DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Darwin DT en el mundial 2018

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2018

Grupo Estrellitas

Integrante	LU	Correo electrónico
Salinas, Pablo	456/10	salinas.pablom@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolás	46/12	nbverboven@gmail.com
Hofmann, Federico	745/14	federico2102@gmail.com

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

Índice

1. Introducción

En este trabajo modelaremos un partido de fútbol, con dos equipos de 3 jugadores cada uno, compitiendo por conseguir más goles que el otro equipo. Nuestro objetivo consistirá en diseñar buenas estrategias para el comportamiento de los equipos. Para ello, modelaremos los distintos aspectos del juego (la cancha con sus dimensiones, los jugadores, la pelota y todas las reglas de movimientos e interacciones entre ellos), y una vez modelado el escenario, aplicaremos diferentes técnicas algorítmicas heurísticas para conseguir buenas estrategias.

Para lograr esto, implementaremos una función que para cada estado del juego (compuesto por las posiciones de los jugadores de ambos equipos y la posición de la pelota) le asigne un puntaje. De esta forma, a cada momento los equipos evaluarán todas las posibles jugadas a ejecutar, eligiendo la que maximice dicha función. Esta función dependerá de ciertos valores parametrizados que le asignarán, intuitivamente, una importancia relativa a cada aspecto del estado del juego. Con esto en mente, el eje central de este trabajo estará en hallar buenos valores para estos parámetros, buscando guiar al equipo para que tome las mejores decisiones posibles.

Una vez definida la función parametrizada, nos valdremos de dos técnicas diferentes para buscar los mejores valores para los parámetros de la misma: Grid-Search y Algoritmos Genéticos.

Finalmente, realizaremos una experimentación detallada comparando las distintas técnicas y estrategias implementadas. Por último, daremos un equipo (es decir, nuestra función para evaluar el estado de un juego, junto con los valores hallados para los parámetros) que sepa jugar en un tablero de 10x5 durante el transcurso de 250 turnos discretos.

1.1. Algoritmos genéticos

La idea principal de estos algoritmos es, dada una población inicial, separar a los individuos mas aptos y combinar propiedades de los mismos creando así nuevos individuos que conserven, preferentemente, las mejores propiedades de la anterior generación. Para determinar los individuos a seleccionar, se debe establecer una función de puntuación para los mismos (función de fitness). Luego, para generar a los individuos de la nueva generación, se les realiza operaciones como mutaciones y cruzamientos aleatorios.

Los algoritmos genéticos han sido aplicados exitosamente en gran variedad de tareas de aprendizaje y en problemas de optimización.

1.1.1. Consideraciones biológicas

En el núcleo de una célula se encuentra su material genético, compuesto por moléculas de ácido desoxirribonucléico (ADN), cada una de las cuales se organiza en una estructura llamada cromosoma. Cada molécula de ADN está conformada por macromoléculas llamadas nucleótidos; los hay de 4 tipos diferentes (adenina, timina, citosina y guanina) y el orden en el que se presentan determina que existan algunas secciones, llamadas genes, que contienen información para sintetizar proteínas.

El conjunto de todos los cromosomas, es decir, de toda la información genética de un individuo se llama genoma y el conjunto de genes contenidos en el genoma se denomina genotipo. En los animales -entre ellos, el ser humano- existen dos copias de cada gen y, por lo tanto, dos juegos de cromosomas.

El fenotipo (características observables) de un individuo surge como una combinación entre el material genético que recibe de sus progenitores -en particular, por su genotipo- y la interacción con el medio ambiente en que se desarrolla.

En el proceso de replicación del ADN, pueden (y suelen) ocurrir errores. El resultado, conocido como mutación, es una secuencia de nucleótidos distinta a la de la molécula original y, si esto se produjera en alguna región que contiene un gen, podría suceder que el cambio en el genotipo se viera reflejado en el fenotipo. La gran mayoría de las mutaciones son silemciosas, pues ocurren en zonas del ADN que no contienen genes. Sin embargo, cuando este no es el caso, pueden ser desfavorables e incluso letales para el organismo o no serlo y, en un pequeño porcentaje de los casos, conferir a sus descendientes alguna ventaja que le permita sobrevivir más fácilmente en su medio.

1.1.2. Definición y factores limitantes

Un algoritmo genético consiste en una función matemática o una rutina de software que toma como entradas a los ejemplares y retorna como salidas cuáles de ellos deben generar descendencia para la nueva generación.

Algunas versiones más complejas de algoritmos genéticos generan un ciclo iterativo que directamente toma a la especie (el total de los ejemplares) y crea una nueva generación que reemplaza a la antigua una cantidad de veces determinada por su propio diseño. Una de sus características principales es la de ir perfeccionando su propia heurística en el proceso de ejecución, por lo que no requiere largos períodos de entrenamiento especializado por parte del ser humano, principal defecto de otros métodos para solucionar problemas, como los Sistemas Expertos.

La aplicación más común de los algoritmos genéticos ha sido la solución de problemas de optimización, en donde han mostrado ser muy eficientes y confiables. Sin embargo, no todos los problemas pudieran ser apropiados para la técnica, y se recomienda en general tomar en cuenta las siguientes características del mismo antes de intentar usarla: Su espacio de búsqueda debe estar delimitado dentro de un cierto rango. Debe poderse definir una función fitness que nos indique cómo es de buena o mala es una cierta respuesta. Las soluciones deben codificarse de una forma que resulte relativamente fácil de implementar en la computadora.

El primer punto es muy importante, y lo más recomendable es intentar resolver problemas que tengan espacios de búsqueda discretos aunque éstos sean muy grandes. No sería conveniente la utilización de algoritmos genéticos en las siguientes situaciones:

- Si se requiere llegar forzosamente al máximo global en el caso de que estemos maximizando una función.
- Si conocemos la función de optimización.
- Si el problema está muy delimitado y se presta a un tratamiento analítico (funciones de una variable).
- Si la función es suave y convexa.
- Si el espacio es limitado. En este caso es mejor enumerar todas las soluciones posibles.

1.1.3. Uso para el modelado de problemas

Es necesario codificar de alguna manera el dominio del problema para obtener estructuras manejables que puedan ser manipuladas por el algoritmo genético. Cada una de estas estructuras constituye el equivalente al genotipo de un individuo en términos biológicos. El elemento del dominio del problema al que se mapea este genotipo es el análogo al fenotipo. Es frecuente que el código de los elementos del dominio del problema utilice un alfabeto binario (0's y 1's).

Una vez que se ha definido la manera de codificar los elementos del dominio del problema y se conoce la forma de pasar de un elemento a su código y viceversa, es necesario fijar un punto de partida. Los algoritmos genéticos manipulan conjuntos de códigos (poblaciones de códigos) en generaciones sucesivas. El algoritmo se encargará de favorecer la aparición en la población de códigos que correspondan a elementos del dominio que estén próximos a resolver el problema. Es decir, dicho algoritmo recibirá como entrada una población de códigos y a partir de ésta generará nuevas poblaciones, donde algunos códigos desaparecerán mientras que otros, que se mapean en mejores soluciones posibles, aparecen con más frecuencia hasta que se encuentra una satisfactoria o hasta que se cumpla alguna otra condición de terminación. Los elementos de la población serán llamados individuos y a los códigos se les denominará indistintamente cromosomas, genotipo, genoma o código genético.

En la naturaleza hay individuos más aptos que otros para sobrevivir y, para reflejar esto, en los algoritmos genéticos es necesario establecer algún criterio que permita decidir cuáles de las soluciones propuestas en una población son mejores respecto del resto de las propuestas y cuáles no lo son.

Para determinar cuáles de estos individuos corresponden a buenas propuestas de solución y cuáles no, es necesario calificarlos de alguna manera. Cada individuo de cada generación de un algoritmo genético recibe una calificación o una medida de su grado de adaptación (fitness). éste es un número real no negativo que será más grande cuanto mejor sea la solución propuesta por dicho individuo.

Una vez calificados todos los individuos de una generación, el algoritmo debe seleccionar a los individuos más calificados para que tengan mayor oportunidad de reproducción. De esta forma se incrementa la probabilidad de tener individuos 'buenos' en el futuro. Si de una determinada generación se seleccionaran sólo aquellos con una calificación mayor o igual que cierto número c para pasarlos a la siguiente generación, es claro que en ésta la calificación superará c y por tanto al promedio de la generación anterior. La selección explota el conocimiento que se ha obtenido hasta el momento, procurando elegir lo mejor que se haya encontrado, elevando así el nivel de adaptación de toda la población.

La cruza de los códigos genéticos de individuos exitosos favorece la aparición de nuevos individuos que heredan de sus ancestros características deseables. En el contexto de los algoritmos genéticos reproducirse significa que, dados dos individuos seleccionados en función de su grado de adaptación, éstos pasen a formar parte de la siguiente generación o, al menos, mezclen sus códigos genéticos para generar 'hijos' que posean un código híbrido. Es decir, los códigos genéticos de los individuos se cruzan.

Ocasionalmente, algunos elementos del código de ciertos individuos de un algoritmo genético se alteran a propósito. Éstos se seleccionan aleatoriamente en lo que constituye el símil de una mutación. El objetivo es generar nuevos individuos que exploren regiones del dominio del problema que probablemente no se han visitado aún. Esta exploración no presupone conocimiento alguno. Aleatoriamente se buscan nuevas soluciones posibles que quizá superen las encontradas hasta el momento. Esta es una de las características que hacen aplicables los algoritmos genéticos a gran variedad de problemas: no presuponer conocimiento previo acerca del problema a resolver ni de su dominio, no sólo en la mutación sino en el proceso total. De hecho, el problema a resolver sólo determina la función de evaluación y la

manera de codificar las soluciones posibles. El resto de los subprocesos que constituyen el algoritmo son independientes y universalmente aplicables.

1.1.4. Principales áreas de aplicación

1. *Análisis discriminante:* El análisis discriminante forma parte del conjunto de técnicas estadísticas diseñadas para resolver el problema de clasificación. Se caracteriza por estudiar la relación entre una variable categórica dependiente (el grupo de clasificación) y un conjunto de variables reales (posiblemente vectoriales) independientes. Estas describen a cada uno de los individuos a clasificar. A través de esta técnica, se construyen reglas de decisión que permiten discriminar o separar grupos mediante funciones de las variables observadas, minimizando la probabilidad de clasificación errónea, o bien maximizándola si la clasificación es correcta. Las reglas más usuales se construyen a partir de un modelo probabilístico, y de muestras de entrenamiento.

2. *Problema del viajante de comercio:* El problema del viajante de comercio, también denominado TSP (Travelling Salesman Problem), consiste en, dada una colección de ciudades con distancias entre ellas, determinar el recorrido de coste mínimo visitando cada ciudad exactamente una vez y volviendo al punto de partida. A lo largo de los años el problema del viajante de comercio ha ocupado la mente de numerosos investigadores por diversos motivos. En primer lugar, el TSP es un problema muy sencillo de enunciar, pero muy difícil de resolver. En segundo lugar, el TSP es aplicable a una gran variedad de problemas de planificación. Finalmente, se ha convertido en una especie de problema test, es decir los nuevos métodos de optimización combinatoria son a menudo aplicados al TSP con objeto de tener una idea de sus potencialidades.

3. *Planificación de actividades:* Los problemas de planificación se encuentran en una gran variedad de campos, incluyendo la manufactura y el servicio industrial. Los problemas de planificación son numerosos y variados. En términos más amplios, la planificación implica el reparto de recursos durante un período de tiempo para llevar a cabo un conjunto de actividades. En general, no hay un algoritmo general que garantice dar una solución óptima y hacerlo en un tiempo polinomial. Por tanto, los problemas de planificación son principales candidatos para la aplicación de la tecnología de la Inteligencia Artificial.

4. *Entrenamiento de redes neuronales profundas:* Los algoritmos genéticos son una buena alternativa a otros algoritmos utilizados en esta área, los cuales se basan en métodos de gradientes, parecidos a diferencias finitas. Si bien podria esperarse que los AGs funcionen peor, por el hecho de ser tan simples y no estar basados en gradientes, resultan ser una alternativa altamente competitiva, y en muchos casos con mejor funcionamiento que los metodos utilizados tradicionalmente en tareas de RL (reinforcement learning), los cuales sí estan basados en gradientes.

5. *Locomoción humanoide:* Los AGs pudieron resolver el desafiante punto de referencia de control contínuo de locomoción humanoide. Si bien estos algoritmos produjeron robots que podian caminar bien, tomó aproximadamente 15 veces mas tiempo que utilizando algoritmos de ES (evolution strategies, otra rama de algoritmos evolutivos), y se obtuvo un rendimiento ligeramente peor que con los ultimos mencionados.

2. Desarrollo

En esta sección se explicarán los algoritmos diseñados para resolver los problemas enunciados en la sección previa.

2.1. Modelado del tablero

En esta parte describiremos la manera en la que son seteados todos los valores iniciales del juego, como el tamaño de la cancha, la posición de los jugadores, la posición de la pelota. Al mismo tiempo, explicaremos cómo fueron programadas las reglas del juego (movimientos válidos para los jugadores, cómo quitar la pelota, patearla, etc).

2.1.1. Valores iniciales

Cuando comienza el juego se crea una clase *player_controller*, dentro de la cual se setean los valores iniciales a partir de los datos recibidos por la entrada estándar. Se configurarán aquí el tamaño de la cancha, la duración del partido (medida en cantidad de jugadas), de que lado jugará cada equipo, los jugadores que formaran parte de dichos equipos y sus posiciones iniciales.

Una vez hecho esto se llama, dentro de la misma clase, a la función *jugar*. Esta función se encargará de, en cada nueva jugada, leer y actualizar el estado del tablero, llamar a la función encargada de realizar un nuevo movimiento, y finalmente realizar dicho movimiento.

2.1.2. Reglas del juego

El juego consiste en una cancha de fútbol de M filas y N columnas donde:

1. $M \geq 3$ es impar.
2. N es par y $N \geq 2M$.
3. Cada posición es una tupla $(fila, columna)$ donde $fila \in [0, M)$ y $columna \in [0, N)$.
4. Un casillero puede contener:
   - Un jugador solo
   - Un jugador con una pelota
   - La pelota sola
   - Dos jugadores de equipos contrarios sin pelota
   - Dos jugadores de equipos contrarios donde uno de los dos jugadores tiene la pelota
5. Los jugadores pueden desplazarse moviéndose de un casillero a otro siempre y cuando estos casilleros compartan un borde o una esquina.
6. En caso de que un jugador se encuentre en el mismo casillero que la pelota pero no tiene la posesión de la misma, si no hay otro jugador en dicho casillero, entonces el jugador pasa automáticamente a poseer la pelota.
7. En caso de que dos jugadores se encuentren en el mismo casillero y uno de ellos tiene la pelota, automáticamente el jugador sin la pelota intenta quitarle la pelota al otro jugador y tiene la probabilidad mostrada en 1 de lograrlo.
8. En caso de que dos jugadores se encuentren en el mismo casillero y la pelota también está en ese casillero pero sin posesión, automáticamente pasan a disputarse la pelota. Para esto, se calcula si uno de los dos jugadores se quedó con la pelota utilizando la probabilidad mostrada en 2; si lo logra, él se queda con la pelota y, si no, el otro jugador se queda con la pelota.
9. El juego tiene una duración en cantidad de pasos. Cada equipo debe decidir en cada paso que movimiento realiza cada uno de sus jugadores. La duración del partido está dada por una variable S.
10. Los movimientos posibles para un jugador son:
    - Moverse a un casillero vecino dentro del tablero, para lo cual, debe indicar una dirección de las indicadas en la Figura 1.
    - Tirar la pelota en una dirección con una cierta fuerza (medida en cantidad de pasos). La dirección de la pelota es una de las 8 direcciones mostradas en la figura 1 y la cantidad de pasos indica durante cuanto tiempo la pelota va a moverse, terminados esos pasos la pelota quedará quieta en el lugar donde se encuentre. **Nota:** La pelota se mueve en cada paso dos casilleros en la dirección indicada en vez de un solo casillero como los jugadores, además no está permitido indicar una cantidad de pasos mayor a $\frac{M}{2}$ ni una cantidad de pasos que dejen a la pelota fuera del tablero.
11. Un jugador intercepta una pelota libre solo si en algún momento ambos (jugador y pelota) se encuentran en un mismo casillero siguiendo las reglas marcadas en 6 y 8 según corresponda. También intercepta la pelota si la misma va de un casillero A a un casillero B y el jugador se encuentra en el casillero intermedio tanto cuando la pelota está en el casillero A como cuando la pelota está en el casillero B. En otras palabras, el jugador está en el camino de la pelota y el mismo se queda quieto mientras la pelota pasa sobre el. Si hay dos jugadores en el casillero intermedio, se utiliza la formula 2 para ver quien se queda con la pelota.
12. El juego comienza con un marcador en 0 para ambos equipos y cada vez que la pelota entra en un arco, se suma 1 punto en el marcador del equipo contrario al arco. Cuando termina el partido, el equipo con más puntos gana, en caso de tener ambos equipos el mismo puntaje, el partido es un empate.
13. Los arcos se encuentran en las posiciones $(\lfloor \frac{M}{2} \rfloor - 1, -1), (\lfloor \frac{M}{2} \rfloor, -1), (\lfloor \frac{M}{2} \rfloor + 1, -1)$ para un equipo y $(\lfloor \frac{M}{2} \rfloor - 1, N), (\lfloor \frac{M}{2} \rfloor, N), (\lfloor \frac{M}{2} \rfloor + 1, N)$ para el otro equipo.
14. Los jugadores inician en la posición que el equipo quiera siempre que esté en su mitad de la cancha.
15. El equipo que comienza (aquel cuyo arco está en la columna -1) tiene un jugador en la posición $(\lfloor \frac{M}{2} \rfloor, \frac{N}{2} - 1)$ que tiene posesión de la pelota.

16. Cada vez que un equipo mete un gol, los jugadores vuelven a la posición inicial pero el equipo al que le metieron un gol tiene la pelota y por ende tiene un jugador en la posición $(\lfloor \frac{M}{2} \rfloor, 2-1)$ si el gol fue en el arco de la columna N o $(\lfloor \frac{M}{2} \rfloor, \frac{N}{2})$ si el gol fue en el arco de la columna -1.

Las reglas del juego están programadas dentro de la clase *logical_board*. Al realizarse cualquier movimiento nuevo en cualquiera de los dos equipos, la función descripta por el algoritmo 1 será la encargada de validarlo o no, de acuerdo a las reglas anteriormente mencionadas.

$$P_{quitar}(p_{pelota}, p_{quitador}) = X \leq \frac{P_{quite}(p_{quitador})}{P_{quite}(p_{quitador}) + (1 - P_{quite}(p_{pelota}))}, X \sim U(0,1) \tag{1}$$

$$P_{ganar}(p_1, p_2) = X \leq \frac{P_{quite}(p_1)}{P_{quite}(p_1) + P_{quite}(p_2)}, X \sim U(0,1) \tag{2}$$

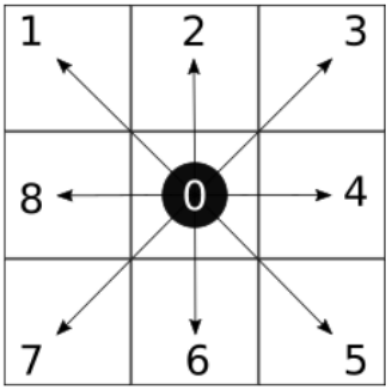

Figura 1: Cada movimiento está asignado a un número. El movimiento 0 indica que el jugador se queda quieto.

Algoritmo 1 Algoritmo para verificar que los movimientos a realizar por un equipo son válidos

Input: Vector de jugadores *team* y vector de movimientos *moves*.
Output: *verdadero* si todos los movimientos en *moves* son válidos y corresponden exactamente a un jugador de *team* y *falso* en caso contrario.

```
1: función isValidTeamMove(team, moves)
2:     result ← verdadero
3:     result ← result ∧ cada jugador j ∈ team tiene asignado exactamente un moviento m ∈ moves
4:     para m ∈ moves hacer
5:         result ← result ∧ m es un movimiento válido
6:     fin para
7:     devolver result
8: fin función
```

2.2. Movimientos de los equipos con el tablero lógico

Para simular cada paso discreto de un partido, definimos el método *makeMove*, dentro de la clase *LogicalBoard*, la cual ejecutará los movimientos de los equipos A y B del paso actual. A su vez, cada una de las acciones que deben ejecutarse sobre el tablero al ejecutar los movimientos de los equipos están modularizadas del siguiente modo:

- *makeTeamMove* se encargará de hacer los movimientos efectivos de un equipo. En el caso de que un jugador tenga la pelota, moverá a la misma con él, y si el movimiento es un pase la marcará como libre y la moverá en la dirección indicada.
- *intercepted* verificará si la pelota fue interceptada.
- $fightBall$ decidirá quién se queda con la pelota en el caso de que dos jugadores la intercepten, de acuerdo a las reglas explicadas.
- $fairFightBall$ resuelve quien se queda con la pelota cuando un jugador que no tenia la pelota se enfrenta contra otro que tenia el control de la misma, resolviendo la disputa según indican las reglas del juego.
- *positionInBoard* verifica si una posición pasada como parámetro pertenece a una posición válida del tablero o no, para asegurar la validez de los movimientos a ejecutar.

• *updateScore* verifica si el último movimiento ejecutado hizo que la pelota entrara al arco, actualizando los puntajes de forma acorde.

A continuación, expresaremos mediante pseudocódigo la lógica completa para ejecutar un paso discreto en función de las jugadas ejecutadas por ambos equipos:

Algoritmo 2 Algoritmo para realizar los movimientos de los dos equipos de un tablero lógico

Input: Tablero lógico *logicalBoard* y dos vectores de movimientos, *movesA* y *movesB*, correspondientes a los que deberán realizar los equipos A y B respectivamente.

Output: Si algún equipo anota un gol como resultado de los movimientos aplicados, el nombre de dicho equipo (*A* o *B* según corresponda). Si ninguno de los dos anota un gol, *NINGUNO*.

```
1: función makeMove(logicalBoard, movesA, movesB)
2:     teamA ← el equipo A de logicalBoard
3:     teamB ← el equipo B de logicalBoard
4:     makeTeamMove(logicalBoard, teamA, movesA)          ▷ Realizo los movimientos de los dos equipos
5:     makeTeamMove(logicalBoard, teamB, movesB)
6:     si ningún jugador estaba en posesión de la pelota entonces
7:         si la pelota fue interceptada por un jugador j entonces
8:             j toma posesión de la pelota
9:         si no si la pelota fue interceptada por dos jugadores j1 y j2 entonces
10:            fairFightBall(j1, j2)
11:        si no
12:            muevo la pelota y vuelvo a revisar si fue interceptada
13:        fin si
14:    si no
15:        si dos jugadores están en la misma posición y uno tiene el control de la pelota entonces
16:            j1 ← el jugador con la pelota
17:            j2 ← el jugador que no posee la pelota
18:            fightBall(j1, j2)
19:        fin si
20:    fin si
21:    result ← updateScore(logicalBoard)
22:    devolver result
23: fin función
```

Algoritmo 3 makeTeamMove

```
1: para todos los jugadores del equipo hacer
2:     si tipo de movimiento = MOVIMIENTO entonces
3:         posición del jugador = movimientos[jugador]
4:         si el jugador tiene la pelota entonces
5:             posición de la pelota = posición del jugador
6:         fin si
7:     fin si
8:     si tiepo de movimiento = PASE entonces
9:         pelota libre = true
10:        dirección de la pelota = dirección del pase
11:        fuerza del pase = cantidad de pasos del movimiento
12:    fin si
13: fin para
```

Algoritmo 4 intercepted

```
comparo posición actual del jugador con su posición anterior
si el jugador se movió entonces
    devolver false
si no
    si posiciónJugador - direcciónPelota = posiciónPelota entonces
        devolver true
    fin si
fin si
devolver false
```

Algoritmo 5 fairFightBall

```
probabilidad jugador 2 = jug2prob/(jug1prob+jug2prob)
si probabilidad random <probabilidad jugador 2 entonces
    jugador 2 pasa a tener la pelota
si no
    jugador 1 pasa a tener la pelota
fin si
trayectoria de la pelota = 0
pelota libre = false
```

Algoritmo 6 fightBall

```
probabilidad de defender = 1 - probabilidad del jugador con la pelota
probabilidad de robar la pelota = normalizar(probabilidad del jugador sin pelota)
si probabilidad random ≤ probabilidad de robar la pelota entonces
    jugador sin pelota pasa a tener la pelota
fin si
```

2.3. Estrategias

En esta subsección explicaremos los diferentes mecanismos que realizamos para obtener buenas estrategias de juego.

2.3.1. Función puntuadora

Creamos una función encargada de darle un puntaje a cada posible estado del juego. Esta función será utilizada mas adelante por distintos equipos, para a partir de esos puntajes obtener los estados mas convenientes y combinaciones entre ellos para generar buenas estrategias.

Para asignar puntajes, consideramos distintos atributos del estado de un partido y a cada uno de ellos le asignamos un 'peso'. Los atributos que consideramos son:

- $ball\_possession$, 1 cuando el equipo tiene la pelota, 0 en caso contrario.
- $ball\_in\_oponent\_possession$, 1 cuando el rival tiene la pelota, 0 en caso contrario.
- $ball\_free$, 1 cuando la pelota está libre, en caso contrario.
- $goal\_distance$, distancia de un jugador de nuestro equipo con la pelota (si es que alguno del equipo la tiene) a nuestro arco.
- $ball\_distance$, distancia de los jugadores a la pelota, cuando la misma está libre.
- $oponent\_with\_ball\_distance$, distancia de los jugadores al jugador del equipo contrario que tiene la pelota.
- *dispersión* distancia entre los jugadores de nuestro equipo.
- $distance\_ball\_oponent\_goal$ distancia de un jugador de nuestro equipo con la pelota (si es que alguno del equipo la tiene) al arco rival.
- $distance\_ball\_our\_goal$ distancia entre un jugador del equipo rival con la pelota (si es que alguno del equipo rival la tiene) a nuestro arco.

Por lo tanto, nuestra función para asignarle puntaje al estado de un tablero se define como:

$$\begin{aligned} puntaje = & k_0 * ball\_possession + k_1 * ball\_in\_oponent\_possession + \\ & k_2 * ball\_free + k_3 * goal\_distance + k_4 * ball\_distance + \\ & k_5 * oponent\_with\_ball\_distance + k_6 * dispersion + \\ & k_7 * distance\_ball\_oponent\_goal + k_8 * distance\_ball\_our\_goal \end{aligned} \tag{3}$$

La lógica para obtener el puntaje de un tablero,tomando en cuenta los atributos que solo aportan puntaje bajo ciertas condiciones (como por ejemplo la distancia al arco rival cuando el equipo tiene la pelota) puede expresarse mediante el siguiente pseudocódigo:

Algoritmo 7 EvaluateBoard

```
1: boardPoints = 0
2: maxDistance = ∛(columnas² + filas²)
3: para jugadores p en mi equipo hacer
4:     dist = distancia(p, pelota)
5:     dist = (maxDistance - dist) / maxDistance
6:     boardPoints = boardPoints + dist * ball_distance
7:     si p tiene la pelota entonces
8:         mejor_dist = distancia entre p y su arco
9:         mejor_dist = (maxDistance - mejor_dist) / maxDistance
10:        boardPoints = boardPoints + mejor_dist * goal_distance
11:    si no
12:        para jugadores op en equipo oponente hacer
13:            si op tiene la pelota entonces
14:                dist = distancia(jugador op, jugador p)
15:                dist = (maxDistance - dist) / maxDistance
16:                boardPoints = boardPoints + dist * oponent_with_ball_distance
17:            fin si
18:        fin para
19:    fin si
20: fin para
21: si mi equipo tiene la pelota entonces
22:    boardPoints = boardPoints + ball_possession
23: si no
24:    si la pelota esta libre entonces
25:        boardPoints = boardPoints + ball_free
26:    si no
27:        boardPoints = boardPoints + ball_in_oponent_possession
28:    fin si
29: fin si
30: dispersión_jugadores = 0
31: para jugadores p en mi equipo hacer
32:    para jugadores q en mi equipo, p ≠ q hacer
33:        dispersión_jugadores = dispersión_jugadores + distancia(p,q)
34:    fin para
35: fin para
36: dispersión_jugadores = dispersión_jugadores / 3*maxDistance
37: boardPoints = boardPoints + dispersión_jugadores * dispersión
38: si mi equipo tiene la pelota entonces
39:    ballDistanceToRivalGoal = distancia del jugador con la pelota al arco rival
40:    ballDistanceToRivalGoal = (maxDistance - ballDistanceToRivalGoal) / maxDistance
41:    boardPoints = boardPoints + ballDistanceToRivalGoal * distance_ball_to_oponent_goal
42: fin si
43: si el equipo rival tiene la pelota entonces
44:    ballDistanceToOurGoal = distancia del jugador con la pelota a nuestro arco
45:    ballDistanceToOurGoal = (maxDistance - ballDistanceToOurGoal) / maxDistance
46:    boardPoints = boardPoints + ballDistanceToOurGoal * distance_ball_out_goal
47: fin si
48: devolver boardPoints
```

Siendo que las posiciones de los jugadores de ambos equipos y de la pelota se reciben por parámetro, todas las propiedades observables que hemos definido para evaluar un tablero pueden obtenerse con complejidad O(1).

Para hallar buenas combinaciones para los pesos asignados a estos parámetros (es decir, los k_i) utilizaremos tres técnicas algorítmicas distintas: Búsqueda Local, Grasp y Algoritmos Genéticos, que explicaremos a continuación.

2.3.2. Búsqueda Local

Este procedimiento consiste, en primer lugar, en armar una vecindad entre todas las posibles combinaciones de los parámetros mencionados en la sección anterior, los cuales consideraremos en el rango de valores entre -1 y 1. Una vez definida la vecindad de una combinación, partiremos de una combinación elegida aleatoriamente y, a partir de ahí, se simulará una serie de partidos entre un jugador que utilice la combinación actual contra los jugadores cuyo comportamiento venga definido por los valores de los k_i vecinos de la combinación actual. Así, cada paso de la búsqueda local se quedará con la combinación vecina que más partidos le haya ganado a la combinación actual. El procedimiento de búsqueda finalizará cuando se llegue a una cantidad de iteraciones preestablecida, y nos quedamos con la última combinación obtenida. Otro posible criterio de finalización podría haber sido encontrar un mismo máximo múltiples veces seguidas. Sin embargo, en las pruebas preliminares se encontró que el tiempo insumido por el algoritmo no permitía (en plazos razonables) llegar a este estado, por lo cual optamos por una cantidad fija de vecindarios a recorrer.

A la hora de analizar combinaciones de genomas obtenidas a partir de la vecindad de un genoma dado, decidimos discretizar dicha vecindad, para poder realizar los cómputos en tiempos razonables. Para el armado de esa vecindad utilizamos dos funciones:

- *getNeighborhood*, que se encarga de, para un genoma dado, llenar un vector de vectores, donde el vector iésimo contiene a los vecinos del gen iésimo del genoma. Para obtener los valores vecinos de un gen, definimos un valor de granularidad, de manera tal que si el gen iésimo tiene el valor g_i, sus dos vecinos son de la forma $g_i - granularidad$ y $g_i + granularidad$. Adicionalmente, para permitir genomas vecinos en los que un gen se mantenga constante y los otros varíen, el vector iésimo también contendrá el valor original del gen iésimo.
- *cart_product*, que recibe el vector de vectores devuelto por la función anterior y obtiene el producto cartesiano entre todos los vectores, efectivamente generando toda la vecindad del genoma actual.

Algoritmo 8 Algoritmo para optimizar los parámetros utilizados para puntuar tableros utilizando la técnica de *grid-search* con búsqueda local

Input: *max_cant_vecindarios*: la máxima cantidad de vecindarios a explorar.
max_cant_partidos: la máxima cantidad de partidos a jugar entre el equipo actual y algún vecino
Output: El genoma que, al terminar de recorrer *max_cant_vecindarios* vecindarios, ganó la mayor cantidad de partidos

```
1: función BUSQUEDALOCAL(max_cant_vecindario, max_cant_partidos)
2:     current_genome ← genoma con valores aleatorios entre -1 y 1
3:     vecinosVisitados ← 0
4:     para i ∈ [0, max_cant_vecindarios) hacer
5:         mi_equipo ← equipo con el genoma current_genome
6:         resultados ← vector para almacenar partidos ganados por el equipo vecino
7:         vecindad ← GETNEIGHBORHOOD(current_genome)
8:         para vecino ∈ vecindad hacer
9:             oponente ← equipo con genoma dado por vecino
10:            para j ∈ [0, max_cant_partidos) hacer
11:                jugar partido entre mi_equipo y oponente
12:            fin para
13:            si oponente ganó al menos tantos partidos como mi_equipo entonces
14:                agrego genoma vecino a resultados
15:            fin si
16:        fin para
17:        si resultados = ∅ entonces
18:            guardo al current_genome como genoma ganador
19:            i ← max_cant_vecindarios ▷ Como no hay ningún genoma localmente mejor, no tiene sentido seguir recorriendo
20:        si no
21:            current_genome ← genoma que ganó mas partidos
22:        fin si
23:    fin para
24:    devolver current_genome
25: fin función
```

Algoritmo 9 Algoritmo para obtener el vecindario de un genoma

Input: Genoma g
Output: Vecinos de g

```
1: función GETNEIGHBORHOOD(g)
2:     vecinos ← inicializar vector de vectores
3:     granularity ← 0,4
4:     para gen ∈ g hacer
5:         si gen + granularity > 1 entonces
6:             agregar {gen − granularity, gen, 1} a vecinos        ▷ Si veo que un extremo supera el límite, saturo
7:         si no si gen − granularity < 1 entonces
8:             agregar {−1, gen, gen + granularity} a vecinos
9:         si no
10:            agregar {gen − granularity, gen, gen + granularity} a vecinos
11:        fin si
12:    fin para
13:    result ← CART_PRODUCT(vecinos)
14:    result ← result, sin el genoma recibido como parámetro
15:    result2 ← reducir la cantidad de vecinos de result para achicar el espacio de búsqueda
16:    devolver result2
17: fin función
```

2.3.3. Grasp

Esta técnica consiste en aplicar búsqueda local varias veces consecutivas, partiendo de distintas combinaciones iniciales aleatorias y, para cada una de ellos, explorar su vecindad y trasladarse al mejor vecino, definiendo al mejor vecino de la misma forma que Búsqueda Local, es decir, como el que le gana más partidos a la combinación actual. Esta lógica puede expresarse mediante el siguiente pseudocódigo:

Algoritmo 10 Grasp

Input: *initial_genomes_count*: cantidad de genomas iniciales a generar aleatoriamente.
max_number_of_searchs: máxima cantidad de vecindades a recorrer.
games_to_play: cantidad de partidos a jugar al comparar un par de equipos.
Output: vector de genomas, correspondientes a ejecutar la busqueda local con *initial_genomes_count* genomas iniciales

```
1:  resultado ← vectorvaciodegenomas
2:  para g ∈ [0, initial_genomes_count] hacer
3:      current_genome = genoma random inicial
4:      para vecinity ∈ [0, max_number_of_searchs] hacer
5:          neighborhood ← getNeighborhood(current_genome)
6:          myTeam ← equipoqueusacurrent_genome
7:          game_results ← vector donde guardaré partidos ganados por el equipo vecino
8:          para neighbor ∈ neighborhood hacer
9:              opponent ← equipocongenomaneighbor
10:             para game ∈ [0, games_to_play] hacer
11:                 jugar partido
12:                 actualizar partidos ganados por cada equipo
13:             fin para
14:             si partidos ganados por myTeam ≤ partidos ganados por opponent entonces
15:                 agregar genoma neighbor a game_results
16:             fin si
17:         fin para
18:         si game_results = ∅ entonces
19:             guardar a current_genome como genoma ganador en el vector resultado
20:             vecinity ← max_number_of_searchs
21:         si no
22:             current_genome ← genomaconlamayorcantidaddepartidosganados
23:             almacenar current_genome en el vector resultado
24:         fin si
25:     fin para
26: fin para
27: devolver resultado
```

2.3.4. Algoritmos genéticos

Para implementar esta nueva técnica definimos a los genomas como los k_i, e implementamos:

- Una población (conjunto de genomas) inicial, con los valores de sus genes obtenidos de forma aleatoria.
- Dos funciones de *fitness*, las cuales se encargarán de hacer la selección de genomas dentro de la población, de acuerdo a los siguientes criterios:
  1. La primera función consistirá en seleccionar aquellos genomas a partir de los cuales se ganen mas partidos, y en el caso en que más de un equipo tenga la misma cantidad de partidos ganados, se resolverá este empate por la cantidad de goles a favor de cada uno.
  2. La segunda función consistirá en seleccionar a los que pierdan la menor cantidad de partidos, y en el caso de empate se definirá por la que menos goles en contra tenga.
- Dos operaciones de *crossover*, las cuales se utilizan para cruzar las propiedades de dos genomas distintos, con los siguientes criterios:
  1. Por corte: se elige un punto de corte aleatorio para los dos genomas, de modo que el genoma resultante tendrá un porcentaje de las propiedades de uno y el resto del otro.
  2. Por manipulación de la representación binaria: convertimos los dos genomas a arreglos de números en binario. Luego generamos una cantidad aleatoria de posiciones de corte dentro de la longitud de los arreglos. Finalmente comenzamos a recorrer uno de los genomas, copiando los valores del otro hasta llegar a un punto de corte, a partir de ahí se continúa copiando los valores del primer genoma en el segundo hasta el segundo punto de corte, y así hasta el final.

- Una operación de mutación, donde se alterará un valor del genoma resultante de manera aleatoria, cuya lógica explicaremos a continuación.
- Dos métodos para seleccionar a los individuos de cada generación:
  1. El primer método será basarse la elección en la evaluación de las funciones de fitness.
  2. El segundo será seleccionar de forma aleatoria a los individuos de la población.

A partir de esto, al igual que en los dos métodos anteriores, lo que buscamos es obtener los mejores genomas para luego utilizarlos en la función puntuadora, y así tener una buena estrategia de juego.

Nuestra implementación de la técnica de algoritmos genéticos estará modularizada de la siguiente forma:

- *EvaluarTodosGenomas*, que recibe una población de genomas y evalúa uno por uno según la función de fitness que se utilice.
- *SeleccionarIndividuosRandom*, que, como su nombre lo indica, a partir de una población devuelve un par de genomas seleccionados de forma aleatoria, para luego ejecutar cross-over y mutación y así obtener sus descendientes.
- *SeleccionarIndividuosByFitness*, que realiza la selección descripta en el anterior item, pero a partir de la función de fitness seleccionada.
- *CruzarGenomesBinary*, que realiza el cruzamiento en binario explicado anteriormente.
- *CruzarGenomesValues*, que realiza el cruzamiento a partir de un punto de corte aleatorio.
- *MutarGenomes*, que realiza la mutación de un genoma con cierta probabilidad.

Introduciremos en los siguientes algoritmos una nueva estructura de datos, a la cual llamaremos *genome_fitness*, que contendrá la siguiente información sobre cada genoma de la población actual: la cantidad de partidos ganados, goles a favor, partidos perdidos y goles en contra. Se actualizará luego de cada partido de prueba.

Algoritmo 11 RunGeneticArgorithm

Input: población de genomas random, un criterio de selección, un método de crossOver, cantidad de iteraciones
Output: población de genomas, resultado de aplicar sucesivas veces el ciclo principal de algoritmos genéticos

```
1: genomePopulationFitness ← EvaluarTodosGenomas(poblacionGenomas)
2: para iteracion ∈ [0, cantidaddeiteraciones] hacer
3:     nuevaPoblacion ← vector de genomas vacío
4:     para i ∈ [0, poblacionGenomas/2] hacer
5:         individuos ← par de individuos
6:         si el criterio de selección es RANDOM entonces
7:             individuos ← SeleccionarIndividuosRandom(poblacionGenomas)
8:         si no
9:             individuos ← SeleccionarIndividuosByFitness(poblacionGenomas, genomePopulationFitness)
10:        fin si
11:        descendiente ← genoma
12:        si el método de crossOver es binary entonces
13:            descendiente ← CruzarGenomesBinary
14:        si no
15:            descendiente ← CruzarGenomesValue
16:        fin si
17:        mutacion ← MutarGenomes
18:        agregar descendiente al vector nuevaPoblacion
19:        agregar mutacion al vector nuevaPoblacion
20:    fin para
21:    poblacionGenomas ← nuevaPoblacion
22:    genomePopulationFitness ← EvaluarTodosGenomas(poblacionGenomas)
23: fin para
24: devolver genomePopulationFitness
```

En el pseudocódigo anterior, vale la pena notar lo siguiente: nuestro esquema de Algoritmos Genéticos representa una variante respecto del esquema convencional. Concretamente, para generar nuevos individuos para la población de la generación siguiente, en lugar de que la cruza genere dos descendientes y la mutación los altere, implementamos un esquema alternativo. En nuestra implementación alternativa, la operación de cruza devolverá un único descendiente de los dos individuos de la generación anterior, y la operación de mutación generará otro individuo más, en base a los mismos antecesores que para la operación de cruza.

Algoritmo 12 EvaluarTodosGenomas

Input: población de genomas
Output: población de $genome\_fitness$

```
populationFitness = vector de genome_fitness
para i ∈ [0, tamao(poblacion)] hacer
    myTeam ← equipo goloso con genoma de poblacion[i]
    para j ∈ [i + 1, tamao(poblacin)] hacer
        oppTeam ← equipo goloso con genoma de población[j]
        para cantidad de partidos hacer
            jugar partido entre myTeam y oppTeam
            actualizar populationFitness[i] con partidos jugados, goles a favor y goles en contra
            actualizar populationFitness[j] con partidos jugados, goles a favor y goles en contra
            si el partido lo ganó myTeam entonces
                sumar 1 a partidos ganados de populationFitness[i]
                sumar 1 a partidos perdidos de populationFitness[j]
            si no
                sumar 1 a partidos ganados de populationFitness[j]
                sumar 1 a partidos perdidos de populationFitness[i]
            fin si
        fin para
    fin para
fin para
devolver populationFitness
```

Algoritmo 13 SeleccionarIndividuosRandom

Input: población de genomas
Output: dos genomas

```
i ← numero entero random entre 0 y el tamaño de la población
individuo1 ← poblacion[i]
eliminar poblacion[i] de poblacion   ▷ Al seleccionar individuos se los elimina para evitar múltiples apariciones en la generación siguiente
j ← numero entero random entre 0 y el nuevo tamaño de la población
individuo2 ← poblacion[j]
eliminar poblacion[j] de poblacion
devolver par(individuo1, individuo2)
```

Algoritmo 14 SeleccionarIndividuosByFitness

Input: población de genomas, población de $genome\_fitness$, función fitness a utilizar
Output: dos genomas

para genoma ∈ población **hacer**
  si criterio es FITNESS_WON_GAMES **entonces**
    $bestFitness$ = seleccionar al que ganó mas partidos
    $scndBestFitness$ = seleccionar al segundo que ganó mas partidos
  si no
    $bestFitness$ = seleccionar al que perdió menos partidos
    $scndBestFitness$ = seleccionar al segundo que perdió menos partidos
  fin si
fin para
borrar a los dos genomas seleccionados de la población
devolver $par(bestFitness, scndBestFitness)$

Algoritmo 15 CruzarGenomesBinary

Input: genoma1 y genoma2
Output: genoma

$values1$ ← transformo genoma1 a binario
$values2$ ← transformo genoma2 a binario
$corte$ ← vector con índices de corte
$usarGenoma2 \leftarrow false$
$IndiceCorte \leftarrow 0$
para $i \in [0, cantidaddebits(genoma)]$ **hacer**
  si i = $corte[IndiceCorte]$ **entonces**
    $IndiceCorte \leftarrow IndiceCorte + 1$
    $usarGenoma2 \leftarrow usarGenoma2$
  fin si
  si $usarGenoma2$ **entonces**
    $nuevoGenomaEnBits[i]$ ← genoma2[i]
  si no
    $nuevoGenomaEnBits[i]$ ← genoma1[i]
  fin si
fin para
$nuevoGenoma$ ← transformar $nuevoGenomaEnBits$ a un genoma normal
devolver $nuevoGenomaEnBits$

Algoritmo 16 CruzarGenomesValues

Input: genoma1 y genoma2
Output: genoma

$iteraciones$ ← numero entero aleatorio entre 1 y cantidad de genes de un genoma-1
$cruza$ ← genoma1
para $i \in [0, iteraciones]$ **hacer**
  $index$ ← número entero random entre 0 y cantidad de genes - 1
  $cruza[index]$ ← genoma2[$index$]
fin para
devolver $cruza$

Algoritmo 17 MutarGenomes

Input: genoma1 y genoma2
Output: genoma

```
1: resultante ← elegir al azar entre genoma1 y genoma2
2: para gen ∈ genes(resultante) hacer
3:     proba = numero aleatorio entre 0 y 1
4:     si proba <PROBABILIDAD_MUTAR_GEN entonces   ▷ definimos la probabilidad de mutar un gen como 0.1
5:         gen ← numero aleatorio proviniente de una distribución normal entre -1.0 y 1.0
6:     fin si
7: fin para
8: devolver resultante
```

2.3.5. Equipo goloso

En base a la función puntuadora y a los tres métodos implementados para optimizar los k_i, construimos un equipo que buscará realizar la secuencia de movimientos que maximice a la función puntuadora en cada jugada, quedándose entonces con el tablero de mayor puntaje en cada nuevo movimiento a realizar. En este equipo goloso, los valores de los k_i se reciben en tiempo de construcción y se toma todas las decisiones con los valores iniciales. A continuación, explicaremos la modularización del equipo goloso.

El método *make_move* del equipo goloso, es el encargado de elegir el movimiento que maximice la función de puntajes. Para elegirlo correctamente, recurrirá a las siguientes funciones auxiliares:

- *setOponentMoves*, que pretende simular los posibles movimientos del oponente.
- *generateMoves*, que genera todos los movimientos posibles a hacer a partir del estado actual del tablero.
- Otras funciones del tablero lógico explicadas previamente.

La lógica del jugador goloso puede expresarse mediante el siguiente pseudocódigo:

Algoritmo 18 make_move

```
1: oponentMoves ← setOponentMoves(tablero)
2: nextMoves ← inicializounvectordemovimientosnulos
3: maxScore ← EvaluateBoard(tablero)
4: potentialMoves ← generateMoves(tablero)
5: para movimientoActual ∈ potentialMoves hacer
6:     si movimientoActual es un movimiento válido entonces
7:         realizar movimientoActual
8:         nuevoPuntajeEvaluateBoard(tablero)
9:         si nuevoPuntaje > maxScore entonces
10:            maxScore ← nuevoPuntaje
11:            nextMoves ← movimientoActual
12:        fin si
13:        deshacer movimientoActual
14:    fin si
15: fin para
```

Algoritmo 19 setOponentMoves

```
1: movimientos ← vector de movimientos
2: para todos los jugadores del equipo contrario hacer
3:     nuevoMovimiento ← desplazamiento con dirección = 0
4:     agregar nuevoMovimiento a movimientos
5: fin para
6: devolvermovimientos
```

Algoritmo 20 generateMoves

```
1: movimientos ← vector de movimientos
2: para i ∈ [0, 8] hacer
3:     para j ∈ [0, 8] hacer
4:         para k ∈ [0, 8] hacer
5:             movimiento_0 ← (tipo : MOVIMIENTO, direccion : i)
6:             movimiento_1 ← (tipo : MOVIMIENTO, direccion : j)
7:             movimiento_2 ← (tipo : MOVIMIENTO, direccion : k)
8:             agregar (movimiento_0, movimiento_1, movimiento_2) a movimientos
9:         fin para
10:    fin para
11: fin para
12: para jugadorActual ∈ jugadores del equipo hacer
13:    si jugadorActual tiene la pelota entonces
14:        para k ∈ [1, longitud(cancha)/2] hacer
15:            para d ∈ [0, 8] hacer
16:                para i ∈ [0, 8] hacer
17:                    para j ∈ [0, 8] hacer
18:                        movimiento_jugadorActual ← (tipo : PASE, direccin : d, distancia : k)
19:                        movimiento_jugadorLibre0 ← (tipo : MOVIMIENTO, direccin : i)
20:                        movimiento_jugadorLibre1 ← (tipo : MOVIMIENTO, direccin : j)
21:                        agregar (movimiento_jugadorActual, movimiento_jugadorLibre0, movimiento_jugadorLibre1) a
   movimientos
22:                    fin para
23:                fin para
24:            fin para
25:        fin para
26:    fin si
27: fin para
28: devolver movimientos
```

3. Experimentación

En esta sección, plantearemos una serie de experimentos con el objetivo de contrastar empíricamente los resultados teóricos obtenidos en el desarollo de los problemas.

3.1. Búsqueda local

Se realizaron dos experimentos con el objetivo de observar, dado un conjunto de parámetros con valores entre -1 y 1, el efecto de recorrer el espacio de todos sus valores posibles mediante búsqueda local.

Para la obtención de los datos, en ambos casos se partió de un genoma con valores entre -1 y 1 generados al azar. Luego de eso, se varió la cantidad de vecindarios recorridos, la granularidad utilizada, la forma de seleccionar sistemáticamente elementos del producto cartesiano y la cantidad de juegos y de steps de la siguiente manera:

- 8 vecindarios, granularidad 0,3, un elemento del producto cartesiano cada 50, 20 juegos, 100 steps.
- 8 vecindarios, granularidad 0,4, un elemento del producto cartesiano cada 50, 20 juegos, 100 steps.
- 15 vecindarios, granularidad 0,3, un elemento del producto cartesiano cada 1000, 20 juegos, 50 steps.
- 15 vecindarios, granularidad 0,4, un elemento del producto cartesiano cada 1000, 20 juegos, 50 steps.
- 15 vecindarios, granularidad 0,5, un elemento del producto cartesiano cada 1000, 20 juegos, 50 steps.

Para cada caso, se realizó una búsqueda local con los parámetros indicados y se almacenaron los mejores genomas de cada vecindario. Luego, se procedió a realizar 50 juegos, de 100 steps cada uno, entre estos y sus correspondientes genomas iniciales, registrando la cantidad de partidos ganados por los primeros. Se trabajó bajo la hipótesis de que, a medida que la cantidad de vecindarios recorridos aumente, mejorará la performance del mejor genoma hallado en cada uno con respecto a aquel del cual se partió.

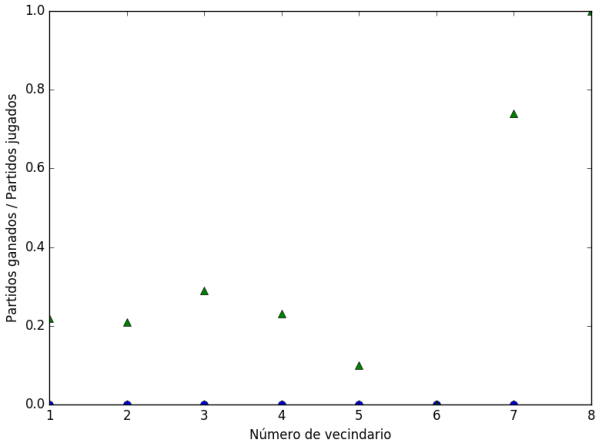

Figura 2: Proporción de victorias sobre el total de partidos disputados para genomas obtenidos de vecindarios calculados con granularidad 0,3 (●) y 0,4 (▲) en función de la distancia (en cantidad de vecindarios recorridos) con respecto al genoma inicial para el caso de 8 vecindarios en total. El caso con granularidad 0,3 llegó únicamente a 7 debido a una interrupción prematura del programa. En todos los casos se hizo jugar al equipo con el genoma inicial del lado derecho del campo.

En las figuras 2 y 3 puede observarse que los únicos casos en que se obtuvieron resultados favorables al jugar contra el genoma inicial (sin considerar los dos últimos genomas obtenidos con granularidad 0.5 y 15 vecindarios recorridos) fueron aquellos en los que se utilizó una granularidad de 0.4 para generar las vecindades.

Una posible explicación para esto es que el hecho de que un equipo juegue mejor que otro con un genoma determinado, no implica necesariamente que vaya a ganer frente a rivales a los que el segundo equipo sea capaz de derrotar.

Otra respuesta puede hallarse en la elección de vecindarios. Dado que se decidió utilizar 8 parámetros para la función de puntuación de tableros, el espacio de búsqueda para la búsqueda local, teniendo en cuenta la forma en que se definieron los vecindarios, era de $3^8 = 6561$ elementos cada vez que se encontraba un nuevo máximo en una vecindad. Debido a que revisar exhaustivamente esta cantidad de genomas hubiera resultado en un costo prohibitivo en términos de tiempo, se tomó la decisión de seleccionar arbitraria y sistemáticamente elementos del vector de vectores que constituía el producto cartesiano para reducir la cantidad de elementos recorridos. Esto provocó que se descartaran 6389 posibles genomas en los primeros dos casos y 6514 en los demás. [1]

También puede atribuirse estos resultados a la forma en la que se definió al vecindario o simplemente al azar. Respecto de este último punto, la elección de los valores 50 y 1000 fue completamente arbitraria. Tal vez, si el valor hubiera sido diferente, otros podrían haber sido los resultados de los partidos.

Con respecto las dos formas de elegir elementos del producto cartesiano, el tiempo de ejecución se redujo de $\tilde{1}2$ hs eligiendo 1 de cada 50 a $\tilde{4}5$ min tomando 1 de cada 1000. Sin embargo, esta reducción en el tiempo de ejecución se obtiene a costa de una menor chance de obtener un buen resultado.

[1] En los primeros casos, se tomó 1 de cada 50 elementos. En el segundo, 1 de cada 1000. Notar que hay 132 valores k desde 0 hasta 6561 tales que $k \equiv 0(\mod 50)$ y 7 valores m tales que $m \equiv 0(\mod 1000)$.

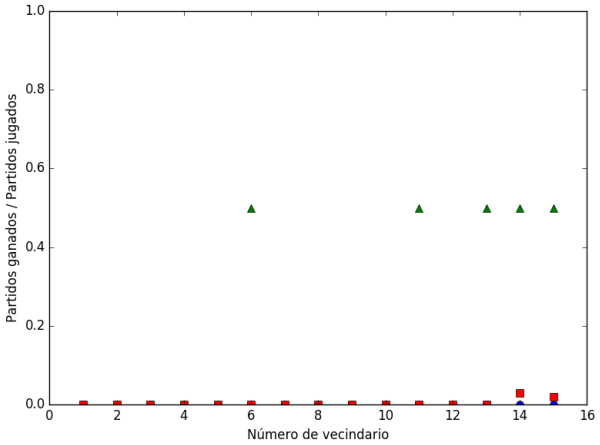

Figura 3: Proporción de victorias sobre el total de partidos disputados para genomas obtenidos de vecindarios calculados con granularidad 0,3 (●) , 0,4 (▲) y 0,5 (■) en función de la distancia (en cantidad de vecindarios recorridos) con respecto al genoma inicial para el caso de 15 vecindarios totales. En todos los casos se hizo jugar al equipo con el genoma inicial del lado derecho del campo.

3.2. Grasp

3.2.1. Comparación de cada genoma contra el inicial de la secuencia

Para evaluar de forma empírica cuánto mejoran los equipos al buscar nuevos valores para sus genomas mediante Grasp, nos basaremos en la siguiente idea: para cada genoma inicial, cada paso de la búsqueda local para obtener un vecino mejor debería, justamente, resultar en un mejor equipo. De esta forma, para cada genoma inicial se genera una sucesión de genomas vecinos que lo mejoran hasta que la búsqueda local se detiene (ya sea por haber hallado un máximo local o por haber alcanzado el límite de vecindades a recorrer).

Para medir esta supuesta mejoría con el correr de las búsquedas por nuevos vecindarios, una forma sencilla consiste en comparar a cada genoma de la sucesión con el primero. Intuitivamente, la motivación de este diseño del experimento se basa en que sucesivas búsquedas para mejorar el genoma deberían ir obteniendo estrategias que vayan mejorando incrementalmente. Para hacer la comparación entre dos genomas, simularemos la ejecución de 20 partidos entre ellos, registrando la proporción de victorias de cada genoma de la sucesión contra el original. Para evitar tiempos de ejecución excesivamente altos, limitaremos la búsqueda a 8 vecindarios, como máximo.

Nuestra intuición inicial indica que, para cada sucesión, deberíamos obtener cada vez más victorias a medida que las búsquedas en cada vecindario vayan devolviendo genomas mejores; es decir, esperamos ver que la cantidad de victorias de cada sucesión de genomas compitiendo contra su genoma inicial se comporte como una función monótona creciente.

Con esta intuición en mente, los resultados obtenidos fueron los siguientes:

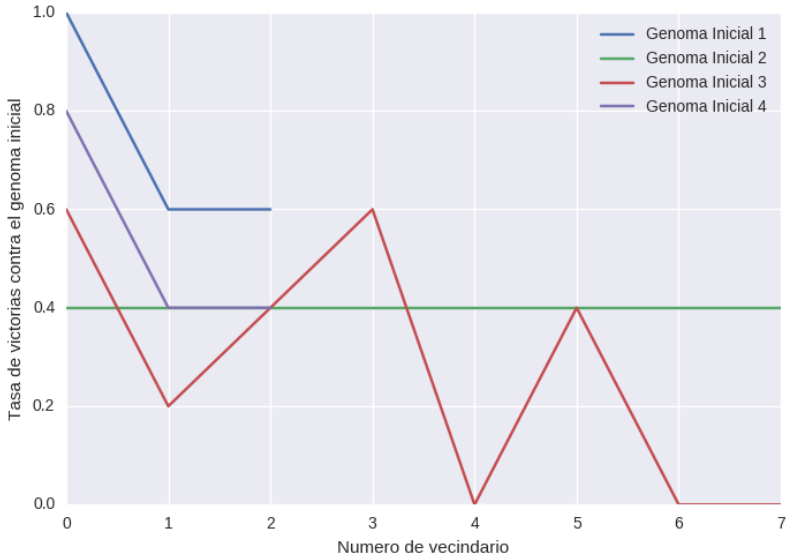

Figura 4: Proporción de victorias en función del vecindario actual, para cada sucesión de genomas

De este gráfico podemos extraer algunas conclusiónes. En primer lugar, para dos de los cuatro genomas iniciales, la búsqueda local se estanca a partir del tercer vecindario. Sin embargo, lo más llamativo es que contradice nuestra intuición inicial, pues a medida que se van recorriendo sucesivamente vecindarios la tasa de victorias va descendiendo en lugar de ascender. De todas formas, creemos poder aventurar una respuesta para este comportamiento: dadas tres estrategias e_1, e_2, e_3 para jugar un partido de fútbol (sin referirnos puntualmente a nuestro modelado del problema en este trabajo) no es necesariamente cierto que si e_2 le gana consistentemente a e_1 y e_3 le gana consistentemente a e_2, esto implique que e_3 vencerá consistentemente a e_1; es decir, la relación entre dos estrategias de que una supere a otra consistentemente no es necesariamente transitiva. Si bien este resultado no formaba parte de lo que esperábamos, sí resulta ser consistente con los resultados obtenidos mediante Búsqueda Local.

3.3. Algoritmos genéticos

3.3.1. Comparando al mejor genoma de cada generación

Para verificar el comportamiento evolutivo de Algoritmos Genéticos, debemos proponer alguna manera de monitorear cuánto mejora cada una de las poblaciones generadas en las iteraciones del ciclo evolutivo. Con este objetivo en mente, una forma posible de observar la mejoría, generación a generación, es quedarse con el mejor genoma de cada generación y usarlo como representante de ella.

Entonces, en este experimento, procederemos de la siguiente forma: para cada generación, haremos competir a todos los individuos entre sí, y definiremos al mejor genoma de la generación como el que más partidos haya ganado. Teniendo esto definido, exhibiremos cómo evoluciona la cantidad de partidos ganados por el mejor genoma con el correr de las generaciones.

Nuestra intuición inicial indica que el resultado debería, de alguna manera, mostrar algún tipo de noción de que las generaciones se van 'estabilizando'; es decir, que las diferencias entre el mejor y el resto (es decir, la cantidad de partidos ganados por el mejor genoma) deberían ir disminuyendo con el correr de las generaciones.

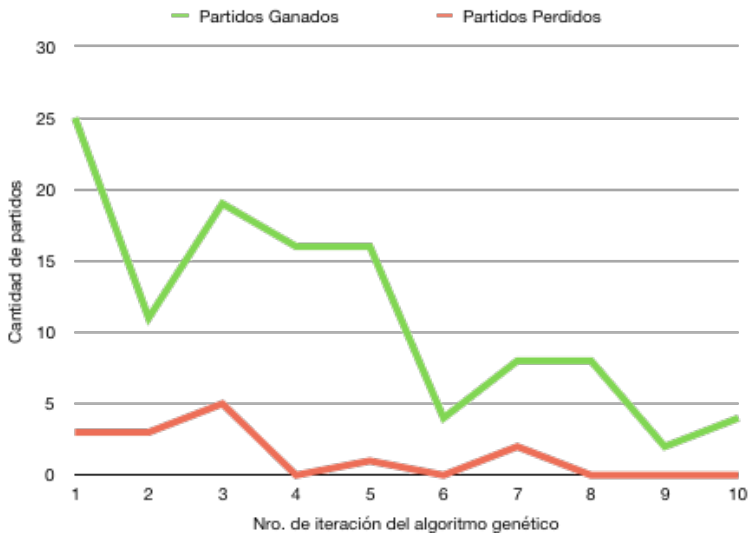

Figura 5: Cantidad de partidos ganados por el mejor al resto de los genomas, en cada generación

En la figura 5 se puede apreciar la estabilización de la población en el algoritmo genético. En la primera iteración, como todos los genomas tienen valores aleatorios, los partidos que juegan unos contra otros son muy desparejos entre sí, provocando resultados drásticos y diferenciando rápidamente los genomas con mejor fitness.

Luego, con cada iteración del algoritmo, los genomas empiezan a mutarse y cruzarse provocando nuevas poblaciones que mejoran el promedio de resultados. Podía decirse que empiezan a estabilizarse o converger los resultados ya que no hay tanta diferencia entre los mismos. Los genomas que representan a un equipo son mas parejos y esto quiere decir que la función de fitness esta evaluando a los genomas e indicandoles una dirección de convergencia.

Vale aclarar que esta convergencia de resultados no indica que los genomas resultantes sean buenos o malos, sino que es el resultado de iterar sobre una población y mutarla o cruzarla hasta que los valores converjan en base al criterio que se escogió en la función de fitness (que es la que establece el criterio de calidad en la población). Entonces, podemos pensar que la convergencia de la población hacia una calidad de genoma esta dada por la función que evalúa a la población (la función de fitness).

Por último, como en este caso la función de fitness evalúa sobre los partidos ganados/perdidos del genoma, la convergencia de la población es a tener una mayor cantidad de partidos ganados (o menor en el caso de perdidos) y como ya dijimos, los resultados en la población hace que el mejor genoma le cueste más destacarse.

Con la misma intuición que para el experimento anterior, también podemos anticipar un resultado análogo: si el mejor de cada población acorta su distancia al resto (en términos de cuánto mejor al resto juega), también es esperable que a medida que se van generando sucesivas poblaciones, el mejor de cada generación sea capaz de convertirle menos goles a los demas. Con esta idea en mente, definiremos al mejor de cada población de la misma forma que para el experimento anterior y mostraremos la suma de los goles convertidos al competir con el resto de la población a la que pertenece:

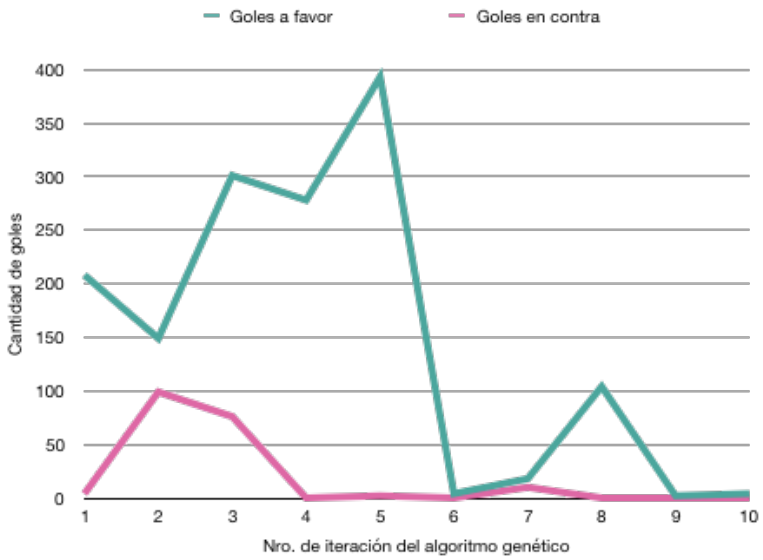

Figura 6: Cantidad de goles convertidos por el mejor al resto de los genomas, en cada generación

Los resultados obtenidos confirman parcialmente nuestra intuición de 'estabilización' de las sucesivas poblaciones: si bien al principio muestra una tendencia creciente, al superar la quinta generación observamos una caída abrupta en la cantidad de goles que el mejor le puede convertir al resto.

Este mismo experimento puede repetirse variando los parámetros asociados al algoritmo genético. Para este caso, utilizaremos la cantidad de partidos ganados como método de fitness, el método de cruza binario y el método de selección basado en el fitness de los individuos, y repetiremos la misma idea: ver cuántos partidos gana y cuántos pierde el mejor de cada generación, esperando ver que a medida que avancen las generaciones, estos valores se estabilicen:

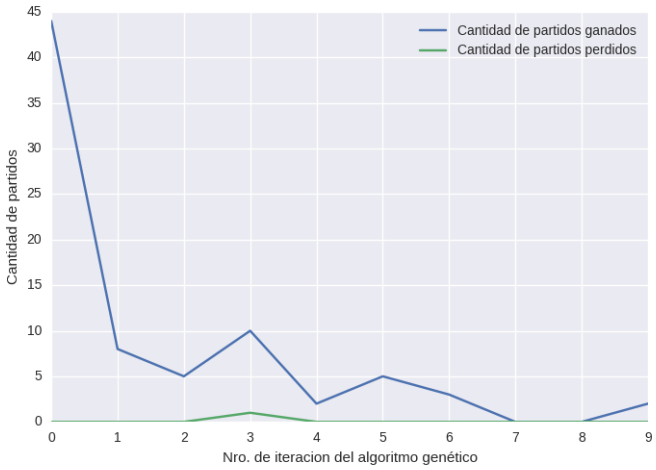

Figura 7: Cantidad de partidos ganados por el mejor al resto de los genomas, en cada generación

De la misma forma, podemos volver a mostrar la cantidad de goles a favor y en contra del mejor genoma de cada generación al competir contra todos los demás:

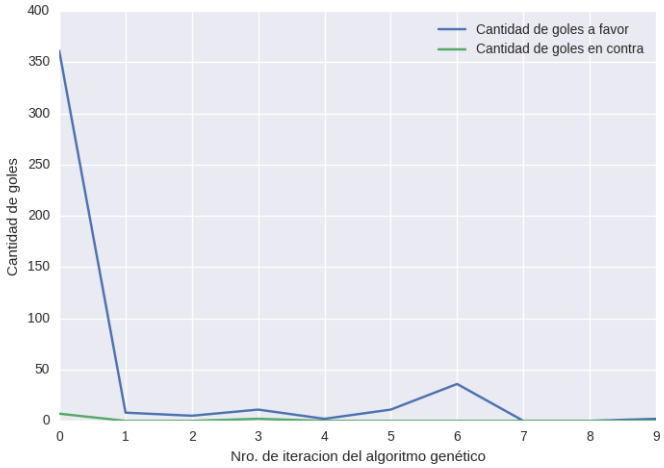

Figura 8: Cantidad de goles a favor y en contra del mejor al competir con resto de los genomas, en cada generación

Estos dos resultados permiten afirmar con cierto grafo de confianza que la tendencia de estabilización detectada en el experimento anterior y este es una característica particular de Algoritmos Genéticos.

Por último, mostraremos los resultados obtenidos al correr este mismo experimento, pero usando el método de selección aleatorio y el método de crossover con un punto de corte, manteniendo la función de fitness como la mayor cantidad de partidos ganados:

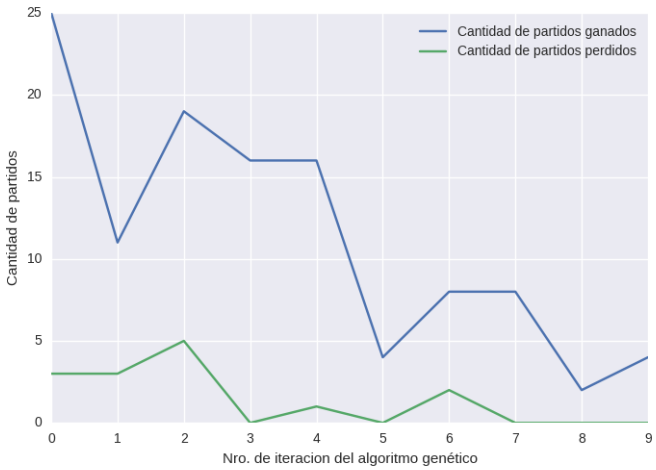

Figura 9: Cantidad de partidos ganados y perdidos por el mejor al competir con resto de los genomas, en cada generación

Y de forma análoga, mostraremos también la cantidad de goles convertidos/recibidos por el mejor de cada generación, para esta combinación de parámetros:

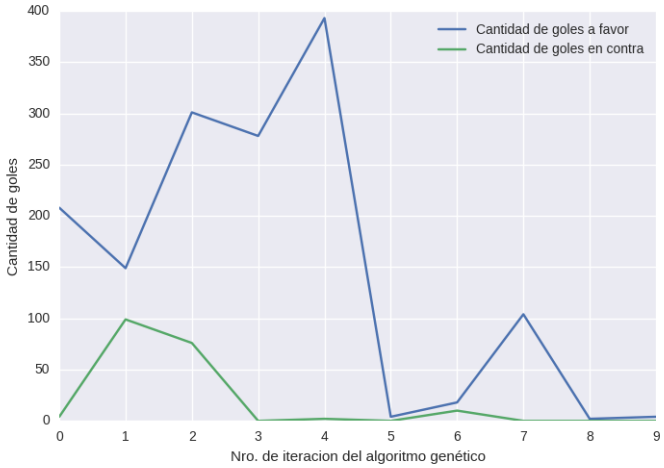

Figura 10: Cantidad de partidos goles a favor y en contra del mejor al competir con resto de los genomas, en cada generación

Una vez más, vuelve a manifestarse la tendencia a achicar la brecha entre el mejor y el resto de la población. Los resultados de este experimento para otras combinaciones de métodos de fitness/selección/crossover son análogos a los obtenidos hasta ahora (pero para evitar redundancia mostrando muchas veces resultados parecidos, no los incluiremos en este informe).

3.3.2. Distintas combinaciones de algoritmos genéticos

Diseñamos este experimento con la intención de obtener los mejores valores para presentar nuestro equipo competitivo. Para ello, hicimos distintas corridas del algoritmo genético con distintos parámetros de entrada, para luego comparar 'a mano' (es decir, utilizando el visualizador provisto por la cátedra) el comportamiento de cada uno de los equipos asociados al mejor genoma de la última generación de cada una de las corridas. Mediante este procedimiento, obtuvimos que la mejor estrategia provino de una corrida del algoritmo genético donde la función de fitness era la que consideraba la cantidad de partidos ganados, la cruza era la versión con valor de corte y la selección de individuos aleatoria; dicha corrida devolvió que el mejor genoma de la última generación considerada tenía los valores:

[0.151093, -0.953879, 0.00476067, -0.751713, 0.371691, -0.38677, 0.204654, 0.74608, 0.851888]

que son los que definen el comportamiento de nuestro mejor jugador encontrado.

4. Conclusiones

4.1. Búsqueda local

Si bien no se obtuvieron los resultados deseados mediante este método, es posible extraer algunas conclusiones con respecto a su uso.

El mayor inconveniente que se tuvo al utilizar esta técnica fue el gran número de variables existentes y, principalmente, la cantidad de parámetros utilizados para puntuar tableros. El hecho de haber utilizado 8 arrojó, considerando la discretización del intervalo $[-1, 1]$ cada 0,1, 6561 candidatos posibles en cada paso del algoritmo, cantidad que obligó a descartar elementos debido al tiempo consumido por el algoritmo. Esto limitó también la cantidad de vecindarios recorridos. Sumado a lo anterior, es precisa una adecuada definición del vecindario a recorrer, pudiendo haber sido nuestra propuesta demasiado simple. Lo mismo podría decirse del punto de inicio.

Restaron muchos posibles experimentos por realizar, principalmente por el tiempo que hubieran requerido para ser llevados a cabo. Entre estos pueden mecionarse: variar las probabilidades de quite de los jugadores, así como sus posiciones iniciales, la forma de generar los movimientos del equipo y las hipótesis acerca de los realizados por el rival.

A pesar de lo arriba mencionado, la búsqueda local resulta ser una alternativa viable para recorrer el espacio de búsqueda en *grid-search*, pero debe ser puesta a punto cuidadosamente para lograr buenos resultados.

4.2. Grasp

Las conclusiones que podemos sacar respecto del uso de Grasp como estrategia para recorrer la grilla son análogas a las obtenidas utilizando Búsqueda Local: la gran cantidad de propiedades que definimos para puntuar el estado de

un partido hace que la grilla sea excesivamente grande. Más aún, como Grasp se basa en realizar sucesivas repeticiones del método de Busqueda Local, variando la combinación inicial por la cual se comienza a recorrer la grilla, lo acotado del tiempo de resolución para este trabajo hizo que fuera inviable experimentar utilizando formas alternativas de dar las combinaciones iniciales de los k_i que no fuera generándolos aleatoriamente, con lo cual nuestra búsqueda utilizando Grasp nos condujo a máximos locales que en la práctica no quedaron asociados a equipos que tomaran buenas decisiones en función de ellos. Sin embargo, vale la pena detenerse a hacer la siguiente salvedad: incluso contando con una cantidad de tiempo considerablemente mayor a la que tuvimos, no hay ninguna garantía de 'acertar' con la combinación inicial de valores de k_i que lleve al máximo global. En definitiva, esto quiere decir que nos topamos con una desventaja de Grasp que conocíamos de antemano: los máximos locales obtenidos mediante la exploración de vecindades no tienen por qué ser máximos globales (y de hecho, ni siquiera tienen por qué ser 'buenas' aproximaciones a una solución al problema en cuestión).

4.3. Algoritmos Genéticos

La técnica de Algoritmos Genéticos nos sirvió como alternativa superadora a las dos anteriores: al no 'estancarse' en máximos locales como Búsqueda Local/Grasp, pudimos hallar mejores combinaciones de parámetros que tal vez no habríamos podido explorar. Además, al tener como uno de los objetivos de este trabajo la búsqueda de un equipo competitivo, visualizar cuán pronunciada es la diferencia entre el mejor de cada generación (que es naturalmente el candidato a presentar como equipo competitivo) y el resto de su generación, nos encontramos con que esa brecha tiende a cerrarse, ya que la cantidad de partidos que el mejor le gana a los demás tiende a bajar con el correr de las generaciones. Esto nos permite introducir otra noción, distinta a la que teníamos para Búsqueda Local y Grasp: en lugar de tener una sucesión de equipos individuales convergiendo *puntualmente* a un máximo local, lo que tenemos son conjuntos de equipos que convergen *uniformemente* (puesto que la diferencia entre el mejor y el resto va acortándose progresivamente) a una buena estrategia de juego.

Por último, vale la pena hacer mención a aquellos experimentos que dejamos fuera de este informe: no planteamos experimentos que se enfocaran en los tiempos de ejecución de los distintos métodos, eligiendo mostrar únicamente la calidad de las soluciones encontradas con cada uno de ellos.