Algoritmos y estructuras de datos III

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

Trabajo Práctico III

C en Linea

Integrante	LU	Correo electrónico
Ansaldi, Nicolas	128/14	nansaldi611@gmail.com
Suárez, Romina	182/14	romi_de_munro@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

En este trabajo práctico se nos da a programar un algoritmo que personifique un jugador dentro del famoso juego 4 en Linea, pero extendido a C, un número natural positivo cualquiera.

En el ejercicio 1, se nos pide buscar un algoritmo exacto basado en la estrategia llamada Minimax, que consiste en buscar los movimientos que lleven a ganar, o en caso de no poder, a empatar. La estrategia minimax es ajena al algoritmo en si, pero su concepto es aplicable a este juego. De la forma en que tenemos 2 jugadores, y el movimiento de uno, simboliza la perdida de puntaje del otro, Minimax busca dentro de un árbol de decisiones, por cada nivel, la máxima ganancia de uno de ellos, y al siguiente, la menor perdida del otro. Así, y suponiendo que el contrario siempre juega de forma óptima, esta técnica buscará el mejor resultado posible. Por razones de eficiencia se utilizará una poda denominada alfa-beta, que se encargará de podar las ramas que ya no generen una mejor solución dentro de los niveles del árbol.

En la segunda etapa, ya no se busca un algoritmo exacto, sino uno que brinde soluciones aproximadas. El mismo toma sus decisiones de forma golosa dado que mira su mejor movimiento de forma local para alguna instancia del juego. Esto provocará que el algoritmo no gane tantas veces como el algoritmo de backtracking, sin embargo, será muchísimo más rápido.

En la última etapa o ejercicio 3, se busca optimizar al algoritmo goloso del ejercicio anterior, utilizando las técnicas de Grid-search y de genética que se encargarán de refinar los parámetros que toma dicho algoritmo. Realizando distintos experimentos lograremos modificar las variables del algoritmo en busca de distintos objetivos: tener un jugador óptimo, conseguir el mejor jugador cuanto antes, etc.

1.1. Uso de Lenguajes

Para todos los ejercicios, se empleó el uso del lenguaje C++, y de python para generar los gráficos y la experimentación.

2. Ejercicio 1

2.1. Explicación del algoritmo y Pseudo-Código

Para este ejercicio levantamos los datos que nos llegan del juez, creamos ”board", la matriz de fichas, la cual tendrá en cada casilla un 1, si es nuestra, 0 si el del contrario y -1 si todavía nadie colocó una ficha en ese lugar y el vector ”PrimDisp"que tendrá el tamaño de la cantidad de columnas del tablero y en cada posición la primera fila que este disponible para colocar ficha. Como dijimos, usamos la estrategia minimax para saber el siguiente movimiento de nuestro jugador. Lo que hacemos es: Dada una instancia del juego utilizamos backtracking, o sea creamos un árbol de decisiones donde vemos todos los posibles movimientos desde el comienzo del juego hasta que este termine, representando los resultados como: 1 si ganamos, -1 si perdemos y 0 si empatamos. Como el juego es por turnos utilizamos 2 funciones, maxmin y minmax, la primera se encarga de nuestro movimiento y la segunda del contrario. Entonces vamos alternando las funciones en los niveles del árbol, por ejemplo si empezamos con maxmin, esta le pasará su movimiento a minmax y dado ese movimiento hará el suyo que se lo pasará a maxmin y asi sucesivamente. Cuando llegamos a los estados terminales para cada jugada, las funciones se quedarán con el movimiento que mas les convenga, o sea, el óptimo. Para no recorrer todo el árbol utilizamos la poda alfa-beta, ésta se encarga de no recorrer una rama que no agregue información a lo que ya tengo. Por ejemplo si me toca mover a mi y tengo una jugada que me lleva a ganar el juego ya no recorro otras posibilidades porque se que no mejorarán esa rama y a lo sumo serán iguales.

Algorithm 1 int maxmin(Matriz board, int fichasRestantes , vector de int primDisp,int posicionesDisp, int alfa, int beta, int altura)

```
1: if (maxMio >= c ) then
2:     return 1
3: end if
4: if  (maxCont >= c ) then
5:     return -1
6: end if
7: if ( no hay mas fichas || no quedan más posiciones libres ) then
8:     return 0
9: end if
10: i = 0
11: res = -Infinito
12: while i < columns do
13:     aux < − primDisp[i]
14:     if  aux <rows then
15:         if  !(i >0 and res >= beta) then
16:             board[aux][i] = 1
17:             aux2 = maxMio
18:             primDisp[i]++
19:             puntosNuevaFicha(aux, i, maxMio, board, 1)
20:             fichasRestantes − −
21:             posicionesDisp − −
22:             altura++
23:             actual = minmax(board, fichasRestantes, primDisp, posicionesDisp, alfa, beta, altura)
24:             altura − −
25:             posicionesDisp + +
26:             board[aux][i] = −1
27:             maxMio = aux2
28:             primDisp[i] − −
29:             fichasRestantes + +
30:             if actual >res then
31:                  res = actual
32:                 movimiento = i
33:             end if
34:             if res >alfa then
35:                 alfa = res
36:             end if
37:         end if
38:     end if
39:     n++
40: end while
41: if altura == 0 then
42:     return movimiento
43: end if
44: return res
```

La función maxmin funciona de la siguiente manera: Toma por parámetros el tablero, la cantidad de fichas restantes, para cortar en caso de que sea 0, el vector de primeras disponibles, la variable posiciones Disponibles, que corta el algoritmo en caso de ser 0, alfa y beta para la poda, y la altura. Los 3 casos base, en donde gana, pierde o empata se verifican al principio, luego para la cantidad de columnas, si es posible colocar una ficha allí, es decir, si la primera fila disponible de la columna es más chica que la cantidad de filas que hay, se coloca la ficha, se guarda el máximo valor mio hasta ahora previo, se recalcula el nuevo maxMio con puntosNuevaFicha, se aumenta la altura y se llama a la recursión con la función minmax. La misma es idéntica a esta función, pero coloca la ficha como si fuese del contrario, toma el mínimo de sus posibles movimientos, y en la recursión llama a maxmin. Luego de volver de la recursión, se deshace el movimiento, y se pregunta si es mejor que el máximo local, entonces se actualiza. La variable movimiento contendrá, solo si la altura es 0, es decir, estoy en

los primeros n hijos, la columna a mandar al juez.
La poda funciona de la siguiente manera: En cada hijo de nivel min se actualizará beta, y en max, alfa, si este es mejor al res local. Y antes de hacer la recursión, en caso de estar en max, se preguntará si el res obtenido es mejor que beta, y en min, con alfa. De ser positivo, se salteará esa rama.

2.2. Calculo de Complejidad

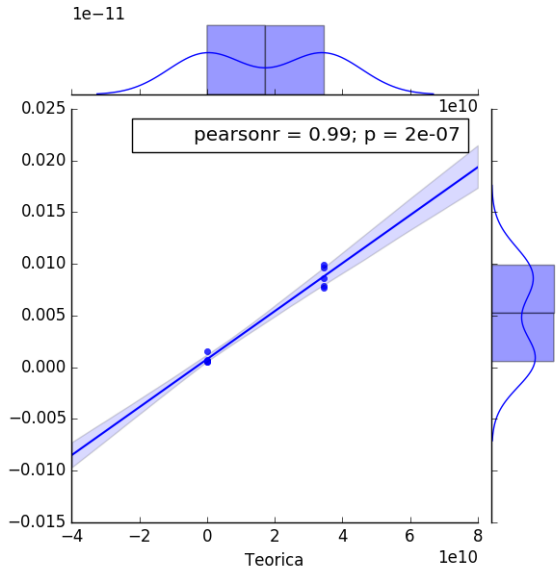
Como ya dijimos este algoritmo se puede pensar como un árbol de decisiones. Cada nodo tiene cantColumnas como hijos, ya que veo todos los posibles movimientos y la altura del árbol es $cantColumnas * cantFilas$ ya que termina cuando el juego termina, y en peor caso es cuando lleno toda la matriz. Luego para pasar de un nivel del árbol a otro pago O($cantColumnas + cantFilas$) para calcular el puntaje, que es la suma de cadenas de fichas, de la nueva ficha agregada al tablero. Entonces la complejidad total del algoritmo es O($(cantColumnas + cantFilas) * cantColumnas^{(}cantColumnas * cantFilas)$).

2.3. Correctitud

Lo que queremos ver acá es que generamos todos los posibles movimientos dado una instancia del juego. Como dado cualquier instancia del juego, vemos el camino de poner una ficha en cada columna y esto lo hacemos para cada nodo podemos decir que recorremos todos los posibles casos. Luego ver que devolvemos el mejor movimiento viene de la estrategia minmax presentada en clase. Entonces con esto generamos todos los posibles movimietos y sabemos que devolvemos el movimiento óptimo para cualquier instancia del juego.

2.4. Experimentación

2.4.1. Pearson



Para corroborar la complejidad propuesta corrimos al backtracking vs el algoritmo random 5 veces en un tablero de 3x3 y 5 veces en uno de 4x4, por otro lado calculamos la complejidad propuesta para estos valores de columnas y filas y corrimos la diferencia. Como vemos, el gráfico muestra que la relación es de 0.99, viendo así una buena dependencia entre ambas complejidades.

3. Ejercicio 2

3.1. Explicación del algoritmo y Pseudo-Código

Aquí lo que buscamos es mejorar la complejidad del ejercicio anterior, para esto evitamos recorrer todas las posibles combinaciones de una instancia del juego dada, y nos centramos en encontrar un movimiento basándonos en el estado actual del juego. Notar que esto puede provocar jugadas que no son las óptimas. Para el algoritmo goloso, mantuvimos la matriz board con el jugador dueño de esa ficha, 1 si es nuestra, 0 si es del contrario y -1 si no es de nadie. Tenemos el mismo vector PrimDisp y 3 nuevos vectores de pesos llamados "pesosPrioridad", cuadrantes y "columnass"los cuales servirán para el ejercicio 3. El vector PesosPrioridad le da pesos a colocar una ficha en la diagonal izquierda, o derecha, o el horizontal o en vertical. El vector Cuadrantes le da pesos a colocar una ficha en los 4 cuadrantes, empezando por el izquierda abajo, izquierda arriba, derecha arriba, y derecha abajo. El vector columnass da peso a cada columna del tablero. Notar que si se cambia la cantidad de columnas a jugar, también cambia la cantidad de pesos que toma el algoritmo. Siguiendo la misma lógica, si nos toca ir primero, o sino, registraremos el movimiento del contrario y llamaremos a la función greedy:

Algorithm 2 int greedy(int** board, int jugador, vector<int>primDisp, vector<int>& pesosPrioridad, vector<int>& cuadrantes, vector<int>& columnass)

```
1: int max = -1000
2: aux2 = fichaGanadora(board, primDisp, 1, aCuantoLeCortoElCombo,pesosPrioridad)
3: if aux2 != -1 then
4:     return aux2
5: end if
6: aux2 = fichaGanadora(board, primDisp, 0, aCuantoLeCortoElCombo,pesosPrioridad)
7: if aux2 != -1 then
8:     return aux2
9: end if
10: while i <columns do
11:     aux = primDisp[i]
12:     if aux <rows then
13:         actual = nuevoValor(board, i, aux, jugador, pesosPrioridad)
14:         if actual >max then
15:             max = actual
16:             res = i
17:         end if
18:     end if
19: end while
```

Algorithm 3 fichaGanadora(int** board, vector<int>primDisp, int jugador, vector<int>pesosPrioridad

```
1: while while (i <columns) do
2:     aux = primDisp[i]
3:     if (aux <rows) then
4:         if (nuevoValor(board, i, primDisp[i], jugador) >= c) then
5:             return i
6:         end if
7:     end if
8: end while
9: return -1
```

La función greedy llama a la función fichaGanadora, la cual se fija por cada columna, si colocando la ficha en esa posición, el algoritmo gana, entonces devuelve esa columna. De no ser así, se llama a la misma función, preguntando por el contrario. Si este va a ganar, se manda esa posición para impedirlo. Y luego, si ninguno puede ganar, entonces se busca la posición que nos de el mejor puntaje. La función nuevoValor buscara en todas las direcciones si las fichas son nuestras y acumulará la máxima de las mismas, sin ningún ponderamiento.

3.2. Calculo de Complejidad

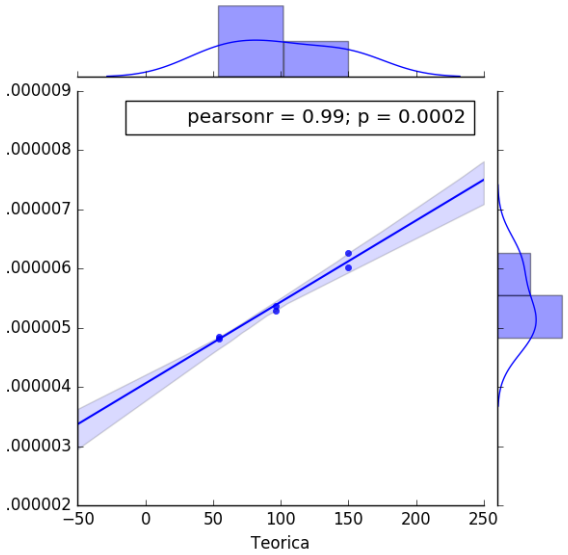
- La función nuevoValor toma O(columnas+filas) en hacer comparaciones.
- La función fichaGanadora toma O(columnas*(columnas+filas)).
- La función greedy toma O(3*columnas*(columnas+filas)).
- Luego, la complejidad propuesta para devolver la próxima ficha a colocar es O($3 * (columnas * (filas + columnas))$).

3.3. Correctitud

Este algoritmo es goloso pues busca dentro del espectro de posibilidades la primera más cercana que le otorgue el mayor valor para acercarse a c. Sin embargo, esto puede llevarlo a perder o empatar. La función greedy hace lo descripto, dentro del árbol de posibilidades la raíz es el tablero que llega como parámetro, y sus hijos son los movimientos posibles, uno por cada columna, pero estos hijos son hojas, pues el algoritmo, no persiste después de ellos y se queda con el mejor de los mismos.

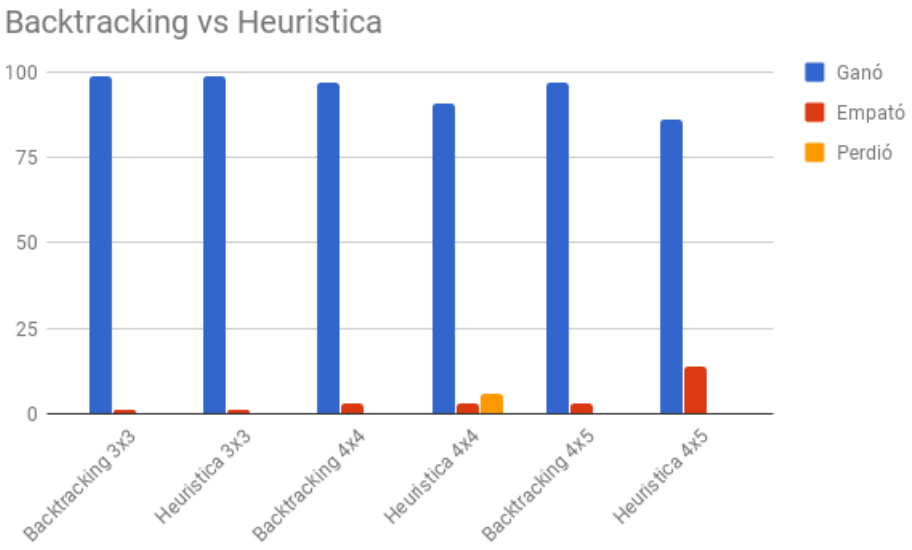
3.4. Experimentación

3.4.1. Pearson



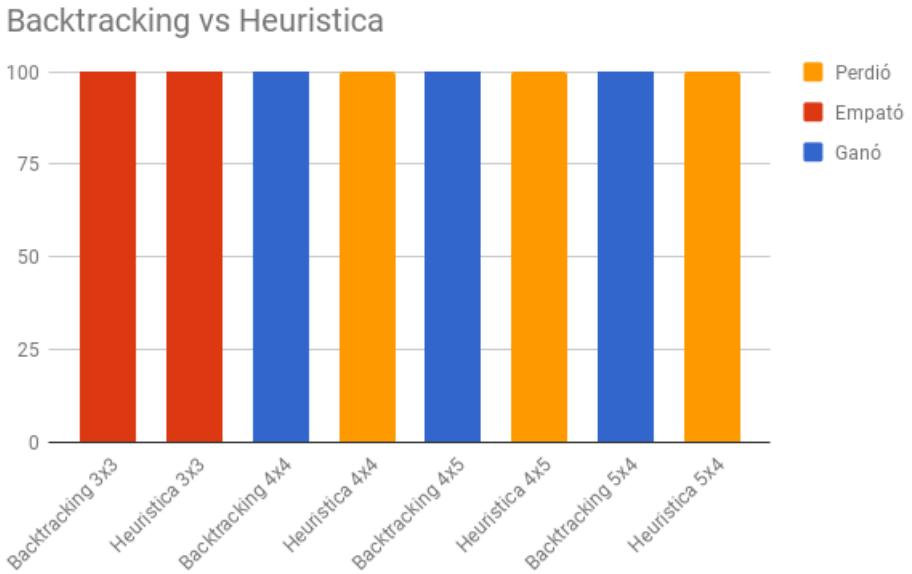
Nuevamente, para corroborar la complejidad propuesta de O(3*columnas*(columnas+filas+)) se corrió el algoritmo 2 veces en cada tablero de 3x3, 4x4, y 5x5 y se tomó el tiempo que toma devolver solo una ficha a jugar. Luego se comparó con la complejidad teórica. Como podemos ver, la relación es casi 1, demostrando que al aumentar una, la otra también aumenta, y viceversa.

3.4.2. Ej1 vs Ej2 contra fijo



En este experimento buscábamos mostrar la diferencia de partidas ganadas, perdidas o empatadas entre el backtracking y la heurística. Lo que hicimos fue correr 100 partidas ambos algoritmos por separado contra un jugador heurístico fijo. Y los corrimos en los tableros de 3 columnas x 3 filas, 4 columnas x 4 filas, y 4 columnas x 5 filas. Para tomar buenos pesos con los que hacer correr la heurística, fueron generados estos 3 jugadores con la técnica de genética:
Pesos para el 3x3: {-69 13 76 -94 -83 67 12 -83 70 -17 -46}
Pesos para el 4x4: {-60 -59 95 -57 -98 -33 -78 -40 -59 -82 95 -88}
Pesos para el 4x4: {-15 32 27 -28 2 -7 64 -66 -53 75 76 -92}
Claramente podemos ver como el algoritmo determinista gana o empata mayor cantidad de partidas que la heurística.

3.4.3. Ej1 vs Ej2 entre ellos



A diferencia del experimento anterior, aquí buscábamos mostrar que pasaba cuando los hacíamos competir entre ellos. Los hicimos correr 100 partidas en los tableros de: 3 columnas x 3 filas, de 4 columnas x 4 filas, y de 4 columnas x 5 filas. Los jugadores de la heurística son los mismos que para el experimento anterior. Sin embargo aquí podemos notar lo siguiente: Para el tablero de 3x3, ambos empatan las 100 partidas. Y esto se debe a que el tablero es muy pequeño para que cualquiera pueda lograr hacer una jugada que lo lleve a ganar, ya que, no olvidemos, que nuestra heurística tapa la jugada del contrario si es que este va a ganar. En los siguientes tableros, el backtracking gana totalmente. Esto lo comprobamos al ver que las partidas se desarrollan de manera en que al algoritmo heurístico no le queda otra opción que colocar una ficha que beneficia al backtracking en el siguiente turno. Cabe notar que como ambos son deterministicos, es decir, la heurística no cambia de pesos, se repite 100 veces la misma partida.

4. Ejercicio 3

4.1. Explicación del algoritmo y Pseudo-Código

Utilizamos el mismo algoritmo que decide en que posición se coloca una ficha, pero le agregamos una nueva función:

Algorithm 4 int nuevoValorGen(vector de vector<int>board, int col, int fila, int jugador, vector<int>pesosPrioridad, vector<int>primDisp, vector<int>cuadrantes, vector<int>columnass)

```
1:  diagizq = pesosPrioridad[0]
2:  diagder = pesosPrioridad[1]
3:  abajo = pesosPrioridad[2]
4:  horiz = pesosPrioridad[3]
5:  while i <rows and board[fila de abajo][misma columna] == jugador do
6:      abajo+1
7:      i++
8:  end while
9:   Asi con el resto de las direcciones
10: res = max(diagder,diagizq, abajo,horiz)
11: res = res + cuadrantes[dondeEstoy(col, fila)]
12:  res = res + columnass[col]
13:  return res
```

Para este ejercicio partimos de un greedy básico y le colocamos 3 vectores de pesos con los que buscaremos cual es el mejor jugador. Mantuvimos la función greedy pero en vez de buscar el próximo valor con la función nuevoValor, la reemplazamos por la anterior descripta, nuevoValorGen. Lo que cambia respecto a la anterior es que los pesos son parámetros y que en base a ellos se decide el puntaje de la ficha a colocar.
Además, se agregó la función horizontaLibre, la cual forma una estrategia mejor, colocando 3 fichas seguidas de forma de poder crear 2 posibilidades para ganar. Esta función se agrego en pos de elevar la cantidad de partidas ganadas. Sin embargo, como dista de darle al algoritmo genético su crédito al encontrar un mejor jugador, lo que hicimos fue incluirla con una probabilidad baja del 10 % de ser usada.

4.2. Grid-search

4.2.1. Explicación del algoritmo y pseudocódigo

Como dijimos antes estamos utilizando el algoritmo descripto en el ejercico 2, con la excepción de que utilizamos un vector de pesos para tomar decisiones sobre donde jugar. Este algoritmo es para refinar o mejorar la proporción de esos pesos. Lo que hacemos es dado 2 pesos los discretizamos en un intervalo (entero). Lo que nos queda es una matriz de los intervalos. Luego para cada par de indices de la matriz, jugamos una cantidad fija de partidas y guardamos el porcentaje de victorias. Una vez que calculamos toda la matriz nos quedamos con el mejor par y ese par va a ser los nuevos pesos. Este proceso lo repetimos para todos los pesos del vector, si la cantidad de pesos es impar hacemos esto para todos los pares de pesos y el último peso lo hacemos separado pero repitiendo el mismo proceso. Para poder correr el algoritmo lo que hacemos es tomar 2 pesos y fijar el resto en una constante aleatoria, una vez que encontramos los valores de esos pesos pasamos a otros 2 pero utilizamos los valores encontrados anteriormente para correr el algoritmo (otra vez fijamos los otros pesos en esa constante aleatoria).

Algorithm 5 int gridSearch(int cantColumnas, int cantPartidas)

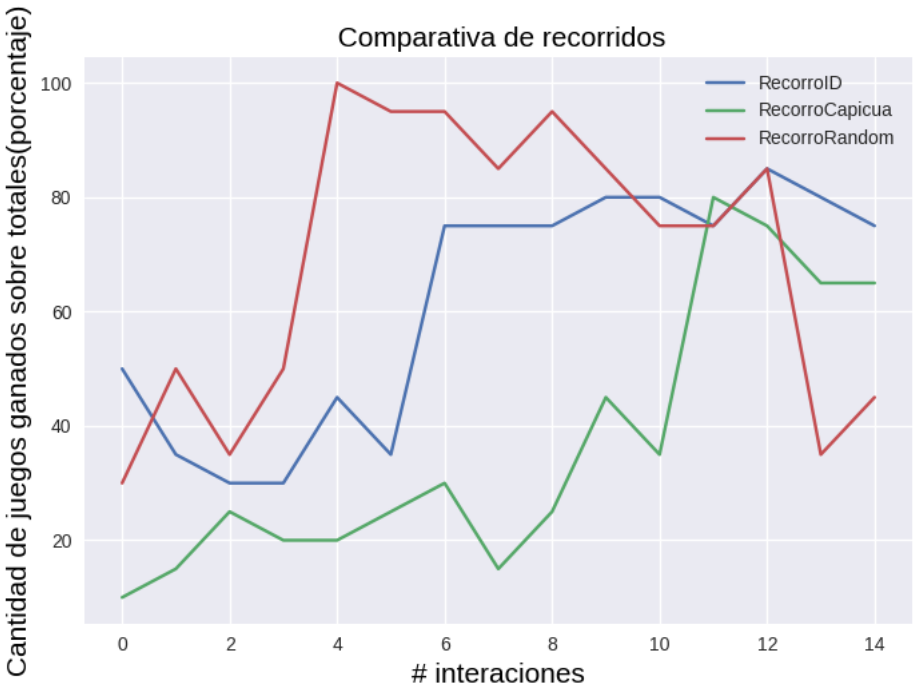
```
1: int r = 0
2: if (cantColumnas %2 == 1) then
3:     r = 8+cantColumnas-1
4: else
5:     r = 8+cantColumnas
6: end if
7:
8: int random k
9: vector de int pesos
10: Cargar los pesos iniciales en el vector pesos
11: for (int z = 0; z <r; z+=2) do
12:     int peso1 = 0
13:     int peso2 = 0
14:     int porcentaje = 0
15:     for (int i = -15; i <15; i++) do
16:         for (int j = -15;j<15; j++) do
17:             Cargar los pesos para correr el grid
18:             Correr el grid
19:             int porcentajeAux = cantPartidasG(cantPartidas)
20:             if (porcentajeAux >porcentaje) then
21:                 peso1 = pesos[z]+i
22:                 peso2 = pesos[z+1]+j
23:                 porcentaje = porcentajeAux
24:             end if
25:         end for
26:     end for
27:     pesos[z] = peso1
28:     pesos[z+1] = peso2
29: end for
30:
31: if (cantColumnas %2 == 1) then
32:     int pesoz = 0
33:     porcentaje = 0
34:     for (int i = -15; i<15; i++) do
35:         Cargar los pesos para correr el grid
36:         Correr el grid
37:         int porcentajeAux = cantPartidasG(cantPartidas)
38:         if (porcentajeAux >porcentaje) then pesoz = pesos[r] + i; porcentaje = porcentajeAux;
39:         end if
40:     end forpesos[r] = pesoz;
41: end if
```

A la hora de cargar los pesos usamos el valor de z y k para cargar el input para grid, la manera en que lo hacemos es la misma que describimos antes. La función cantPartidasG toma la cantidad de partidas jugadas y devuelve el porcentaje de victorias para ese vector de pesos, luego utilizamos ese porcentaje contra el mejor hasta el momento. Al finalizar la matriz nos quedamos con el mejor porcentaje y guardamos esos pesos en el vector pesos. El último if sirve para cuando tenemos un número de impar de parámetros, esto lo sabemos por la cantColumnas, entonces nos falta recorrer el último peso, para ello hacemos lo mismo que antes pero para este peso solo.

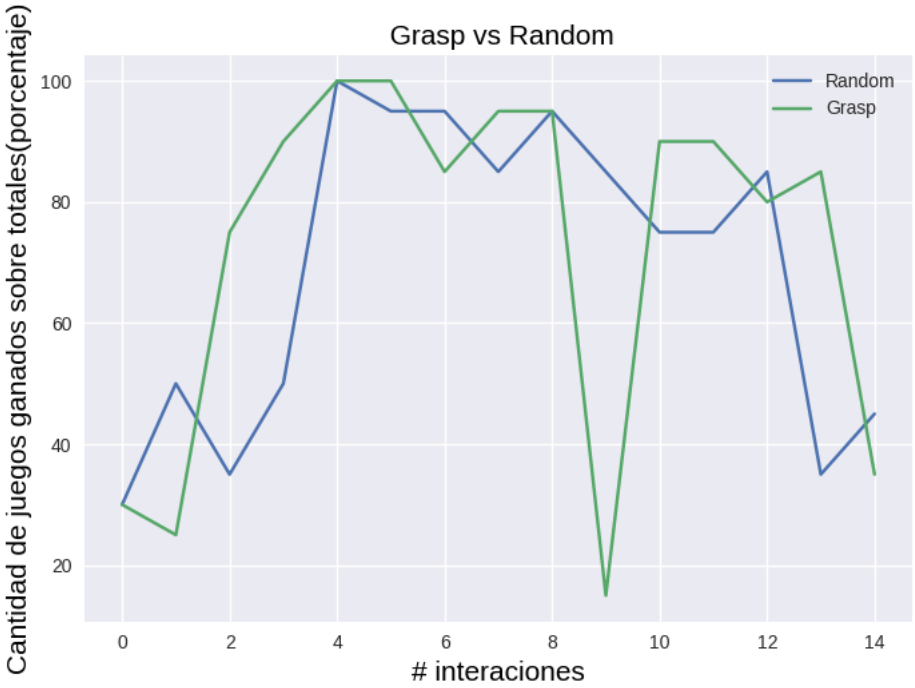
4.2.2. Experimentación

El algoritmo que planteamos antes recorre los pesos de izquierda a derecha, entonces quisimos ver que pasaba cuando recorríamos de otra manera. Para esto planteamos 2 opciones más una random, donde el orden en que recorremos los pesos es aleatorio, y otro capicúa, donde los pesos los recorremos de los extremos hacia el centro. Para realizar los experimentos utilizamos el tablero de 4 en línea (7 columnas, 6 filas y 21 fichas) y tomamos 20 partidas por cada par de pesos. Después para cada una de estas formas de recorrer los pesos hacemos un

total de 15 repeticiones de este algoritmo dado unos pesos aleatorios (los pesos pueden ser tanto positivos como negativos) y el intervalo que estamos tomando es de 30 ([-15,15]). Además nuestro jugador siempre mueve segundo y esta jugando contra un jugador con los pesos fijos.



Lo que podemos ver es que el recorrido Random no solo alcanza un máximo sino que crece más rápido que el resto de funciones. Además el recorrido Capicúa crece más rápido que el recorrido ID, además de que alcanza un pico más alto. Entonces podemos concluir que el recorrido Random es la mejor forma de recorrer los pesos para hacer el proceso descripto anteriormente. El motivo por el cual el gráfico varía es porque una vez que encuentra un máximo local eventualmente se aleja de ese máximo.
Para el siguiente experimento quisimos ver que pasaba cuando teníamos una matriz grande donde no podíamos recorrerla toda, o sea, donde la discretización de los pesos es muy grande. Para esto usamos Grasp para generar un vector aleatorio de incrementos. Lo que hacemos es tomar uno de esos incrementos y sumarlo al peso que tenemos, luego con búsqueda local vemos un intervalo de ese peso y nos quedamos con el mejor de ellos. La forma en que recorremos los pesos es la misma que hicimos antes (izquierda a derecha). Para este experimento tomamos 2 incrementos usando grasp y tomamos el mismo intervalo que veníamos tomando con la misma cantidad de repeticiones. Esto lo comparamos contra el mejor del experimento anterior, o sea, el Random.



Lo que podemos ver es que el grasp con búsqueda local alcanzamos un máximo pero el gráfico tiene muchos altibajos, esto se debe a que como no estamos recorriendo toda la matriz dependemos de que la submatriz que recorremos tenga algún punto alto o se encuentre cerca de algún máximo local.

4.3. Genética

Para el algoritmo genético básico comenzamos generando una población de 10 jugadores, rellenando el vector de pesos de cada uno con números randoms de -100 a 100. Luego llamamos a la función "soy adoptado", la cual se encargará de darnos la siguiente población a correr.

Algorithm 6 vector de vectores de int soyAdoptado(vector de vector de int población)

```
1: vector<float >rank
2: vector de vector de int poblacionaux = población
3: for int i = 0; i <jugadores; i++ do
4:     vector<int>pesos = población[i]
5:     Levanto cada jugador pasado por parámetro, uno a la vez
6:     int ganadas = 0
7:     int perdidas = 0
8:     int empatadas = 0
9:     Corremos el ej3 contra el jugador fijo unas 100 veces en el tablero del 4-Linea
10:    leo y cuento cuantas partidas ganó, empató y perdió
11:    float aux = (float) perdidas/ (float) 100
12:    rank.push_back(aux)
13: end for
14: int padre1 = maxVector(rank)
15: if (rank[padre1] <= elMejor) then
16:    Me guardo al mejor y su vector de pesos
17: end if
18: vector<int>padre1v = poblacion[padre1]
19: //Crossover
20: for int i = 0; i <jugadores; i ++) do
21:    vector<int>hijo = poblacionaux[i]
22:    for (int i = 0; i <cantidadDePesos; i+=2 do
23:        if probabilidad del 50 % then
24:            hijo[i] = padre1v[i]
25:            hijo[i+1] = padre1v[i+1]
26:        end if
27:    end for
28:    población[i] = hijo
29: end for
30: //Mutación
31: for int i = 0; i <jugadores; i++ do
32:    vector<int>pesos = población[i]
33:    for int i = 0; i <cantidadDePesos; i++ do
34:        if probabilidad del 10 % then
35:            if  probabilidad del 50 % then
36:                pesos[i] = random positivo
37:            else
38:                pesos[i] = random negativo
39:            end if
40:        end if
41:    end for
42: end for
43: return poblacion
```

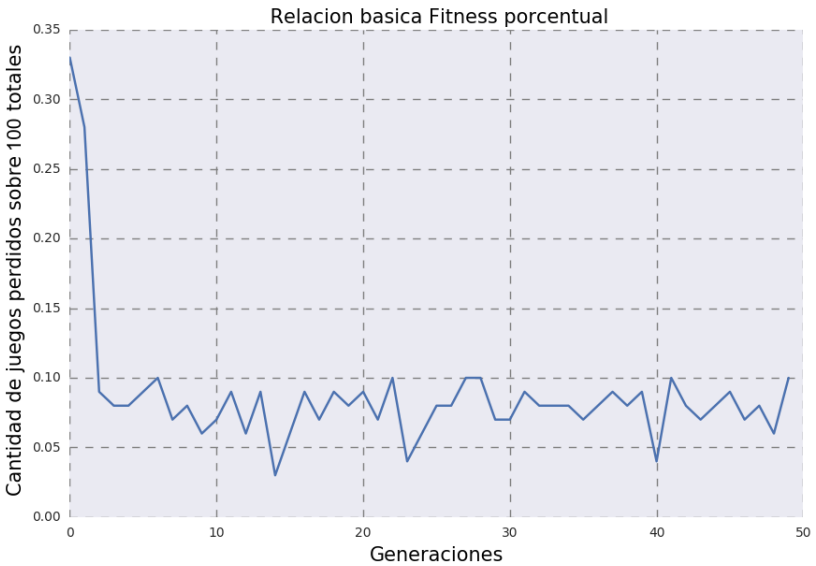
La función MaxVector buscará el máximo dentro del vector rank que posee lo que dio la fitness por cada jugador. ElMejor es una variable global que se guarda cual es el mejor fitness de jugador de todas las generaciones, y elMejorv es el vector de pesos del mismo.
En todas las corridas se compitió contra un jugador con pesos fijos, siendo este, el primero en jugar siempre y para no tener siempre la misma partida, se le dio una probabilidad de tomar en cuenta o no uno de sus vectores de pesos.

Parámetros Básicos:
El método de Corte seleccionado fue de 50 Generaciones.
El método de selección es ponderado según la función de fitness actual.
Crossover: Tomamos al máximo de esa generación y lo cruzamos con el resto de jugadores, teniendo cantidad de pesos/2 puntos de corte. Es decir, tomaremos cada 2 pesos, y con probabilidad del 50 % dejaremos los pesos del máximo de la generación, o del jugador original. De esta forma, si todos los pesos se copian del máximo, este se verá replicado en la siguiente generación, y de la misma manera, si ninguno se copia, se replicará el jugador original a la siguiente generación.
Por ultimo mutamos cada uno de los pesos de cada jugador de la población obtenida con probabilidad del 10 %.
Por Ejemplo:

Genetico	Basico
Terminacion:	50 Generaciones
Seleccion:	Ponderada
Fitness:	Cantidad de Partidas perdidas sobre totales
CrossOver:	7 puntos de corte
Mutacion:	0.1 de probabilidad

4.3.1. Fitness porcentual vs puntaje

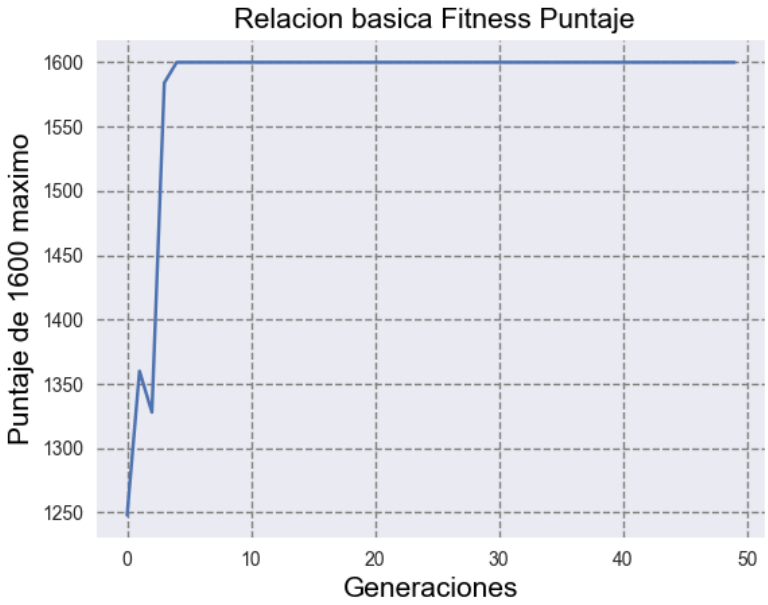
Para este experimento las variables fueron: Cada jugador corre contra nuestro jugador de pesos fijos 100 partidas. El método de corte son 50 generaciones para ambas funciones. La probabilidad de mutación es de 0.1. El crossover se realiza de la manera descripta anteriormente. Y por ultimo la población es de 10 jugadores. En este experimento buscábamos relacionar las 2 funciones de fitness y elegir cual seria la mejor. Como el C-Linea no posee un jugador óptimo, nuestro objetivo se dividió en 2: Queríamos un jugador que perdiera lo menos posible? o Queríamos el que ganara lo más posible? Estas 2 preguntas se ven plasmadas en las 2 funciones de fitness.
La primera que vemos a continuación toma para cada jugador, el numero de partidas perdidas sobre las 100 jugadas, y coloca en el gráfico al mejor de cada generación, siendo 0.0 el mejor resultado posible a obtener.



Y la siguiente, crea un puntaje y asigna:

Partida Ganada	15 puntos
Partida Empatada	10 puntos
Partida Perdida	-1 punto

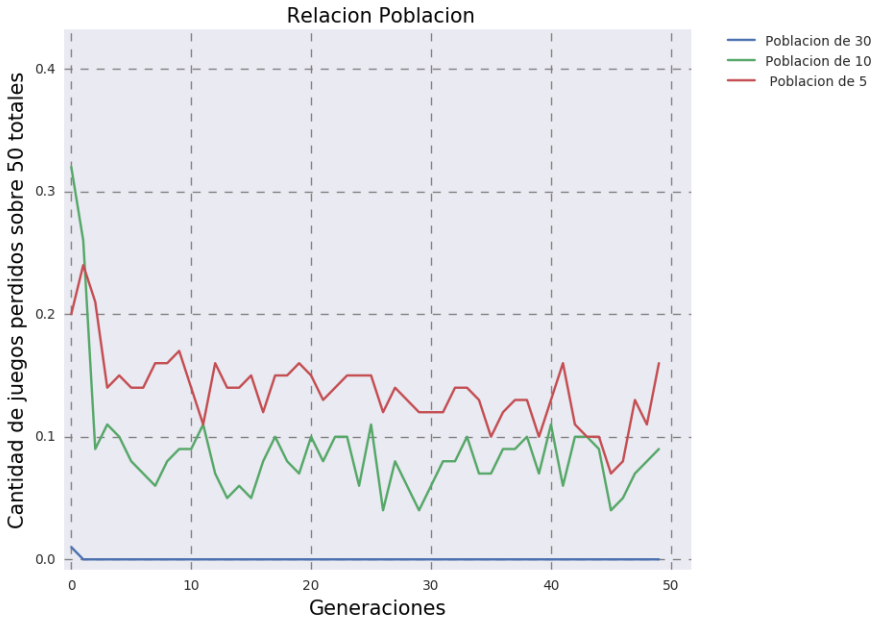
Para evitar problemas con números negativos, le sumamos 100 al total, dando así el mínimo puntaje posible 0 y el máximo, 1600. Igual que la fitness porcentual, muestra al mejor jugador de cada generación:



A diferencia de lo mostrado por el paper del TaTeTi, a nosotros nos dió un resultado más efectivo tomar la fitness de puntaje. Esto se debe gracias a que la fitness de perdidas sobre totales no discrimina entre partidas empatadas o ganadas, consiguiendo así jugadores que a la larga pueden no ser tan buenos, ya que muchas partidas empatadas contra un jugador más apto podrían convertirse en partidas perdidas.

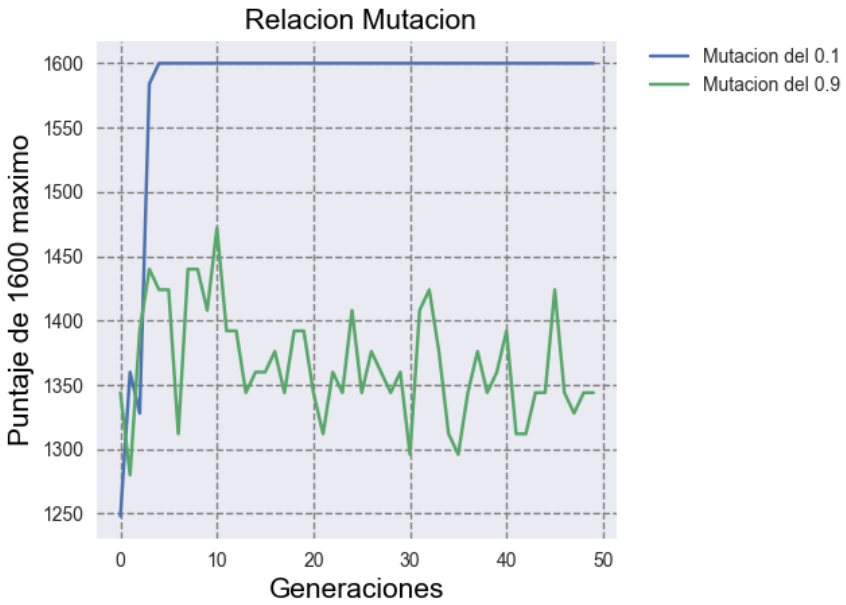
4.3.2. Población

Con este experimento queríamos analizar como varia la cantidad de jugadores en la población con respecto al semi-óptimo de las primeras generaciones. Como sabíamos que la primera generación de cada función es random, pudimos comprobar nuestra hipótesis: Al tener más cantidad de jugadores, se obtiene una mayor probabilidad de comenzar con un buen jugador dentro de las primeras generaciones. Como vemos con la generación de 30 jugadores, en la 2da generación ya consigue el óptimo y se queda con él. Sin embargo, a la población de 5 jugadores le ocurre exactamente lo contrario. Y como vemos en el gráfico, las 50 generaciones no fueron suficientes para que obtuvieran un óptimo.



4.3.3. Mutación

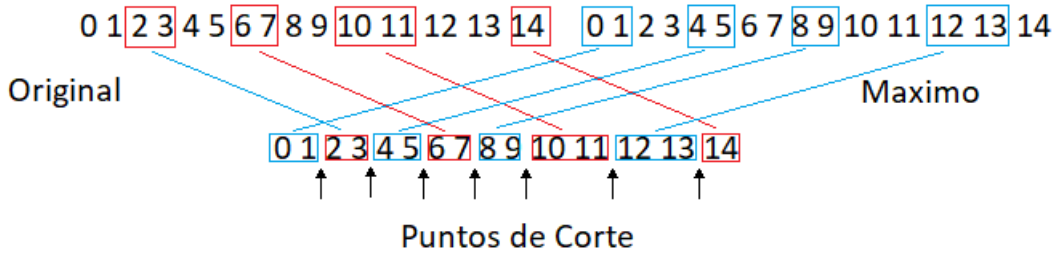
En este experimento buscábamos ver la diferencia entre mutar con mucha probabilidad o con poca, cada peso de cada jugador de la generación. Para todos los experimentos tomamos niveles bajos de probabilidad, es decir, un 10 %. Ahora, queríamos ver que ocurría si nos íbamos al otro extremo, tomando de probabilidad un 90 %. Como vemos en el gráfico, al tener más probabilidad de mutar, la generación pierde homogenidad, que si bien no es algo malo de por si, pues podríamos eventualmente conseguir un nuevo máximo local, sino que también se pierde al posible buen jugador de la generación anterior.



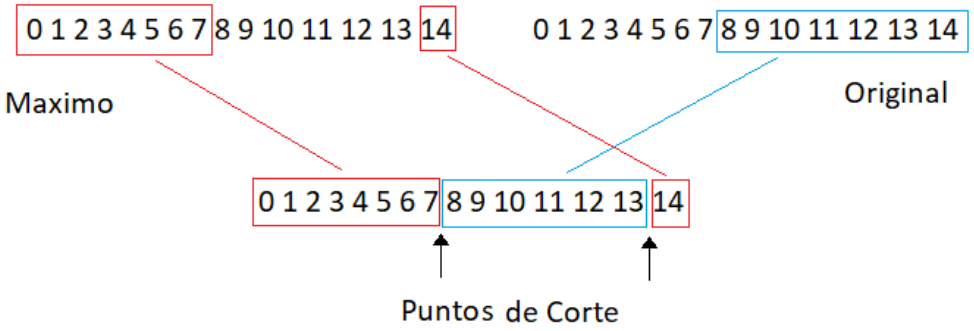
Parecería luego que seria bueno tener una probabilidad de mutación nula. Sin embargo, esto no es cierto, pues puede ocurrir que en el área de búsqueda actual no se encuentre un buen jugador y gracias a ella se puede evitar el estancamiento y lograr conseguir nuevos máximos locales.

4.3.4. Crossover

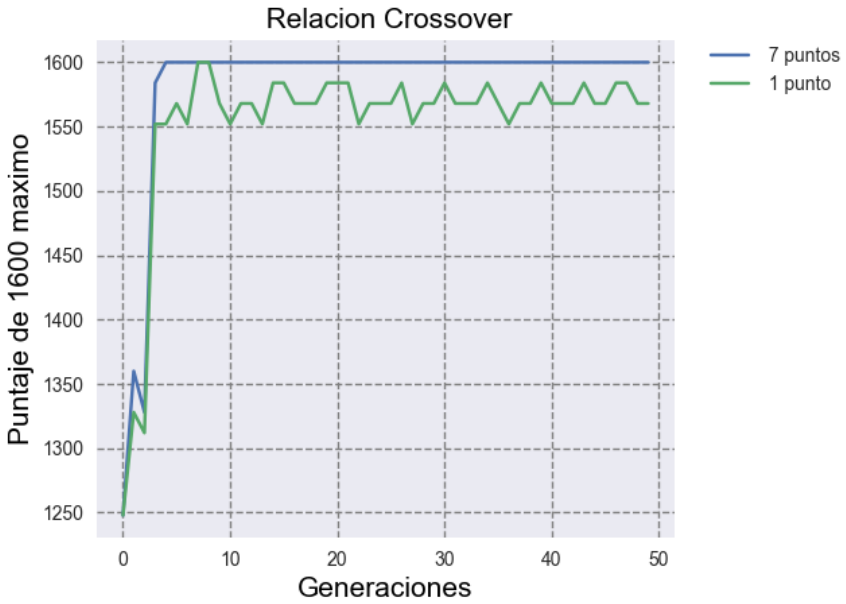
En este análisis modificamos los puntos de corte del crossover: Tomamos uno que posee 7 puntos de corte. El algoritmo elije tomar un grupo de 2 nuevos pesos del vector que fue el máximo de la generación pasada, con probabilidad de 50 %, y si no, deja los pesos del vector original.



En el segundo crossover utilizamos 2 puntos de corte, 1 justo por la mitad del vector de pesos como muestra la figura y para el ultimo peso del vector dejamos el original. Notar que si los pesos fueran pares, se dejarían los 2 hasta 7 pesos del original, y luego 7 del padre, y así sucesivamente.



Con este experimento buscábamos ver como influye la forma en que tomamos los pesos del padre, y si el crossover elegido era más o menos destructivo, en cuanto a lo que hereda del padre máximo, sea un beneficio, y cuanto mayor este último, mejor.



Como vemos en el gráfico, al tener 7 posibilidades de obtener beneficios del máximo de la generación pasada, más homogénea se volverá la población, ya que posee más herencia de su padre. En cambio, al tener un solo corte, y la probabilidad del 50 % de no heredar del padre, las generaciones se volverán más dispersas, perdiendo así al buen candidato que habían obtenido, mostrándonos un claro ejemplo de crossover destructivo.

4.3.5. Selección

A la hora de comparar selecciones nos volcamos por la Ponderada y la simple o random:

En este gráfico buscábamos ver plasmada la hipótesis que teníamos. Si el método de selección es puramente azaroso, puede no heredarse beneficios en la nueva generación. Como vemos, en esta corrida, el algoritmo de selección simple no logra mantener a su buen jugador en la siguiente generación, y por eso varia entre buenos y malos jugadores. En cambio, el método de selección ponderada, va seleccionando lo bueno de la generación anterior, y si logra pasar el crossover, se mantendrá en la nueva, apreciándose así en el gráfico.

5. Conclusiones

En este trabajo práctico pudimos ver la diferencia entre correr un algoritmo totalmente deterministico, el cual, si hay una forma de ganar, o de empatar en peor caso, la va a encontrar, a costas del tiempo que le tome. Y a diferencia de esto, vimos la heurística, que posee el mismo objetivo, pero al no ser exacta, no logra con su cometido el 100 % de las veces. Para ello, se nos pedía buscar los mejores candidatos que hicieran que la misma se acercara al jugador óptimo. A diferencia del paper propuesto como referencia, nuestro juego no posee un óptimo, y por lo tanto fuimos modificando las variables en juego para conseguir acercarnos al mejor jugador posible. También se nos ocurrió que la forma de considerar los datos del tablero, juegan un papel crucial en este tipo de problemas ya que una u otra forma se acercaran más al óptimo. En el paper del Tateti pudieron encontrar las 857 jugadas posibles y trabajar con ellas. Sin embargo nuestro juego puede tener más de 857 posibles partidas dependiendo del tamaño del tablero y por lo tanto no fue posible esta consideración.

5.1. Jugadores Pedidos para 4-Linea

Grid-Search = { 65 -62 -2 -21 -93 13 23 -46 -11 -32 -16 -24 -64 41 -31 }
Genética = {-20 -26 -53 43 24 17 71 43 -94 -47 22 91 -96 -2 -56 }