejor.

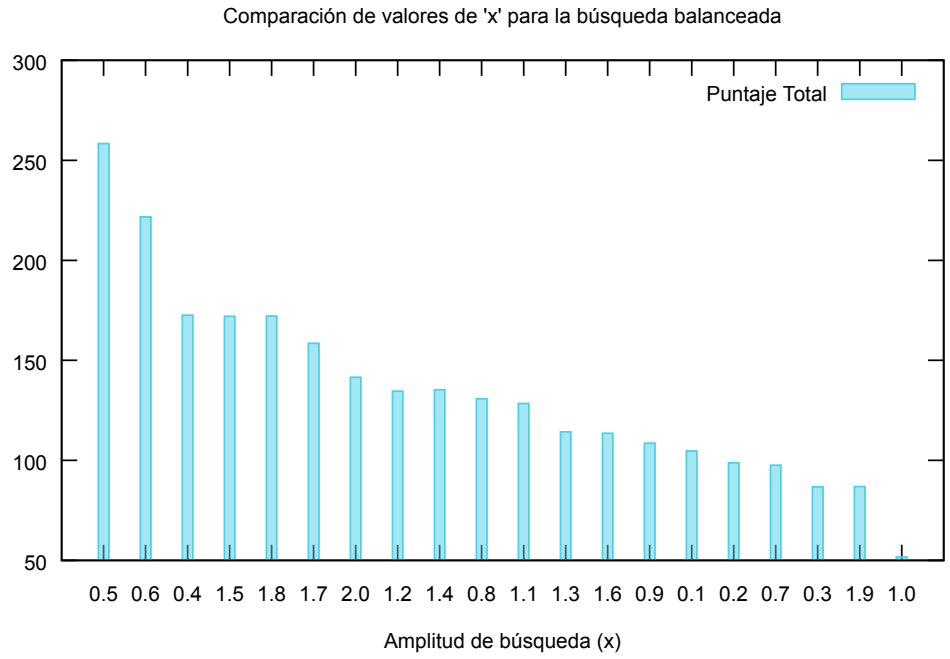

Figura 10: Puntaje del jugador adaptativo según la amplitud de búsqueda (n^x) con $k = 3$

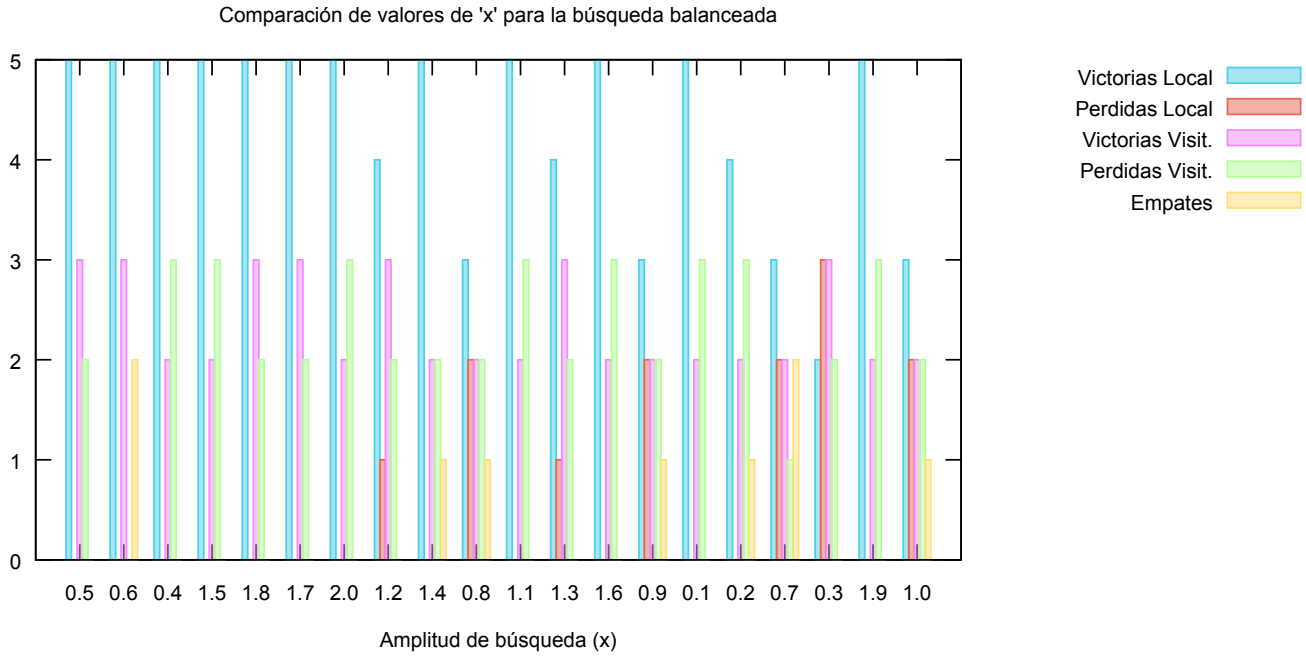

Figura 11: Estadísticas del jugador adaptativo según la amplitud de búsqueda (n^x) con $k = 3$

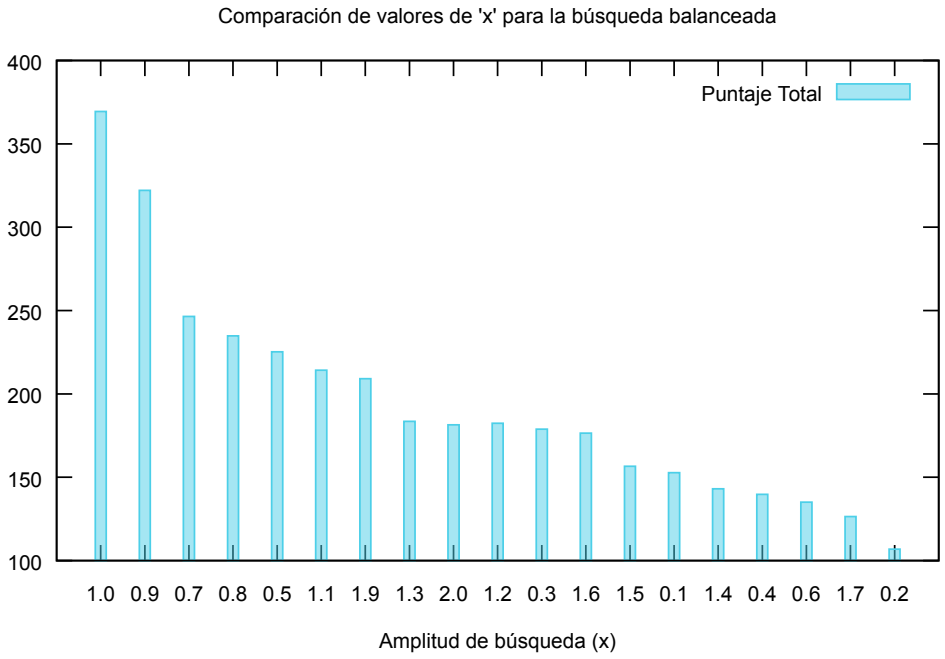

Figura 12: Puntaje del jugador adaptativo según la amplitud de búsqueda (n^x) con $k = 4$

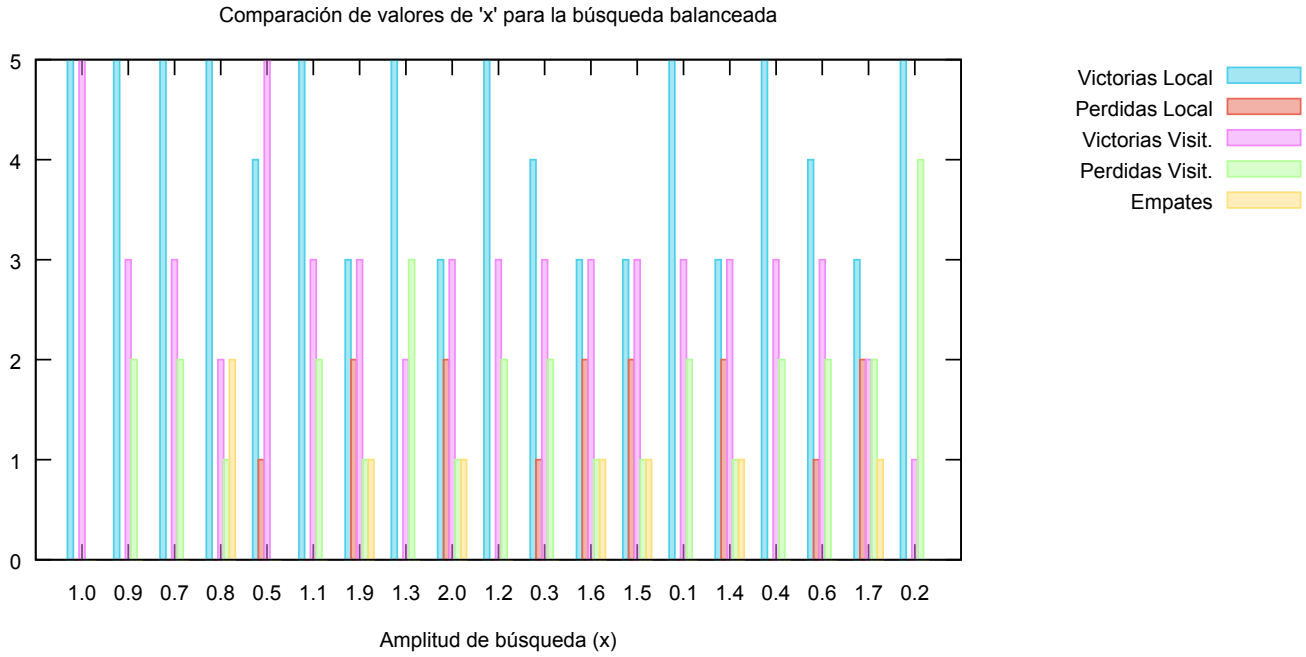

Figura 13: Puntaje del jugador adaptativo según la amplitud de búsqueda (n^x) con $k = 4$

El algoritmo goloso debía, en cada movida, revisar todas las jugadas posibles y para cada una de ellas hacer un análisis estático del tablero resultante. Como fue dicho anteriormente, la búsqueda de las jugadas posibles y la evaluación estática toma una cantidad de tiempo $\mathcal{O}(n^2)$. Dado que la cantidad de jugadas se acota por la misma complejidad, este algoritmo requiere al menos $\mathcal{O}(n^4)$. De esta forma, si queremos respetar una complejidad $\mathcal{O}(n^{2k})$, k debe ser mayor o igual a 2. En particular, dado que en nuestro algoritmo calculamos la altura con la que podemos hundirnos en el árbol de jugadas mediante la fórmula $h = (2 \times k - 4)/x$, Si $k < 2$ la fórmula da 0, haciendo que el algoritmo no funcione. Es por esto que requerimos $k \geq 3$. Por el otro lado,

corriendo el algoritmo efectivamente, el tiempo requerido para procesar una jugada con $k \geq 5$ hacía poco viable cualquier estudio. En consecuencia, los únicos valores de k que pudimos analizar fueron 3 y 4.

Tanto para $k = 3$ como $k = 4$ vemos que hay una diferencia fuerte en la cantidad de puntos acumulados por los jugadores con distintos valores de x. Si bien cambia qué valores logran el mayor puntaje, es incluso notable la similitud en la forma de los gráficos. Viendo la escala de estos notamos que $k = 4$ y la visión adelantada que le permiten hace que pueda obtener muchos más puntos (370 a 260). Por otro lado, dado que un k mayor da más profundidad de búsqueda, vemos que en $k = 4$ esta profundidad no es tan beneficiosa como aumentar la amplitud de búsqueda. Muestra de esto es que para $k = 3$ el x óptimo fue 0.5, mientras que para $k = 4$ fue 1.

Corriendo nuevamente las partidas jugadas, vemos que los dos casos donde el algoritmo adaptativo perdió con $k = 3$ y $x = 0{,}5$ fue al jugar contra nuestro goloso de visitante. Con $x = 0{,}6$ esto ya no pasa. Pasa entonces que $x = 0{,}5$ se consagra por haber obtenido más puntos de heurb. Viendo que $x = 0{,}6$ logra vencer a nuestra heurística consistentemente y la distancia en puntaje de ambos jugadores no es tan grande, consideramos este valor como el más competitivo, ya que consideramos que, en promedio, se posiciona mejor frente a una variedad más amplia de jugadores.

Es notable que con un incremento en el valor de k, cambie radicalmente el valor óptimo de x. En el caso $k = 4$, si el jugador usa $x = 1{,}0$, vemos en la tabla, es el único valor que no perdió ninguna partida. Además la diferencia de puntos con el inmediatamente anterior es de 80, lo cual es prácticamente la suma dos fuertes victorias en un tablero de 11×11. En este caso no quedan dudas que $x = 1{,}0$ es óptimo dentro de la muestra utilizada. Sin embargo esto es razonable. Es bueno ver cierta cantidad de pasos adelante; por lo tanto cuando el k es chico es preferible resignar jugadas analizadas en favor de la profundidad, ya quedade otra manera esta sería casi nula. Lo que nos muestran los gráficos es que no es cierto, de todas maneras, que aumentar la profundidad sea la forma más clara de ganar. Para $k = 4$ el ganador es $x = 1{,}0$, que elije ver varios nodos y resignar tener mayor profundidad. Aquí se comienza a ver que ambas medidas son importantes, y que se deben combinar bien. Suponemos que se podrían haber llegado a resultados interesantes viendo lo que sucedía con valores mayores de k, pero el tiempo que llevaba analizarlo era excesivo.

Amplitud (n^x)	**Puntaje**	**Vic. Local**	**Perd. Local**	**Vic. Visit**	**Perd. Visit**	**Empates**	**Tiempo Prom. (ms)***
0.5	258.396	5	0	3	2	0	1414.38
0.6	221.686	5	0	3	0	2	1248.45
0.4	172.585	5	0	2	3	0	725.13
1.5	172.038	5	0	2	3	0	429.69
1.8	172.132	5	0	3	2	0	269.54
1.7	158.537	5	0	3	2	0	317.44
2.0	141.514	5	0	2	3	0	271.85
1.2	134.628	4	1	3	2	0	325.36
1.4	135.309	5	0	2	2	1	385.53
0.8	130.807	3	2	2	2	1	604.06
1.1	128.426	5	0	2	3	0	233.79
1.3	114.243	4	1	3	2	0	311.9
1.6	113.566	5	0	2	3	0	277.69
0.9	108.603	3	2	2	2	1	750.35
0.1	104.728	5	0	2	3	0	996.54
0.2	98.841	4	0	2	3	1	532.46
0.7	97.558	3	2	2	1	2	554.44
0.3	86.762	2	3	3	2	0	2106.71
1.9	86.932	5	0	2	3	0	256.61
1.0	51.696	3	2	2	2	1	994.99

*Tiempo promedio en milisegundos de las partidas local y visitante contra el mismo oponente

Cuadro 13: Tabla de pruebas de amplitud con $k = 3$

Amplitud (n^x)	**Puntaje**	**Vic. Local**	**Perd. Local**	**Vic. Visit**	**Perd. Visit**	**Empates**	**Tiempo Prom. (ms)***
1.0	369.366	5	0	5	0	0	13935.13
0.9	322.137	5	0	3	2	0	7893.26
0.7	246.458	5	0	3	2	0	11349.56
0.8	234.827	5	0	2	1	2	5539.57
0.5	225.205	4	1	5	0	0	41597.0
1.1	214.244	5	0	3	2	0	4767.05
1.9	209.153	3	2	3	1	1	5817.96
1.3	183.553	5	0	2	3	0	6308.14
2.0	181.464	3	2	3	1	1	7500.48
1.2	182.366	5	0	3	2	0	5170.77
0.3	178.866	4	1	3	2	0	131467.24
1.6	176.465	3	2	3	1	1	3093.32
1.5	156.588	3	2	3	1	1	2407.46
0.1	152.775	5	0	3	2	0	1386.25
1.4	143.107	3	2	3	1	1	2031.07
0.4	139.744	5	0	3	2	0	13685.83
0.6	135.046	4	1	3	2	0	17832.47
1.7	126.42	3	2	2	2	1	3714.71
0.2	106.909	5	0	1	4	0	989.78
1.8	98.409	3	2	2	2	1	4503.41

*Tiempo promedio en milisegundos de las partidas local y visitante contra el mismo oponente

Cuadro 14: Tabla de pruebas de amplitud con $k = 4$

5.3.6. Evaluación del uso del tiempo

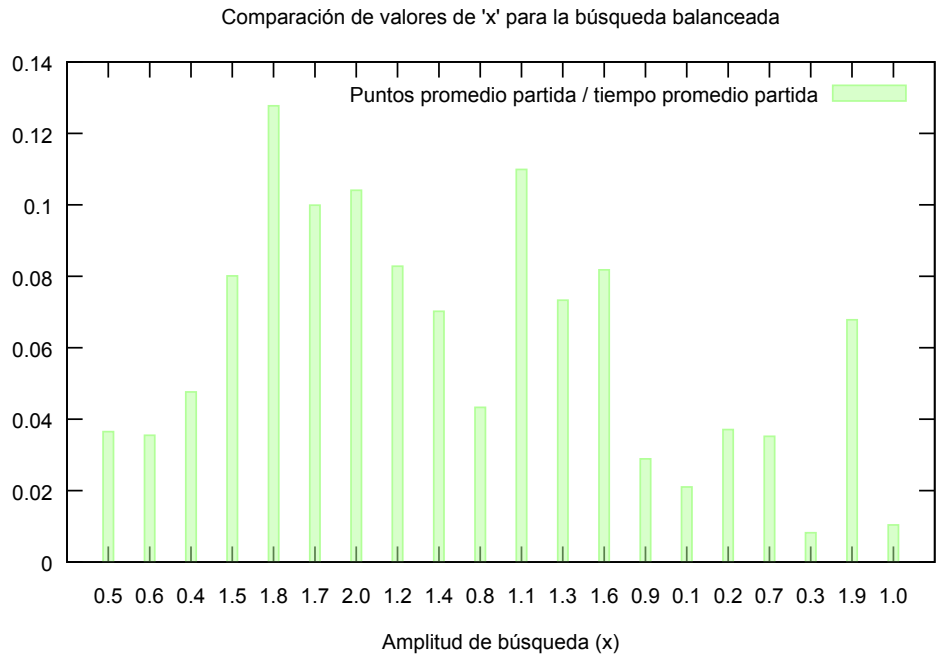

Figura 14: Relación puntaje/tiempo promedios de partida según la amplitud de búsqueda (n^x) con $k = 3$

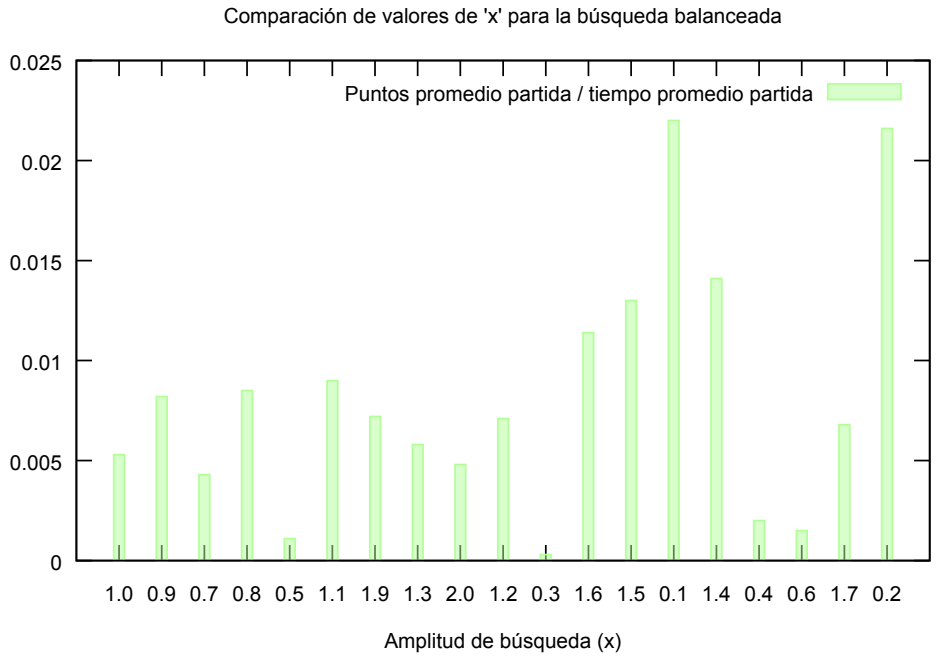

Figura 15: Relación puntaje/tiempo promedios de partida según la amplitud de búsqueda (n^x) con $k = 4$

A partir de los datos de las tablas es difícil obtener información sobre el comportamiento relacionado con el tiempo del algoritmo. De los valores de las tablas no parece haber ningún patrón que relacione el valor de x con el tiempo de duración de la partida. Esta impredecibilidad puede estar relacionada con el efecto del $\alpha - \beta$ *prunning*, o por los elementos de aleatoriedad en `heurb` que codifican de forma no determinista que subárbol de juego se va a seguir. Además, no parecería haber una relación entre el tiempo de juego y el puntaje obtenido. Si bien el tiempo de juego de los primeros en la tabla es de los más altos, no es el máximo y este es alguien en una posición más bien baja de la tabla.

Por otro lado en el gráfico vemos que no son necesariamente los jugadores de mayor puntaje los que hacen mejor uso del tiempo: cantidad de puntos por unidad de tiempo. En particular, los dos jugadores que ganaron obtuvieron una cantidad más bien chica de puntos por unidad de tiempo, en contraposición a otros jugadores más abajo en la tabla que alcanzan picos más altos en el gráfico.

5.4. Conclusiones y elección de nuestro jugador representativo

Habiendo implementado diversas estratégias para encontrar mejores parámetros y por lo tanto mejores jugadores usando nuestras heurísticas de manera golosa decidimos que nuestro jugador más fuerte fue aquel que sumó más puntos en el total de partidos que disputó. Con estos parámetros y nuestra idea `adaptive` para ver variables anchos y profundidades dentro del árbol de jugadas, hicimos el torneo ya expuesto.

Siguiendo con nuestro favoritismo por aquellos jugadores que concretan mayor cantidad de puntaje en el total de los partidos disputados, nuestro jugador representante será el ganador del Torneo Masters con un x de 1.0 y un k de 4.0 pues logró una diferencia de puntaje notable frente al resto de sus contrincantes.

La manera de invocarlo será entonces:

```
./jugador –alphabeta –funcion 1 –params 187.0 163.0 221.12 211.2 202.62 –adaptive 1 4
```