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

Tesis de Licenciatura en Ciencias de la Computación

Obtención de Políticas Mediante Aprendizaje por Refuerzo Utilizando un Simulador para Fútbol de Robots

Diciembre 2007

Alexis G. Prus
Carlos E. Weiss

Director de Tesis: Dr. Juan M. Santos

Resumen

La aplicación del aprendizaje automático en sistemas muti-robots puede traer aparejado dificultades derivadas de la representación del espacio de estados o estado-acción como así también la necesidad de realizar un número elevado de experimentaciones que a veces conllevan a declinar su uso. Una de las técnicas que mejor se ajusta al aprendizaje en robots es el aprendizaje por refuerzo y su aplicación a casos de sistemas multi-robots ha sido acotada por las razones expuestas recién.

Para abordar este problema, se ha escogido al Fútbol de Robots como caso de aplicación, el cual está bien difundido y es frecuentemente usado como benchmark para robótica móvil y sistemas multi-robots. La intención de usar este problema es tratar de extraer conclusiones y resultados que después podrían ser de utilidad para problemas similares o mas simples.

En este trabajo se describe un análisis detallado de los diferentes aspectos de la aplicación del aprendizaje por refuerzo para la obtención de políticas que seleccionan comportamientos en Fútbol de Robots. Aquí fue necesario tratar con el problema de espacios de acción-estado de cardinal suficientemente grande. A tal efecto, se usaron tres técnicas de clasificación (clustering): Coarse Coding, Q-Kohon y Q-RBF a efectos de compararlas y seleccionando la que mejor se adecuaba para el caso de estudio. Para poder implementar la técnica de aprendizaje por refuerzo, fue necesaria la creación de un simulador que permitiera realizar experimentos con realismo, para lo cual se escogió ODE (Open Dynamics Engine), un motor de simulación de sistemas dinámicos de cuerpos rígidos.

Se estudiaron los casos para 1 x 1 (un robot contra otro robot), 2 x 2 y 3 x 3, para los cuales se realizaron extensas series de simulaciones, incluyendo el uso de técnicas de aprendizaje multi-agente con coordinación. En todos los casos, el simulador facilitó la obtención de resultados compatibles con la realidad y que permitieron extraer conclusiones sobre la cuestión de aprendizaje en sistemas multi-robots.

Abstract

The use of automatic learning in multi-robot systems can present difficulties derived from the state space or state-action space representation, as well as the need to perform a large number of experiments that sometimes discourage its utilization. One of the techniques that better suits robot learning is Reinforcement Learning, and its application to the multi-robot systems problem has had a limited use so far, due to the reasons exposed above.

In order to approach this problem, Robot Soccer has been chosen as an application case, which is widely spread and is often used as benchmark for mobile robotics and in multi-robot systems. The purpose of using this domain is to attempt to come up with conclusions and results that then can be useful for similar or simpler problems.

A detailed analysis of the different aspects of the application of Reinforcement Learning for obtaining policies that select behaviors in Robot Soccer is described in this work. Here, it was necessary to deal with the problem of state-action spaces with a sufficiently large cardinal. For this reason, 3 methods of classification (clustering) were used: Coarse Coding, Q-Kohon and Q-RBF. They were compared and the one that best suited the study case was selected. In order to be able to implement the Reinforcement Learning technique, it was necessary to create a simulator that allows performing realistic experiments. ODE (Open Dynamics Engine), a simulation engine for rigid body dynamic systems, was chosen for implementing this simulator.

The scenarios involving 1 x 1 (one robot against another robot), 2 x 2 and 3 x 3 were studied, for which extensive series of simulations were performed, including the utilization of coordinated, multi-agent learning techniques. In all these cases, the simulator helped to achieve results that were compatible with reality and allowed to draw conclusions regarding the learning problem in multi-robot systems.

Tabla de contenidos

Capítulo 1

Introducción

En el año 2001, en el Departamento de Computación de la FCEyN, se creó el Grupo de Inteligencia Computacional Aplicada a Robótica Cooperativa. En el marco de dicho grupo se inició el proyecto UBASot, cuyo principal objetivo fue el desarrollo de equipos de fútbol de robots para participar en las competencias internacionales relacionadas con el tema.

El fútbol de robots es muy similar al fútbol que todos conocemos, aunque la principal diferencia radica en que los jugadores son robots en vez de personas. Involucra a dos equipos rivales de múltiples jugadores cada uno, moviéndose en un entorno altamente dinámico. En un partido, cada equipo tiene como objetivo el de ganar el juego convirtiendo una mayor cantidad de goles que el equipo adversario. Por supuesto que los objetivos de los dos equipos son incompatibles y, por eso, puede verse al equipo oponente como un entorno dinámico que puede dificultar el logro del objetivo común.

En estos momentos hay dos campeonatos de fútbol de robots reconocidos mundialmente y con fuerte participación. Son eventos anuales organizados por RoboCup y FIRA, cada uno con múltiples categorías, tales como la utilización de jugadores simulados o la utilización de robots reales, cada una de las cuales posee desafíos específicos.

El Grupo de Inteligencia Computacional de la FCEyN compitió dos veces en torneos de la FIRA y obtuvo importante experiencia en ellos, tanto en categorías simuladas como con robots reales. Aprovechando esto, el grupo tuvo la iniciativa de desarrollar robots especializados para este tipo de competencias y, simultáneamente, desarrollar métodos de control para su utilización. Dado que no es factible programar los comportamientos de un robot para todas las situaciones posibles, y considerando la incertidumbre en los datos de los sensores y en la ejecución de las acciones, dentro del grupo surgió la inquietud de investigar técnicas de aprendizaje automático para abordar el problema. En particular, se decidió profundizar en una técnica llamada aprendizaje por refuerzo.

La aplicación del aprendizaje por refuerzo en sistemas multi-agente puede traer aparejado dificultades derivadas de la representación del espacio de estados o estado-acción, como así también la necesidad de realizar un número elevado de experimentaciones que a veces conllevan a declinar su uso. Imaginemos el caso

de dos equipos de 5 robots cada uno, donde para cada comportamiento que se intente aprender debemos ejecutar decenas de experimentos y, en cada uno, posicionar manualmente a cada robot. Vemos rápidamente que para lograr de esta forma la experiencia necesaria para obtener un resultado razonable, el problema se torna en algo impracticable.

De aquí nace la necesidad de contar con un simulador que modele el campo de juego y la dinámica de los robots, de manera de poder rápidamente extender los resultados obtenidos a través de la simulación y aplicarlos a equipos de fútbol de robots reales. Los simuladores generalmente utilizados en competencias como RoboCup y FIRA utilizan modelos físicos muy simplificados. Por esto, es menester disponer de un simulador que modele y aproxime tanto las características físicas reales del juego como la de los robots. Además, el uso del aprendizaje por refuerzo en Fútbol de Robots es de una complejidad tal que justifica profundizar el análisis de las diferentes decisiones y procedimientos que involucran la obtención de políticas por parte de un agente.

Luego del capítulo 1, que realiza una introducción a este trabajo, la tesis está dividida en dos partes.

La primer parte describe la implementación de un simulador que permite realizar experimentos con realismo, en tiempos más reducidos y sin la supervisión requerida por los robots reales. Este simulador será utilizado para evaluar distintas técnicas en un juego de fútbol de robots de la categoría MiroSot de la FIRA. En esta parte, el capítulo 2 introduce algunos conceptos de simulación, evalúa las características deseables para nuestro simulador y describe la elección de un motor de simulación. Luego, el capítulo 3 describe el diseño y la implementación del simulador.

En la segunda parte se presentan los resultados obtenidos al implementar un método de aprendizaje por refuerzo. Para esto, en el capítulo 4 nos detendremos a analizar las características principales del aprendizaje por refuerzo. El capítulo 5 describe el método de aprendizaje por refuerzo en particular que implementamos, así como también la problemática que encaramos al especializar el problema utilizando al fútbol de robots como base. En el capítulo 6 se detallan algunos de los experimentos realizados, junto con sus resultados, al analizar la aplicación del aprendizaje por refuerzo al fútbol de robots y evaluamos diferentes alternativas relacionadas con los siguientes puntos:

- Comportamientos básicos de los robots
- Definición del espacio de situaciones
- Inicialización de parámetros de los métodos de aprendizaje utilizados
- Técnicas de discretización
- Cantidad de jugadores en cada equipo

- Algoritmos de coordinación para sistemas multi-agente

En todos los casos, el simulador facilitó la obtención de resultados compatibles con la realidad y que permitieron extraer conclusiones sobre la cuestión de aprendizaje en sistemas multi-agente.

Por último, el capítulo 7 expone las principales conclusiones y trabajos futuros.

Capítulo 2

Elección de un motor de simulación

2.1 Tipo de simulador a utilizar

Teniendo en claro que para nuestros experimentos necesitaríamos de un simulador, nos preguntamos qué tipo de simulación era conveniente para nuestros propósitos. Como mencionamos anteriormente, nuestro propósito principal es lograr una simulación que a efectos prácticos sea tan cercana a la realidad como sea posible, manteniendo bajos los requerimientos de tiempo y poder computacional necesario. Nuestra hipótesis en este caso es que, logrando tal simulación, es más fácil trasladar el modelo implementado a los robots reales para obtener resultados comparables a los simulados.

En un principio partimos de la idea de mejorar algunos componentes de software que el Grupo de Inteligencia Computacional del departamento [1] ya poseía. Estos componentes implementan conceptos básicos de la física para modelar el movimiento y la interacción de partículas. Es decir que, en este caso, los robots y la pelota se modelan como partículas. Cabe acotar que estos componentes no incluían, entre otras cosas, los conceptos de fricción y un detector de colisiones avanzado para modelar un sistema multi-agente. Rápidamente evaluamos que nuestra simulación debería contar con estos elementos.

También fueron evaluados los simuladores de RoboCup [2] y FIRA [3]. Ellos nos otorgaban la facilidad de tener modelos listos para la simulación de equipos de fútbol de robots, pero esto mismo implicaba alejarnos del modelo realista que queríamos implementar para nuestros robots. Por ejemplo, el simulador de RoboCup [4] provee acciones para cada robot predefinidas y un modelo de movimiento, aceleración y fricción simplificado. Por otra parte, y al igual que el simulador de RoboCup, el simulador de FIRA es muy restrictivo en cosas como la cantidad de jugadores que componen cada equipo y la posición de juego inicial de la pelota. Por estos motivos, avanzamos en la identificación de otros modelos de simulación que estuvieran alineados con nuestros principios rectores.

Otra opción evaluada fue la simulación más realista posible utilizando cuerpos deformables y complejos modelos de fricción y colisiones. Pero este tipo de simulación suele ser muy costosa en tiempo y recursos, lo que se contrapone con los lineamientos identificados.

Nuestra elección, basándonos en los propósitos mencionados más arriba, es la de utilizar un simulador de sistemas dinámicos de cuerpos rígidos.

2.2 Cuerpos rígidos

Un cuerpo rígido es similar a un sistema de partículas y por lo tanto, la mayoría de las ecuaciones para un sistema de partículas se pueden utilizar en la dinámica de un cuerpo rígido. Por supuesto que la principal diferencia radica en que un cuerpo rígido es rígido. No hay movimiento de masa dentro del mismo. Como resultado, la posición relativa entre las partículas que lo componen no cambia [5].

Muchas de las cosas con las cuales estamos en contacto diariamente son rígidas para todos los propósitos prácticos. Aunque objetos como una silla o un cuchillo son flexibles y presentan vibraciones, estos efectos no tienen un gran impacto en nosotros. Algunos ejemplos de objetos que claramente no se comportan como cuerpos rígidos pueden ser gotas de agua, telas y los gases.

2.3 Dinámica de cuerpos rígidos

Un sistema dinámico es cualquier colección de cosas que se mueven o cambian. La dinámica de cuerpos rígidos (rigid body dynamics) trata del diseño de modelos matemáticos y algoritmos para predecir el movimiento de cuerpos y las fuerzas de contacto, incluyendo la fricción, que surgen entre ellos.

2.4 Motores de simulación de sistemas dinámicos de cuerpos rígidos

La simulación es la ejecución de un modelo matemático de un sistema dinámico. La simulación del movimiento de un cuerpo rígido se basa en la simulación del movimiento de una partícula. Pero a diferencia del caso de una partícula, si estamos trabajando con cuerpos rígidos, para determinar la posición de un objeto también tenemos que encontrar su orientación en el espacio.

El motor de simulación física se utiliza para contestar a la pregunta: “Dada la entrada (torque de los motores, gravedad, etc.), dime cómo se moverá el robot y qué fuerzas de contacto surgirán”. Por ejemplo, si un robot se mueve hacia delante y colisiona con la pelota, la solución al problema nos permitirá predecir la dirección y velocidad resultantes de la pelota, entre otras cosas.

En general, la simulación de sistemas dinámicos de cuerpos rígidos requiere la solución de un problema de complementariedad diferencial (differential complementarity problem o DCP), que puede ser pensado como un análogo a una ecuación diferencial ordinaria (ordinary differential equation u ODE). Las

ODEs son una categoría de ecuaciones diferenciales que involucran funciones de una sola variable y sus derivadas [6].

En la simulación, el tiempo se fracciona en intervalos discretos. Cada vez que se avanza el tiempo actual un paso dado, se realiza lo que se conoce como paso de integración, donde se ajustan el estado de todos los cuerpos rígidos para el nuevo valor de tiempo.

Los métodos de time-stepping para resolver DCPs requieren la solución de uno o más problemas de complementariedad por cada paso de tiempo, al igual que las soluciones numéricas de ODEs requieren varias evaluaciones de la función derivada.

2.5 Elementos de un sistema dinámico de cuerpos rígidos

Los motores de simulación ofrecen diferentes elementos a la hora de modelar un sistema. Los principales son cuerpos rígidos de diferentes formas (por ejemplo, cubos, esferas y cilindros), articulaciones con variadas características (por ejemplo, bisagras y pistones), contactos, colisiones y un modelo de fricción. Aparte de estos conceptos básicos, algunos proveen funcionalidad para modelar resortes y otros accesorios.

2.6 Características deseables de nuestro motor de simulación

Al momento de seleccionar un motor de simulación, y adicionalmente a nuestros lineamientos originales en cuanto al realismo de la simulación y los requerimientos de tiempo y recursos computacionales, consideramos adecuado que el motor elegido cuente con las siguientes características:

- Poder especificar fácilmente el modelo de los robots utilizados
- Soportar la simulación de un partido de fútbol de robots de dos equipos de hasta once integrantes cada uno
- Soporte para poder inicializar de manera automática cada episodio de simulación
- No requerir la supervisión de un operador
- Tener acceso al código fuente del motor y, en lo posible, que su licencia sea GPL o similar
- Que posea una buena comunidad de desarrolladores con participación activa en foros de discusión y otros medios
- Desarrollo robusto, estable y probado en el tiempo
- Soportar ambientes de desarrollo Windows ó Linux en arquitecturas x86

2.7 Elección de nuestro motor de simulación

Cuando enfrentamos la tarea de seleccionar un motor de simulación que cubra con nuestras necesidades encontramos una gran variedad de trabajos, cada uno con madurez y orientaciones diferentes. Algunos eran desarrollos incipientes y en sus primeros meses de vida, otros se orientaban a la simulación de un tipo de sistema dinámico en particular.

Luego de estudiar algunos motores y ver cómo cada uno se ajustaba a nuestros requerimientos, evaluamos dos alternativas: DynaMechs [7] y ODE (Open Dynamics Engine) [8].

DynaMechs es un motor de simulación en desarrollo desde 1991 que, aunque originalmente ideado para simular sistemas hidrodinámicos, posee las habilidades de simulación que necesita nuestro modelo. Es un paquete multiplataforma, orientado a objetos y desarrollado en C++. Posee varios componentes que resuelven las ecuaciones del modelo dinámico, llamados integradores, que permiten balancear la necesidad de exactitud y estabilidad numérica. Las principales falencias de este paquete es que no tiene un desarrollo y una comunidad activos.

Por otro lado, ODE es un desarrollo más reciente, cuyos comienzos datan del año 2000. A pesar de esto es bastante estable, posee un grupo de desarrolladores numeroso, tiene un buen número de proyectos que lo usan y una comunidad amplia y participativa. También es multiplataforma, está desarrollado en C++ y ofrece una API en C.

Para la implementación de nuestro simulador elegimos ODE (Open Dynamics Engine) v0.5, dado que cumple con los principios y características requeridas, aunque la robustez y estabilidad del mismo en algunas de nuestras pruebas no fueron especialmente buenas.

Cabe acotar que tiempo después encontramos otro motor, Newton [9], que puede ser evaluado para trabajos futuros teniendo en cuenta su estabilidad y velocidad.

Capítulo 3

Implementación del simulador

3.1 Organización en capas de nuestro simulador

Nuestro simulador está organizado en una arquitectura con las siguientes capas, donde cada una se implementa en base a la anterior:

- Motor de simulación física (ODE)
- Framework de simulación (implementado por nosotros)
- Modelos de los robots (implementado por nosotros)

Por otro lado, el framework de simulación puede operar en conjunto con un motor de visualización para mostrar la evolución de la simulación.

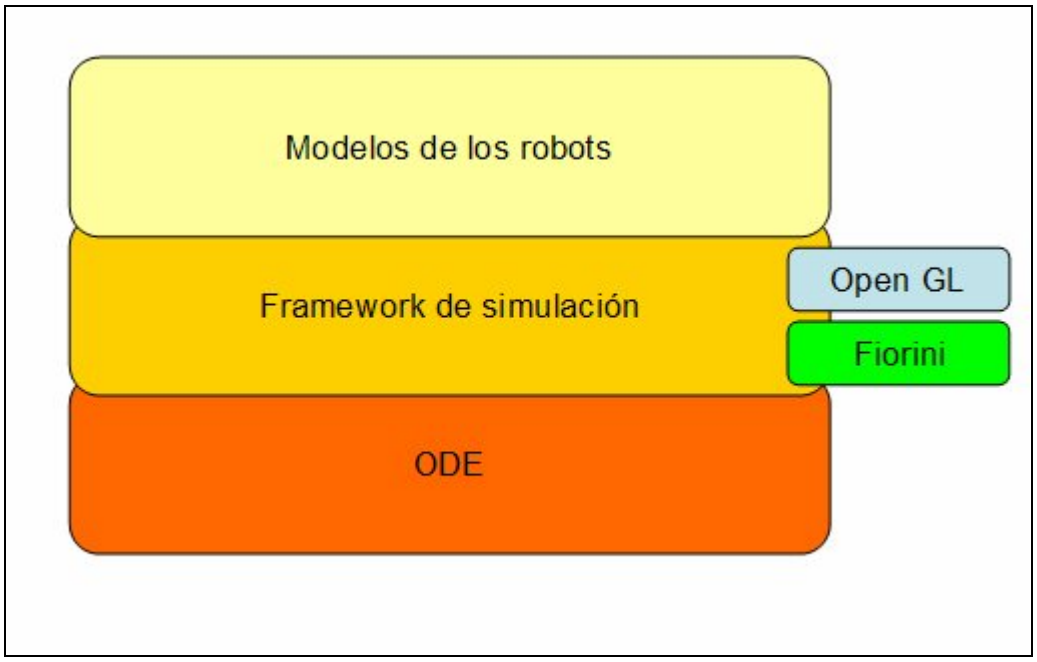

Figura 3.1: Esquema de las capas del simulador mostrando su relación entre sí o con componentes auxiliares u opcionales.

A continuación detallamos algunas características de cada capa.

3.2 ODE

ODE es nuestra elección de motor de simulación de sistemas dinámicos de cuerpos rígidos. Una de las más importantes diferencias con otros motores es que enfatiza la velocidad y la estabilidad de la simulación sobre la exactitud física de la misma. También provee mecanismos y opciones que permiten balancear la velocidad de cómputo y la exactitud de las soluciones, como ser:

- Modelo de contacto y fricción basado en la aproximación de Coulomb [10], donde mediante un coeficiente de fricción podemos simular distintos tipos de materiales
- Ecuaciones de movimiento derivadas de un modelo basado en multiplicadores de Lagrange (Trinkle/Stewart [11] y Anitescu/Potra [12])
- Un integrador de primer orden que, a menos que el paso temporal sea muy pequeño, no es muy exacto pero es rápido
- Método de time-stepping iterativo que no resuelve todo el modelo simultáneamente

ODE trabaja con ciertos conceptos que conviene enumerar. Los elementos básicos son los cuerpos rígidos, con vectores de posición, orientación, velocidad lineal y angular. Estos cuerpos se conectan entres sí mediante juntas para formar islas. Las juntas pueden ser de diferentes tipos, tales como bisagra, bola y hueco o pistón y restringen las relaciones espaciales que pueden tener los cuerpos [13].

ODE tiene dos componentes principales: el motor de simulación propiamente dicho y un motor de detección de colisiones, donde los contactos son duros (es decir que no hay penetración cuando dos cuerpos chocan). El motor de simulación tiene como objeto primario a uno llamado mundo, el cual contiene cuerpos. Por su parte, el motor de detección de colisiones se basa en un espacio, el cual puede contener otros espacios. A su vez, cada espacio puede contener formas geométricas básicas (geoms) que pueden ser cubos, paralelepípedos, esferas o cilindros. El modelo implementado tiene que asociar los cuerpos del motor de simulación con los geoms del detector de colisiones. En la figura 3.2 se puede ver parte del código que implementa los cuerpos y los geoms de un robot.

```
// Create chassis body
bodies[CHASSIS_BODY] = dBodyCreate(world);

spaceGroup = dHashSpaceCreate(space);

// Create chassis geom associated to the chassis body
geoms[CHASSIS_GEOM] = dCreateBox(spaceGroup, CHASSIS_LENGTH, CHASSIS_WIDTH,
        CHASSIS_HEIGHT);
dGeomSetBody(geoms[CHASSIS_GEOM], bodies[CHASSIS_BODY]);

// Create center wheel bodies and rotate them
bodies[CENTER_WHEEL_BODY] = dBodyCreate(world);
bodies[CENTER_REAR_WHEEL_BODY] = dBodyCreate(world);
```

```
// Create rear wheels
for (int i = LEFT_WHEEL_BODY; i <= RIGHT_WHEEL_BODY; i++)
  {
    // Create wheel
    bodies[i] = dBodyCreate(world);

    // Create wheel geom
    if (i == LEFT_WHEEL_BODY)
        {
            geoms[LEFT_WHEEL_GEOM] = dCreateSphere(spaceGroup, WHEEL_RADIUS);
            dGeomSetBody(geoms[LEFT_WHEEL_GEOM], bodies[i]);
        }
    else
        {
            geoms[RIGHT_WHEEL_GEOM] = dCreateSphere(spaceGroup, WHEEL_RADIUS);
            dGeomSetBody(geoms[RIGHT_WHEEL_GEOM], bodies[i]);
        }
  }

  // Create joint between the chassis and the left wheel
  joints[LEFT_WHEEL_JOINT] = dJointCreateHinge(world, 0);
  dJointAttach(joints[LEFT_WHEEL_JOINT], bodies[CHASSIS_BODY],
       bodies[LEFT_WHEEL_BODY]);

  // Create joint between the chassis and the right wheel
  joints[RIGHT_WHEEL_JOINT] = dJointCreateHinge(world, 0);
  dJointAttach(joints[RIGHT_WHEEL_JOINT], bodies[CHASSIS_BODY],
       bodies[RIGHT_WHEEL_BODY]);
```

Figura 3.2: Parte del código del método que inicializa un robot, donde se van creando y uniendo los objetos en ODE. Al objeto *world* se le van agregando los cuerpos que modelan el chasis y las ruedas (ver sección 3.4). Al mismo tiempo, al objeto *space* se le agregan *geoms* (caja y esferas), también representando al chasis y las ruedas, que luego se asocian a los correspondientes cuerpos de *world*. Por último se crean las juntas entre el chasis y las ruedas, en este caso del tipo bisagra. Para evitar mostrar una cantidad de texto excesiva, se eliminó de este cuadro lo referente a las ruedas de estabilidad delantera y trasera y donde se definen los parámetros de la masa del cuerpo. Sobre esto último cabe agregar simplemente que el centro de masa del chasis se trasladó hacia abajo, para lograr una mejor estabilidad.

Algunos problemas que se presentaron a la hora de implementar nuestro modelo fueron:

- la inestabilidad (causada, por ejemplo, por pasos de tiempo largos, grandes fuerzas, grandes masas interactuando con masas pequeñas, mucha fricción)
- la gran cantidad de parámetros para ajustar, y
- la baja velocidad obtenida en sistemas de mediana complejidad para que el modelo sea mayormente estable

3.3 Framework de simulación

Por encima del motor de simulación de ODE y utilizando su API en C, diseñamos e implementamos un pequeño framework orientado a objetos en C++ cuyos propósitos principales son abstraernos de las particularidades de ODE y facilitarnos la implementación de modelos de robots para experimentar con ellos.

Este framework es específico para el fútbol de robots ya que trabaja con conceptos tales como el campo de juego, los jugadores y la pelota.

La clase principal del framework se llama Simulation y es un wrapper para el proceso de simulación usando ODE. Provee un esquema de simulación basado en episodios, donde cada episodio finaliza cuando un equipo convierte un gol (o, mejor dicho, cuando la pelota ingresa en un arco) o bien cuando se alcanza un tiempo límite predeterminado. Se encarga de inicializar y restaurar ODE al comienzo y al fin de la simulación, inicializa los objetos a simular, y también interactúa con el motor de visualización para mostrar la simulación en forma gráfica.

El punto más importante de la clase Simulation es que provee puntos de expansión. Los puntos de expansión son puntos específicos en el diseño que se utilizan para aumentar o cambiar una funcionalidad dada. En particular, podemos mencionar los puntos de expansión utilizados para aumentar la funcionalidad de los métodos que inicializan/restauran el estado de un episodio, y el método que se ejecuta en cada paso de simulación. Para cada experimento de simulación se crea una clase derivada de esta.

Las clases Ball y Field implementan y modelan en ODE una pelota de golf y una cancha de fútbol, ambas según el reglamento de la categoría MiroSot [14] de FIRA.

En la figura 3.3 se puede ver un diagrama de clases simplificado de esta implementación.

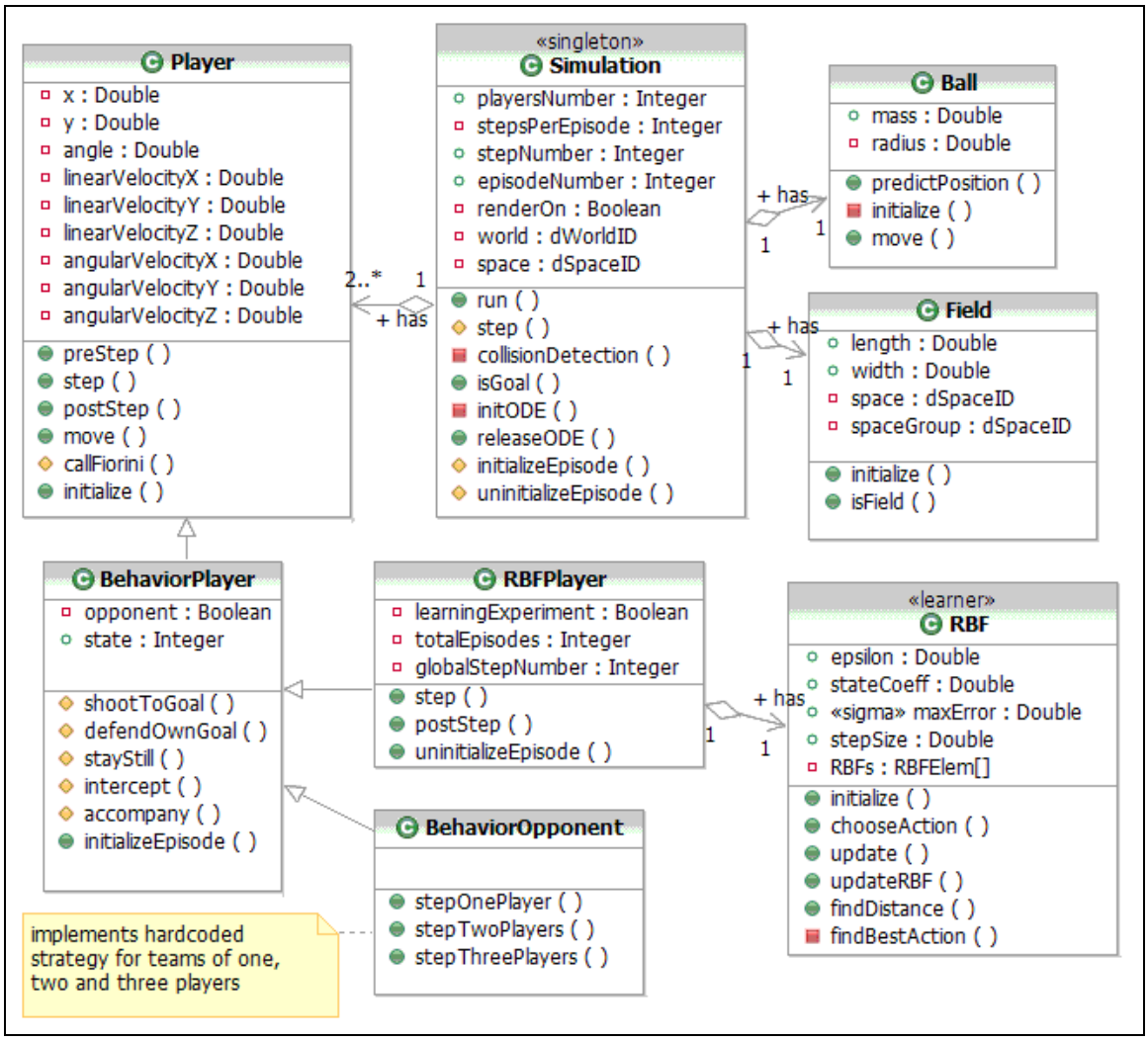

Figura 3.3: Diagrama de clases simplificado, conteniendo las principales clases del simulador. Las especializaciones de la clase Player se entenderán leyendo el Capítulo 5.

3.4 Modelos de los robots

Por último se proveen dos clases, Player y BehaviorPlayer, que sirven como clases base para modelar nuestros robots. La clase Player inicialmente modelaba un robot Yujin YSR-A [15], pero luego el grupo de inteligencia computacional del departamento implementó localmente un robot llamado CheBot [16]. Como estos últimos serían los que se continuaran utilizando para competencias de la FIRA decidimos implementar y optimizar el modelo para que, dentro de las posibilidades, se acerque a un CheBot real.

Para simular el CheBot se usó un paralelepípedo para el chasis y esferas para las ruedas, dado que la implementación de cilindros presentaba problemas de estabilidad. En total son cuatro ruedas: dos pequeñas en la parte delantera y

trasera para otorgar estabilidad, y las principales a los costados para la tracción. Las juntas entre las ruedas de estabilización y el chasis son de tipo bisagra, al igual que las juntas entre las ruedas de tracción y el chasis.

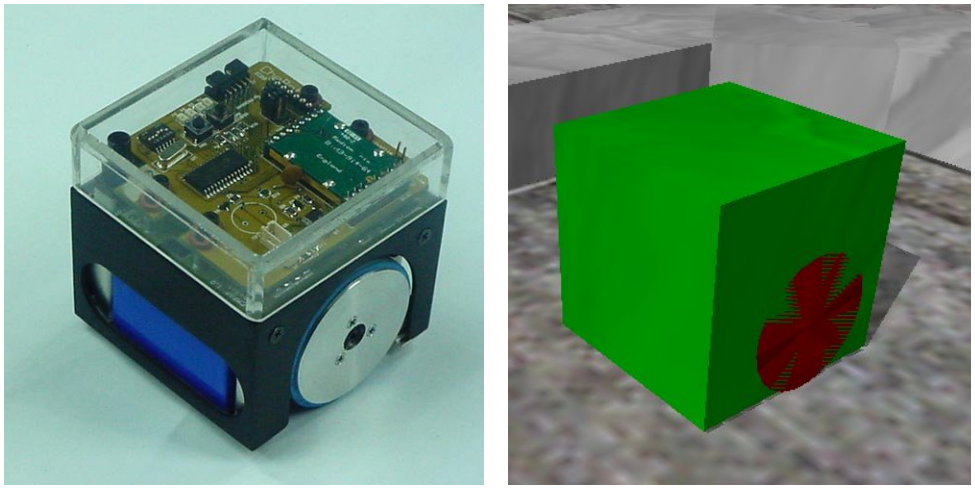

Figura 3.4: Comparación de un robot CheBot real [17] y uno generado por nuestro simulador y visualizado mediante OpenGL. Si bien las ruedas se modelan como esferas, en la visualización se muestran como cilindros, para que el aspecto general resulte más familiar.

La clase Player provee un método Player::move que utiliza un generador de trayectorias desarrollado por otro tesista del departamento [18]. Éste se basa en el método de velocity obstacle propuesto por Fiorini y Shiller [19] y provee una forma de calcular las velocidades de las ruedas de un robot para dirigirse hacia un objetivo especificado por una coordenada espacial, un ángulo de dirección y las velocidades de las ruedas con que se desea llegar.

La clase BehaviorPlayer deriva de Player e implementa diversos comportamientos básicos que se describirán más adelante.

3.5 Motor de visualización

El motor de visualización es un componente opcional del simulador, dado que éste puede utilizarse sin ninguna interfaz de usuario y con el único propósito de ejecutar un experimento.

El motor de visualización utilizado es el DrawStuff, incluido en ODE. Es un pequeño motor basado en OpenGL, de bajos requerimientos en cuanto a recursos computacionales, que corre en un thread independiente.

Este motor permite visualizar en tres dimensiones el desarrollo de un experimento, pudiendo durante su transcurso cambiar la posición de la cámara y dirigirla en cualquier dirección.

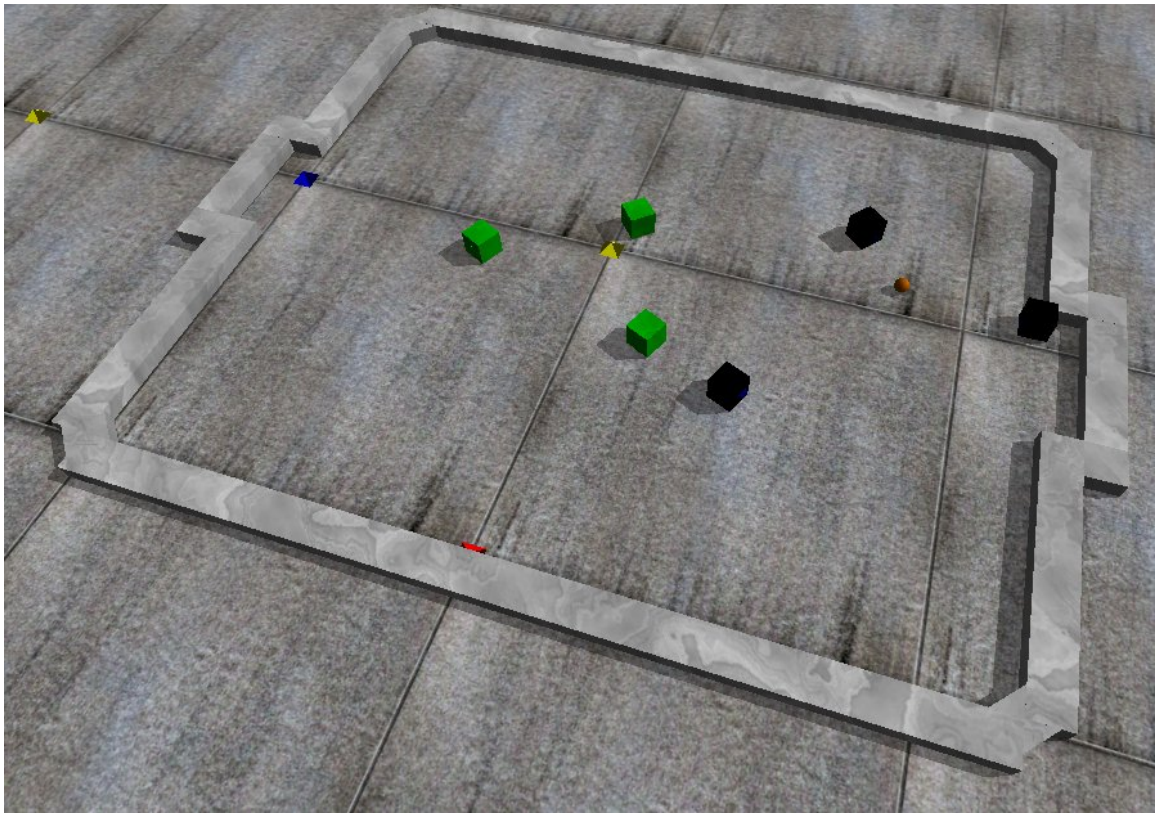

Figura 3.5: Un partido del simulador visualizado mediante OpenGL. En este caso, se utilizaron 3 jugadores por equipo. Los cubos verdes representan al equipo de robots que aprenden, mientras que los negros son el equipo rival, que usa una estrategia fija.

Capítulo 4

Aprendizaje por refuerzo

Existen diversas maneras de encarar el problema del aprendizaje en robótica. El aprendizaje supervisado trata del aprendizaje basado en ejemplos provistos por un agente externo entrenado. Muchas veces se torna muy difícil obtener los ejemplos necesarios del comportamiento deseado que sean a la vez correctos y representativos del espacio de situaciones que el agente deberá enfrentar. Esto se nota especialmente cuanto mayor es la dimensionalidad del espacio de situaciones. Si, por ejemplo, se considera un partido de fútbol de robots donde cada equipo cuenta con 11 jugadores, se deberán tener en cuenta a los 22 robots junto con la pelota. Y cada uno de ellos se asocia con múltiples dimensiones (por ejemplo, posición, rotación y velocidad). Vemos aquí lo complejo que puede llegar a ser la obtención de dichos ejemplos.

Desde otro punto de vista, los métodos evolutivos intentan alcanzar la solución óptima modelando elementos de la evolución biológica, tales como reproducción, mutación o selección natural. Los individuos de una población (soluciones candidatas a resolver un problema) se combinan y mutan a través de las generaciones y se van eligiendo los más aptos (las soluciones que obtienen mejor resultado en cierta función de aptitud). Por ejemplo, en los algoritmos genéticos se parte de una población inicial generada al azar, y en cada generación se les da más posibilidades de combinarse y generar nuevos individuos a las soluciones que mejor se adaptan. Esto se logra tomando dos soluciones padres y eligiendo alternadamente segmentos de la secuencia genética (que puede ser, por ejemplo, un arreglo de bits) de alguno de ellos. Además se introducen mutaciones, cambiando al azar el contenido de una solución, con el objetivo de no perder de vista posibles soluciones y evitar llegar a mínimos locales. A pesar de funcionar bien en muchos casos, estos métodos no interactúan con el entorno, lo que los hace poco adaptables a las particularidades de cada caso. El Teorema de No-Hay-Almuerzo-Gratis [20] nos dice que no hay un algoritmo que funcione en forma óptima para todos los problemas, sino que para cada uno va a haber una solución que se adapta mejor que otra.

A diferencia de otros enfoques, el aprendizaje por refuerzo (AR) se concentra más en el aprendizaje dirigido por un objetivo, basándose en la interacción con el entorno. Se trata de encontrar un mapeo entre situaciones y acciones, de manera tal de maximizar la suma de una señal numérica denominada refuerzo o recompensa. De esta manera, no se le dice al agente qué acciones debe elegir ante una situación dada, sino que él mismo debe descubrir qué acciones le otorgan la

mayor recompensa. Para descubrirlas deberá ir probando diferentes acciones durante su interacción con el entorno.

Esta técnica resulta particularmente atractiva en el caso de robots móviles. Como sugieren William D. Smart y Leslie P. Kaelbling en [21], el hecho de proveer una especificación de alto nivel del objetivo a lograr (mediante la elección de una función de refuerzo apropiada) resulta más fácil que describir los comportamientos de control de bajo nivel y, como resultado, adicionalmente está menos sujeta a posibles errores en su implementación.

4.1 Características del aprendizaje por refuerzo

4.1.1 Elementos

Un sistema de AR posee cuatro elementos principales [22]: una política, una función de refuerzo, una función de valor y, opcionalmente, un modelo del entorno.

Una política, comúnmente denominada π, define la manera en que un agente se comportará en un momento determinado. Es el mapeo entre los estados percibidos del entorno y las acciones que se pueden realizar desde ellos. El concepto de política es central para el AR dado que es suficiente para determinar el comportamiento de un agente.

La función de refuerzo, generalmente r, es la que define el objetivo que se persigue. Es un valor real determinado por el entorno que nos dice cuán deseable es un estado o un par estado-acción. El único objetivo del agente es maximizar la recompensa total obtenida a lo largo del proceso, llamada retorno, y por lo tanto es crítico definir correctamente con ella lo que deseamos hacer y no cómo hacerlo. Esto quiere decir que lo que se busca maximizar no es la recompensa inmediata, sino la acumulada a lo largo del tiempo.

La función de valor, denominada $V(s)$, caracteriza cuán deseable es un estado en un sentido de largo plazo. El valor de un estado según π, $V^{\pi}(s)$, es la recompensa total que un agente puede esperar obtener en el futuro a partir de ese estado, siguiendo una política π dada, tomando en cuenta los estados que pueden seguirle y las recompensas asociadas a ellos. Asimismo, la función de acción-valor $Q(s, a)$ nos dice lo propio acerca de un par estado-acción y $Q^{\pi}(s, a)$ indica el valor de tomar la acción a en el estado s y siguiendo π a partir de ahí. Uno de los componentes más importantes en un algoritmo de AR es un método eficiente para estimar estos valores. Estas funciones se van actualizando a lo largo de la tarea del aprendizaje mediante las llamadas operaciones de backup, donde se

transfiere información a un estado (o a un par estado-acción) desde los estados (o pares estado-acción) sucesores.

La función de refuerzo puede o no contener conocimiento previo acerca del modo de lograr el objetivo. Es decir, se puede hacer que esta función premie y castigue al agente solamente en base a la concreción del objetivo de la tarea y no indicando cómo hacerlo asignando refuerzo a los pasos intermedios, o bien usar algún conocimiento del problema para sesgar la función. Por ejemplo, en el dominio de fútbol de robots se podría premiar la cercanía a la pelota, sabiendo que es más fácil convertir goles de esa manera, y no solamente la concreción de un gol.

Por último, podemos tener un modelo del entorno. Por ejemplo, dado un par estado-acción, el modelo podría predecir el siguiente estado. El AR no requiere un modelo para su aplicación, pero puede utilizarse si está disponible o puede aprenderse.

Todos los agentes que utilizan AR tienen un objetivo específico, reciben información de su entorno, y eligen acciones que influyen sobre él. Más específicamente, el agente y el entorno interactúan en una secuencia de pasos en el tiempo. En cada momento el agente recibe cierta representación del estado del entorno, y en base a eso selecciona una acción del conjunto de acciones disponibles desde ese estado. En el siguiente paso, posiblemente en parte como consecuencia de la acción elegida, el agente recibe una recompensa numérica y se modifica el estado del entorno. Podemos ver esto en forma esquemática en la Figura 4.1.

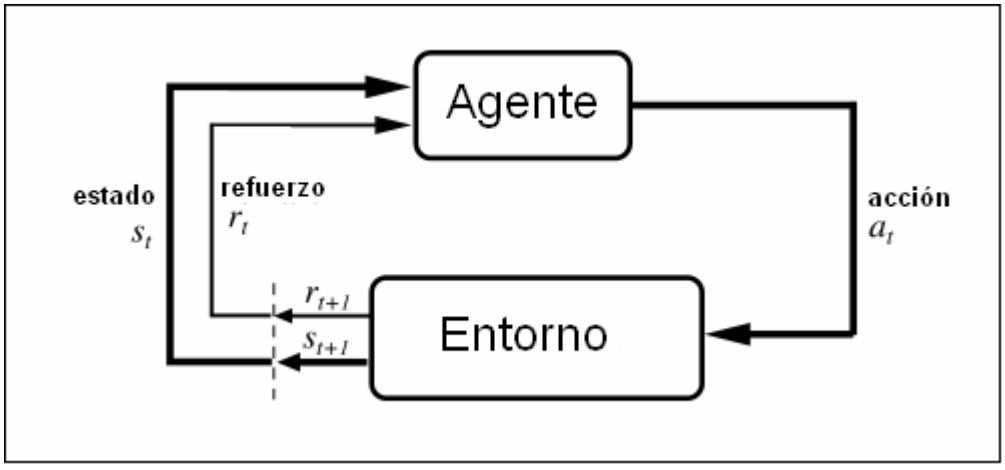

Figura 4.1: Esquema general de interacción entre los elementos del aprendizaje por refuerzo. El agente recibe información del entorno por medio de la lectura de los estados (la situación del entorno en cada momento) y del refuerzo obtenido como consecuencia de la última acción tomada, y a su vez influye sobre éste por medio de las acciones.

Una tarea de AR puede ser partida en secuencias de pasos llamadas episodios, donde cada uno finaliza en un estado terminal, o bien podemos encontrarnos con tareas continuas, donde no hay un límite para la ejecución de la misma.

4.1.2 Propiedad de Markov

En los modelos donde se verifica la propiedad de Markov, se mantiene toda la información relevante en el estado actual, sin necesidad de recurrir al historial de estados y acciones que llevaron a un entorno a dicho estado. De ahí se desprende que la mejor política para elegir una acción en función de un estado de Markov es tan buena como si lo fuera en función del historial completo. En algunos entornos no se cumple estrictamente esta propiedad, lo que hace que no podamos asegurar los resultados teóricos que se verifican en modelos que sí la poseen, aunque se pueden usar los mismos algoritmos.

A una tarea de AR que satisface la propiedad de Markov se la llama MDP (Proceso de Decisión de Markov), y si su espacio de estados y acciones es finito, tenemos un MDP finito. Un MDP se define mediante un conjunto de estados y acciones, las probabilidades de transición (cuán probable es pasar de un estado *s* a un estado *s'* ejecutando la acción *a*):

$$P^{a}_{s\,s'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \qquad (4.1)$$

y el refuerzo esperado dado un estado, una acción y un estado siguiente:

$$R^{a}_{s\,s'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \qquad (4.2)$$

Para buena parte de la teoría de AR se asume que los entornos son MDP finitos.

4.1.3 Prueba y error

La diferencia básica entre el AR y otros tipos de aprendizaje radica en la utilización de información de entrenamiento que evalúa las acciones tomadas, en lugar de tener que instruir al agente indicándole las acciones correctas. Esto hace que en el AR la búsqueda de una política óptima esté basada en la prueba y el error. El agente prueba diferentes acciones, para una misma situación, y utiliza el error devuelto por el entorno, el refuerzo, para evaluar su desempeño. Este tipo de retroalimentación indica cuán buena fue la decisión de tomar una acción determinada, pero no dice nada acerca de si fue la mejor o la peor acción posible, como en los métodos de aprendizaje supervisado.

4.1.4 Refuerzo demorado

En general, cada acción puede afectar no sólo su refuerzo inmediato, sino que también puede afectar la siguiente situación y, a través de ella, los refuerzos subsiguientes. Por ello, una acción que tenga asociada una recompensa inmediata muy baja puede, en realidad, ofrecer mucha recompensa en el futuro al influir sobre las situaciones venideras. A esto se lo conoce como el problema del refuerzo demorado, donde una acción puede estar involucrada en la obtención de un alto nivel de recompensa independientemente de su recompensa inmediata.

En general, los sistemas de un agente que aprende mediante refuerzos en un dominio de tiempo extendido (es decir, que involucran secuencias de al menos dos acciones) y los sistemas multi-agente comparten una dificultad común conocida como el problema de asignación de crédito. Los sistemas multi-agente presentan el problema de asignación de crédito estructural consistente en tratar de determinar la contribución de un agente en particular en una tarea común. Por otro lado, los sistemas de un agente en un dominio de tiempo extendido conllevan el problema de asignación de crédito temporal, donde se trata de determinar la contribución de una acción en particular en la calidad de la secuencia de acciones completa.

4.1.5 Exploración vs. explotación

En una situación determinada, el agente debe elegir una acción de un conjunto de acciones posibles, quizás limitada por la situación en la que se encuentra. De este conjunto habrá una o más acciones cuyo valor estimado sea mayor. Dichas acciones se denominan acciones golosas.

Se entiende por explotación aquellos momentos donde el agente selecciona acciones golosas. Es decir, elige las acciones que hasta el momento le dieron mejores resultados.
Pero para poder descubrir este tipo de acciones deberá elegir en algunos momentos acciones diferentes a la golosa. En estos casos, se dice que explora otras posibilidades para hallar acciones asociadas a mayores recompensas, de manera de poder posteriormente explotarlas. La exploración posibilita mejorar la estimación de los valores de los estados, al considerar las acciones no golosas.

Un agente debe explotar su conocimiento para acumular mayor recompensa, pero también debe explorar posibilidades para poder hacer mejores decisiones en el futuro. Una de las características distintivas del AR es esta necesidad de encontrar un balance entre la exploración y la explotación.

4.1.6 Complejidad del espacio de estados-acción

La complejidad del espacio de estados-acción es un problema ampliamente tratado en AR y en otras áreas de investigación. En [23] se destaca que el número de estados considerados en un algoritmo tradicional de AR crece exponencialmente con la dimensionalidad del espacio de estados y esta a su vez crece linealmente con la cantidad de agentes. Estos problemas los encontraremos en el problema que tratamos en nuestro trabajo, ya que veremos que usamos varias dimensiones y varios agentes. De todas maneras, antes de eso, debemos resolver como discretizar el espacio de estados, ya que las variables que elegiremos para usar como dimensiones del espacio de estados de nuestro problema son continuas y por ende infinitas, mientras que la técnica de AR se basa en un conjunto de estados finito. Estas razones nos llevan a necesitar de un método de discretización que, además de modelar en forma apropiada los estados reales del problema, reduzcan considerablemente la cantidad de estados, intentando que esta representación se aleje de la realidad lo menos posible.

4.1.7 Descuento

El descuento (discount), representado por el parámetro γ, nos dice cuánto valen en el presente los retornos futuros. El retorno con descuento R_t es igual a $r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^n r_{n+1}$, donde γ es la tasa de descuento. Si esta tasa vale 0, al agente se lo considera miope, dado que sólo le importa el paso actual. Por otro lado, una tarea sin episodios no puede tener una tasa de descuento que valga 1, ya que la suma de los retornos futuros sería infinita.

4.2 Métodos de aprendizaje por refuerzo

4.2.1 Programación dinámica

Existen diversos métodos para resolver el problema del AR. Uno de ellos es la programación dinámica. El principal problema es que supone un conocimiento completo del modelo del entorno y resulta poco aplicable en la práctica, aunque es importante teóricamente.

Los métodos de programación dinámica actualizan las estimaciones de los valores de los estados en base a estimaciones de los valores de los estados sucesores. Esto se conoce como bootstrap. El objetivo es hallar las políticas óptimas, para lo cual hay que hallar las funciones de valor y acción-valor óptimas (V^* y Q^* respectivamente). Éstas deben satisfacer las ecuaciones de optimalidad de Bellman:

$$V^*(s) = \max_a E\{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\} = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] \quad (4.3)$$

$$Q^*(s, a) = E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1, a'}) \mid s_t = s, a_t = a\} = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_a Q^*(s', a') \right] \quad (4.4)$$

$\forall s \in S$, $a \in A(s)$, $s' \in S$ (siendo S el conjunto de estados posibles y $A(s)$ las acciones posibles en casa estado).

Luego, modificando la ecuación de Bellman para convertirla en una regla de asignación para una política π, se obtienen aproximaciones sucesivas:

$$V_{k+1}(s) = E_\pi\{ r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s\} = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]. \quad (4.5)$$

El Algoritmo de Evaluación de Política utiliza este método para obtener el valor de V para una política π. Luego de inicializar en 0 las $V(s)$ de todos los estados), en cada paso se actualizan las estimaciones de todos los $V(s)$ según la ecuación 4.5, repitiéndose este proceso hasta que converja (hasta que las diferencias de los nuevos valores respecto a los viejos no superen cierta cota).

```
inicializar V(s) ← 0, ∀ s ∈ S
repetir
     dif ← 0
     para cada s ∈ S
          v ← V(s)
          V(s) ← Σ_a π(s,a) Σ_s' P^a_ss' [R^a_ss' + γV(s')]
          dif ← max(dif, | v − V(s) | )
hasta que dif < θ (un número positivo cercano a cero)
```

Figura 4.2: Pseudocódigo del Algoritmo de Evaluación de Política

A la par de evaluar la política, en el transcurso de la tarea de aprendizaje corresponde preguntarse si se puede mejorar la política que viene siendo usada, para lo cual se utiliza el Teorema de Mejoramiento de Política (Policy Improvement Theorem):

$\forall s \in S$, si $Q^\pi(s, \pi'(s)) \geq V^\pi(s) \Rightarrow \forall s \in S$, π' es igual o mejor que π,

es decir que si el valor esperado de un estado al seguir la nueva política es mayor o igual que el valor de ese estado con la política actual (para todos los estados), entonces la nueva política es mejor o igual que la política en uso. Esta constatación de un posible cambio de la política a nivel de un estado se puede extender a todos los estados, considerando la nueva política golosa π' (la que da un mejor resultado en el corto plazo) dada por

$$\pi'(s) = arg\ max_a\ Q^{\pi}(s, a) \qquad (4.6)$$

es decir, hacer que en la nueva política, la acción tomada en cada estado *s* sea la que ofrece un mejor resultado con la información disponible.

Entre estos dos procesos se va generando un ciclo llamado de iteración de políticas (policy iteration) que alterna entre el algoritmo de Mejoramiento de Política y el de Evaluación de Política. De este modo, partiendo de una política y función de valor arbitraria se llega a la política y función de valor óptimas. El algoritmo de este ciclo se puede ver en la figura 4.3 y su esquema general, en la figura 4.4.

```
1 – inicialización
V(s) ∈ ℜ, π(s) ∈ A(s) arbitrariamente ∀ s ∈ S

2 – Evaluación de Política
repetir
      dif ← 0
      para cada s ∈ S
            v ← V(s)
            V(s) ← Σ_s' P^a_ss' [R^a_ss' + γV(s')]
            dif ← max(dif, | v − V(s) | )
hasta que dif < θ (un número positivo cercano a cero)

3 – Mejoramiento de Política
Política-estable ← verdadero
para cada s ∈ S
      b ← π(s)
      π(s) ← arg max_a Σ_s' P^a_ss' [R^a_ss' + γV(s')]
      si b ≠ π(s)
          política-estable ← falso
          ir a paso 2
      si no
           Parar
```

Figura 4.3 – Algoritmo del Ciclo de Iteración de Políticas. A(s) es el conjunto de todas las acciones posibles realizadas a partir del estado s.

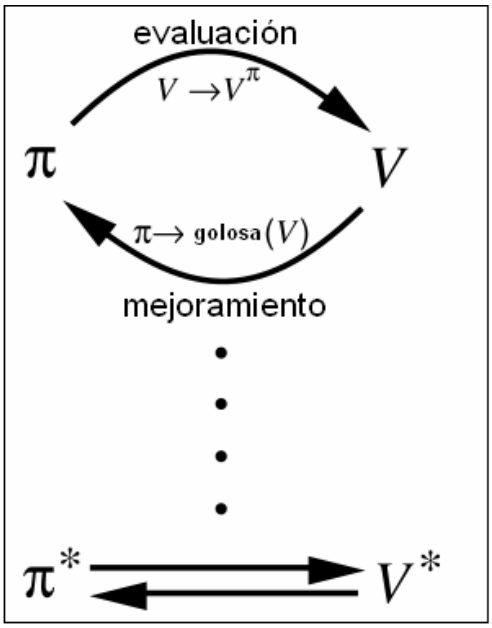

Figura 4.4: Esquema general del Ciclo de Iteración de Políticas

4.2.2 Monte Carlo

Otra clase de métodos usados en AR son los llamados métodos de Monte Carlo. A diferencia de lo que ocurre en programación dinámica, no se asume un completo conocimiento del comportamiento del entorno, con todas las probabilidades de transición entre estados, sino que solamente se hace uso de la experiencia adquirida hasta ese momento por el agente. Es necesario armar un modelo, pero éste solo necesita contener transiciones de muestra y no la distribución completa de probabilidades que se requiere en programación dinámica. Monte Carlo resuelve el problema de AR en base a promedios de retornos de muestra (sample returns). Otra particularidad de estos métodos es que son incrementales por episodios y no por pasos temporales, es decir que las estimaciones de valores y cambios de política se realizan al final de cada episodio. Dado que se permite enfocar el problema en un subconjunto pequeño de estados, se logra facilidad y eficiencia en comparación con programación dinámica.

El algoritmo de control del método de Monte Carlo implementa la iteración de políticas explicada en la sección anterior, aunque al no haber disponible un modelo no es suficiente la información que brindan los valores de los estados, y se hace necesario considerar también las acciones, con lo que la función que se quiere estimar pasa a ser Q^* en vez de V^*. La figura 4.5 muestra el pseudocódigo de este algoritmo.

```
Inicialización (∀ s ∈ S, a ∈ A)
      π ← política arbitraria
      Q(s, a) ← arbitraria
      Retornos (s, a) ← lista vacía

Repetir eternamente:
      Generar un episodio usando π
      Evaluación - Para cada par s, a que aparece en el episodio:
            R ← retorno a partir de la primera ocurrencia de s y
                        a en el episodio
            Agregar R a Retornos (s, a)
            Q(s, a) ← Promedio (Retornos (s, a) )
      Mejoramiento - Para cada s que apareció en el episodio:
            π (s) ← arg max_a Q (s, a)
```

Figura 4.5: Algoritmo de control del método de Monte Carlo.

Este método asume que todos los pares estado-acción serán visitados infinitas veces en el transcurso de una infinita cantidad de episodios, algo que en la práctica es imposible aplicar. Por esto se implementan algoritmos en donde se mantiene la exploración, es decir que no siempre se utiliza la acción que sugiere la política para un cierto estado.

4.2.3 Diferencias temporales

En los métodos denominados de diferencias temporales, al igual que en los métodos de Monte Carlo, se aprende directamente de la experiencia, sin un modelo del entorno. Al igual que los métodos de programación dinámica, se obtienen las estimaciones basándose en otras estimaciones aprendidas, es decir que hacen bootstrap.

La ventaja de los métodos de diferencias temporales es que no requieren un modelo del entorno y se implementan naturalmente de una forma on-line e incremental. Para realizar las actualizaciones de los estimados no hay que esperar hasta el fin del episodio, sino que éstas se realizan en cada paso. Esto

posibilita su aplicación cuando los episodios se extienden mucho o cuando no hay episodios sino un proceso continuo. Además, algunos métodos Monte Carlo deben ignorar parte de los episodios donde se toman acciones experimentales, lo cual retarda el aprendizaje. La figura 4.6 muestra un algoritmo para estimar V^{π} mediante diferencias temporales.

```
inicializar V(s) arbitrariamente, π con la política a evaluar

repetir (para cada episodio)
     Inicializar s
     repetir (para cada paso del episodio)
          a ← la acción dada por π para s
          ejecutar a
          observar el retorno r y el siguiente estado s'
          V(s) ← V(s) + α [r + γ V(s') − V(s)]
          s ← s'
      hasta que s sea terminal
```

Figura 4.6: Algoritmo de evaluación de politica de Diferencias Temporales.

La implementación de los métodos de diferencias temporales puede ser on-policy (como el algoritmo Sarsa), donde se intenta mejorar o evaluar la política usada para tomar decisiones, o bien off-policy, donde la política que se estima está separada de la que se usa para generar un comportamiento (behavior policy).

4.2.3.1 Q-Learning

Un método de diferencias temporales off-policy es Q-Learning. El paso de actualización de la función de estimación está dado por:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\,[r_{t+1}, + \gamma\, max_a\, Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (4.7)$$

donde α es el tamaño del paso (step size) que determina la tasa de cambio del valor actual hacia la nueva estimación, y Q (la función acción-valor usada para aprender) aproxima Q* (la función óptima), dada en este caso por el máximo de los valores del paso siguiente, independientemente de la política que se aplica, lo cual simplifica drásticamente el análisis del algoritmo. La figura 4.7 muestra el algoritmo de Q-Learning completo.

```
inicializar Q(s, a) arbitrariamente
```

```
repetir (para cada episodio)
     s ← un estado arbitrario
     repetir (para cada paso del episodio)
          elegir a de s usando una política derivada de Q
          ejecutar a; observar r y s'
          Q(s, a) ← Q(s, a) + α [r + γ max_a' Q(s', a')
                        - Q(s, a)]
          s ← s'
      hasta que s sea terminal
```

Figura 4.7: Algoritmo de Q-Learning.

La convergencia de Q-Learning fue probada por Watkins en [24]. El hecho de que su convergencia haya sido demostrada y, adicionalmente, las ventajas que comentamos más arriba de los métodos de diferencias temporales, son los motivos que nos llevaron a elegir Q-Learning como método de aprendizaje para nuestros experimentos.

Capítulo 5

Caso de aplicación

Como se concluyó en el capítulo anterior, para validar nuestro simulador evaluaremos experimentalmente la técnica Q-Learning aplicada al fútbol de robots.

En general, el problema que abordaremos en esta parte del trabajo es el de la obtención de políticas en un sistema multi-agente sin coordinación. En un sistema multi-agente sin coordinación el control es descentralizado y no existe comunicación entre los agentes. Cada uno de ellos es independiente y sigue su propia política. Con estrategia nos referimos a una política que permita la selección de diferentes comportamientos base para cada jugador. En particular, nos basaremos en partidos de fútbol de robots con los lineamientos definidos en el reglamento de la categoría MiroSot de la FIRA.

Se dividió la tarea en episodios, cada uno correspondiendo a un partido que finaliza con un gol de cualquier equipo, o bien por haberse cumplido el tiempo, que se fijó en 60 segundos.

En el último campeonato de FIRA llevado a cabo, la categoría MiroSot tuvo dos ligas: middle (5 jugadores en cada equipo) y large (11 jugadores). La cancha varía sus dimensiones, dependiendo de la liga, desde 220 cm x 180 cm a 400 cm x 280 cm. En todos los casos la cancha tiene dos arcos, se utiliza una pelota de golf naranja y cada equipo cuenta con una cámara posicionada sobre el campo de juego para obtener una visión global del partido. Los robots están limitados a dimensiones de hasta 7.5 cm x 7.5 cm x 7.5 cm y a un peso de 650 g.

5.1 Tratamientos existentes del problema

El fútbol de robots ya cuenta con cierta madurez y las formas de atacar este problema son muy variadas.

Un enfoque muy utilizado es la especialización de cada jugador para que realice comportamientos que, posiblemente, no comparte con otro jugadores [25]. En este caso, cada jugador puede comportarse únicamente como un arquero, defensor o atacante, por ejemplo. Cada uno de estos roles puede tener un área de acción determinada, donde únicamente se activará alguno de sus comportamientos cuando se detecte algún cambio dentro de ella. Una desventaja

de este enfoque es la especialización permanente de sus jugadores, probablemente impidiendo que actúen en casos necesarios, imposibilitando la cooperación con otros jugadores de su mismo equipo. Una posible mejora es la asignación dinámica de roles durante el juego [26].

Un paso adelante están los enfoques que proponen la utilización de formaciones para coordinar a todo un equipo [27]. Una formación es una estructura del equipo que define un conjunto de roles o posiciones con comportamientos asociados. Los robots son dinámicamente mapeados a esas posiciones y están equipados con el conocimiento necesario para jugar en cualquier posición de varias formaciones.

El enfoque utilizado por el equipo del departamento, UBA-Sot [28], es la obtención de comportamientos cooperativos en forma emergente a partir de comportamientos de más bajo nivel basados en el desplazamiento de un robot desde una posición a otra de la cancha. Cada comportamiento de bajo nivel tiene una precondición diferente asociada a él, y éstos se utilizan junto a predictores para el movimiento de la pelota y los otros robots para conformar un método de planificación muy rápida. De esta manera se pretende utilizar la planificación para obtener comportamientos más cercanos a una postura reactiva que a una deliberativa.

Una característica de los enfoques anteriores es que involucran un gran volumen de conocimiento a priori que se utiliza para modelar y definir las estrategias de un equipo. El aprendizaje por refuerzo estuvo recientemente recibiendo mayor atención, involucrando poco o ningún conocimiento a priori y otorgando mayor capacidad de comportamientos reactivos y adaptativos.

Existen diversos enfoques basados en AR para el fútbol de robots. Como ejemplo podemos mencionar el uso de modular Q-Learning [29]. Su arquitectura consiste en múltiples módulos de aprendizaje y un módulo mediador. El módulo mediador selecciona la acción apropiada para el agente basándose en el valor-Q obtenido por cada módulo de aprendizaje. Para obtener una mejor performance, aparte de los valores-Q, el módulo mediador también considera la información de estado en el proceso de selección de una acción.

En [30] se propone un mecanismo llamado VQQL, que integra Q-Learning como técnica de AR y cuantización vectorial como técnica de generalización del espacio de estados. Luego se constata su utilidad aplicándolo en el dominio del fútbol de robots y en particular, usando el simulador de RoboCup. El método consta de dos fases: aprendizaje del cuantizador y aprendizaj de la función Q. Para la primera se usan clusters definidos por sus centroides para representar a todos los estados y se implementa el Algoritmo Generalizado de Lloyd, que en

base a muestras sucesivas de estados reales va recalculando los centroides de los clusters hasta que el algoritmo converge y se obtiene el vector cuantizador (un mapeo de los estados originales a los clusters generados). Para la segunda, se utiliza el algoritmo de Q-Learning visto en la sección 4.2.3.1, con una tabla donde para cada cluster generado en la primera fase hay un valor Q para cada acción.

En una línea similar, existe un trabajo [31] donde se aplica AR a un dominio que consiste en mantener a un robot con forma de auto dentro de una pista. Al utilizar un modelo cinemático, en donde el controlador conoce información relativa al camino actual (curvatura, error de orientación, posición) pero no al camino futuro, se afirma que esta idea se podría utilizar en entornos más dinámicos, como RoboCup. Para discretizar el espacio de estados se utiliza un aproximador de funciones basado en casos. Cada caso corresponde a un estado visitado con anterioridad y guarda información respecto al valor Q para cada acción. Se toman los casos para los que la distancia de su componente de estado es menor a cierto umbral respecto al estado real y de entre ellos se elige la acción con mayor valor Q.

5.2 Definiciones y base del aprendizaje

5.2.1 Efectividad

Para medir el desempeño de un robot o un equipo y poder compararlo con el del rival, con el de alguna otra política o con el de la misma política en diferentes fases del aprendizaje fue necesario definir una medida de efectividad. Las variables del dominio que se tuvieron en cuenta fueron la cantidad de goles a favor y en contra, así como también el tiempo promedio de gol de uno y otro equipo. Finalmente, por simplicidad, se resolvió considerar solamente los goles de los equipos (es decir, los episodios terminados en gol, lo que hace descartar los empates), y definir efectividad de la siguiente manera:

Efectividad = # goles equipo propio / (# goles equipo propio + # goles equipo rival) *(5.1)*

5.2.2 Función de refuerzo

Como vimos en el capítulo anterior, al mencionar los elementos de AR, la política es aprendida en base a una única fuente de información denominada valor de refuerzo, que evalúa las acciones tomadas en un estado y tiene en cuenta el estado al que se llegó luego de realizada la acción: se recibe un valor de refuerzo positivo o negativo de acuerdo a cómo esté definida la función de refuerzo. De esta manera, el aprendizaje por refuerzo permite eludir los problemas de construir un modelo explícito del comportamiento a aprender o recabar una base de aprendizaje necesaria por el aprendizaje supervisado.

La función de refuerzo define los valores de refuerzo para cualquier elemento en el dominio S x A x S, donde S es el espacio de situaciones y A es el de acciones. El último elemento denota la situación resultante luego haber ejecutado una acción determinada en una situación determinada. Generalmente se invierte mucho esfuerzo en refinarla. Es uno de los componentes principales en el aprendizaje y su diseño hará que se logre la convergencia esperada o no a la política deseada. Mediante ella se puede incorporar conocimiento a priori a la política deseada y, de esa manera, sesgar el proceso de aprendizaje, probablemente mejorando la velocidad de convergencia. Cabe acotar que introduciendo dicho sesgo se podría estar vulnerando la capacidad de un método para encontrar nuevas soluciones a un problema, no previstas anteriormente.

La función de refuerzo usada para nuestros experimentos se definió de la siguiente manera:

$$r(s, a, s') = \begin{cases} 1 \text{ si s pertenece a las situaciones donde la pelota está en el arco contrario} \\ -1 \text{ si s pertenece a las situaciones donde la pelota está en nuestro arco} \\ 0 \text{ en cualquier otro caso} \end{cases}$$

En este caso no estamos introduciendo información adicional ya que la función de refuerzo está directamente relacionada con el objetivo del juego de la manera más inmediata (premio al convertir un gol, castigo al recibirlo). También se puede ver que es una función de refuerzo demorado, por el hecho de que un estado no en situación de gol (por ejemplo, cuando la pelota está en la mitad de la cancha) deberá esperar que pasen varios pasos temporales para que las operaciones de backup hagan efecto sobre dicho estado.

5.2.3 Navegación

Para calcular la trayectoria de un robot a su destino nos orientamos al enfoque que eligió P. Fiorini en [32]. En dicho trabajo se ataca el problema basándose en evitar obstáculos en un entorno dinámico, afirmando que, dado que el cálculo de trayectorias en un entorno cambiante es un problema intratable, se deben usar heurísticas para resolverlo.

Primero se resuelve el problema cinemático (encontrar una trayectoria que considere posición y velocidad de los obstáculos, así como una aproximación a las restricciones dinámicas del robot) y luego se usa una optimización dinámica para minimizar el tiempo de movimiento.

Se realiza un planeamiento jerárquico, en donde primero se prioriza evitar cualquier obstáculo, luego llegar al punto de destino, luego reducir el tiempo de movimiento, luego elegir la trayectoria, etc.

En todo momento se calcula el Velocity Obstacle (VO) para cada obstáculo, que servirá para determinar las velocidades del robot que evitarán una colisión entre el robot y el entorno en algún momento futuro.

Se calcula el llamado cono absoluto de colisiones CC_B, aseverando que las velocidades que caigan en ese cono van a garantizar una colisión del robot A con un obstáculo B. Por otro lado, el conjunto de velocidades alcanzables de A en cierto tiempo ($FV_A(t_0)$) estará determinado por el segmento circular dado por los círculos de máxima y mínima aceleración partiendo de A y un cono representado por los ángulos de máximo giro, siempre en un intervalo Δ_t. Finalmente, el VO de A debido a B se define como la intersección del cono de colisión absoluto con el conjunto de velocidades alcanzables, y $SV_A(t_0)$ (el conjunto de velocidades seguras para A en t_0) se define como la diferencia de conjuntos entre $FV_A(t_0)$ y $VO_B(t_0)$ (el VO de A debido a B en t_0):

$$SV_A(t_0) = FV_A(t_0) \setminus VO_B(t_0) \qquad (5.2)$$

Para múltiples obstáculos, se define MVO como la unión de todos los VO individuales y las velocidades que evitan colisiones deben caer en la diferencia de conjuntos entre $FV_A(t_0)$ y MVO.

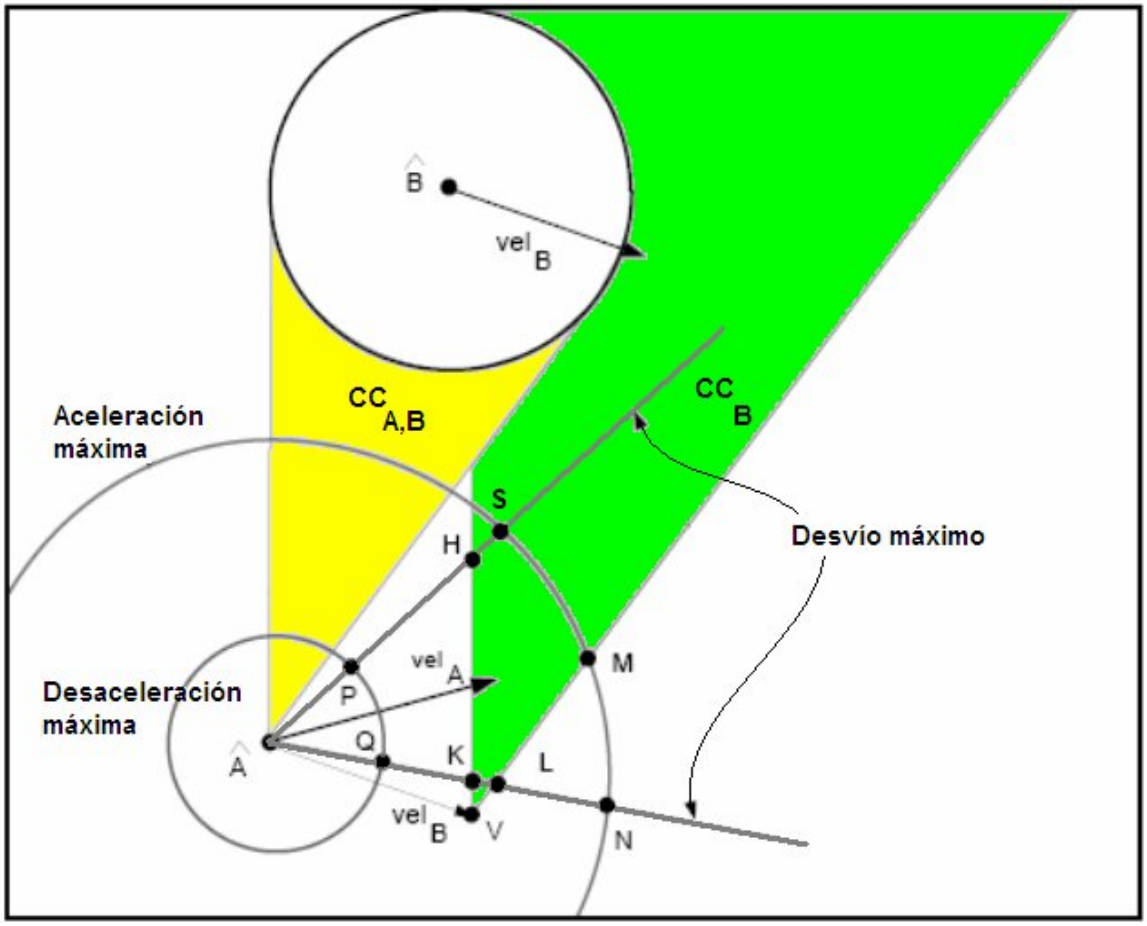

Figura 5.1: A es reducido a un punto (su centro) y a B le es incrementado el radio de A. El cono de colisión relativo (en amarillo) tiene su vértice en A e indica que si la velocidad relativa entre A y B cae en el cono, habrá colisión. El cono de colisión absoluto CC_B (en verde) tiene su vértice en A trasladado en la velocidad de B e indica que si la velocidad de A cae en él, habrá colisión. FV_A está delimitado por el segmento circular con vértices PQNS. Finalmente, SV_A (t_0) está delimitado por la intersección entre CC_B y FV_A, lo cual se indica en el segmento KLMSH.

Una vez logrado el objetivo de evitar obstáculos (el más importante en la jerarquía), se evalúa si se pueden alcanzar las tareas de menor jerarquía (como llegar más rápido). El análisis de la estructura de los VOs permite seguir refinando la maniobra para evitar colisiones, definiendo si va a pasar por delante o por detrás del obstáculo. El robot puede planificar una trayectoria agresiva, pasando delante de los obstáculos, con el riesgo de moverse al límite de sus posibilidades y posiblemente colisionar con un obstáculo, o una trayectoria más conservadora, donde puede elegir (dependiendo de factores como el tamaño y la velocidad) pasar o ceder el paso a otro vehículo / robot.

S. Soria y J. Santos encontraron algunas dificultades para aplicar esta propuesta al dominio de fútbol de robots. Una de ellas es cuando el conjunto de velocidades para elegir es vacío y se debe mover al robot. Otra dificultad es que el objetivo

contempla sólo las coordenadas cartesianas y no el ángulo o la velocidad final -algo necesario a tener en cuenta en fútbol de robots-. Finalmente, otro impedimento para aplicar VO al fútbol de robots es que la forma de los obstáculos no necesariamente es circular –por ejemplo, las paredes de la cancha-. Los autores exponen en [33] un conjunto de ideas que intenta resolver esos problemas. Esta última versión fue la que aplicamos a nuestro simulador, delegándole entera responsabilidad para determinar las velocidades de las ruedas (único factor necesario para determinar la velocidad y dirección de los robots), una vez establecido el punto, velocidad y ángulo que se quiere alcanzar en el destino.

5.2.4 Predicción de la posición de la pelota

Luego de observaciones preliminares de algunos comportamientos, vimos que los resultados no eran muy satisfactorios. Nos dimos cuenta de que para calcular el objetivo nos estábamos basando en la posición actual de la pelota, lo cual hacía que una vez que estuviésemos allí, probablemente la pelota se hubiera movido y tuviésemos que recalcular la trayectoria. Otro problema es que al no predecir con certeza la posición de la pelota, al tocarla aumentaba la probabilidad de rozarla de costado o en un ángulo incorrecrto, con lo cual la misma no salía dirigida limpiamente hacia el centro del arco rival sino desviada. Por estos motivos, y para mejorar la precisión de los comportamientos, surgió la necesidad de predecir la posición de la pelota en un tiempo posterior al actual.

Luego de algunos experimentos, decidimos implementar el predictor para que modele la posición de la pelota en $t = t_0 + 2\ s / cm * \Delta$, siendo t_0 el momento actual (en segundos) y Δ la distancia actual entre el robot y la pelota (en centímetros). Está claro que cuanto más lejos estemos, mayor será el tiempo necesario para poder alcanzar la pelota, con lo cual es lógico que el tiempo crezca junto con la distancia. Pero el hecho de que este crecimiento sea lineal y con un coeficiente igual a 2 fue un criterio obtenido experimentalmente, que sin embargo mejoró la efectividad de la acción.

Una vez definido el tiempo, basándonos en que:

$$Pos.\ x_t = Pos.\ x_0 + vel.\ x_0 * t \qquad (5.3)$$
$$Pos.\ y_t = Pos.\ y_0 + vel.\ y_0 * t \qquad (5.4)$$

calculamos las posiciones de *x* e *y* en el tiempo deseado, ya que conocemos la posición y velocidad actuales de la pelota, al ser éstas últimas elementos de la situación leída por el agente en cada paso temporal. Este procedimiento se repite en cada paso temporal, con nuevos valores leídos de posición y velocidad y obteniéndose un nuevo tiempo *t* cada vez.

Si esta posición excede los límites de la cancha, consideramos sucesivos rebotes en las paredes de ésta, donde asumimos que no se modifica la velocidad y que la dirección es simétrica respecto de la coordenada perpendicular a la pared. Por ejemplo, en la Figura 5.1 vemos como la pelota inicialmente está en *a* con la dirección indicada por la flecha. En *b*, al toparse con una pared, la predicción espeja el ángulo sobre el eje *y* (en líneas punteadas), para finalmente predecir la posición *c* en el tiempo deseado.

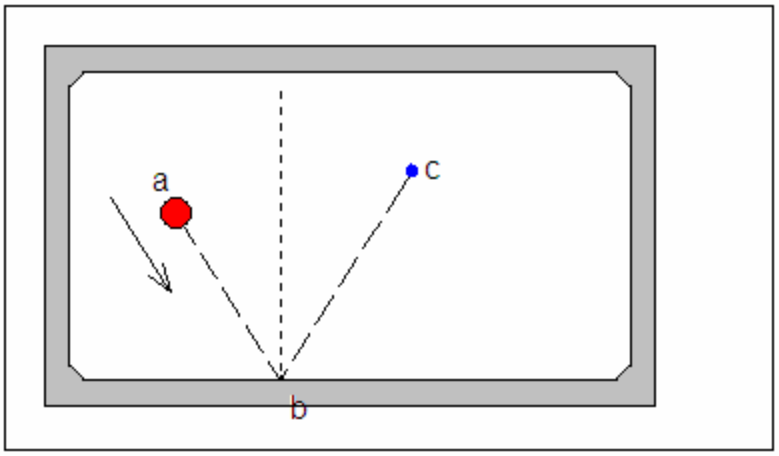

Figura 5.2: Predicción de la posición de la pelota.

Para medir la mejora en la efectividad del comportamiento "Patear al Arco" (ver sección 5.2.6.1), comparamos una serie de experimentos, donde cada uno consistió en inicializar la pelota y el jugador al azar en cualquier lugar de la cancha, efectuar el comportamiento y esperar que la pelota toque la línea de fondo rival, computándose gol o desviado según corresponda, luego de lo cual finalizaba el episodio. Sin predicción de la trayectoria, la efectividad fue de un 55% (porcentaje de episodios terminados en gol) y con predicción, 71%, sobre un total de 50000 intentos.

5.2.5 Dimensiones del espacio de estados

Desde el punto de vista de un agente aprendiendo en el dominio de fútbol de robots (inicialmente con un jugador por equipo), un estado podría representar información acerca de sí mismo, del rival y de la pelota. Respecto a los agentes, importa fundamentalmente la posición dentro de la cancha y su ángulo u orientación.

Dos situaciones donde la posición de los agentes es la misma, al igual que la posición y velocidad de la pelota, pero donde la orientación de uno de los agentes es distinta va a significar probablemente que la trayectoria de este agente

para alcanzar la pelota (y por ende el tiempo destinado a realizar esta tarea) sea distinto, con lo cual nos conviene considerarlos como situaciones distintas.

Por ejemplo, en una situación donde el agente está cerca del arco y la pelota delante del mismo, apuntar al arco o a la esquina va a generar trayectorias claramente distintas (Figura 5.3), dependiendo de la orientación del robot.

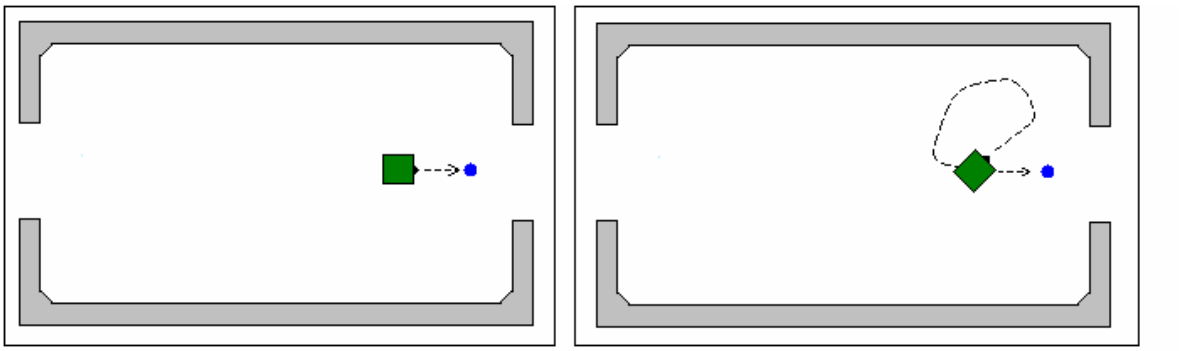

Figura 5.3: En la situación de la izquierda, el agente apunta hacia el arco y genera una trayectoria directa hacia la pelota; en la situación de la derecha, el agente apunta hacia el corner, con lo que para posicionarse para patear al arco, debe realizar una vuelta que le lleva mucho más tiempo.

Respecto a la pelota, importa su posición pero no su orientación (de hecho no la tiene, no confundir hacia dónde apunta con hacia dónde se dirige). Otro factor que se tuvo en cuenta fue la velocidad en ambas coordenadas, tanto para la pelota como para los agentes. Aquí se decidió considerar la velocidad de la pelota pero no la de los agentes, por razones de complejidad (para la pelota necesitamos agregar solo dos dimensiones mientras que para los agentes debemos agregar 2 * # agentes) y además porque luego resultó necesario saber hacia dónde se dirige la pelota para poder predecir su posición, algo factible siempre y cuando contemos con información de su velocidad.

Con todo esto, un estado pasa a estar representado por las siguientes 4 + 3 * # jugadores = 10 dimensiones:

Dimensión	Rango (middle league)	Unidad de medida
Posición *x* propia	0 ... 220	cm
Posición *y* propia	0 ... 180	cm
Ángulo propio	0 ... 2 Π	radianes
Posición *x* del rival	0 ... 220	cm
Posición *y* del rival	0 ... 180	cm
Ángulo del rival	0 ... 2 Π	radianes
Posición *x* de la pelota	0 ... 220	cm
Posición *y* de la pelota	0 ... 180	cm

Velocidad *x* de la pelota	-200 ... 200	cm/s
Velocidad *y* de la pelota	-200 ... 200	cm/s

5.2.6 Espacio de acciones

Con la idea de mantener bastante simplificado el proceso de aprendizaje, para el caso de un solo jugador por equipo, delineamos únicamente dos comportamientos básicos. Uno (patear al arco), relacionado con el objetivo de hacer goles, y el otro (atajar) para evitar que se los hagan.

5.2.6.1 Patear al arco

Patear al arco utiliza un predictor para aproximar la posición de la pelota en un tiempo futuro (ver sección 5.2.4) y luego ejecuta el método de navegación para poder alcanzar dicha posición con la dirección necesaria para patear la pelota al medio del arco contrario, tratando de alcanzar en el momento del impacto una velocidad alta (176 cm/s), de forma tal que la pelota salga a la mayor velocidad posible.

En la figura 5.4 se puede apreciar el posicionamiento del agente para ejecutar la acción. En el tiempo t_0, la pelota y el agente están en cualquier posición de la cancha. Se predice que en tiempo t_1 la pelota va a estar en determinada coordenada (PredicX, PredicY). El agente se dirige hacia allí, de forma tal de poner la pelota entre él y el arco, y eligiendo un ángulo que haga empujar la pelota hacia el centro del arco rival (*x* = Largo de Campo, *y* = Ancho de campo / 2), de la siguiente manera:

a = Ancho de campo / 2 – PredicY
b = Largo de campo – PredicX
α = atan (a / b)

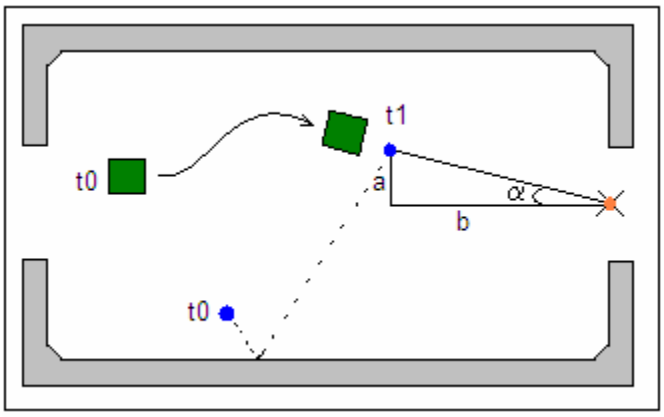

Figura 5.4: Posicionamiento del agente para patear al arco.

5.2.6.2 Atajar

El agente puede estar en dos estados: Posicionándose y Defendiendo. Si está en el primero, usa las funciones de navegación de Fiorini hasta colocarse delante del arco propio (a 5 cm) y con una orientación paralela al arco ($\pi/2$, o equivalentemente $3/2*\pi$). Al llegar a esa zona, pasa al estado Defendiendo y a partir de ahí se moverá en una línea paralela al arco, en un área restringida a la longitud del mismo, intentando interceptar la pelota cuando ésta pase por la línea del arco.

Las funciones de navegación de Fiorini que usamos tienen el problema de que al intentar ir hacia atrás no invierten la velocidad de las ruedas sino que ejecutan una vuelta, yendo siempre hacia adelante. Por lo tanto, en el estado Defendiendo no utiliza estas funciones sino que asigna valores directamente a las velocidades de las ruedas, permitiendo que éstas se desplacen hacia adelante o atrás.

A partir de ahí, se realiza un control proporcional de la posición, es decir que la señal de control (la velocidad de las ruedas) será proporcional al error entre la posición actual y la posición en la coordenada *y* donde se predice que pasará la pelota (PredicY). Como corremos el riesgo de que cuando el error disminuya, la señal de control también lo haga en gran medida y el agente nunca llegue a su objetivo, introducimos una componente integrativa, que acumula el error de los últimos pasos temporales (5 en nuestro caso), con lo que finalmente la velocidad se define como:

$$Vel = k_p\,(PredicY - Pos.\ y\ actual\ del\ agente) + k_i * Integral\ del\ error \qquad (5.5)$$

donde k_p es la constante proporcional y k_i es la constante integrativa.

5.2.7 Estrategia hardcodeada

Inicialmente, para entrenar al agente fue necesario simular un entorno similar a la realidad, es decir con un oponente que trate de ganar los partidos. Para esto, se pensó en una política fija (hardcodeada) que logre la mayor efectividad posible (según fue definida en la sección 5.2.1), que luego le fue asignada al equipo-agente rival.

Como base para el equipo adversario de un solo jugador se tomó la siguiente estrategia: si la pelota se encuentra en la mitad propia se ejecuta el comportamiento atajar; por el contrario, si la pelota se encuentra en la mitad opuesta, se ejecuta el comportamiento patear al arco. Utiliza los dos comportamientos básicos, poniendo como punto de corte el centro del campo, que puede parecer arbitrario, pero intuitivamente (y basándose en cómo se comporta un jugador de fútbol en la realidad) se pensó que es mejor patear al arco al estar la pelota cerca del arco rival y atajar cuando hay peligro de gol (al

estar la pelota cerca del arco propio). A pesar de su simpleza, esta estrategia demostró ser eficiente. De hecho, enfrentada con otras estrategias hardcodeadas, algunas más complejas, resultó siempre vencedora.

Además, se realizaron episodios de aprendizaje donde el rival no tenía una estrategia definida, sino que en cada paso temporal se tomaba una acción al azar. Con esto, aspiramos a comparar la efectividad de nuestro agente en ambas experiencias, esperando que la efectividad alcanzada al enfrentarse con una política al azar sea mayor a la obtenida contra la estrategia hardcodeada.

5.3 Discretización

Como se explicó en el capítulo anterior, necesitamos pasar de un espacio de estados-acción continuo, que es como se presenta el dominio del problema, a uno discreto, que es una condición necesaria para utilizar AR. Los métodos de AR vistos hasta ahora no escalan apropiadamente para tratar espacios muy grandes, posiblemente infinitos o continuos. Existe una gran cantidad de métodos de discretización ampliamente estudiados, cada uno con sus características particulares en cuanto a la eficiencia de la cobertura del espacio de estados-acciones, las estructuras de datos que usan o la complejidad de sus cálculos.

Entre otras características de la discretización, debemos cuidar por un lado que ésta no sea demasiado fina, ya que el problema será tanto la memoria necesaria para almacenar tablas muy grandes como el tiempo y los datos necesarios a recolectar por parte del agente para utilizarla correctamente [22], y por el otro lado no debe ser demasiado gruesa, para evitar un problema adicional llamado cluster aliasing, que explicaremos en la siguiente sección.

Para discretizar nuestro espacio de estados-acciones se evaluaron 3 alternativas: coarse-coding, Q-Kohon y RBF, que serán explicadas en sus correspondientes secciones en este capítulo. En [32] se afirma que no hay resultados teóricos firmes que sugieran que un método sea mejor que otro, por lo que la decisión para elegir uno de esos métodos se basó en los resultados que obtuvimos en nuestra experiencia.

5.3.1 El problema del aliasing

Un problema inherente a cualquier generalización es el del aliasing. Perceptual aliasing se da cuando el agente percibe (o representa) diferentes estados del mundo real (o simulado) en uno solo, lo cual influye negativamente en la capacidad del sistema de aprender una política de control adecuada, ya que confunde estados del mundo real (o simulado) que necesariamente debe distinguir para resolver la tarea [33].

Un problema parecido es el de cluster aliasing, que ocurre cuando se toma la acción incorrecta de un estado porque éste está dentro de un cluster junto a un estado que es el que realmente determina la acción. Se han realizado diferentes trabajos donde se intentan reducir los efectos negativos de este problema. En [34] se propone una métrica para medir el grado de cluster aliasing y un algoritmo para determinar la cantidad apropiada de clusters. En [35] se afirma que cuanto más grande el cluster, más alta es la probabilidad de obtener diferentes refuerzos para los clasificadores pertenecientes a la subpoblación correspondiente y menor es la aptitud del sistema para aprender la acción correcta, lo cual es una motivación para que el cluster sea lo más pequeño posible. Teniendo esto en cuenta, en la elección de nuestro método de discretización intentaremos reducir este aliasing en la medida de lo posible.

5.3.2 Coarse-coding

En [22] se introduce el concepto de generalización, tratando de responder cómo la experiencia con un conjunto limitado puede ser generalizada hasta producir una buena aproximación en un subconjunto mucho mayor, con estados que probablemente nunca se hayan visitado.

Para lograr eficiencia es importante elegir las características (features) adecuadas de acuerdo al problema que se quiere modelar. Coarse-coding significa representar a un estado con un conjunto de estos features, y la generalización de un estado a otro está dada por la cantidad de features que estos dos estados tienen en común. Si la forma de ver a los features es que estrictamente estén o no presentes dentro de un estado, se llaman features binarios. Siempre citando a [22], se afirma que más que el tamaño o la forma de los campos receptivos de los features, lo que realmente determina la correctitud a un nivel de ajuste fino es la cantidad de features usados.

Una de las formas de coarse-coding más simples y que fácilmente podemos adaptar a nuestro escenario es tile-coding. Aquí se realiza una partición del espacio de estados en baldosas (tiles) y cada tile va a coincidir con un feature binario.

Nuestra implementación de tile-coding consistió en partir cada dimensión en intervalos regulares, primeramente de 2 y luego de 4 valores, con lo que la cantidad total de clusters fue (siempre en el caso de un jugador por equipo):

$\#\text{ intervalos}^{\#\text{ dimensiones}} = 2^{10} = 1024$ en el primer caso y $4^{10} = 4096$ en el segundo.

La función para pasar de un estado del mundo real (o simulado) a la clusterización consiste simplemente en determinar, para cada dimensión, a qué intervalo pertenece el valor observado. La combinación de los índices de los intervalos corresponde a un

elemento de una tabla general, donde se tiene un valor Q(*s*, *a*) para cada acción *a* posible, siendo *s* el estado generalizado apuntado por el elemento de la tabla.

5.3.3 Q-Kohon

Una crítica del método anterior es que la simplicidad en la cual se parte el espacio de estados no es muy eficiente. En efecto, puede haber sectores del espacio de estados "marginales", es decir que se visiten muy rara vez a lo largo del aprendizaje y que sin embargo ocupen uno o varios tiles con el método anterior. O bien, un sector muy visitado, donde en un mismo tile haya situaciones que tal vez sería mejor tenerlas en distintos clusters. Para mejorar esta discretización, usamos una estructura basada en un SOM (Self Organizing Map) de Kohonen, para lo cual nos valimos del trabajo de Claude Touzet, Q-Kohon [36].

El SOM es una red neuronal cuya topología es un conjunto de celdas ubicadas en un rectángulo de diversas maneras (en forma de grilla, hexagonal, etc.). Cada celda contiene un vector con información llamado vector represenante, que se va ajustando de acuerdo a los datos de entrada que van llegando en sucesivas iteraciones, luego de las cuales se produce el orden final del SOM [37]. El proceso de aprendizaje es competitivo y no supervisado, o sea que no hay una entidad externa que defina la salida correcta (en este caso la celda a la cual se mapea la entrada) para una entrada. En el algoritmo básico del SOM, las celdas empiezan con sus vectores inicializados de alguna manera (por ejemplo, al azar). Luego se repite lo siguiente hasta que haya convergencia: a un vector de entrada cuya estructura coincide con los de las celdas, se busca cuál es la celda cuyo vector más se asemeja. A ésta se la llama BMU – Best Matching Unit o celda ganadora. El vector representante de la BMU se modifica mediante la siguiente ecuación, de forma tal que se parezca un poco más al vector de entrada y lo mismo (en menor medida al ir alejándose de la BMU) ocurre con las celdas vecinas:

$$m_i\,(t+1) = m_i\,(t) + h_{c(x),\,i}\,(x(t) - m_{(i)}(t)) \qquad (5.6)$$

donde *t* es el tiempo, $h_{c(x),\,i}$ una función decreciente basada en la distancia entre la BMU y la celda con índice *i*, *x* el valor del vector de entrada y *m* el valor del vector de la celda *i*.

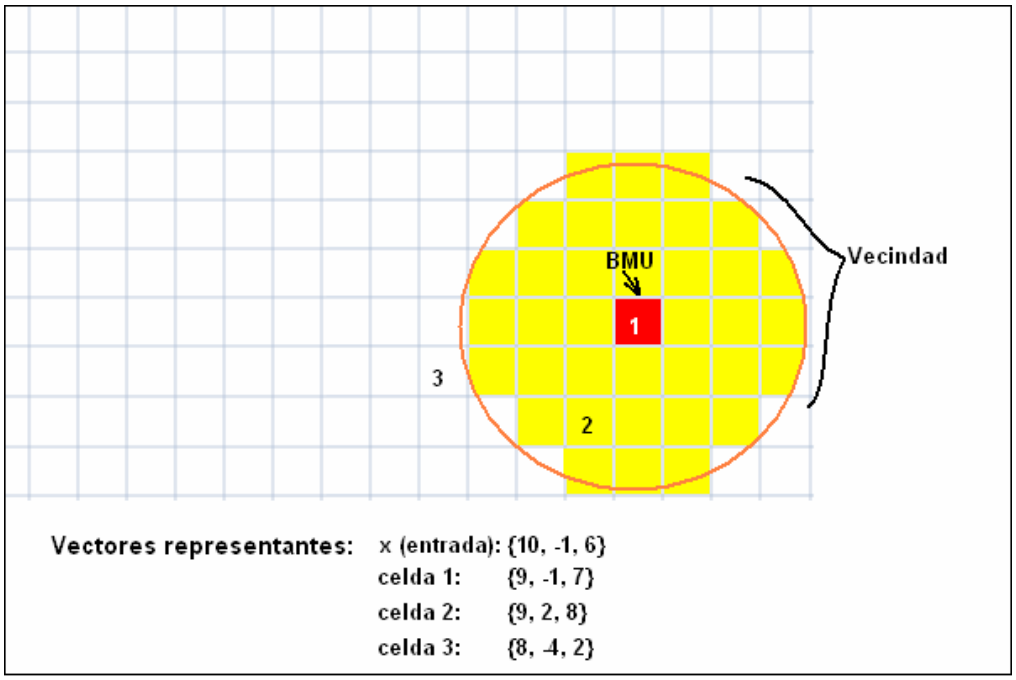

Figura 5.5: Ejemplo de una red de Kohonen, en este caso, con topología de grilla. La celda 1 es la que posee el vector representante con menor distancia al vector de entrada, con lo que pasa a ser la BMU. Las celdas vecinas (en amarillo) también verán modificado su vector representante.

Una clara ventaja de esta estructura respecto a tile-coding es que no es estática sino que se va modificando con el tiempo (dado que los vectores representantes se van moviendo hacia las situaciones observadas), lo cual da una respuesta al problema planteado al inicio de la sección.

En Q-Kohon (una implementación de Q-Learning que utiliza un SOM [38]), los componentes del vector representante de las celdas del SOM son el estado o situación, la acción y el valor Q. En cada iteración, la BMU es la que tiene la distancia mínima con la situación del vector de entrada y un valor de Q de + 1:

$$d(i) = | \ WorldSituation - W_{situation,\,i} \ | \ + \ | \ 1 - W_{Qvalue,\,i} \ | \qquad (5.7)$$

Al buscar la mínima distancia entre el valor Q de la celda y el +1, es decir un valor de Q óptimo, se prioriza la búsqueda de celdas con recompensa alta y que a la vez sean parecidas a la situación actual, con lo cual la acción de esta celda es la que debería ofrecer la mejor recompensa en la situación actual.

Luego se debe actualizar la celda correspondiente a la situación y a la acción efectuada. La distancia usada en este caso es distinta a la anterior, ya que en este caso se usa la distancia entre 1 y la acción en vez del valor de Q (para lo cual, es necesario asignar un valor a las diferentes acciones).

$$d(i) = | \ WorldSituation - W_{situation,\ i} \ | + | \ 1 - W_{Action,\ i} \ | \qquad (5.8)$$

Nuestra implementación de Q-Kohon fue levemente distinta, ya que hicimos que el vector de pesos asociados a cada celda del SOM se corresponda con los componentes del espacio de estados. Cada celda, a su vez, tiene asociada una estructura donde cada acción tiene su correspondiente valor Q. De este modo, cada vez que una acción es ejecutada, es actualizado el valor Q para esa acción, y por otro lado, durante la explotación es utilizada la acción que tiene el mayor valor de Q para esa celda.

La función para obtener la distancia se simplifica a:

$$d(i) = | \ WorldSituation - W_{situation,\ i} \ | \ . \qquad (5.9)$$

En este trabajo, usamos grillas en forma de cuadrícula para la capa de salida de la red de Kohonen y con diferentes cantidades de celdas. Además, se usaron los siguientes coeficientes de actualización del SOM:

CoefEstado: indica por cuánto se multiplica a cada elemento del vector representante de un cluster. La actualización de cada componente del vector representante se realiza en cada paso temporal de la siguiente manera:

SOM[fila][columna].estado.comp$_i$ += coefEstado * (últimaSituación.comp$_i$ - SOM[fila][columna].estado. comp$_i$)

CoefQNeighb: indica por cuánto se multiplica el valor Q, sólo de los vecinos (recordar que el valor Q de la BMU se actualiza mediante el algoritmo de Q-Learning) y, nuevamente, la línea de actualización en cada paso temporal queda:

SOM[fila][columna].AcciónRealizada.qValue += coefQNeighb * (qValue$_{BMU}$ - SOM[fila][columna].AcciónRealizada.qValue)

Para decrecer estos 2 coeficientes a medida que nos alejábamos de la BMU se usó la siguiente fórmula:

$$CoefEstado_i = CoefEstado \ / \ (2 * distancia(_{BMU,\ i})) \qquad (5.10)$$

$$CoefQNeighb_i = CoefQNeighb \ / \ (2 * distancia(_{BMU,\ i})) \qquad (5.11)$$

En la primera mitad del aprendizaje (fase de ordenamiento) el ancho de la vecindad comienza siendo igual al ancho del SOM y va decreciendo linealmente hasta llegar a 0 al final de la fase. Los coeficientes son inicializados en el episodio 0 y decrecen linealmente hasta llegar a 0 al final de la segunda fase.

```
chooseAction(Situation currentSituation)
   explore ← Bernoulli(epsilon);

   found ← false;
   if explore
       // se elige un comportamiento al azar
      lastAction.behaviorId ← Equilikely(0, BEHAVIORS - 1);
   else
      find chosenCol, chosenRow:
            the column and row with the shortest distance to currentSituation

      find bestAction, the action corresponding to
          the element with highest Q-Value in the action list of
          the BMU (the cell referenced by chosenRow, chosenCol)

      lastAction.behaviorId ← bestAction

   lastSituation ← currentSituation;

-----
update(Situation newSituation)

   r ← getImmediateReward

   find newChosenCol, newChosenRow:
      the column and row with the shortest distance to newSituation

   SOM[chosenRow][chosenCol].qValue[bestAction] +=
        stepSize * (r + discount * SOM[newChosenRow][newChosenCol].qValue[bestAction] -
        SOM[chosenRow][chosenCol].qValue[bestAction]

        // Actualizo el vector representante de la BMU
        updateCell(chosenRow, chosenCol, coefficState, coefficQNeighb,
                                lastSituation, bestAction, 0, true)
        // Actualizar vecindad
        for each row i in SOM
           for each column j in SOM
                // hallo distancia euclideana
                distanceToBMU ← sqrt(chosenCol - j) ^ 2 + (chosenRow - i) ^ 2);
                if (distanceToBMU < neighborhoodWidth and
                   (i != chosenRow or j != chosenCol))

                    updateCell(i, j, coefficState / (distanceToBMU * 2),
                       coefficQNeighb / (distanceToBMU * 2), lastSituation, bestAction,
                       SOM[chosenRow][chosenCol].qValue[bestAction], false);

-----
updateCell(int vertIndex, int horIndex, float neighbCoeffState, float neighbCoeffQ,
               Situation lastSituation, Action action, float qValueBMU, bool isBMU)

        // Actualizo cada dimensión del estado según el coeficiente
        SOM[vertIndex][horIndex].state.x += neighbCoeffState *
           (lastSituation.x - SOM[vertIndex][horIndex].state.x)
        SOM[vertIndex][horIndex].state.y += neighbCoeffState *
           (lastSituation.y - SOM[vertIndex][horIndex].state.y)
        SOM[vertIndex][horIndex].state.angle += neighbCoeffState *
           (lastSituation.angle - SOM[vertIndex][horIndex].state.angle)
        SOM[vertIndex][horIndex].state.opponentX += neighbCoeffState *
           (lastSituation.opponentX  - SOM[vertIndex][horIndex].state.opponentX)
        SOM[vertIndex][horIndex].state.opponentY += neighbCoeffState *
           (lastSituation.opponentY  - SOM[vertIndex][horIndex].state.opponentY)
        SOM[vertIndex][horIndex].state.opponentAngle += neighbCoeffState *
           (lastSituation.opponentAngle - SOM[vertIndex][horIndex].state.opponentAngle)
        SOM[vertIndex][horIndex].state.ballX += neighbCoeffState *
           (lastSituation.ballX - SOM[vertIndex][horIndex].state.ballX)
        SOM[vertIndex][horIndex].state.ballY += neighbCoeffState *
           (lastSituation.ballY - SOM[vertIndex][horIndex].state.ballY)
```

```
SOM[vertIndex][horIndex].state.ballLinearVelX += neighbCoeffState *
   (lastSituation.ballLinearVelX - SOM[vertIndex][horIndex].state.ballLinearVelX)
SOM[vertIndex][horIndex].state.ballLinearVelY += neighbCoeffState *
   (lastSituation.ballLinearVelY - SOM[vertIndex][horIndex].state.ballLinearVelY)
if (!isBMU)
        SOM[vertIndex][horIndex][bestAction].qValue += neighbCoeffQ *
            (qValueBMU - SOM[vertIndex][horIndex].qValue[bestAction])
```

Figura 5.6: Pseudocódigo de la implementación de Q-Kohon realizada en este trabajo. chooseAction() es invocado antes de ejecutar la llamada a ODE que modifica el entorno (el paso temporal) y update() es llamada con posterioridad a ese momento, donde newSituation es el nuevo estado leído.

5.3.4 RBF

El último método de discretización que contemplamos es Radial Basis Functions (RBF). Se puede ver como una generalización de coarse-coding a features con valores continuos, ya que un feature no está ausente o presente de un estado en forma absoluta sino que lo está en algún grado en el intervalo [0, 1] (determinado por la distancia euclidiana de un punto al centro de un cluster).

Este método es bastante usado en aproximación de funciones y la suavidad con la que logra estas aproximaciones lo ubican en ventaja frente a las discretizaciones con features binarios. La principal desventaja que presenta es una mayor complejidad computacional. Una ventaja sobre Q-Kohon es que este método permite crear clusters en forma dinámica, con lo cual la cantidad de éstos se va adaptando a las necesidades del problema. La cantidad de clusters se controla automáticamente y no crece en forma indiscriminada, ya que con el correr de la tarea de aprendizaje, y al haberse cubierto en forma apropiada el espacio de estados-acción, no se crean nuevos clusters.

Nuestra implementación estuvo basada en Q-RBF, un algoritmo propuesto por J. Santos [38], en el que se aplica un método dinámico de creación de clusters. Durante el aprendizaje, se mide la distancia entre el estado actual y los centros de los clusters existentes. Si la distancia al cluster más cercano excede cierto umbral, se crea un nuevo cluster con centro en la posición actual. Si no, corresponde usar el cluster existente cuyo centro tenga la distancia mínima con el estado actual, tanto para actualizar su valor Q como para ejecutar la acción con mayor valor Q (en el caso de que se esté explotando). Al igual que en el caso de Q-Kohon, se simplificó el algoritmo eliminando la acción y el valor Q de los componentes que determinan un cluster, quedando en el mismo solamente las variables que determinan un estado.

Los centros de los clusters pueden mantenerse fijos o variables (haciendo que se muevan hacia la situación actual de forma similar a Q-Kohon). En nuestro trabajo probamos ambas alternativas. En la figura 5.7 se puede ver el pseudocódigo de la implementación usada en este trabajo.

```
chooseAction(Situation currentSituation)

   explore ← Bernoulli(epsilon);

   found ← false;
   if explore
      // se elige un comportamiento al azar
      lastAction.behaviorId = Equilikely(0, BEHAVIORS - 1);
   else
      find chosenCluster, minDistance:
            the cluster with the shortest distance to currentSituation
            and that shortest distance respectively

   if minDistance <= sigma
      found ← true;

      // Leo la acción del cluster ganador
      lastAction.behaviorId = findBestAction(chosenCluster);
   else
      // Si no encontró un cluster a  distancia menor a sigma,
      // ejecuta una acción al azar
      lastAction.behaviorId = Equilikely(0, BEHAVIORS - 1);

   lastSituation ← currentSituation;


update(Situation newSituation)

   r ← getImmediateReward

   // En el caso de que se haya explorado, busco el cluster
   // más próximo a la situación anterior para ver si encuentro uno
   // para actualizar o tengo que crear uno nuevo
   if explore

      find chosenCluster, minDistance:
            the cluster with the shortest distance to lastSituation and
            that shortest distance respectively

      if (minDistance <= sigma)
          found ← true;

   if (!found)

      newRBFElement.state ← lastSituation;

      for each behavior i
         newRBFElement.qValue[i] = 0.0;

      RBFs.add(newRBFElement)
      // hago que el cluster a actualizar sea el que acabo de crear
      chosenCluster ← RBFs.size() - 1;

   // Hallo celda ganadora, en base a newSituation (para obtener el maxQ)
   find newChosenCluster, newMinDistance:
        the cluster with the shortest distance to newSituation and that
        shortest distance respectively
```

```
maxQAction ← findBestAction(newMinRBF);
RBFs[chosenCluster].qValue[lastAction.behaviorId] +=
      stepSize * (r + discount * RBFs[newChosenCluster].qValue[maxQAction] -
      RBFs[chosenCluster].qValue[lastAction.behaviorId])

// en esta implementación, permitimos que los clusters
// se muevan. Movemos las coordenadas de estado del cluster
updateRBF(chosenCluster, stateCoeff, lastSituation);
```

Figura 5.7: Partes principales de la adaptación del algoritmo Q-RBF. Se usan los mismos métodos chooseAction y update y se invocan en los mismos momentos que en Q-Kohon.

5.3.4.1 Elección del sigma

Debemos hallar un balance entre la cantidad de clusters creados durante el aprendizaje y el cluster aliasing derivado de ésta. Si aumentamos la cantidad de clusters haremos decrecer el cluster aliasing asociado pero esto hará necesaria una mayor cantidad de episodios para lograr convergencia, ya que se necesita invertir más pasos en exploración. Por otro lado, si consideramos una cantidad menor de clusters, será necesaria una menor cantidad de episodios para converger, pero el cluster aliasing producido se incrementará.

Estos tres elementos asociados (cantidad de clusters, cantidad de episodios, cluster aliasing) son los que influyen al seleccionar el umbral o sigma para controlar la creación de clusters.

Siempre podemos seleccionar el menor sigma posible, digamos σ_1, tal que toda situación visitada se asocie a un solo cluster. A partir de éste se puede realizar una búsqueda binaria en el intervalo $(0, \sigma_1]$ para encontrar valores donde se observe una correcta convergencia en la creación de clusters.

En la práctica, a medida que disminuimos σ_1 en escenarios con 2000 episodios, no se observa una convergencia total hacia una cantidad de clusters, si bien se evidencian sectores donde la curva decrece considerablemente.

Por este motivo, definimos la siguiente métrica para buscar el episodio que satisface nuestro criterio de convergencia: Para un coeficiente k, tomamos el mínimo episodio e tal que la cantidad de clusters creados entre 0 y e sea k veces la cantidad creada entre e y $2e$. Por ejemplo, si $k = 6$, si en 250 episodios se crearon 1200 clusters, si en los siguientes 250 episodios se crearon 200, y además no hubo un episodio anterior a 250 donde se observara esa relación, entonces 250 va a ser el episodio que consideramos como el momento donde se observa la convergencia, y que en la sección siguiente usaremos para dividir las fases de un experimento. Intuitivamente, esto quiere decir que el episodio de corte será el

que deje creados *k* veces la cantidad de clusters que creará luego en una cantidad de episodios igual.

Además, debemos asegurarnos una cantidad mínima de clusters, ya que si por ejemplo elegimos $k = 6$, y además en 2 episodios se crean 12 clusters y en los siguientes 2 se crean otros 2, no nos sirve mucho, aun si se satisface el criterio explicado arriba. En conclusión, el sigma utilizado para los experimentos se halla mediante la búsqueda binaria hasta satisfacer el criterio de convergencia explicado arriba.

5.3.4.2 Fases del experimento

Cuando utilizamos RBFs como método de discretización se pueden notar dos fases diferentes durante el aprendizaje que van a influir en todo el experimento. La primera fase se caracteriza por una creación de clusters acelerada, esperándose que la mayor parte sea creada en ella. Luego se espera que esta relación disminuya y se desacelere, creándose cada vez menos clusters. Nombraremos a la primera fase como fase de exploración y a la segunda fase como fase de convergencia.

Teniendo en cuenta esta división se advierte que cada fase tiene necesidades claras y diferentes. Para cada elemento de nuestra solución a configurar analizaremos cuáles son estas necesidades y evaluaremos diferentes hipótesis de trabajo.

Episodios de cada fase:

Como según nuestro criterio de convergencia el episodio de corte (el que separa ambas etapas) es la mitad del total, haremos que las etapas tengan la misma cantidad de episodios.

Alfa o step size:

Como dijimos anteriormente, durante la fase de exploración se creará la mayor cantidad de clusters. Si utilizamos un step size variable y decreciente durante esta fase se dificultará la convergencia de los clusters creados hacia el final de la fase, con lo que haremos que éste sea fijo en esta fase.
En la fase de convergencia se crearán muy pocos clusters y, como su nombre lo indica, lo preponderante es conseguir una correcta convergencia. Por otro lado, nuestro problema tiene en cuenta un oponente estacionario (no cambia su estrategia a través del tiempo). Esto quiere decir que no necesitamos un step size constante mayor que 0 para adaptarnos a posibles cambios. Además, el problema de cluster aliasing sumado a un alfa constante puede entorpecer el aprendizaje haciendo que los valores Q presenten mucha varianza. Por este motivo, utilizaremos un step size variable y decreciente hasta 0 durante la fase de convergencia.

Factor de exploración:
Si se eligió correctamente el sigma de modo que las fases estén bien delimitadas, se crearán pocos clusters en la etapa de convergencia. Sin embargo, no queremos que se dé cierta anomalía.
Veamos qué ocurre al agregarse un cluster con la implementación de RBF que usamos: se elige una acción al azar para ser efectuada en el paso actual (dado que se carece de información sobre ese estado), se actualiza el valor Q correspondiente a dicha acción y todos los demás valores del vector (dimensiones de la situación) quedan en 0. Si no hay exploración, a menos que el valor Q de la acción elegida al azar sea negativo, esa acción será siempre elegida en las sucesivas visitas al cluster, con lo que otra acción posiblemente mejor nunca será evaluada. Por este motivo no queremos que haya una parte del aprendizaje donde convivan la creación de clusters y la falta total de exploración, por lo que dejamos decrecer el factor de exploración a lo largo de todo el aprendizaje hasta llegar a 0 y no solamente en la fase de exploración.

Coeficiente de actualización de clusters:
Los valores que representan el estado de cada cluster se actualizan cada vez que éste es visitado, de acuerdo a un coeficiente. Una posibilidad es mantenerlo siempre nulo, con lo que los clusters se mantendrían fijos. Esto solucionaría un posible riesgo, que es que un cluster dado se desplace a través del tiempo y pierda la referencia al estado que representaba originalmente, pero a la vez, una zona del espacio de situaciones más visitada a través de los episodios estaría representada por una mayor densidad de clusters, lo que ofrece una mayor granularidad. También es posible que este coeficiente tenga un valor fijo no nulo a través de todo el aprendizaje o bien que decrezca desde el principio hasta el fin. Pero, por las mismas razones que en el caso del step size ninguna de estas dos alternativas parece la más viable, por lo que utilizaremos un coeficiente constante en la fase de exploración y uno variable y decreciente en la fase de convergencia.

5.4 Escenario con varios robots por equipo

Hasta ahora hemos analizado únicamente el caso de un agente por cada equipo. Si bien por su simpleza nos sirvió para probar los diferentes algoritmos y técnicas de discretización, éste no es el escenario de la realidad y en particular de las competencias de la FIRA, donde hay 5 o más robots por equipo.

Al escalar el problema de esta manera surgen varias dificultades. Una de ellas tiene que ver con el aumento de la complejidad de cada estado y el crecimiento del cardinal del espacio de estados-acción. En el primer caso ocurre porque ahora cada agente mantiene en cada estado los datos de todos sus compañeros y todos sus rivales, además de los de sí mismo, aunque este crecimiento es lineal. En el

segundo caso, como se agregan elementos, el espacio de estados-acción crece en forma exponencial.

Para ejemplificar, veamos qué ocurre con tile-coding (si bien ya descartamos su uso). Supongamos un espacio de situaciones unidimensional *S: [-1,1]*. Utilizando tile coding con intervalos regulares podemos discretizar *S* dividiéndolo en k tiles. Esta división está asociada con un tamaño de cada tile, que será el mismo para todos.
Si extendemos *S* a *n* dimensiones con dominios similares *(S': [-1,1]^n)*, podemos pensar una discretización tile coding en donde cada dimensión esté dividida en la misma cantidad de tiles y, a su vez, cada tile mantiene el mismo tamaño. En este caso, la cantidad de tiles necesaria para discretizar *S'* aumenta exponencialmente en relación directa a la cantidad de dimensiones del espacio. Específicamente,

tiles = k^n.

Para dar un ejemplo relacionado con nuestro problema, podríamos decir que si en un espacio de 10 dimensiones tenemos 4000 tiles y deseamos extender este espacio a uno de 16 dimensiones (resultante de agregar un jugador para cada equipo, cada uno con 3 dimensiones), tendremos

tiles = $(4000^{1/10})^{16} \approx 580000$.

No debemos perder de vista que el caso mencionado no es el nuestro debido a varios factores. Con la discretización elegida (RBF) aspiramos a lograr el nivel de aprendizaje deseado con una cantidad manejable de clusters, es decir que el crecimiento de su cantidad será considerablemente menor que un crecimiento exponencial.

Por otro lado, la estructura de nuestro espacio de situaciones posee determinadas características que hacen que ciertas dimensiones se encuentren relacionadas y no sean totalmente independientes como en el caso general.

Un caso de esto se evidencia cuando hay una relación "semántica" entre dos o más dimensiones. Por ejemplo, dos robots no pueden ocupar el mismo espacio físico. Otro caso son las relaciones específicas que se derivan de la dinámica del problema que se encara. Éste hará que la distribución de las probabilidades de encontrar una situación determinada no sea uniforme y que pueda haber porciones del espacio con muy baja o nula probabilidad de ser visitados.

En cuanto a la implementación, definir la estrategia hardcodeada del rival se vuelve una tarea más compleja, ya que hay que contemplar las diferentes

combinaciones de los demás jugadores en el espacio de situaciones (mediante un árbol de decisiones mucho más complejo que para el caso de un jugador por equipo), lo que hace que para 2 jugadores haya que pensar una estrategia totalmente distinta que para 3 y así sucesivamente. Además, cuanto más jugadores haya, es más factible descuidar situaciones y más difícil dar con las acciones correctas para cada jugador.

Otro problema se da porque ya no alcanza con el algoritmo de Q-Learning tal como lo veníamos implementando, donde un agente pudo haber aprendido una determinada acción en base a una situación en la cual están involucrados los dos agentes del mismo equipo. En un caso, el compañero pudo efectuar una acción y en otro caso otra, y el agente no tiene forma de diferenciar ambos casos. Esta falta de coordinación nos lleva a orientar el problema hacia métodos pensados especialmente para considerar el aprendizaje en un entorno multiagente.

Al agregarse agentes, resultan insuficientes las 2 acciones detalladas en el Capítulo 4. Sólo patear al arco o atajar como opciones provocaría que se solapen las posiciones de los agentes y dejar zonas del campo de juego (como el campo propio exceptuando el arco) sin cubrir. Hay que pensar en acciones que participen de la jugada aunque no directamente sobre la pelota y ocuparse de la parte defensiva. Para eso, se agregaron 3 acciones nuevas:

Quedarse quieto: Es una acción trivial que consiste en detenerse donde se esté, apuntando al campo rival (ángulo = 0). La idea es no realizar acciones innecesarias mientras la acción se desarrolla lejos.

Interceptar: Es la acción más parecida a lo que haría un defensor. Se posiciona apuntando al campo rival (ángulo = 0) y se detiene, a una distancia de 0.2 m de la pelota en el eje *x* hacia el arco propio, y en la misma coordenada *y*. Con esto se busca entorpecer el avance del atacante rival, o bien rechazar la pelota hacia adelante, haciendo que rebote (si no viene acompañada de un rival).

Acompañar: Consiste en acompañar al jugador que patea al arco. En realidad, la acción no debe tener información sobre el entorno, ya que no hay comunicación entre agentes (y de hecho no la tiene), pero está pensada asumiendo que hay un compañero que está pateando al arco. Para esto, ubica al agente en la misma posición *x* que la pelota, y a una distancia en *y* igual a la mitad del ancho del campo, con la idea de tomar un rebote si el compañero falla en la definición. La velocidad que se toma es la misma que la de la pelota. Puede ocurrir que la pelota se mueva a lo ancho del campo y que no quede espacio suficiente para mantener la distancia. En ese caso, el agente se cruza con la pelota e intenta mantener la distancia por el otro lado.

Hubo que delinear una estrategia hardcodeada para el caso de 2 jugadores por equipo y una adicional para el de 5 jugadores por equipo, siempre con nivel creciente de dificultad. Por la complejidad de este último escenario, donde tenemos que:
- el tamaño del estado crece de 3 * 4 + 4 = 16 dimensiones (en 2 vs. 2) a 3 * 10 + 4 = 32 (en 5 vs. 5) y, dado que en cada episodio hay que analizar todas las dimensiones (al leer la posición y al calcular la distancia), cada episodio va a durar mucho más.
- dado que va a aumentar la cantidad de clusters, también hay que aumentar la cantidad de episodios del aprendizaje para obtener convergencia en una cantidad de episodios manejable.
Esto hace que se vuelva inviable el análisis con 5 jugadores por equipo. En el Capítulo 6 se explicarán en detalle las simulaciones que fueron corridas, pero para tener una idea, lo que en 1 vs. 1 jugadores nos llevaba 2 horas, en 5 vs. 5 nos llevaría 20 días, con lo cual debimos usar como escenario multiagente más complejo el caso de 3 jugadores por equipo.

Para el caso de 2 jugadores por equipo, el árbol de decisiones tomó en consideración la posición *x* propia y de la pelota y la proximidad del agente a la pelota. Se prioriza el jugador más cercano a la pelota para atacar o atajar, si se está en el campo rival o propio respectivamente. Quedarse quieto si se está lejos de la pelota (en distintas mitades del campo) tiene por objeto mantener la posición sobre el campo de juego para estar listo si la acción pasa a ejecutarse en el otro campo y no abalanzarse sobre la pelota.

El pseudocódigo es el siguiente:

```
Si pelota en campo rival
    Si estoy en campo rival
        Si soy el más cercano a la pelota
            Patear al arco
        Si no
            Acompañar
    Si no
        Quedarse quieto
Si no
    Si estoy en campo rival
        Quedarse quieto
    Si no
        Si soy el más cercano a la línea de fondo
            Atajar
        Si no
            Interceptar
```

Figura 5.8: Pseudocódigo de la estrategia hardcodeada para 2 jugadores.

Para 3 jugadores por equipo vemos como, si bien la estrategia elegida no es mucho más compleja, ésta puede parecer caprichosa y posiblemente lejos de la óptima, aunque sigue guías similares al caso de 2 jugadores. La cantidad de jugadores justifica que haya agentes especializados en un comportamiento. En este caso, el jugador número 1 (de cierto orden de una lista de los mismos) va a atajar siempre:

```
Si soy 1er jugador
    Atajar
Si no
    Si pelota en campo rival
        Si soy el más cercano a la pelota
            Patear al arco
        Si no
            Si estoy en campo rival
                Acompañar
            Si no
                Quedarse quieto
    Si no
        Si soy el más cercano a la pelota
            Interceptar
        Si no
            Si estoy en campo rival
                Quedarse quieto
            Si no
                Acompañar
```

Figura 5.9: Pseudocódigo de la estrategia hardcodeada para 3 jugadores.

5.5 Multi Agent Reinforcement Learning

Existen diversas maneras de encarar el aprendizaje por refuerzo para varios agentes, donde se abren varias alternativas para los diferentes aspectos.

En [39] se menciona el aprendizaje social usado para adquirir reglas sociales. Parte de evidencias de la biología, donde la observación de congéneres es usada como fuente de aprendizaje. Se basa en que todos los agentes tienen el mismo objetivo y se derivan tres formas de refuerzo involucradas en el aprendizaje social: percepción individual del progreso relativa al objetivo actual (maximiza el beneficio individual), observación del comportamiento de los congéneres (donde un comportamiento similar se asocia con refuerzo positivo y fomenta la exploración de comportamientos que no dan un beneficio inmediato) y refuerzo indirecto (vicarious), donde un premio o castigo de un congénere repercute de la misma manera en un agente, dado que le pasaría lo mismo a él en una situación similar. La misma autora propone en [40] alternativas para ayudar al aprendizaje en un entorno ruidoso y dinámico como el que hay en un escenario multiagente, reformulando el espacio de aprendizaje y reduciéndolo de estado-acción a condiciones-comportamientos.

En [41] se propone un método que se basa en la experiencia del aprendizaje individual para luego usarlo eficientemente en un entorno multiagente. Se muestra el algoritmo MAPLE, que combina merging dinámico y Q-Learning, donde la cota superior se inicializa como la suma de las funciones acción-valor de los MDPs de un agente, y la inferior como el máximo de éstas; se van ajustando la cota superior y la inferior de la función acción-valor del conjunto de agentes; luego, para cada estado se van eliminando las acciones cuya cota superior sea inferior a la cota inferior de alguna otra, ya que no pueden ser óptimas, repitiéndose el proceso hasta que haya convergencia.

Por su parte, Ono y Fukumoto exponen en [42] el problema de cómo crece la complejidad del espacio de estados (exponencialmente) en un entorno multiagente. Se enfocan en el Pursuit Problem, donde 4 cazadores intentan cazar a una presa y proponen una arquitectura que consiste en 3 módulos de aprendizaje y un módulo mediador. Cada módulo de aprendizaje realiza Q-Learning pero guarda sólo la posición de la presa y la de un compañero, con lo cual, si el campo visual es pequeño, el especio de estados es tratable. El módulo mediador combina las políticas de los módulos de aprendizaje usando una heurística que produce la decisión final, basado en la acción que produce el valor Q más alto. Los resultados muestran que hubo dos clases de comportamiento colectivo: ir en grupo y especialización de la funcionalidad (los agentes terminan casi siempre haciendo prevalecer una acción sobre las demás, posiblemente con cada agente con una acción distinta). Dado que los estados donde los agentes van en soledad no producen recompensa, aprenden rápido a ir en grupo. En cuanto a la especialización, no se ahonda demasiado en las razones por las que se produce. [43] Es un trabajo similar donde se agrega que los cazadores aprenden a comportarse de manera altruista, es decir que cuando localizan la presa, no van inmediatamente a posicionarse junto a ella, sino que se mantienen a cierta distancia, permitiendo a los compañeros acercarse también.

Entre los aspectos abordados en estos trabajos, y los que son importantes para nuestra tarea, podemos incluir si el aprendizaje es centralizado o distribuido, si existe o no comunicación entre los agentes acerca de sus decisiones, o si el entorno sobre el que se trabaja se considera estocástico o determinístico. Como se mencionó, una dificultad que se presenta al ir sumándose más agentes es el crecimiento exponencial del espacio de estados y de acciones (que pasa a llamarse acción conjunta). En cualquiera de las decisiones que tomemos, habrá que lidiar con este problema.

En nuestro caso, trataremos el problema como de aprendizaje distribuido, sin comunicación entre agentes y en un entorno estocástico.

Es distribuido por una decisión del enfoque que queremos darle a nuestro trabajo, donde nos interesa cómo se desenvuelve cada agente en forma autónoma. En un esquema centralizado, por otro lado, al ir incorporando cada vez más agentes, se vuelve difícil la escalabilidad. En un esquema distribuido, en principio cada agente modela su acción. Más adelante veremos que existen técnicas para, o bien inferir lo que ocurre con los demás, o bien modelar parte del resto del entorno.

Respecto a la comunicación entre agentes, Claus y Boutilier definen en [44] el caso de Joint Action Learners (JAL), donde cada agente puede observar tanto su comportamiento como el de los demás en contraposición a Independent Learners (IL), donde un agente sólo conoce su propio comportamiento. En nuestro caso, elegimos el esquema IL porque queremos minimizar la comunicación entre agentes y así evitar tanto un overhead innecesario como la necesidad de establecer un canal de comunicación.

Queda definir si consideraremos al entorno estocástico o determinístico. No podemos asegurar que si a un estado *S* le aplicamos una acción *a* obtenemos un estado *S'* (definición de determinístico). Esto ocurre porque con la discretización que usamos, dos situaciones distintas del mundo real podrían caer en el mismo cluster y con la misma acción obtenemos dos nuevos estados que no necesariamente caen en el mismo cluster de nuestra discretización. Otro motivo para considerar estocástico al entorno es que existe un rival que sigue una política desconocida por nosotros y que influye directamente sobre el entorno, modificándolo.

Otras características del entorno sobre el cual trabajaremos, que en general no plantean discusión pero que sin embargo conviene mencionar, son que la función de refuerzo es la misma en todos los agentes y que no nos interesa forzar especializaciones en los comportamientos. Vimos que en algunos casos [42],[43] esta especialización se produce, posiblemente por una necesidad de complementar los comportamientos, pero si se da así dejaremos que suceda sin nuestra intervención.

5.6 Implementación de un algoritmo existente

Lauer y Riedmiller exponen en [45] un algoritmo cuyo dominio es el que necesitamos (aprendizaje distribuido, sin comunicación entre agentes y estocástico) y muestran resultados exitosos para ciertos experimentos. La principal idea del algoritmo es la llamada "coordinación implícita" entre los agentes. Cada agente mantiene una lista para cada estado, en la cual cada elemento contiene un componente de una acción conjunta (implícita) y el valor Q asociado a esa acción. Se debe asegurar que en todo momento todos los agentes

usen el mismo índice para acceder a sus respectivas listas, de modo que el refuerzo se asigne en forma unívoca y así el valor Q se actualiza correctamente. Para asegurar esta elección simultánea se sugiere usar la misma semilla de un generador de números al azar en todos los agentes, cosa que hicimos. Además, para lograr una eficiente exploración se sugiere ordenar las listas por el valor Q (en forma decreciente), dándole mayor prioridad a los primeros de la lista, siempre y cuando todas las acciones tengan probabilidad mayor que 0 de ser elegidas.

La versión para la cual se asegura convergencia es la llamada de *listas completas*, en la cual se contemplan todas las posibles acciones conjuntas. Esta alternativa es poco práctica, ya que el tamaño de las listas crece exponencialmente con el número de agentes. Una alternativa es la llamada de listas reducidas, donde se mantienen sólo las acciones con mayor valor Q y periódicamente se dejan ingresar otras.

```
Inicialización():
para cada agente i = 1 … m
    para cada estado s = 1 … N
        para cada índice l = 1 … l_max
            L^i(s)[l].Q ← 0.0
            L^i(s)[l].a ← ElegirAccionCoordinada()

aprenderTransición (estado s):
para cada agente i = 1 … m
    l_max ← SeleccionarIndice
    a_i ← L^i(s)[l_i].a
aplicar a = (a_1, … , a_m)
observar s', r
para cada agente i = 1 … m
    L^i(s)[l_i].Q ← L^i(s)[l_i].Q + α_i[r + γ max_j L^i(s')[j].Q]
                                  - L^i(s)[l_i].Q
    ordenarListasPorQDecreciente(L^i(s))

principal:
inicialización()
repetir
    aprenderTransicion(estadoActual)
hasta criterio de convergencia
```

Figura 5.10: Pseudocódigo del algoritmo AR estocástico, distribuído y con coordinación implícita [44], en la versión de listas completas. El paso donde se selecciona la acción se reemplaza por la selección de un índice *l*. $L_i(s)[l]$ es el elemento *l* de la lista del agente i para el estado s y contiene una tupla con el valor Q y a_i (el componente i del vector de acción conjunta referenciado por *l*). Para un índice *l* la acción efectuada es

L(s)[*l*].a. elegirAccionCoordinada() debe ser capaz de dejar para cada indice l un componente del vector de acción conjunta correspondiente para cada agente.

En nuestro trabajo, usamos la versión de listas reducidas, con una extensión máxima de 25 elementos de las listas (el total de acciones conjuntas posibles, para el caso de 2 jugadores y un quinto del total para el de 3) y quedándonos con los 10 mejores elementos al realizar la poda periódica. Adaptamos las listas a la estructura de RBF, es decir que cada cluster contiene una lista con los componentes de las acciones conjuntas correspondientes al agente. La manera en que veníamos implementando Q-Learning en los diferentes algoritmos, que era eligiendo la acción con máximo valor Q al explotar y cualquier otra al explorar, se adapta al requerimiento de elegir la acción con mayor valor Q con mayor probabilidad.

Capítulo 6

Experimentación y Resultados

En este capítulo se muestran los resultados obtenidos a partir de varias series de experiencias, las cuales ayudaron a profundizar el estudio del simulador como herramienta para implementar la técnica de aprendizaje Q-Learning con equipos de uno o varios robots y, a partir de esto, poder extraer algunas conclusiones.

6.1 Definiciones

Primeramente, definimos algunos términos que luego utilizaremos:

En el capítulo 5, sección 5.2.1, se definió la siguiente medida de efectividad *(5.1)*:

Efectividad = # goles equipo propio / (# goles equipo propio + # goles equipo rival)

Un episodio equivale a un partido, que finaliza o bien por haber concluído el tiempo máximo (en nuestro caso, 60 segundos) o bien cuando alguno de los equipos convierte un gol.

Un escenario es una situación en la que cada equipo sigue una cierta política.

Una experiencia es una serie de episodios en un escenario dado.

6.2 Encuadre experimental

Esta sección describe cómo fueron aplicadas las definiciones de la sección anterior en las simulaciones.

La configuración inicial del entorno de cada episodio se elige de la siguiente manera:

1. La posición de los robots y de la pelota se selecciona aleatoriamente con distribución uniforme sobre toda la superficie de la cancha
2. Las velocidades lineales de los robots es cero
3. Las velocidades lineales de la pelota se seleccionan aleatoriamente de manera uniforme en el intervalo [-200 cm/seg, 200 cm/seg]

Las políticas utilizadas se basan en los comportamientos de atacar (patear la pelota al arco contrario) y defender (atajar la pelota), y son las siguientes:

1. Azar: Política que, en cada paso de simulación, selecciona un comportamiento al azar.
2. Atacar: Política que selecciona constantemente el comportamiento atacar
3. Defender: Política que selecciona constantemente el comportamiento defender
4. Hardcode: Política que selecciona un comportamiento teniendo en cuenta la estrategia hardcodeada definida en la sección 5.2.7.
5. Aprendida: Política que utiliza la información (extraída de una tabla u otra estructura) de un aprendizaje de Q-Learning previo.

A continuación se enumeran los escenarios donde se ejecutaron las simulaciones. Se debe leer como "política seguida por los robots del equipo A (o propio)" vs. "política seguida por los robots del equipo B (o rival)":

1. Atacar vs. Azar
2. Azar vs. Azar
3. Aprendida vs. Azar
4. Atacar vs. Defender
5. Azar vs. Defender
6. Aprendida vs. Defender
7. Atacar vs. Hardcode
8. Azar vs. Hardcode
9. Aprendida vs. Hardcode

Cada experiencia estuvo formada por una serie de 2000 episodios. En los casos en donde se utilizó una política aprendida, además se dividió esta experiencia en dos fases. En la primera, se corrieron 2000 episodios para el aprendizaje. En la segunda, se ejecutaron 2000 episodios sin aprendizaje (fijando las tablas obtenidas en la primera fase), con la idea de utilizar lo aprendido.

La adopción de este número para la fase de aprendizaje estuvo motivada por el hecho de que a partir de las experiencias realizadas se pudo observar que luego de los 1000 episodios, la medida de efectividad se estabiliza, y su valor varía en menos de 1% en los 1000 episodios que restan hasta completar 2000. Es decir, el valor de la efectividad a partir de los 1000 episodios es prácticamente constante, por lo que al realizar series con 2000 episodios nos extendemos para dar cierto margen de seguridad a esta medida, aunque no tanto como para que la experiencia requiera demasiado tiempo. En el siguiente gráfico se observa que luego de cierta inestabilidad inicial, a partir del episodio 600 aproximadamente, la variación de la efectividad es mínima.

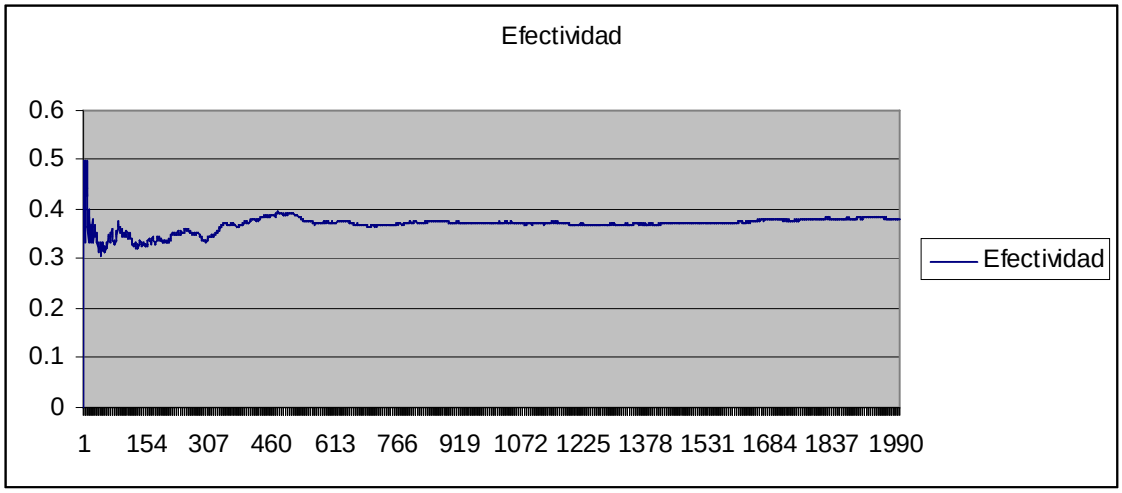

Figura 6.1: Evolución de la medida de efectividad acumulada a lo largo de una serie de episodios de aprendizaje.

En los casos donde se utilizaron políticas aprendidas, se usó una discretización de Tile Coding, dividiendo cada una de las 10 dimensiones del espacio de estados en 2 tiles. Esto da como resultado una tabla Q de 1024 estados, que al tener 2 comportamientos, genera 2048 valores.

6.3 Análisis del aprendizaje

En la tabla 6.1 se muestran los resultados de las experiencias con un jugador por equipo en los distintos escenarios:

	Escenario								
	Atacar vs. Azar	Azar vs. Azar	Aprendida vs. Azar	Atacar vs. Defender	Azar vs. Defender	Aprendida vs. Defender	Atacar vs. Hardcode	Azar vs. Harcode	Aprendida vs. Hardcode
Total de Goles:	1620	1254	1456	1435	1191	1192	1764	1600	1672
Total de goles propios:	927	618	828	757	619	620	871	622	736
Total de goles rival:	693	636	628	678	572	572	893	978	936
Efectividad:	**57.22%**	**49.28%**	**56.87%**	**52.75%**	**51.97%**	**52.01%**	**49.38%**	**38.88%**	**44.02%**

Tabla 6.1: Efectividad en los distintos escenarios.

Comparando la efectividad entre las distintas políticas del robot propio en los distintos escenarios posibles y fijando la política del rival, se puede ver que la efectividad de la política aprendida supera con bastante claridad a la obtenida

mediante una que elige acciones al azar, excepto cuando el rival tiene como política solamente defender (56.87% a 49.28%, 52.01% a 51.97% y 44.02% a 38.88%). No obstante, la política aprendida no logra alcanzar la efectividad de una política fija, como lo es la que sólo elige atacar, cosa que se evidencia especialmente cuando el rival usa la estrategia hardcodeada definida en el capítulo 4 (56.87% a 57.22%, 52.01% a 52.75% y 44.02% a 49.38%).

Deteniéndonos en el caso de que el robot propio utiliza una política aprendida y comparando las efectividades en escenarios con distintas políticas del rival, vemos que la mayor efectividad se logra, como es lógico de esperar, cuando el rival efectua un comportamiento al azar. Cuando se enfrenta a la política hardcodeada, esta efectividad se reduce considerablemente.

6.4 Evaluación de la medida de efectividad

Los valores obtenidos en el cuadro anterior nos muestran que la medida de efectividad elegida nos puede servir para estudiar el comportamiento de la técnica de aprendizaje implementada en el presente trabajo. En efecto, es razonable que una política al azar, es decir sin información, dé un valor menor que una política pensada de antemano o producto de un aprendizaje, como vimos más arriba.

Por otro lado, también es esperable el valor cercano al 50% obtenido en Azar vs. Azar y además nos dice que las condiciones de simulación no producen sesgo. Un valor similar en Atacar vs. Hardcode (49.38%) sugiere que ambas políticas fijas son igualmente eficaces.

6.5 Comparación de diferentes discretizaciones

En la sección 5.3 se mencionó que no existe un método de discretización que a priori sea mejor que otro y que nos basaríamos en los resultados (en particular en la efectividad en cada caso) para elegir un método. Para esto, realizamos o reusamos experimentos representativos de las 3 discretizaciones con las que trabajamos. En este caso, los experimentos consistieron siempre en una fase de aprendizaje y otra de 2000 episodios en la que se usa la política aprendida enfrentando a un rival que usa la política hardcodeada definida en la sección 5.2.1. La cantidad de episodios de aprendizaje se mantuvo en 2000 para Tile-Coding y Q-Kohon, aunque para Q-RBF se utilizaron 6000 episodios, producto de usar el procedimiento para ajustar el sigma y la cantidad de episodios definido en la sección 5.3.4.1. En la siguiente tabla se comparan los resultados de las 3 discretizaciones. Para Tile-Coding usamos el resultado de Aprendida vs. Hardcode mostrado más arriba. Para Q-Kohon, se realizaron pruebas con grillas

de 20 x 20 y de 60 x 60 celdas. Por último, para Q-RBF mostramos dos resultados con distinto step-size.

Método de Discretización	Particularidad	Total goles equipo propio	Total goles equipo rival	Efectividad
Tile-Coding		736	936	44.02%
Q-Kohon	20 x 20 celdas	824	1036	44.30%
	60 x 60 celdas	752	1078	41.10%
RBF	Step size = 0.1	829	1001	45.30%
	Step size = 0.3	771	1062	42.06%

Tabla 6.2: Comparación de diferentes técnicas de clustering.

En la tabla 6.2 se puede ver que no hay una clara ventaja para ninguno de los métodos, simplemente una diferencia del 1% en favor de Q-RBF. Por ese motivo y por las ventajas que ofrece Q-RBF, en particular la de crear clusters solamente en la medida de la necesidad de cada caso, se optó por este método de discretización para continuar con el análisis.

6.6 Ajuste de la elección

Para intentar obtener mejores resultados, en esta sección ajustaremos los valores de algunos parámetros. Primero, buscaremos el valor de σ más apropiado, según el criterio de convergencia definido en la sección 5.3.4.1. Luego probaremos distintos valores de α (paso temporal) y nos quedaremos con el que obtengamos la mayor efectividad.

Para aplicar el criterio de convergencia, a *k* le asignamos 5, es decir que el σ debe ser tal que en la mitad de los episodios de aprendizaje cree 5/6 de los clusters totales. Fue en este momento que se vio que con 2000 episodios y una cantidad de clusters razonable (mayor a 100) no se lograba convergencia con ningún σ, por lo cual se optó por aumentar la cantidad de episodios de aprendizaje y llevarlos a 6000. Luego de descartar valores de σ que hacían que la cantidad de clusters fuera demasiada (más de 10000) o demasiado poca (menos de 100), nos quedamos con cuatro valores:

Sigma	0.8	1.0	1.2	1.4
Clusters	5535	3802	1548	292
episodio de corte	3844	3513	3620	3474

Tabla 6.3: Aplicación del criterio de convergencia a distintos valores de σ.

Como vemos, hubo que relajar el criterio de convergencia, ya que en ningún caso se lograba que el episodio de corte (el episodio que deja *k / (k + 1)* de los clusters en la primera mitad y *1 / (k + 1)* en la segunda) quede justo en la mitad. Con esto, no había un valor de σ que mostrara claras ventajas sobre los demás, pero se eligió $\sigma = 1$ por ser uno de los que tienen el episodio de corte más cerca de 3000 (la mitad de la cantidad total de episodios de aprendizaje).

Con el σ y la cantidad de episodios fijadas, se ejecutaron experimentos con distintos α para ver con cuál se obtenía la mayor efectividad.

Alfa	Goles nuestros	Goles rival	Efectividad
0.05	822	1038	44.19%
0.1	829	1001	45.30%
0.2	807	1027	44.00%
0.3	771	1062	42.06%
0.4	791	1048	43.01%
0.5	689	1119	38.11%
0.9	774	1069	42.00%

Tabla 6.4: Comparación de la efectividad para diferentes valores de α.

Nuevamente, la diferencia no es notoria para uno u otro valor, pero $\alpha = 0.1$ supera levemente a los demás valores, por lo que se eligió para seguir con las pruebas.

6.7 Rendimiento de AR cuando crece la cantidad de agentes

En esta sección evaluaremos resultados obtenidos en escenarios de 2 y 3 jugadores por equipo, para así volver sobre las problemáticas expuestas en la sección 5.4, donde comentamos los problemas que surgen al agregar agentes. En estos casos, los experimentos consisitieron en una búsqueda binaria para hallar el sigma apropiado (al igual que en el caso de 1 vs. 1 jugador), y luego comparar la efectividad entre estas políticas de nuestros agentes, enfrentadas a las políticas hardcodeadas (para 2 o 3 jugadores, según correspondiera) definidas en el capítulo 5:

1. al azar
2. independiente, que consiste en usar Q-Learning tal como se venía haciendo cuando había un jugador por equipo, y solamente adaptar las dimensiones del espacio de estados-acciones a las correspondientes de cada caso.

3. coordinado, que consiste en implementar el algoritmo de MARL definido en la sección 5.6.

Como al incrementarse el espacio de estados también lo hace la cantidad de episodios necesaria para la convergencia, decidimos incrementar linealmente esta última (aunque la primera lo hace en forma exponencial). Para el caso de 2 vs. 2 jugadores se utilizaron 12000 episodios de aprendizaje, mientras que en 3 vs. 3 jugadores se utilizaron 18000. La cantidad de episodios de ejecución (utilizando las tablas obtenidas en el aprendizaje sin modificarlas) se mantuvo en 2000.

La siguiente tabla muestra los últimos pasos de la búsqueda binaria para llegar a elegir el sigma entre 3 valores distintos. Un sigma mayor significaría una cantidad de clusters creados similar al caso de 1 vs. 1 jugador. Dado que una posible causa de que la efectividad en este caso haya sido menor al 50% era que los clusters no alcanzaban a cubrir adecuadamente el espacio de estados, al utilizar más agentes e incrementarse este espacio, se optó por usar un sigma no mayor a 1.7. Por otro lado, un sigma menor a 1.6 hubiese implicado una cantidad de clusters muy grande y difícil de manejar.

Sigma	**1.6**	**1.65**	**1.7**
Clusters	6923	5406	4388
Episodio de corte	7472	7426	7360

Tabla 6.5: Aplicación del criterio de convergencia a distintos valores de σ en escenarios de dos jugadores por equipo.

Como se ve, en este caso también hubo que relajar el criterio de convergencia, ya que en todos los casos el episodio de corte supera los 6000 episodios. Como este criterio no arrojaba un valor claramente más favorable que otro, se prefirió elegir σ en base a la cantidad de clusters creados, tratando de quedarnos con la mayor cantidad posible (dentro de lo manejable), con lo cual se optó por $\sigma = 1.6$.

Con σ ya fijado, en la tabla 6.6 se pueden comparar los resultados de las 3 políticas definidas más arriba:

	Política		
	Azar	**Independiente**	**Coordinado**
Goles nuestros	461	600	576
Goles rival	1395	1292	1319
Efectividad	24.84%	31.71%	30,40%

Tabla 6.6: Comparación de diferentes políticas en escenarios de dos jugadores por equipo, enfrentando a un equipo que sigue la política hardcodeada para dos jugadores (ver figura 5.8 en sección 5.4).

Hubo una clara mejora en las políticas aprendidas con respecto a la política al azar. Sin embargo, el algoritmo multiagente no mejoró respecto al independiente.

Pasando al caso de 3 vs. 3 jugadores, utilizando $\sigma = 2.1$ se obtuvieron estos resultados:

	Política		
	Azar	**Independiente**	**Coordinado**
Goles nuestros	373	408	479
Goles rival	1512	1502	1433
Efectividad	19.79%	21.36%	25,05%

Tabla 6.7: Comparación de diferentes políticas en escenarios de tres jugadores por equipo, enfrentando a un equipo que sigue la política hardcodeada para tres jugadores (ver figura 5.9 en sección 5.4)..

Aquí, si bien la política al azar sigue siendo la que produce la peor efectividad, la política independiente no es mucho mejor, siendo claramente superada por la política con coordinación. Esto hace pensar que los problemas de un algoritmo sin coordinación expuestos en la sección 5.4 se hacen evidentes cuanto más crece la cantidad de agentes.

6.8 Métrica de Cluster Aliasing

Como vimos en el Capítulo 5, existe un problema inherente a cualquier discretización conocido como Cluster Aliasing, que dificulta el aprendizaje. Para tener una idea de cuánto puede estar afectando este cluster aliasing la tarea del agente, se pensó en medirlo de la siguiente manera.

Para cierto sigma, se divide el espacio de estados en regiones y luego se mide la proporción de clusters que son seleccionados desde más de una región (es decir, cuando la situación actual cae en al menos dos regiones distintas y se selecciona el mismo cluster).

Las regiones definidas en este caso fueron 2, determinadas por si la pelota está a la izquierda o derecha de la linea central. Se dejaron correr normalmente 100 episodios para 5 diferentes valores de sigma, midiendo en todos los pasos temporales de estos episodios la selección de clusters desde las 2 regiones, obteniéndose estos resultados:

Sigma	**CA**
1.00	0.116
1.25	0.284
1.50	0.438
1.75	0.641
2.00	0.971

Tabla 6.8: Medida de Cluster Aliasing para diferentes valores de σ.

Se ve claramente que a mayor σ, mayor CA, lo cual probablemente sea una de las causas por las que la efectividad se ve afectada. Se deberá obtener un balance entre el grano de cluster y el tiempo de aprendizaje para lograr la mejor efectividad posible.

6.9 Conclusiones

La mejora producida por el aprendizaje respecto a una política al azar se sigue verificando al agregar agentes. Sin embargo, la efectividad comparada entre escenarios con distintas cantidades de agentes, muestran que ésta disminuye al ir creciendo la cantidad de agentes, tanto con la política coordinada como con la no coordinada. Tal vez el espacio de estados sea realmente inabarcable de la forma que fue planteado y por eso no alcancen los episodios de aprendizaje que utilizamos, a pesar de haberlos incrementado.

Se vio que una vez elegido el método de discretización RBF, se esperaba que hubiese convergencia en cierto punto en la creación de clusters. En la práctica eso no ocurría ya que para cualquier cantidad de episodios totales que tomáramos (siempre dentro de lo posible para medir), el episodio era posterior a la mitad. Esto quiere decir que aún en etapas tardías del aprendizaje se seguían creando una considerable cantidad de clusters, cuando se suponía que esto no debería ocurrir por haberse visitado ya la mayoría de las situaciones posibles.

Capítulo 7

Conclusiones y trabajos futuros

En esta tesis se estudió la problemática de implementar el aprendizaje por refuerzo en el dominio del fútbol de robots y se implementó un simulador que nos permitió realizar experimentos en forma realista.

Se introdujo el tema de aprendizaje por refuerzo, mencionándose diferentes métodos de aplicación y en particular la técnica de Q-Learning. Luego, se implementó esta técnica y se definieron las dimensiones del espacio de situaciones como así también los comportamientos básicos para los robots en un equipo. A la hora de elegir una técnica de discretización adecuada, se evaluaron las alternativas Coarse-Coding, Q-Kohon y Q-RBF, quedándonos con esta última. Con esta elección se validó la hipótesis de que la incorporación dinámica de clusters es más efectiva que una cantidad fija (Q-Kohon) o una partición elemental del espacio de estados (Coarse-Coding).

Para llevar a cabo la experimentación, se diseñó un framework de simulación, que permite simular partidos con duración, tamaño de paso temporal y cantidad de robots a elección. El framework delega la resolución de las trayectorias de los robots de un punto a otro en una biblioteca existente, que utiliza las funciones de navegación de Fiorini. El hecho de que esta biblioteca haya sido desarrollada en el Departamento de Computación de la FCEN y por ende haber podido interactuar con los autores, y además que resuelve bien el problema de los obstáculos móviles, nos llevó a adoptarla. Con este framework se logró una madurez y robustez aceptable, lo que lo vuelve una opción a considerar para futuras investigaciones de aprendizaje en fútbol de robots

Se analizaron las diferentes alternativas en cuanto al tipo de motor de simulación y finalmente se eligió implementar el simulador mediante ODE, un motor de simulación de sistemas dinámicos de cuerpos rígidos, cuyo nivel de detalle para el modelado de entornos físicos, sumado a que cumplía la mayoría de las características que se esperaban de un simulador (ver sección 2.6), lo acercó a nuestro objetivo de conseguir simulaciones con realismo. Sin embargo, se presentaron algunos problemas relacionados a la estabilidad de los robots que no pudieron ser resueltos totalmente, por lo que para un trabajo futuro se puede investigar con profundidad otras alternativas disponibles (Newton [9], por ejemplo).

En los experimentos realizados, luego de una cantidad suficiente de episodios de aprendizaje se consiguió que el agente logre una efectividad similar a la obtenida mediante una política hardcodeada. Al pasar a escenarios donde había más de un jugador por equipo, comenzaron a evidenciarse problemas derivados del Cluster Aliasing y el crecimiento exponencial del espacio de estados. Se investigaron algoritmos existentes de aprendizaje por refuerzo Multiagente, haciendo hincapié en la coordinación entre los agentes y se implementó el que más se acercaba a las características del entorno, que se comportó mejor que el no coordinado. Aun así, queda bastante por investigar en este aspecto, sabiendo de antemano que el cardinal del espacio de estados es un problema de difícil tratamiento.

Referencias

[1] http://www-2.dc.uba.ar/proyinv/robotica/spa/pg_index.php

[2] http://www.robocup.org/

[3] http://www.fira.net/

[4] http://sserver.sourceforge.net/

[5] Baraff, D. (1997) *An Introduction to Physically Based Modeling: Rigid Body Simulation*, SIGGRAPH'97 Tutorial Notes.

[6] Hartman, Philip (2002), *Ordinary Differential Equations, 2nd Ed.*, Society for Industrial & Applied Math.

[7] http://dynamechs.sourceforge.net/

[8] http://ode.org/

[9] http://www.newtondynamics.com/

[10] Trinkle, J., Pang, J., Sudarsky, S. y Lo, G. (1995) *On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction*, Mathematical Programming, 73, pp. 199–226.

[11] Stewart, D. y Trinkle J. (1996) *An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Inelastic Collisions and Coulomb Friction*, International Journal for Numerical Methods in Engineering, 39, pp. 2673 - 2691.

[12] Anitescu, M. y Potra F. (1997) *Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction As Solvable Linear Complementarity Problems*, ASME Nonlinear Dynamics, 14, pp. 231–247.

[13] Smith, R. (2004) *Open Dynamics Engine v0.5 User Guide*, http://ode.org/odelatest-userguide.html.

[14] http://www.fira.net/soccer/mirosot/overview.html

[15] http://www.edrobot.com/english/product/ysra.asp

[16] http://www.dc.uba.ar/people/proyinv/robotica/spa/pg_chebot.php

[17] http://www-2.dc.uba.ar/proyinv/robotica/spa/pg_images.php

[18] Soria, S., Santos, J., (2003). *Velocity and Trajectory Computing Using the Velocity Obstacle Approach in Robot Soccer*, Proceedings of the FIRA Robot World Congress 2003 October 1 – 3, 2003, Vienna, Austria.

[19] Fiorini, P., (1995). *Robot Motion Planning Among Moving Obstacles*, PhD Thesis, University of California.

[20] Wolpert, D. H., Macready, W. G. (1997). *No Free Lunch Theorems for Optimization*, IEEE Transactions on Evolutionary Computation, Vol 1, No. 1, pp. 67-82.

[21] Smart, William D., Pack Kaelbling, Leslie (2002). *Effective Reinforcement Learning for Mobile Robots*, in Proceedings of the International Conference on Robotics and Automation (ICRA-2002), volume 4, pp. 3404–3410.

[22] Sutton, Richard S., Barto, Andrew G. (1998). *Reinforcement Learning, an introduction*, The MIT Press.

[23] Uther, William T. B., Veloso, Manuela M. (1998). *Tree Based Discretization for Continuous State Space Reinforcement Learning*, in Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI, pp. 769–774.

[24] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD. thesis, Cambridge University.

[25] Kim, J-H, et.al. (1997). *Action Selection and Strategies in Robot Soccer Systems*, in Proceedings of the 40th Midwest Symposium on Circuits and Systems, 1, pp. 518-521.

[26] Han, K-H, et.al. (2002). *Robot Soccer System of SOTY 5 for Middle League MiroSot*, 2002 FIRA Robot Congress, pp. 632-635.

[27] Veloso, M., Stone, P., Han, K-H. y Achim, S. (1997) *CMUnited: A Team of Robotic Soccer Agents Collaborating in an Adversarial Environment*, In Hiroaki Kitano, ed., RoboCup-97: The First Robot World Cup Soccer Games and Conferences, Springer Verlag, Berlin.

[28] Santos, J. , Scolnik, H., Laplagne, I., Daicz, S., Scarpettini, F., Fassi, H., y Castelo, C. (2001) *UBA-Sot: An Approach for Control and Team Strategy in Robot*

Soccer, International Journal of Control, Automation and Systems, 1(1), pp. 149 - 155.

[29] Park, K-H., Kim, Y-J. y Kim, J-H. (2001) *Modular Q-Learning Based Multi-Agent Cooperation for Robot Soccer*, Robotics and Autonomous Systems 35, pp. 109-122.

[30] Fernández, F. y Borrajo, D. (1999) *VQQL. Applying Vector Quantization to Reinforcement Learning*, RoboCup-99: Robot Soccer World Cup III (1999) pp. 292-303. Springer.

[31] Baltes, J., Lin, Y. (1999) *Path Tracking of Non-holonomic Car-like Robot with Reinforcement Learning*, RoboCup-99: Robot Soccer World Cup III (1999) pp. 162-173. Springer.

[32] Sutton, Richard S. (1996) *Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding*, Advances in Neural Information Processing Systems 8. MIT Press.

[33] Whitehead, Steven D., Ballard, Dana (1991) *Learning to Perceive and Act by Trial and Error*. Machine Learning, 7, pp. 45-83. Kluwer Academic Publishers.

[34] Matuk, Rosana (2001) *Cluster-Aliasing en Aprendizaje en Robótica* – Tesis de Licenciatura, Universidad de Buenos Aires.

[35] Bonarini, A., Bonacina, C., Matteucci, M. (1999) *Fuzzy and Crisp Representations of Real-valued Input for Learning Classifier Systems*. In Proceedings of IWLCS99, Cambridge, MA, pp. 228-235, AAAI Press.

[36] Touzet, Claude (2003) *Q-Learning for Robots*. In The Handbook of Brain Theory and Neural Networks (Second Edition), M. Arbib (Ed.), pp. 934-937. MIT Press.

[37] Kohonen, T. (1995) *Self-Organizing Maps*. Springer.

[38] Santos, J. (1999) *Contribution to the study and the design of reinforcement functions*. PhD Thesis, Universidad de Buenos Aires and Université d'Aix-Marseille III.

[39] Matarić, M. (1994) *Learning to Behave Socially*. In Cliff, D., Husbands, P., Meyer, J.-A., & Wilson, S. W. (Eds.), From Animals to Animats 3: Proceedings of

the Third International Conference on Simulation of Adaptive Behavior, Cambridge, MA, pp. 453–462. MIT Press / Bradford Books.

[40] Matarić, M. (1997) *Reinforcement Learning in the Multi Robot Domain*. Autonomous Robots, 4(1), pp. 73–83.

[41] Ghavamzadeh, M., Mahadevan, S. (2002) *A Multiagent Reinforcement Learning Algorithm by Dynamically Merging Markov Decision Process*. In Proceedings of the First International Conference on Autonomous Agents and Multi-agent Systems, Bologna, Italy.

[42] Ono, N., Fukumoto, K. (1996) *Multi-agent Reinforcement Learning: A Modular Approach*. In Proceedings of the Second International Conference on Multi-agent Systems, Kyoto, Japan, pp. 252-258. AAAI Press.

[43] Ono, N., Fukumoto, K. (1996) *A Modular Approach to Multi-agent Reinforcement Learning*. Selected papers from the Workshop on Distributed Artificial Intelligence Meets Machine Learning, Learning in Multi-Agent Environments, pp. 25-39.

[44] Claus, C., Boutilier, C. (1998) *The dynamics of reinforcement learning in cooperative multiagent systems*. In Proceedings of the Fifteenth National Conference on Artificial Intelligence, Menlo Park, CA. AAAI Press.

[45] Lauer, M. Riedmiller, M. (2004) *Reinforcement Learning for Stochastic Cooperative Multi-Agent-Systems*, in Proceedings of AAMAS 2004, pp. 1516-1517.