DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº3

Darwin DT en el mundial 2018

28 de Junio de 2018 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bonaccini, Adrián	207/04	abonaccini@dc.uba.ar
Catalano, Arístides	279/10	aristidescata@gmail.com
Musso, Bruno Martín	676/11	brunomusso91@hotmail.com
Romera, Joaquin	183/16	jromera@dc.uba.ar

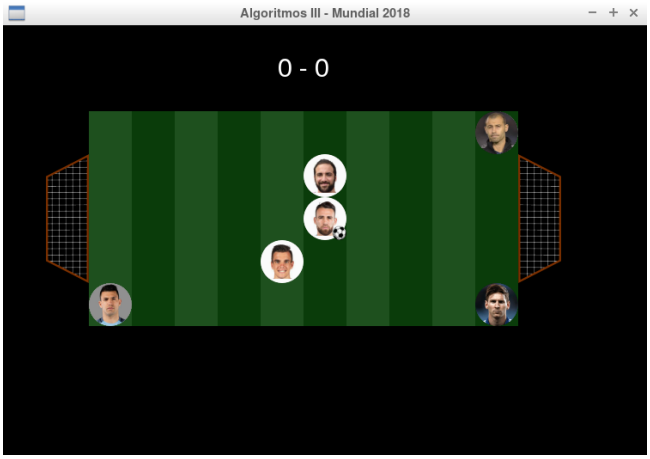
Entrega
Reentrega

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

Índice

1. Introducción

En el presente informe se describen las metodologías y estrategias utilizadas para resolver el trabajo práctico Nº3. El objetivo del mismo es modelar computacionalmente partidos de fútbol implementando estrategias con diferentes técnicas algorítmicas. En particular se nos pidió una *función parametrizable* que dado un estado cualquiera de un tablero, la misma le asigne un puntaje, y un jugador *goloso* que elija el movimiento que maximice el puntaje de dicha función. Para poder implementarlos se requirió experimentar con distintas heurísticas para determinar las mejores combinaciones de parámetros. Dichas heurísticas incluyen *búsqueda local* y *grasp*, para restringir el espacio de soluciones al buscar usando la técnica *grid-search*, y algoritmos *genéticos* para optimizar los parámetros.



A continuación detallamos cómo se procedió en cada instancia de la resolución del trabajo práctico. También incluímos una breve investigación sobre el estado del arte de los algoritmos genéticos, sus principales áreas de aplicación y sus limitaciones. Finalmente en la sección de Experimentación detallamos los experimentos realizados en el contexto de este trabajo.

1.1. Consideraciones previas

Para todos los algoritmos implementados se partió de la implementación básica otorgada por la cátedra del *static player*. Cada uno de los jugadores presentados en nuestro trabajo utiliza la misma interfaz para respetar la compatibilidad con *Elizondo.py*. Asimismo la lógica del tablero fue traducida de Python a C++ según los contenidos de *LogicalBoard.py* de manera de encapsular la lógica y reglas de juego.

A lo largo del trabajo nos referimos a movimientos posibles o válidos, por esto nos referimos a movimientos o pases que están en conformidad con las reglas detalladas en el enunciado.

1.2. Archivos

El trabajo puede compilarse a través del **Makefile**, este archivo compila los jugadores *static*, *greedy* y *grid search*. El código de cada uno puede revisarse en los siguientes archivos:

- static_player.hpp
- greedy_player.hpp
- gs_player.hpp
- auxiliars.hpp

2. Función Puntuadora Parametizable

2.1. Evaluate Board

Para poder decidir cuál es el próximo movimiento a realizar implementamos una función que evalúa el estado de un tablero según una serie de criterios y le asigna un puntaje al mismo. Esta función a su vez es parametizable, podemos elegir para cada criterio de los evaluados, qué peso tendrá en el puntaje otorgado. Los criterios que consideramos oportuno evaluar fueron los siguientes:

- La pelota está dentro del arco contrario.
- La distancia de los jugadores al arco contrario.
- La distancia de cada jugador al oponente más cercano.
- La dispersión de los jugadores en el tablero.
- La distancia de los jugadores a la pelota.

Cada uno de estos criterios es evaluado positiva o negativamente dependiendo de si la pelota la tiene nuestro equipo, el equipo contrario, o si la pelota está libre. De todos los criterios el primero, que equivale a un gol de acuerdo a las reglas del deporte, otorga puntaje máximo.

2.2. Implementación y pseudocódigo

```
1:  function EVALUATE BOARD(boardStatus currentBoard, array[int] moves, array[double] weights)
2:      boardStatus updatedBoard ← updateBoard(currentBoard, moves)                    ▷ O(1)
3:      double result ← 0                                                              ▷ O(1)
4:      if scoredGoal(updatedBoard) then
5:          result ← ∞
6:          return result
7:      if weHaveTheBall() then
8:          for player in updatedBoard.team do
9:              result ← result − distancePlayerOpponnentGoal(player) · weight[0]    ▷ O(1)
10:             result ← result + distancePlayerClosestOpponnent(player) · weight[1] ▷ O(1)
11:             result ← result + dispersion(player) · weight[2]                     ▷ O(1)
12:     if opponnentsHaveTheBall() then
13:         for player in updatedBoard.team do
14:             result ← result − distancePlayerBall(player) · weight[3]             ▷ O(1)
15:             result ← result − distancePlayerClosestOpponnent(player) · weight[4] ▷ O(1)
16:             result ← result + dispersion(player) · weight[5]                     ▷ O(1)
17:     if boardHasTheBall() then
18:         for player in updatedBoard.team do
19:             result ← result − distancePlayerBall(player) · weight[6]             ▷ O(1)
20:             result ← result − distancePlayerOpponnentGoal(player) · weight[7]    ▷ O(1)
21:             result ← result + dispersion(player) · weight[8]                     ▷ O(1)
22:         for player in updatedBoard.oponnentteam do
23:             result ← result + distancePlayerBall(player) · weight[9]             ▷ O(1)
24:     return result
                                                                                     ▷ Total: O(1)
```

2.3. Análisis de complejidad

Cada instrucción es asintóticamente O(1), inclusive los ciclos iteran por los equipos, cuyos tamaños estan acotados por 3, lo cual no implica que el tiempo de ejecución sea equivalente al que toma cualquier operación O(1) ya que realiza varios cálculos para dictaminar el valor asociado al tablero pasado por parámetro.

3. Greedy Player

Este algoritmo recibe una variable de tipo *board status* y elige el mejor movimiento para cada jugador del equipo según la función puntuadora y la parametrización establecida. La forma de hacerlo es la siguiente: itera por todos los movimientos posibles y válidos de cada jugador del equipo, si algún jugador tiene la pelota evalúa también hacer un pase en todas las direcciones válidas y con todos los valores de *steps* posibles (hasta $\frac{m}{2}$ y sin salir del tablero). Por cada una de estas combinaciones evalúa el estado del tablero resultante con *Evaluate Board* y se devuelve la combinación de movimientos que otorgó el mayor puntaje. Esta técnica golosa asegura obtener la solución óptima para la parametrización y el estado del tablero dados.

3.1. Implementación y pseudocódigo

```
1: function SEARCH MOVE(boardStatus currentBoard)
2:     double maxRank ← −∞                                                    ▷ O(1)
3:     double currentRank ← −∞                                                ▷ O(1)
4:     array[{int,int,int,int,int}] result ← ∅                                ▷ O(1)
5:     for i in moves do
6:         if insideBoard(updatedBoard.team,i) then
7:             for j in moves do
8:                 if insideBoard(updatedBoard.team,j) then
9:                     for k in moves do
10:                        if insideBoard(updatedBoard.team,k) then
11:                            for jugador in jugadores do
12:                                if hasTheBall(jugador) then
13:                                    maxSteps ← calculateMaxSteps(player,currentBoard)   ▷ O(m)
14:                                    for steps in [1, maxSteps] do
15:                                        currentRank ← evaluateBoard(currentBoard,i,j,k,player,steps) ▷ O(1)
16:                                        if currentRank ≥ maxRank then
17:                                            maxRank ← currentRank          ▷ O(1)
18:                                            result ← {i, j, k, player, steps}   ▷ O(1)
19:                            if validPositions(updatedBoard.team,i,j,k) then   ▷ O(1)
20:                                currentRank ← evaluateBoard(currentBoard, i, j, k, 0, 0)
21:                                if currentRank ≥ maxRank then
22:                                    maxRank ← currentRank                  ▷ O(1)
23:                                    result ← {i, j, k, 0, 0}
24:    return result
                                                                              ▷ Total: O(m)
```

```
1: function CALCULATEMAXSTEPS(playerStatus player, int dir, int rows, int columns)
2:     int middleRow ← rows/2                                                 ▷ O(1)
3:     int steps ← 0                                                          ▷ O(1)
4:     for i in [0, middleRow] do
5:         if isValidKick(player,dir,i,rows,columns) then
6:             steps ← k                                                      ▷ O(1)
7:     return steps
                                                                              ▷ Total: O(m)
```

3.2. Correctitud

El algoritmo genera todas las combinaciones posibles de movimientos considerando tanto aquellos de tipo *MOVIMIENTO* como los de tipo *PASE* para cada uno de los jugadores del equipo. Luego cada combinación recibe una puntuación y se devuelve la que mejor puntaje haya alcanzado. La implementación del algoritmo utiliza la técnica de *Backtracking* con *podas por factibilidad*. Estas *podas* consisten en excluír de la evaluación aquellas combinaciones de movimientos que produjeran jugadas inválidas. Una validación similar se aplica respecto de todas las posibles jugadas de *PASE* que podrían hacerse en caso de que algún jugador posea la pelota.

3.3. Análisis de complejidad

El pseudocódigo detallado en la sección anterior posee tres ciclos principales de manera de iterar por todas las combinaciones de movimientos posibles para el equipo. A medida que construye la combinación de movimientos, chequea que la elección se mantenga dentro de los rangos definidos por el tablero. Luego valida que la combinación de movimientos obtenidos genere una instancia válida y, por cada jugador del equipo, en caso de estar en posesión de la pelota, la posibilidad de patear y con qué potencia (*steps*). Cada uno de los tableros que se generan de ejecutar dicho movimiento son evaluados por la función puntuadora seleccionando el de valor máximo. Una vez verificados todos los tableros que posibilitan realizar un *PASE*, hacemos un segundo cálculo que mide el tablero donde cada jugador realiza un *MOVIMIENTO*. Como los ciclos de movimientos y jugadores iteran por arreglos de tamaño constante *evaluateBoard* tiene complejidad O(1), el único cálculo realizado cuyo costo es superior coincide con el cálculo de la máxima potencia del pase (*calculateMaxSteps*). Dicha función, en base a la dirección del *PASE* y la posición del jugador, calcula el movimiento de la pelota incrementando la cantidad de *steps* hasta llegar a la iteración en que la pelota sale del tablero obteniendo así el máximo.

En el peor de los casos, la pelota podría ir de arco a arco realizando $\frac{n}{2}$, pero es condición del enunciado que el valor máximo permitido sea $\frac{m}{2}$. Como máximo el algoritmo itera $\frac{m}{2}$ veces y se obtiene que el orden de complejidad del algoritmo es de O(m).

4. Algoritmos Genéticos

4.1. Definición

En las ciencias de la computación, los algoritmos genéticos son meta heurísticas inspiradas en el proceso natural de selección donde se busca a partir de una población inicial ir obteniendo una nueva que se comporte mejor que la inicial utilizando procesos inspirados en la cruza de individuos, mutaciones y selección.

Las llamadas poblaciones son un conjunto de individuos con un grupo de propiedades (a las que denominaremos genes y a su conjunto un genoma) que se elige inicialmente de forma aleatoria y a la cual se le aplicará una función de fitness que dictaminará de todos mis individuos los más aptos en base a lo que nos interesa observar y luego a ellos los mantendremos para la siguiente muestra y reemplazaremos a los demás por nuevos individuos formados en base a los que mantendremos (crossover) y/o modificando ciertos genes aleatoriamente (mutación).

Este proceso se repite hasta que las poblaciones obtenidas sean "óptimas" (estemos satisfechos con los resultados o no haya mucha mejora entre generaciones) o hasta un cierto número de iteraciones, ambos son dos criterios distintos de corte que se pueden aplicar juntos o por separado.

4.2. Áreas de aplicación

A continuación se detallarán algunas de las aplicaciones de esta estrategia para optimizar ciertos problemas.

Uno de los principales campos de aplicación se encuentra en la robótica. Los algoritmos genéticos se utilizan para lograr que el robot alcance el objetivo que tiene programado y se adapte a los problemas y obstáculos con los que se pueda llegar a encontrar.

Otra aplicación muy recurrente se da en la economía, el mundo de las acciones y bolsas de valores es muy caótico donde todo sube y baja de precio en cuestión de segundos, lo que se intenta lograr con los algoritmos genéticos es predecir cuales son las mejores estrategías para invertir evitando así la pérdida de dinero en acciones o bolsas de valores en las que, según los resultados obtenidos, no convenga invertir.

Yendo a un plano más de la vida cotidiana podemos encontrar a las aplicaciones para calcular las mejores rutas o caminos para llegar de un lugar a otro donde utilizando información sobre el tráfico del día a día se puede estimar cuales van a ser las calles más congestionadas y en qué horarios al momento de decidir por dónde transitar.

Por último, podemos considerar las inteligencias artificiales de los videojuegos en las cuales a medida que un jugador va utilizando el juego va mejorando sus habilidades y por ende requiere rivales más habilidosos y a veces las dificultades proporcionadas por los juegos no son lo suficientemente desafiantes. Aplicando estos algoritmos podemos desarrollar en base a las partidas del jugador nuevas estrategias para ofrecerle un modo donde encuentre el desafio que está buscando y no se aburra del mismo.

4.3. Limitaciones

Si bien los algoritmos genéticos son buenos para conseguir optimizaciones, tienen ciertas limitaciones. Por ejemplo:

- La creación de la función de fitness tiene que ser lo suficientemente robusta como para poder soportar los cambios aleatorios de cada iteración.
- No solo alcanza con que la función sea robusta, si se elige una mala función también tendremos el problema de que haya casos favorables que se pierdan impidiendo así conseguir las mejores soluciones posibles.
- Si bien la función de fitness es importante, tampoco hay que descuidar los demás procesos como el crossover y las mutations ya que son los que logran variar la población entre cada iteración para intentar obtener mejores resultados. Si ellas fallan, podría no solo retrasar el tiempo hasta encontrar una solución eficiente,

sino también lograr que nunca se llegue a una, por ejemplo que en el crossover se elijan los genes que menos favorezcan o con la mutación reemplazar los mejores por valores malos.

- Como las "buenas soluciones"son buenas en comparación a otras, tampoco podemos asegurar que lleguemos a las mejores posibles, como detallamos anteriormente, si no se varían bien las muestras entre cada iteración para obtener resultados nuevos y variados.
- Los algoritmos genéticos no funcionan muy bien para problemas de optimización donde los valores posibles de sus genes son solo "si o no", "verdadero o falsoza que no hay forma de converger a una solución. En estos casos una búsqueda aleatoria puede resultar más eficiente.

4.4. Referencias

- https://en.wikipedia.org/wiki/Genetic_algorithm
- https://www.doc.ic.ac.uk/project/examples/2005/163/g0516312/Algorithms/
- https://www.brainz.org/15-real-world-applications-genetic-algorithms/