UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

APRENDIZAJE REFORZADO ORIENTADO A LA TOMA DE DECISIONES EN EL FÚTBOL ROBÓTICO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

PABLO RECABAL GUIRALDES

PROFESOR GUÍA:
SR. JAVIER RUIZ DEL SOLAR SAN MARTÍN

MIEMBROS DE LA COMISIÓN:
SR. HECTOR AGUSTO ALEGRÍA
SR. PABLO GUERRERO PÉREZ

SANTIAGO DE CHILE
ABRIL 2009

Resumen de la Memoria para optar al
Título de Ingeniero Civil Electricista
Por: Pablo Recabal Guiraldes
Prof. Guía: Dr. Javier Ruiz del Solar
Santiago, Marzo de 2009

"Aprendizaje Reforzado Orientado a la Toma de Decisiones en el Fútbol Robótico"

En el contexto del fútbol robótico la toma de decisiones es un problema interesante y complejo de resolver. El objetivo del presente trabajo es desarrollar un algoritmo que permita al robot tomar la decisión de qué hacer cuando está en posesión de la pelota, de modo de mejorar los resultados finales de cada partido. Para esto, se propone un algoritmo de aprendizaje reforzado, el cual mediante la experimentación del mundo, representado por estados, ajuste los parámetros del sistema de modo de maximizar cierta entrada o recompensa.

El problema se modela con un espacio de estados reducido de modo de conseguir una generalización más amplia. Se implementa un algoritmo basado en *Q-Learning* y otro basado en *SARSA*, acercamientos levemente distintos de aprendizaje reforzado. Los experimentos consisten en partidos de diez minutos con cuatro jugadores por lado en donde uno de los dos equipos juega utilizando el algoritmo propuesto y el otro utiliza una estrategia estándar. Tanto para *Q-Learning* como para *SARSA* se alternan períodos en los que se juega utilizando el algoritmo "puro", con otros en donde se eligen algunas acciones al azar.

Analizando la diferencia de goles correspondiente a cada período y a cada algoritmo, se puede observar en ambos casos una leve tendencia creciente en la diferencia de goles, sin embargo, esta no es categórica debido a la alta dispersión de los datos. Además, es posible observar como *SARSA* presenta mejores resultados si se considera los resultados globales, mientras que *Q-Learning* presenta una tendencia creciente más pronunciada para las pruebas que involucran al algoritmo puro. Los objetivos no se satisfacen completamente, pues después de más de 90 horas de entrenamiento ninguno de los dos algoritmos es capaz de superar a la estrategia estándar.

Dedicatoria

A mi familia.

Agradecimientos

Quiero agradecer a los miembros del Laboratorio de Robótica de la Universidad de Chile, especialmente a mis amigos Pablo Guerrero, Javier Testart y Ricardo Dodds, por su ayuda y paciencia durante la realización de este trabajo; y a mi profesor guía, Dr. Javier Ruiz del Solar, por su confianza y orientación durante la carrera, y en particular para el desarrollo de esta memoria.

Pablo Recabal G.

Santiago, Diciembre de 2008

Índice General

Índice de figuras

Capítulo 1

Introducción

1.1. Antecedentes Generales

La Robótica es una disciplina muy compleja y con múltiples desafíos a resolver. En particular, un gran subconjunto de la robótica, el diseño de robots autónomos es el que más desafíos presenta y por ende el más interesante de abordar. Un robot se considera autónomo cuando es capaz de actuar y tomar decisiones sin una intervención humana, es decir, de manera desatendida.

El funcionamiento de un robot autónomo es más complejo a medida que el medio en el que éste se desenvuelve es más dinámico. Dado el nivel de desarrollo tecnológico actual, es necesario trabajar en ambientes acotados, los cuales en este trabajo se definen como ambientes en los cuales las variables presentes de interés para el robot son siempre las mismas. En este contexto, el fútbol robótico ha demostrado ser un ambiente fértil para el desarrollo de nuevas tecnologías: se trata de un ambiente acotado, pues cuenta con reglas definidas y tareas específicas a realizar por los jugadores en un tiempo establecido[1], pero no por esto exento de dificultad ni desafíos.

La presente memoria se enmarca dentro del trabajo realizado por el equipo del Laboratorio de Robótica de la Universidad de Chile, que desde el año 2003 participa en el mundial de fútbol robótico organizado cada año por la RoboCup^{TM}, organización internacional dedicada

[1]Mayor detalle acerca de las reglas, se entrega en la sección 1.3.1

a promover la inteligencia artificial, la robótica y las disciplinas relacionadas[2].

Para el presente trabajo se utiliza un simulador para los robots AIBO desarrollado por el equipo de robótica de la Universidad de Chile[3], el cual será explicado en detalle en la sección 1.3.3.

1.2. Motivación

Desde un punto de vista biológico y motriz, jugar al fútbol es una de las actividades más complejas que un ser humano puede realizar. A modo de ejemplo se puede considerar la habilidad que debe tener un jugador de fútbol para simultáneamente; estimar la trayectoria de la pelota; correr, esquivando rivales, en dirección a esa trayectoria; calcular el instante en el cual debe saltar para poder intersectar dicha trayectoria con su cabeza; posicionar la cabeza, y luego moverla en el momento del impacto para darle una nueva trayectoria a la pelota, la cual también tiene un cálculo asociado a un lugar del arco para que se convierta en un gol. Así también sucede con casi todas las situaciones y aspectos del fútbol.

En particular, el juego en equipo es una tarea difícil de resolver. En el caso del fútbol robótico ¿Cómo puede programarse un robot para que tenga la habilidad de dar pases a sus compañeros, de controlar la pelota, de tirar la pelota al arco o a algún punto específico? Y si este tema ya está resuelto, ¿Cómo puede programarse un robot para que éste decida cual de estas acciones es más conveniente? Por ejemplo, para una situación específica ¿es más conveniente dar un pase a un compañero en vez de tirar al arco? Definir manualmente un conjunto de reglas es el primer acercamiento para resolver este problema, pero se cree que no es una solución muy efectiva, ya que a priori parece haber muchos escenarios posibles, dificultando mucho una programación de este tipo. Otro acercamiento consiste en el aprendizaje: construir un algoritmo para que el robot aprenda mediante su propia experimentación qué decisiones son favorables en cada caso. Muchos esquemas de aprendizaje están originalmente basados en la forma en que los humanos aprenden, por lo que resultan bastante

[2] www.robocup.org
[3] www.robocup.cl

intuitivos de implementar.

1.3. El Fútbol Robótico

1.3.1. La competencia y sus reglas

La propuesta de esta memoria está enfocada a robots que participan de la "plataforma estándar", competencia que es parte del mundial de robótica organizado por la RoboCup^{TM}. A continuación se explica brevemente las reglas de esta liga, que resultan importantes de entender para la programación de la solución. La plataforma estándar es una liga de fútbol robótico en la que dos equipos de cuatro robots cada uno juegan fútbol en una cancha como la que se observa en la figura 1.1.

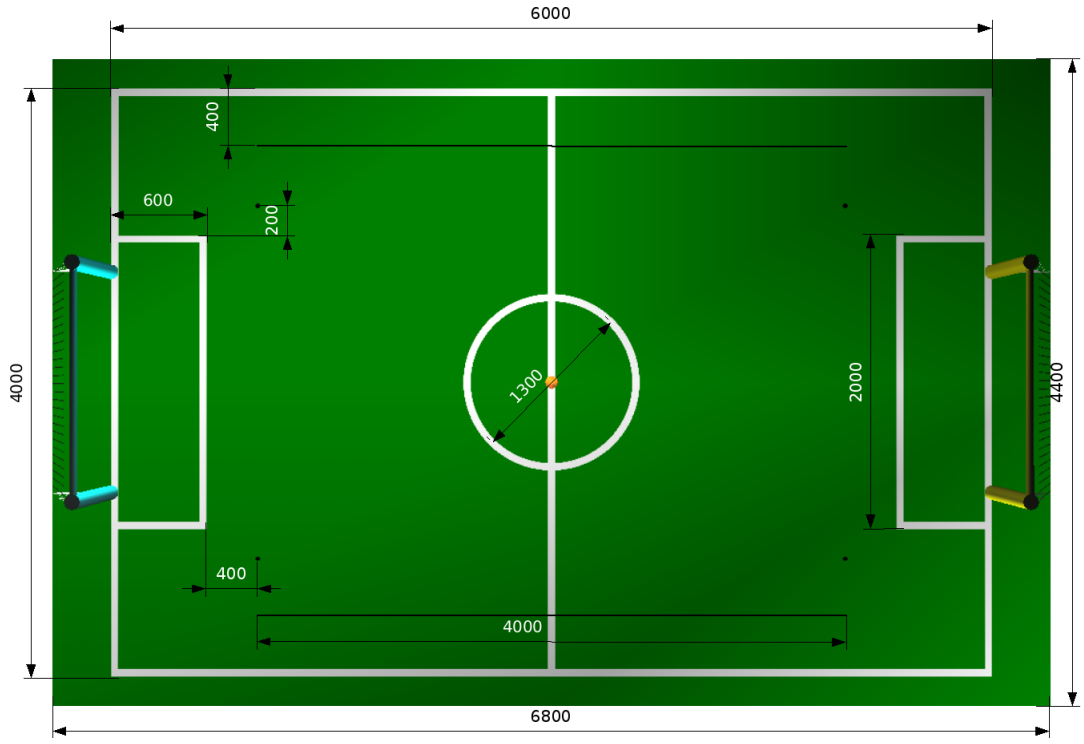

Figura 1.1: Cancha utilizada para la plataforma estándar (dimensiones en milímetros).

Ambos equipos deben usar el mismo tipo de robots[4], por lo que la única diferencia entre

[4]Desde el año 2008 se usa para esta liga los robots NAO, desarrollados por la compañía francesa Aldebaran Robotics

equipos queda determinada únicamente por la programación de los robots. Las reglas son lo más similares posible a las del fútbol humano, con limitaciones de tiempo y tamaño principalmente, impuestas tanto por el propio *hardware* de los robots (por ejemplo, duración de las baterías) como por las dificultades que existen en la percepción de los objetos (que obligan a contar con iluminación controlada y colores definidos de los distintos objetos)[5]. Pese a lo anterior, se debe mencionar que se avanza cada vez más hacia escenarios más realistas.

1.3.2. Problemáticas a Resolver

A modo de explicar las diversas problemáticas que deben resolverse, se explicará las distintas tareas que debe realizar un robot para poder jugar al fútbol, basándose en el esquema de programación realizado por el equipo de la Universidad de Chile [7]. En este esquema, se dividen las tareas globales en cuatro módulos independientes, que son: Visión, Localización, Actuación y Estrategia. Estos módulos se describen brevemente a continuación:

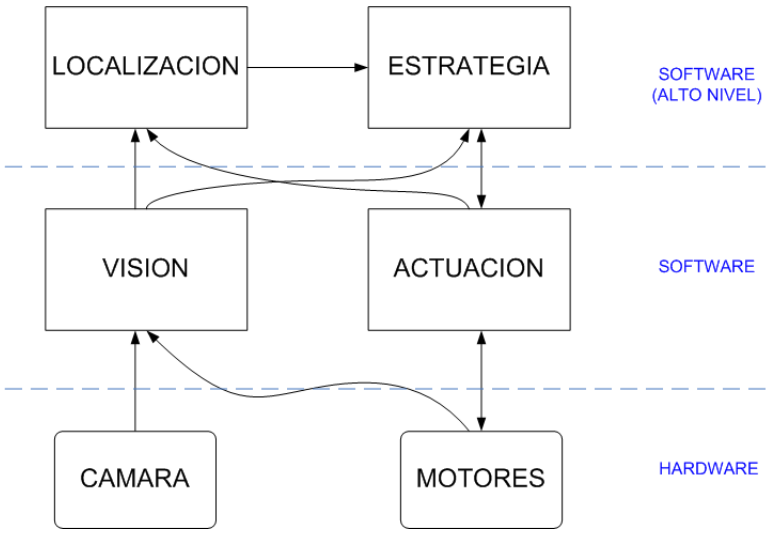

Figura 1.2: Módulos en los que se separa la programación de los robots por parte del equipo de la Universidad de Chile. Cada módulo esta representado por una caja rectangular. Las flechas representan las comunicaciones entre módulos, las cuales incluyen información relevante para el funcionamiento de cada módulo.

Cuando se habla de visión se alude al procesamiento que se realiza a la secuencia de

[5]Para un mayor detalle de las reglas consultar www.robocup.org.

imágenes (video) capturada por el robot. En el caso de la RoboCup^{TM}, los robots están equipados con cámaras, las cuales son las encargadas de recibir la mayor parte de la información externa al robot. Dentro de las problemáticas que se consideran parte del módulo de Visión, se puede mencionar la detección de la pelota, la detección de los arcos, de los compañeros y de los oponentes, todas tareas sumamente complejas.

El módulo de Localización es el que calcula la ubicación del robot dentro de un sistema de coordenadas. Para esto existen distintos acercamientos, los cuales se basan en la información recibida por los sensores del robot (sensores visuales, sonares, sensores de tacto, sensores infrarrojos, etc.) y en el conocimiento de la relación entre el movimiento de los motores y el desplazamiento. La importancia de la localización reside en que para cada robot es clave conocer al menos su propia ubicación. Así podrá aportar al juego.

Para los robots articulados, la caminata consiste en el algoritmo encargado de hacer que el robot se mueva. En el caso de la RoboCup^{TM}, existe la categoría de robots cuadrúpedos y la de robots bípedos. En ambos casos se debe generar una secuencia de movimiento de los motores para producir un desplazamiento del robot. Es deseable que la caminata sea estable, lo más rápida posible y minimice la energía consumida[6]. De esto y de todos los movimientos del robot se encarga Actuación.

Al igual que en un partido de fútbol real, la estrategia caracteriza el tipo de juego que se va a realizar ¿Cómo será el juego en equipo? ¿Qué debe hacer un "jugador" cuando esté en posesión de la pelota? ¿Y qué debe hacer cuando no lo esté? Todas estas interrogantes son respondidas al definir la estrategia. Ésta es una componente muy importante del problema del fútbol robótico, pues evidentemente, el comportamiento de los jugadores es crucial para el resultado de un partido. En particular, como se explicó en la sección 1.3.1, para la categoría en la que se enmarca este trabajo, todos los equipos utilizan el mismo tipo de robot, por lo que la "inteligencia"de cada robot es aun más relevante en el resultado final al ser lo único que diferencia el juego de cada equipo.

[6]Entre la velocidad de la caminata y el consumo energético, al igual que en los seres vivos, existe un compromiso. Por esta razón se debe definir criterios que indiquen cuando usar las distintas alternativas existentes. (Por ejemplo, los seres humanos pueden correr muy rápido por intervalos cortos de tiempo, de modo de no consumir toda su energía).

La estrategia es importante también porque es una problemática que presenta muchos desafíos sin resolver y en la que quedan muchísimas mejoras por realizar para que los robots puedan igualar el nivel del ser humano en esta disciplina. Es dentro de este tema que el algoritmo propuesto por la presente memoria se enmarca. En términos simples, propone una solución a la toma de decisiones, parte importante de la estrategia.

En general el problema de la toma de decisiones de los robots es abordado con estrategias que van desde utilizar aprendizaje de máquinas[7] o un algoritmo genético, hasta una programación del robot basada en reglas heurísticas por casos. En este trabajo, se utiliza una estrategia basada en el aprendizaje reforzado, tópico que se explica en la sección 2.2.

1.3.3. El simulador de alto nivel

La programación de los robots se hace en módulos independientes, cada uno orientado a realizar las tareas específicas descritas en la sección 1.3.2. Esquemáticamente, los módulos pueden ser representados como se observa en la figura 1.2, en donde cada módulo está representado por una caja rectangular y la información recibida y transmitida por cada uno se representa por flechas.

Debido a que el presente trabajo se enfoca en el módulo de Estrategia, se dará una breve explicación de la información que recibe y que entrega este módulo. Desde Localización recibe las coordenadas de cada robot, la pelota y los arcos de la cancha; recibe desde Visión información que es utilizada para saber con qué confianza es recibida la información desde Localización; como salida, le dice a Acutación que acción ejecutar y desde Actuación recibe de vuelta información acerca de qué acción fue realmente ejecutada. Mayor detalle acerca del esquema utilizado para programar estos robots se encuentra en [7].

El simulador de alto nivel sólo se preocupa de simular los módulos "de alto nivel"que procesa el robot, es decir, la Estrategia y en algunos casos la Localización. No se encarga de simular variables de bajo nivel como el movimiento de los motores, la cinemática de los robots o la captura y el procesamiento de imágenes. En la figura 1.2 se puede observar qué módulos

[7]Traducido del termino inglés, *Machine Learning*

son los correspondientes a cada nivel, donde el nivel más bajo y el más concreto es el *hardware*, y el nivel más alto corresponde a los módulos que realizan tareas más abstractas, como son Localización y Estrategia.

1.4. Objetivos

1.4.1. Objetivos Generales

- Desarrollar e implementar un algoritmo de aprendizaje reforzado, que permita a los robots tomar decisiones convenientes en términos de los resultados finales de cada partido.

- Mejorar el esquema actual de toma de decisiones, basado en probabilidades, logrando una generalización entre situaciones similares.

1.4.2. Objetivos Específicos

- Generar una o más medidas que permitan evaluar el desempeño del algoritmo para comparar su rendimiento con el de otros algoritmos. Esta medida además permite ajustar los parámetros y características del método a desarrollar de modo de obtener un rendimiento óptimo de éste.

- Dado que la aplicación final del trabajo propuesto es en robots que participan de la competencia de fútbol robótico, existe una limitante importante en términos de capacidad de procesamiento de la CPU de cada robot y la cantidad de memoria con que cada robot está equipado. Por esta razón, se tiene como objetivo específico que el algoritmo desarrollado minimice el uso del procesador y de la memoria requerida.

1.5. Hipótesis y Metodología

El presente trabajo propone un algoritmo de aprendizaje reforzado, el cual se postula permite a los robots simulados aprender a tomar decisiones de un modo tal que les permita

ganar los partidos. Además, se propone utilizar este algoritmo en conjunto con una codificación del espacio de estados conocida como *Tile Coding*[8], que permitirá un ahorro de memoria y una mayor generalización por parte del algoritmo. Esto último se refiere a la capacidad que tendrá el algoritmo de abstraer situaciones similares de modo de que el aprendizaje que se obtiene de una situación pueda ser aplicado en otra.

Para demostrar esto, el trabajo se divide en 3 etapas:

- Diseño del algoritmo

  Consiste en la definición detallada del algoritmo y sus parámetros. Para definir esto, se utilizan criterios orientados al correcto funcionamiento del algoritmo y logro de los objetivos.

- Implementación

  El modelo ya diseñado se lleva a nivel de código, programándolo en los robot que juegan en el simulador. Esta etapa consta de un diseño eficiente y pruebas del código hasta que éste funcione coherentemente con la solución propuesta.

- Pruebas y ajuste de parámetros

  Se lleva a cabo una etapa en la que los robots aprenden a tomar decisiones mediante el sólo hecho de jugar. Se prueba el algoritmo primero con un esquema simplificado de uno o dos robots, para luego pasar a la etapa del juego verdadero. Este consiste en partidos de 4 robots por equipos, en donde un equipo juega con una estrategia probada y el otro juega con el algoritmo aquí propuesto. A medida que se juegan más partidos, los resultados a favor del algoritmo propuesto debiesen mejorar hasta superar a la otra estrategia.

El principal aporte de la presente memoria, es que toma técnicas conocidas de aprendizaje que han sido probadas en escenarios diversos, y las aplica al fútbol robótico para una configuración real de cuatro jugadores por lado; un escenario que como se verá en la sección 2.1

[8]En español, puede entenderse como una codificación de "mosaicos" o "azulejos", la cual será explicada en detalle en la sección 2.3.

no ha sido explorado antes (en su totalidad) por métodos como éste. Además, propone una reducción del espacio de estados que permite generalizar situaciones similares que ocurren en la cancha, del mismo modo que el ser humano discierne qué elementos son más importantes dada cierta situación en una cancha de fútbol.

En el capítulo siguiente se presenta los antecedentes teóricos necesarios para la comprensión del tema. Luego, en el capítulo 3 está descrito extensamente el trabajo desarrollado. Los análisis y discusiones de los resultados obtenidos se encuentran en el capítulo 4. Finalmente, se concluye en el capítulo 5.

Capítulo 2

Marco Teórico

2.1. Trabajos Anteriores

El presente trabajo se enmarca dentro del aprendizaje reforzado multiagente, que corresponde a una extensión del aprendizaje reforzado a sistemas en los que el entorno puede ser modificado por las acciones de más de un agente. En este contexto, se debe mencionar como trabajos relacionados los acercamientos propuestos por Peter Stone correspondientes a la tarea del *Keepaway* ("Mantener Alejado") [9] y del *Half Field* ("Medio Campo") [4]. Ambos corresponden a simplificaciones o subtareas del fútbol robótico de la RoboCup, y son descritas a continuación.

La tarea *Keepaway* consiste en dos equipos, uno de los cuales —los *keepers* (o mantenedores)— intenta mantener posesión de la pelota dentro de una región limitada, mientras que el equipo oponente —los *takers* (o tomadores)— intenta ganar la posesión. Cuando los *takers* toman posesión de la pelota, o la pelota sale fuera de la región, el episodio termina y los jugadores son reiniciados para un nuevo episodio (con los *keepers* en posesión de la pelota nuevamente). Esta es una tarea mucho más simple que el juego descrito en la sección 1.3, debido a que considera limitaciones en el número de jugadores, en el tamaño de la región y principalmente en que el objetivo se reduce sólo a mantener posesión de la pelota.

Junto con la presentación de la tarea del keepaway, Stone, Sutton y Kuhlmann propusieron (en [9]) un esquema de aprendizaje reforzado conocido como *SARSA(λ)*, el cual fue combinado con *Tile Coding*. Como función de pagos utilizan recompensas asociadas a pases

correctamente realizados. Los tiempos de duración de cada episodio obtenido fueron mayores a cualquier otro algoritmo de aprendizaje reforzado.

Otro esquema para la resolución del *Keepaway* fue propuesto por Victoriano [12], en el cual propone una solución extendiendo el aprendizaje reforzado a sistemas multiagente mediante el uso de teorías de juego, para lo cual es necesario conocer las acciones del oponente. Los tiempos de duración de cada episodio no son tan buenos como los obtenidos por Stone.

La tarea *Half Field*, también propuesta por Peter Stone como una aproximación más compleja a la competencia de la RoboCup, puede ser considerada una extensión del *Keepaway*. Ésta se lleva a cabo en una mitad del campo de fútbol, mucho más grande que la región utilizada en el *Keepaway*. En cada episodio, el equipo atacante debe convertir goles, lo que involucra mantener la posesión de la pelota, moverse por el campo en dirección al arco y anotar goles. El equipo defensivo por su parte intenta no permitir que le hagan goles.

Para resolver la tarea del *Half Field*, Stone, Kalyanakrishnan y Liu proponen (en [4]) un esquema similar al utilizado en el *Keepaway*, pero con modificaciones a la función de pago que logran reducir el tiempo de convergencia. En primer lugar, se agrega una recompensa por hacer un gol. En segundo lugar, las recompensas por pases bien realizados son comunicadas entre los jugadores de un mismo equipo de modo de ser acumulativas, por ejemplo para todos los agentes que participan de una jugada de gol. Esta última modificación es la que permite la convergencia del algoritmo en tiempos comparables a los del *Keepaway*, los cuales habían aumentado considerablemente debido al aumento en la dimensionalidad del espacio de estados y al aumento en la duración de cada episodio, ambos problemas que hacen más relevante el problema de sistemas multiagente.

En el ámbito de los resultados más teóricos, Singh, Jaakkola, Littman y Szepesvari demostraron la convergencia de *SARSA* para un solo paso temporal [8]. Por otra parte, Takadama y Fujita investigaron (en [11]) la sensibilidad frente al modelamiento del problema en sistemas multiagente, mediante la comparación de los resultados para *Q-Learning* y *SARSA*. Concluyeron que una diferencia mínima en el modelamiento tiene una influencia esencial en los resultados. Además, obtuvieron guías básicas para el modelamiento de los problemas, y

compararon lo sucedido en ambientes dinámicos y estáticos.

Peng y Williams propusieron (en [6]) un algoritmo de *Q-Learning* que involucra más de un paso temporal para realizar las actualizaciones de la función de valor, basándose en el esquema TD(λ) descrito por Sutton [10] y obtuvieron resultados bastante mejores que los del *Q-Learning* clásico para el problema del control del péndulo invertido.

Finalmente, se debe mencionar el trabajo realizado por Bab y Brafman [2], que consiste en una comparación de los métodos más importantes para resolver sistemas de aprendizaje reforzado multiagente. Primero, define de manera general el problema del aprendizaje reforzado multiagente, y luego diferencia entre dos tipos de problemas: los juegos de "suma fija", en donde los agentes tienen objetivos completamente opuestos; y los juegos de "interés común", en donde los agentes tienen un objetivos comunes e intereses que no entran en conflicto. Luego, presenta 3 métodos para cada caso y los compara mediante la implementación de éstos para la resolución de problemas estándar.ente, se debe mencionar el trabajo realizado por Bab y Brafman [2], que consiste en una comparación de los métodos más importantes para resolver sistemas de aprendizaje reforzado multiagente. Primero, define de manera general el problema del aprendizaje reforzado multiagente, y luego diferencia entre dos tipos de problemas: los juegos de "suma fija", en donde los agentes tienen objetivos completamente opuestos; y los juegos de "interés común", en donde los agentes tienen un objetivos comunes e intereses que no entran en conflicto. Luego, presenta 3 métodos para cada caso y los compara mediante la implementación de éstos para la resolución de problemas estándar.

Los trabajos presentados relacionados con el *Keepaway* y el *Half Field* corresponden a aproximaciones al problema completo de cuatro jugadores por lado, que utilizan aprendizaje reforzado de manera exitosa. El presente trabajo busca resolver el problema completo.

2.2. Aprendizaje Reforzado

2.2.1. El Problema

El aprendizaje reforzado de un "agente", consiste en que éste aprenda qué "acciones" realizar cuando se encuentra en cierto "estado", de modo de maximizar una recompensa.

El agente interactúa con un entorno definido, y las acciones que éste realiza modifican dicho entorno. De aquí se puede desprender dos características de este tipo de aprendizaje: primero, debe definirse el entorno y una representación discreta de éste, que corresponde al estado; segundo, el agente debe ser capaz de identificar en qué estado se encuentra y las recompensas asociadas a cada estado[1].

La principal diferencia del aprendizaje reforzado con otros tipos de aprendizaje es que usa como información para el entrenamiento la evaluación de las acciones y el resultado de éstas en el entorno, en vez de instruir al agente diciéndole que acción tomar. Por ejemplo, para el aprendizaje supervisado se conoce a priori un set de pares entrada-salida, es decir, para cierta entrada o estado, está dada la salida o acción esperada. Es ésta la información utilizada para entrenar al agente, y en función de éstos ejemplos se modifican los parámetros internos del sistema de modo de aproximar lo más fielmente la función que mapea la entrada con la salida. En la figura 2.1 se puede observar un esquema del funcionamiento del aprendizaje supervisado. Por otra parte, en el aprendizaje reforzado, no se conoce la salida esperada, sino que se ejecutan acciones de acuerdo a cierta regla, y posteriormente se evalúa el resultado de cada acción, usándose esa evaluación —o recompensa— como información para el entrenamiento. Lo que se desea es encontrar las salidas —o acciones— que maximicen la recompensa a largo plazo.

En la figura 2.2 se observa un esquema básico del funcionamiento del aprendizaje reforzado. El agente debe experimentar el entorno mediante la realización de acciones. Los estados a los que llega el agente como resultado de dichas acciones llevan asociados una recompensa, la cual se trata de maximizar en el largo plazo por el agente mediante una "política", o en otras palabras, mediante un mapeo de estados a acciones.

El aprendizaje reforzado puede ser aplicado a tareas episódicas y a tareas no episódicas. Las primeras corresponden a tareas con un final claro, como cualquier tipo de competencia o juego, por ejemplo, el fútbol robótico. En tanto, tareas no episódicas son tareas que no tienen un final claro, sino que se ejecutan de manera permanente, por ejemplo, un proceso

[1]El contenido de esta sección está basado principalmente en la explicación realizada por R. Sutton y A. Barto en su libro "Reinforcement Learning: An Introduction". Para mayor detalle, consultar [10].

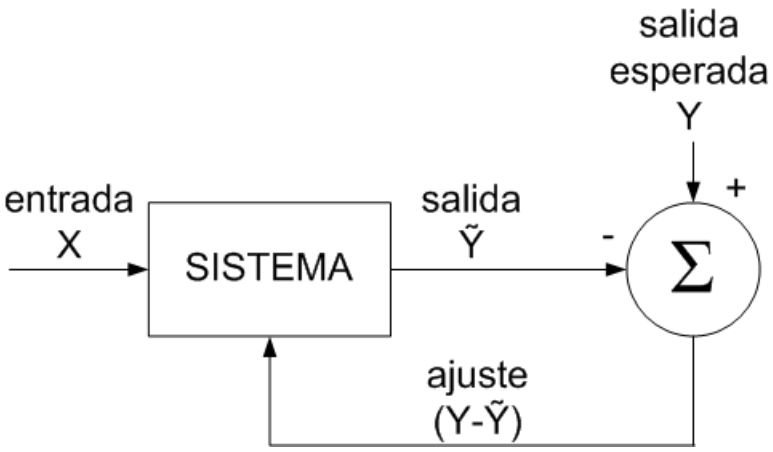

Figura 2.1: Esquema del aprendizaje supervisado

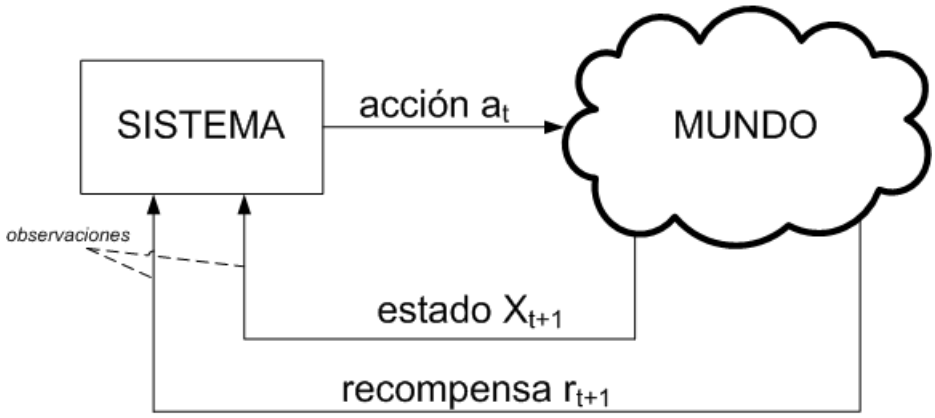

Figura 2.2: Esquema del aprendizaje reforzado

de control continuo. Si bien existen diferencias entre ambos tipos de tarea, en el presente trabajo se utiliza una notación unificada que es apropiada para ambos esquemas.

En términos concretos, lo que el agente debe maximizar es el retorno esperado, lo cual corresponde a una función de las recompensas recibidas paso a paso. En general —y especialmente en tareas no episódicas— conviene usar una función de las recompensas que utilice un factor de descuento para las recompensas futuras, de modo de considerar el valor presente de las recompensas recibidas. De esta manera, se fomenta la búsqueda de recompensas en un tiempo mínimo. En la ecuación 2.1 se observa una función de las recompensas en cada

instante de tiempo, que utiliza el factor de descuento, γ.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.1}$$

($T = \infty$ en el caso de tratarse de una tarea no episódica)

La mayoría de los algoritmos de aprendizaje reforzado se basan en la aproximación de una función de valor, la cual dependiendo del enfoque, es un indicador de cuán bueno es estar en cierto estado o de cuán conveniente es estar en cierto estado y ejecutar cierta acción. "Cuán conveniente" queda definido en función de las recompensas futuras a obtener. La función que representa el valor de estar en el estado s en el instante t y luego seguir la política π puede escribirse como $V^\pi(s)$ (ecuación 2.2), mientras que la función que representa el valor de estar en el estado s en el instante t, ejecutar la acción a y luego seguir la política π, puede escribirse como $Q^\pi(s, a)$ (ecuación 2.3).

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} \tag{2.2}$$

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\} \tag{2.3}$$

Para el presente trabajo se utilizará el enfoque representado por la ecuación 2.3, por lo que en todas las definiciones venideras se considera sólo esta variante.

Se dice que una política π es mejor que otra política π' si su retorno esperado es mayor al de π' para todo estado. Siempre existe al menos una política que es mejor o igual que todas las otras políticas. Esta corresponde a una política óptima. Si se conoce la función de valor para una política óptima (sin necesariamente conocer dicha política), basta con que en cada instante de tiempo el agente tome como decisión la acción que maximice esta función de valor. De esta manera, estará maximizando el retorno esperado. Entendiendo este concepto, se entiende la importancia que tiene la aproximación de la función de valor por parte de los algoritmos de aprendizaje reforzado.

Las políticas óptimas comparten una función de valor común, dada por:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.4}$$

La función de valor se puede estimar de distintas formas en base a la propia experiencia del agente, mediante el cómputo de los retornos inmediatos (en cada paso temporal) o acumulados (en cada episodio). Más detalles sobre esto se darán en la sección 2.2.2

2.2.1.1. Recompensa

La recompensa puede ser desde una señal numérica, hasta un concepto general cómo el "sentirse bien". Ésta debe estar alineada con el objetivo que se desea alcanzar tras el aprendizaje. Además, el agente debe ser capaz de identificar u observar las recompensas asociadas a cada estado que alcanza. En cada instante de tiempo, la señal de recompensa pasa del ambiente al agente. Esta es la recompensa inmediata r_t.

2.2.1.2. Espacio de Acciones

El espacio de acciones puede ser de bajo nivel, como es un set de órdenes específicas de movimiento a los motores, o tan de alto nivel como desplazarse a cierta ubicación y patear la pelota. Esta es una decisión que se toma al momento de desarrollar la aplicación. Sin embargo, se debe dejar en claro que el espacio de acciones está limitado a las tareas que el robot puede realizar antes de implementar el aprendizaje reforzado[2].

El set de acciones que se escoja debe estar orientado al logro de los objetivos y al aprendizaje adecuado del agente, es decir, a una modificación efectiva del entorno mediante sus acciones. Si por ejemplo, se desea implementar aprendizaje reforzado para un robot que recoja basura, lo mínimo que puede esperarse del espacio de acciones es que una de estas involucre el recogimiento de objetos, o para un robot cuya tarea involucra el recorrer un espacio determinado, se espera como mínimo que sus acciones le permitan alcanzar cada lugar de dicho espacio.

[2]Esto quiere decir que las acciones no pueden definirse como un comportamiento para el cual el robot no está programado o capacitado.

2.2.1.3. Espacio de Estados

Para poder implementar el aprendizaje reforzado, el sistema debe interactuar con el mundo[3]. Este mundo debe ser representado en variables que el sistema o robot pueda ver, las cuales son, naturalmente, discretas. A una configuración específica del mundo en un momento dado se le llama "Estado". Las distintas configuraciones que puede presentar el mundo bajo la mirada del robot corresponden al "Espacio de Estados".

Se debe mencionar que el estado puede ser en algunos casos "parcialmente observable", lo que significa que los sensores del agente no le proporcionan una observación completa del estado en cada momento. (Un entorno puede ser parcialmente observable debido al ruido y a la existencia de sensores poco exactos o porque los sensores no reciben información de parte del sistema). El aprendizaje reforzado aplicado a problemas con estados parcialmente observables es un tema muy amplio del que no se tratará en este trabajo. Para mayor información consultar [5].

2.2.2. La solución

Existen diversas estrategias para resolver el problema del aprendizaje reforzado, las cuales en general difieren en la forma en que se estima la función de valor, o en la forma en la que se busca la política óptima.

Por una parte, la Programación Dinámica (DP)[4] trata de resolver este problema mediante el cómputo de la función de valor óptima utilizando un modelo de las transiciones que ocurren en el ambiente. Para esto, se escribe la función de valor de la siguiente manera:

$$Q^*(s,a) = \sum_{s'} p(s'|s,a)[r(s'|s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{2.5}$$

donde $p(s'|s,a)$ corresponde a la probabilidad de alcanzar el estado s' si se está en el estado s y se ejecuta la acción a, y $r(s'|s,a)$ corresponde al valor esperado del retorno a obtener si se está en el estado s y mediante la ejecución de la acción a se alcanza el estado s'.

[3]Cuando se habla de "el mundo"se refiere al entorno particular al que el sistema pertenezca. En el caso del fútbol robótico, el mundo corresponde al campo de juego, incluyendo éste a los compañeros, los oponentes, la pelota, los arcos y el árbitro.

[4]Del inglés "Dynamic Programming'.

Para encontrar la función de valor óptima, primero se calcula la ecuación 2.5 para una política arbitraria π, de forma iterativa. Este procedimiento se conoce como evaluación de política (*policy evaluation*). Una vez calculada la función de valor para dicha política, se puede realizar lo que se llama un mejoramiento de la política (*policy improvement*). Esto significa volver a experimentar el entorno utilizando la política π' que corresponde a:

$$\pi'(s) = arg\,\max_{a} Q^{\pi}(s,a) \tag{2.6}$$

Este procedimiento se puede repetir, entrando en una etapa llamada iteración de política (*policy iteration*), en donde la política converge a una política óptima.

La iteración de política presenta el inconveniente de que cada una de sus iteraciones involucra una evaluación de política, la cual puede ser un proceso lento y computacionalmente costoso. Es posible truncar la evaluación de política de distintas formas sin perder las garantías de convergencia que da la iteración de política. Un caso importante es cuando la evaluación de política se trunca justo después de una pasada (recordar que esto es un proceso iterativo). Este algoritmo se conoce como iteración de valor (*value iteration*). (Para mayor detalle consultar [10], capítulo 4).

Uno de los principales problemas de la programación dinámica, es que para poder realizar este cómputo, es necesario disponer de un modelo completo del sistema, que permita calcular la función de transición de probabilidades entre estados, y las recompensas asociadas a dichas transiciones, lo que limíta bastante el campo de aplicación de esta técnica. Además, involucra cálculos computacionalmente costosos, los cuales se incrementan notablemente al aumentar el número de estados. Esta situación fue descrita por Bellman como la *maldición de la dimensionalidad* [3].

Otra forma de abordar el problema es tratándolo como un problema de Monte Carlo (MC). A diferencia de la programación dinámica, los métodos basados en MC no asumen un conocimiento completo del sistema, sino que requieren sólo experimentación del ambiente por parte del agente, que involucre distintos estados, acciones y recompensas de muestra. Estos métodos se basan en obtener promedios de muchos retornos obtenidos. Sólo al término de un

episodio es cuando la estimación de la función de valor es actualizada y la política se cambia.

A pesar de las diferencias existentes entre los métodos basados en MC y los métodos basados en DP, ambos utilizan las mismas ideas básicas. Esto es: cómputo de la función de valor para una política arbitraria, mejoramiento de política e iteración de política[5]. Para realizar la evaluación de política, una alternativa es simplemente promediar todos los retornos obtenidos desde el estado s en adelante. Esta alternativa también funciona cuando se desea obtener el valor de un par (s, a), pero se debe tener cuidado de generar alguna forma en que para cada estado, todas las posibles acciones sean exploradas.

Dada cierta experiencia obtenida al seguir una política arbitraria, los métodos MC actualizan su estimación de $Q(s_t, a)$. Si s_t es un estado no terminal visitado en el tiempo t y luego se ejecuta la acción a, ambos métodos actualizan su estimación de acuerdo a lo que ocurra después de esa visita. En términos simples, los métodos MC esperan a que el retorno que sigue a la visita se conozca, y luego actualizan su estimación de Q. Un método sencillo de actualización basado en MC se muestra en la ecuación 2.7, en donde α corresponde a la tasa de aprendizaje.

$$Q(s_t, a) = Q(s_t, a) + \alpha[R_t - Q(s_t, a)] \tag{2.7}$$

Una tercera alternativa, en la que están enmarcados los métodos utilizados en este trabajo es utilizar los métodos de Diferencia Temporal (TD)[6]. Este tipo de métodos combina las ideas combina las ideas de DP y MC, y se basa en una aproximación de la función de valor que va siendo modificada en cada intervalo de tiempo del episodio mediante la utilización de la recompensa observada en cada instante. Al igual que DP y MC, considera una evaluación, un mejoramiento y una iteración de política.

Para evaluar la política, tanto los métodos basados en TD como los basados en MC usan la experiencia. A diferencia de los métodos MC —que deben esperar hasta el fin del episodio para determinar el incremento, los métodos TD sólo esperan hasta el próximo episodio para

[5]Cada una de estas ideas es extendida a los métodos MC, en donde sólo está disponible la experiencia obtenida en la experimentación

[6]Del inglés "Time Delay".

actualizar su estimación. En el instante $t+1$ pueden realizar una actualización de la función de valor observando la recompensa y el estimado de la función de valor en t+1. Nótese que para poder evaluar Q en $t+1$, se necesita conocer una acción. Existen distintos esquemas que utilizan TD. En el presente trabajo se utiliza *Q-Learning* y *SARSA*, los cuales actualizan la función de valor de forma diferente, como se detalla a continuación.

2.2.2.1. *Q-Learning*

Q-Learning es un caso particular de aprendizaje reforzado, el cual intenta aproximar la función de valor mediante la ecuación 2.8. Para esto, se debe inicializar de alguna forma el valor de $Q(s,a)$ para todos los pares (s,a) y guardar estos en una tabla[7]. La ecuación 2.8 es aplicada en cada instante de tiempo, con lo cual se reemplaza el valor de $Q(s,a)$ almacenado por un valor que pondera la estimación antigua y la nueva (consistente esta última en la suma de la recompensa del estado al que se llega y el máximo valor de dicho estado). La tasa de aprendizaje se denota por α y el factor de descuento por γ.

$$Q(s_t,a) = (1-\alpha)Q(s_t,a) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a)] \tag{2.8}$$

La ecuación 2.8 también puede escribirse como la ecuación 2.9, en donde se observa claramente cuánto se ajusta el valor almacenado de Q(s,a).

$$Q(s_t,a) = Q(s_t,a) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a)] \tag{2.9}$$

Se habla de *Q-Learning* como un algoritmo de control *Off-Policy* —o fuera de la política, pues la estimación de la función de valor no depende de la política, o más concretamente, de la acción escogida. Esto se observa en el término correspondiente a la maximización sobre a del valor de $Q(s_{t+1},a)$ de la ecuación 2.9, en donde no existe una dependencia con la acción escogida por el agente en el instante $t+1$ y por lo tanto la actualización de la tabla es independiente de la exploración.

[7]De ahí el nombre de la "tabla" del *Q-Learning*, referido más adelante en este documento y en el código implementado

2.2.2.2. *SARSA*

A diferencia de *Q-Learning*, que utiliza el máximo valor de Q en el instante s' para actualizar la tabla, *SARSA* utiliza el valor de la función para la acción escogida en dicho instante, es decir $Q(s', a')$, tal como se observa en la ecuación 2.10. De ahí el nombre *SARSA*, pues este algoritmo involucra para la actualización de la función de valor a las variables s, a, r, s' y a'.

$$Q(s_t, a) = (1 - \alpha)Q(s_t, a) + \alpha[r(s_{t+1}) + \gamma Q(s_{t+1}, a_{t+1})] \tag{2.10}$$

Se habla de *SARSA* como un algoritmo de control *On-Policy* —o dentro de la política, pues la estimación de la función de valor depende de la política, o más concretamente, de la acción escogida. Esto se observa en el término $Q(s', a')$, pues la actualización depende directamente de la exploración.

Cualquier enfoque de TD puede ser combinado con *Trazos de Elegibilidad*[8]. Los trazos de elegibilidad son un registro de la ocurrencia de un evento, como por ejemplo estar en el estado s y escoger la acción a. Este "trazo", marca los parámetros en la memoria asociados con dicho evento como "elegibles" para ser objeto de los cambios asociados al aprendizaje. Cuando exista una actualización de TD, sólo los valores asociados a pares (s, a) elegibles son modificados. De esta forma, la propagación de las recompensas ocurre de forma mucho más rápida que cuando sólo se trabaja con un paso temporal de diferencia.

Por ejemplo, supóngase que cierto episodio comienza en el instante t_1, con el estado s_1 y se ejecuta la acción a_1, llevando a una recompensa r_2 asociada al siguiente estado. La sucesión de estados, acciones y recompensas puede numerarse como:

$$s_1, a_1, r_2, s_2, a_2, r_2, ..., s_{\tilde{t}}, a_{\tilde{t}}, r_{\tilde{t}+1}, ...$$

Si en el instante $\tilde{t}+1$ se obtiene la recompensa $r_{\tilde{t}+1}$, tanto en el caso de *Q-Learning* como en el de *SARSA* —sin la utilización de trazos de elegibilidad— el valor de $Q(s_{\tilde{t}}, a_{\tilde{t}})$ es el

[8]Traducido del inglés *Eligibility Traces*, es también referido como *Multistep TD, o TD(λ)*

único que sufre una modificación inmediata. Ahora, si se utiliza trazos de elegibilidad, esta recompensa puede propagarse —con ponderaciones temporales— hasta el primer evento de la cadena de pares (s, a) que condujo al estado alcanzado en $\tilde{t} + 1$, en este caso, hasta el par (s_1, a_1).

2.3. Aproximación de funciones mediante *Tile Coding*

Tile Coding es un método para aproximar funciones continuas, que se basa en realizar distintas particiones del dominio de la función. Cada partición se llama *tiling* (embaldosado) y cada elemento de la partición se llama *tile* (azulejo). Un estado cualquiera, queda representado por un sólo *tile* de cada partición, sumando para cada estado, un total de *tiles* igual al número de particiones [1] [13]. Cada *tile* además tiene asociado un valor o peso, el cual es modificado en función de la función que se desee aproximar.

En la figura 2.3 se observa un ejemplo para un dominio de una dimensión, el cual ha sido particionado con 3 *tilings* de iguales características, cada una desplazada por un desfase (*offset*) constante.

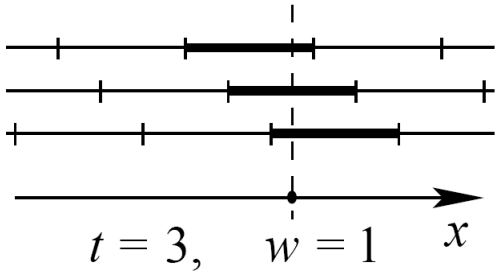

Figura 2.3: Codificación de un estado unidimensional en 3 *tilings*. El estado actual se muestra con línea punteada. Éste activa un *tile* correspondiente a cada partición, los que son representados como *tiles* más oscuros. El número de *tilings* se denota por t, y el peso de cada *tiling*, en este caso igual a 1, se denota por w.

Si se desea recuperar el valor de la función en un estado s cualquiera, basta con calcular:

$$f(s) = \sum_i w(t_i(s)) \tag{2.11}$$

en donde $t_i(s)$ corresponde al í-esimo *tile* activado por s y $w(t)$ es el peso almacenado para el *tile* t.

La implementación descrita por la ecuación 2.11 asume la existencia de un peso específico asociado a *tile*, por lo que la memoria necesaria para almacenar los pesos crece considerablemente al aumentar el numero de *tilings* y de *tiles*. Concretamente, el número de parámetros que deben almacenarse es igual a $m \cdot n$, en donde m es el número de *tilings* y n es número de *tiles* de cada *tiling* [13].

Una forma de reducir los requerimientos de memoria es mediante la implementación de una función de resumen (o como más se le conoce por su denominación inglesa, función de *hash*), la cual asocia a cada dato o elemento sobre la que se aplica una clave que lo representa de manera casi unívoca. Esto se traduce en un resumen de información, que por lo general se traduce en una disminución del tamaño total del conjunto de datos sobre el que se esté aplicando la función. Esta disminución se explica porque más de un dato puede tener asociada la misma clave al aplicarle la función de *hash*, con lo que el total de datos a almacenar es menor, y por ende, la memoria para almacenarlos también.

En la ecuación 2.12, se muestra una función de *hash* aplicada un elemento t, que representa un *tile* cualquiera. El resultado de aplicar la función a t entrega el peso w_j asociado a dicho *tile*, mas si j es menor al número total de *tiles*, es altamente probable que el peso w_j este asociado a más de un *tile*. Esto último es lo que se conoce como una colisión, y es un factor que agrega cierto error a la información, sin embargo, puede lograrse una gran disminución de memoria con pequeñas pérdidas en el rendimiento. Esto es posible debido a que una alta resolución es requerida solo en una pequeña fracción del espacio de estados (por lo que es poco probable que *tiles* que presenten colisiones interfieran en realidad entre sí).

$$hash(t) = w_j, \quad j = 1...k \tag{2.12}$$

Para el cómputo del valor de la función en cada estado se hace algo similar a lo explicitado

por la ecuación 2.11, pero previamente se debe calcular el *hash* asociado a cada *tile*, como:

$$f(s) = \sum_i hash(t_i(s)) \tag{2.13}$$

Si se desea aproximar una función desde cero, basta con presentar iterativamente ejemplos conocidos $(s, f(s))$ de dicha función al sistema, y actualizar el valor de los pesos mediante:

$$hash(t_i(s)) = hash(t_i(s)) + \alpha(f(s) - hash(t_i(s))), \quad 0 < \alpha < 1, \ \forall i \tag{2.14}$$

Las ideas expuestas en esta sección serán utilizadas en combinación con el aprendizaje reforzado. La función a aproximar descrita corresponde a la función de valor, y el dominio de ésta, al espacio de estados.

Capítulo 3

Trabajo Desarrollado

En este capítulo se describe extensamente la solución ofrecida por esta memoria al problema de la toma de decisiones en el fútbol robótico. En particular se aborda el problema de la toma de decisiones cuando un jugador está en posesión de la pelota. Para enfrentar este problema se utiliza en un principio un algoritmo llamado *Q-Learning* y al no presentar buenos resultados éste, se implementa posteriormente un algoritmo llamado *SARSA*. Naturalmente, por esta razón, una serie de definiciones correspondientes a *Q-Learning* y a *SARSA* deben ser establecidas. Esto se hace en la sección 3.1.

Una vez definido el modelo correspondiente a cada algoritmo, éste debe ser implementado. En este caso se trabaja sobre un Simulador de Alto Nivel desarrollado por el equipo del laboratorio de robótica de la Universidad de Chile. Tanto el simulador como la implementación del algoritmo sobre éste están explicados en la sección 3.2.

3.1. Modelamiento del Problema

Como se vio en el capítulo 2, un problema de aprendizaje reforzado y en particular de *Q-Learning* o *SARSA* debe tener un *espacio de estados* definido (y capaz de ser percibido por el agente), un conjunto definido de posibles acciones a realizar por el agente o *espacio de acciones*, y una señal correspondiente a la recompensa o *reward* asociada a cada estado.

Naturalmente, el fútbol robótico se trata de una tarea episódica: cada partido tiene una duración acotada y esto podría considerarse la forma más simple de un episodio. Sin embargo,

se puede profundizar más aún, y observar que dentro de cada partido existen distintos tipos de eventos que subdividen al partido en episodios más pequeños. Por una parte, para el acercamiento que utiliza *Q-Learning*, se definen episodios que terminan cuando alguno de los dos equipos hace un gol. Por otra parte, para el acercamiento que utiliza *SARSA*, se definen episodios que terminan cuando hay un gol, o cuando el oponente recupera la posesión de la pelota (esta diferencia se justifica en la sección 3.1.3).

En este trabajo el espacio de estados y el espacio de acciones se definen de igual forma para ambos acercamientos (*Q-Learning* y *SARSA*). Sin embargo, debido a lo explicado en el párrafo anterior, para la recompensa se hace necesario utilizar esquemas diferentes. A continuación se fundamenta la elección de estas variables para el diseño de cada algoritmo.

3.1.1. Definición del Espacio de Estados

Por simplicidad se propone una aplicación de aprendizaje reforzado orientada sólo a la toma de decisiones para el robot que en un instante dado esta en posesión de la pelota[1]. En adelante nos referiremos a este robot como "el agente tomador de la decisión", o simplemente, "el agente".

Debido a que se desea lograr una generalización adecuada, sumado al hecho de que la memoria en cualquier sistema es limitada, se debe procurar mantener un espacio de estados no demasiado grande. Incluso aunque el sistema sea capaz de almacenar una tabla Q en memoria grande, si el espacio de estados y/o acciones es muy grande se corre el riesgo de que ciertas combinaciones (s, a) nunca sean visitadas en el período de entrenamiento, por lo que no se estaría cumpliendo el objetivo al no existir una generalización adecuada. Es por esto que resulta conveniente escoger un número limitado de variables del entorno como representación del estado.

Por otra parte, esta representación del entorno debe ser lo suficientemente amplia de modo tal que se pueda aprender de ésta, o dicho de otro modo, existe una dimensionalidad mínima en el modelamiento del problema en la que hace sentido el hecho de que un equipo gane un

[1]Es decir, en cualquier sistema de coordenadas la posición de la pelota y el robot se asume como la misma.

partido al utilizar la estrategia en cuestión. Para ser más claros, las variables indispensables que deben ser representadas son la posición del robot en la cancha, la posición de la pelota y la posición de los arcos[2]. Este es un modelamiento básico en el cual el robot podría llegar a aprender a hacer goles en el arco rival y evitar hacer goles en el arco propio, pero en ningún caso a jugar en equipo, o a jugar contra uno o más oponentes. Para lograr esto, se debe incluir en el modelamiento del estado variables que representen la posición de los oponentes y de los compañeros.

Se ha decidido entonces hacer dos tipos de experimento; uno reduciendo de 4 a 1 los compañeros representados en el estado (para lo cual se escogerá de una manera inteligente al mejor compañero) y de 4 a 2 los oponentes representados en el estado (para lo cual se escogerá de una manera inteligente a los 2 oponentes mas peligrosos, o peores oponentes); el otro, reduciendo de 4 a 2 los compañeros representados en el estado y de 4 a 3 los oponentes representados en el estado.

Considerando los requisitos sobre las variables que representen el estado, se ha decidido utilizar un sistema de coordenadas polares, centrado en el robot que toma la decisión. Se representa en este sistema, el arco oponente y la posición del mejor compañero y de los dos peores oponentes[3]. Este sistema puede observarse en la figura 3.1. Además, se incluye como variable de estado el ángulo del robot que toma la decisión con respecto a un sistema absoluto o fijo a la cancha, pues esto permite matemáticamente conocer la posición del arco propio y además representar todas las variables antes mencionadas en un sistema absoluto o fijo a la cancha. En resumen, las variables de estado representadas, para el esquema con un mejor compañero y dos peores oponentes se listan a continuación[4]:

Sistema polar relativo al robot:

- Posición del arco oponente (r_{EN_GOAL}, θ_{EN_GOAL})

[2]O bien la posición relativa de los arcos con respecto al robot.

[3]La pelota no necesita ser representada en este sistema pues su posición coincide con el origen.

[4]Notar que este listado es fácilmente extensible al caso con dos mejores compañeros y tres peores oponentes, mediante la adición de las variables (r_{B_P2}, θ_{B_P2}) y (r_{W_E3}, θ_{W_E3}).

- Posición del mejor compañero (r_{B_P1}, θ_{B_P1})
- Posición de los dos peores oponentes (r_{W_E1}, θ_{W_E1}), (r_{W_E2}, θ_{W_E2})

Sistema fijo a la cancha (absoluto):

- Ángulo propio absoluto (θ_{SELF})

A continuación, se muestra cómo escoger al mejor compañero (seccción 3.1.1.1) y a los peores oponentes (sección 3.1.1.2). El análisis que se hace es fácilmente generalizable al caso de 2 mejores compañeros y 3 peores oponentes. De hecho, es generalizable a N compañeros y M oponentes para un partido con un número cualquiera de jugadores. Para esto, sólo basta con aplicar los mismos criterios descritos a continuación de modo encontrar el número de compañeros y oponentes a ser representados.

3.1.1.1. Elección del Mejor Compañero

Para elegir al mejor compañero dado cierto agente, se realiza un cálculo heurístico que involucra el cálculo de un puntaje —o *score*— para cada compañero. Este *score* debe dar cuenta de cuán bueno es cada compañero, y se intenta que el resultado de esta heurística coincida con lo que la intuición nos dice acerca de cuál elegiríamos como mejor compañero. Para el cálculo del *score* se itera sobre cada compañero del agente, calculándose para cada uno un puntaje en base a la distancia entre el receptor y la pelota[5] (d_{P_B}), la distancia entre el receptor y los dos oponentes más cercanos a él (d_{P_O1}), (d_{P_O2}) y la distancia entre el receptor y el arco (d_{P_G}).
Finalmente, el *score* del compañero i-ésimo se calcula como:

$$Score_{BP}(i) = -0{,}25 d_{P\_B} + 0{,}1(d_{P\_O1} + d_{P\_O2}) - 0{,}17 d_{P\_G} \tag{3.1}$$

Se escoge entonces como mejor compañero, al compañero que presenta el *score* mayor. Nótese que este cálculo puede ser fácilmente generalizado a más compañeros si se decide

[5]La cual es igual, recordemos, a la distancia entre el receptor y el agente

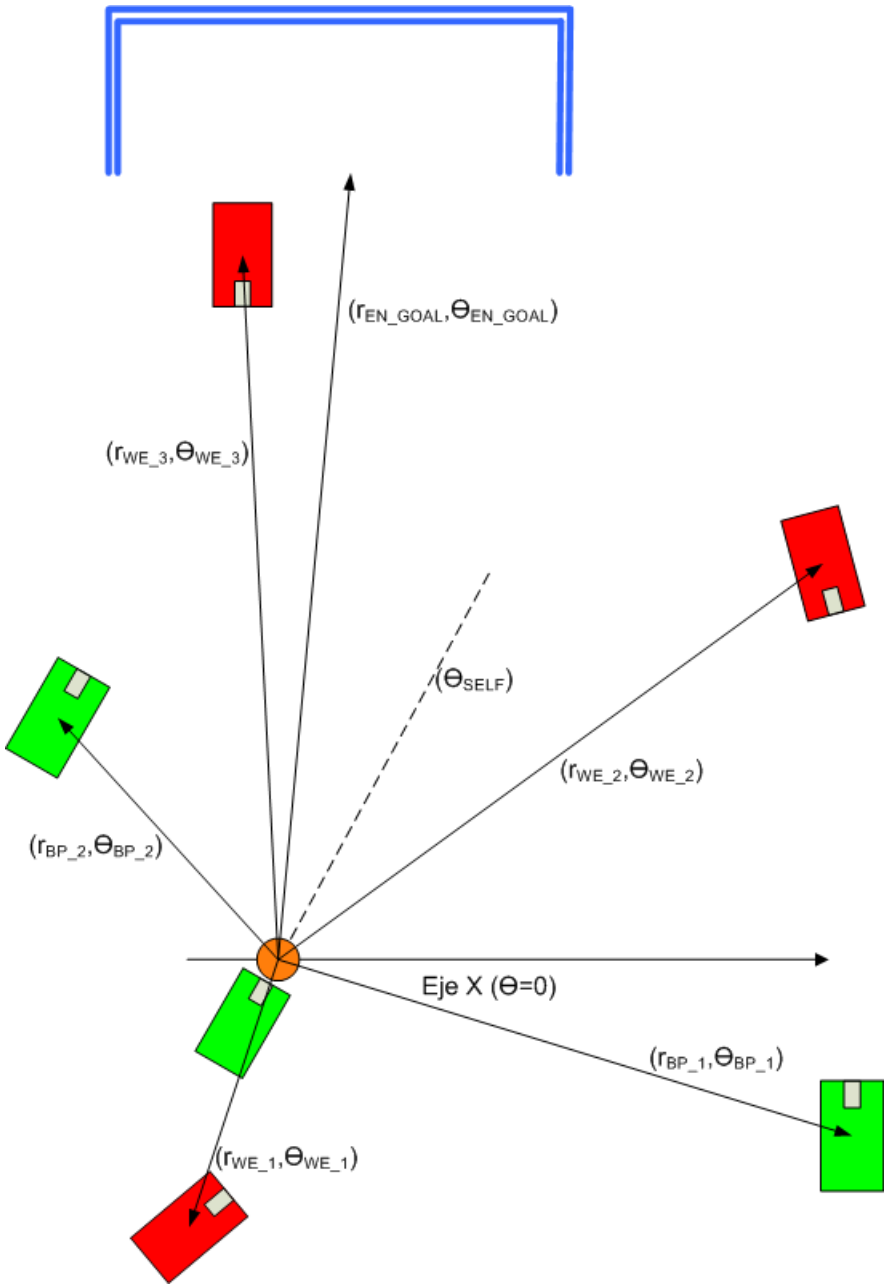

Figura 3.1: Espacio de estados para una configuración dada de jugadores, en donde se muestra sólo los mejores compañeros y peores oponentes (caso en que se escogen 2 mejores compañeros y 3 peores oponentes.).

incluir la representación de estos en el espacio de estados, mediante la simple elección de los dos mayores puntajes.

3.1.1.2. Elección de los Peores Oponentes

La elección de los peores oponentes se hace en base a tres criterios, los cuales para cada oponente generan un puntaje —o *score*. Al igual que en el caso anterior, cada criterio genera un un puntaje individual y para el cálculo del *score* global, se suman dichos puntajes. Se realiza una cálculo para cada oponente, en donde se evalúa: cuánto obstaculiza el oponente el tiro al arco (S_{1W_E}), cuánto obstaculiza el oponente al mejor compañero (S_{2W_E}) y cuánto obstaculiza el oponente el arco al mejor compañero (S_{3W_E}). Dentro del cálculo de estos puntajes está considerado además la distancia a la que se produce la obstaculización (en caso de haberla), mediante el uso de una función ponderadora dependiente de la distancia. A continuación se explica cómo se calcula cada uno de estos puntajes.

- **El oponente obstaculiza el tiro al arco**

  Para asignar un puntaje de acuerdo a este criterio se generan áreas cónicas con vértice en el jugador que tiene la pelota y un área circular alrededor del oponente[6]. Si esta área cónica contiene o está contenida dentro del área cónica del arco, se asigna un puntaje igual a 2. Si esta área cónica intersecta al área cónica formada por el arco oponente se asigna un puntaje igual a 1. Si no hay intersección, el puntaje es cero. Esto se observa más claramente en la figura 3.2. Luego, este puntaje es ponderado por la función que se observa en la figura 3.3 la cual tiene como objetivo darle mayor peso a los oponentes que se encuentren cerca, y no considerar tan peligrosos a los oponentes que se encuentren más lejos.

- **El oponente interfiere en algún sector angular al que puede llegar la pelota**. (Por simplicidad esto queda definido en función del mejor compañero ya escogido).

  Este criterio asigna puntajes de la misma forma que el anterior, con la salvedad que el área cónica de comparación pasa de estar formada por el arco oponente a estar formada

[6]Esto debido a la incerteza que hay en la posición del oponente, debido a tiempo que transcurre entre la ejecución de la decisión y el momento en que la pelota alcanza al oponente.

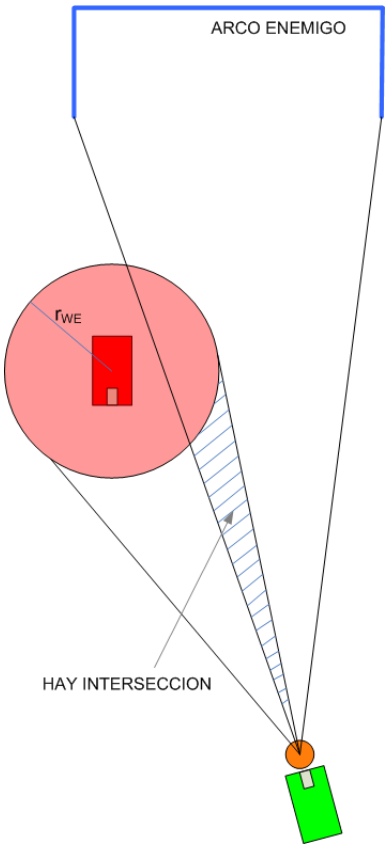

Figura 3.2: Primer criterio para la asignación del peor oponente

por el área de una circunferencia alrededor del mejor compañero. Esto se observa más claramente en la figura 3.4.

- **El oponente obstaculiza el tiro al arco a los compañeros**. (Por simplicidad esto queda definido en función del mejor compañero ya escogido).
  Este criterio es igual al primer criterio (fig. 3.2), sólo que se considera como punto de origen al mejor compañero. Al igual que en el primer caso, se multiplica por la función ponderadora mostrada en la figura 3.3.

Nótese que para cada criterio se debe definir áreas circulares de incerteza alrededor de los oponentes. El radio de estas áreas es un parámetro ajustable que permite ponerse en peores o mejores escenarios.

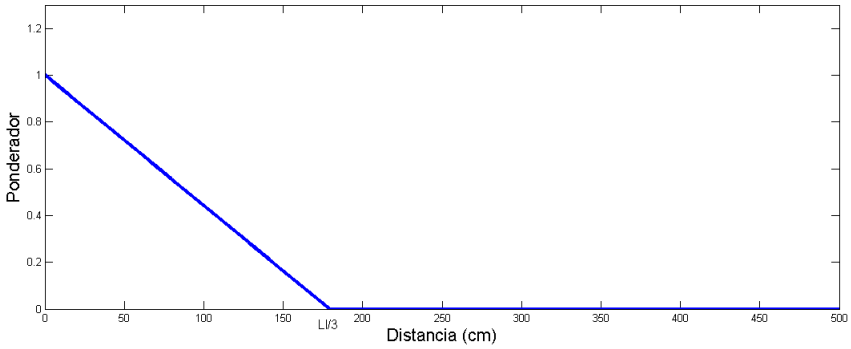

Figura 3.3: Ponderación entregada al puntaje en función de la distancia del origen a la que se encuentra el robot que obstaculiza. LI es igual a 537 cm, que corresponde al largo interno de la cancha, es decir a la distancia entre los arcos. Cualquier distancia mayor a $LI/3$ se considera con una ponderación igual a cero. Mediante el uso de esta función, se le da más peso a los oponentes que están más cercanos, pues naturalmente, estos son más peligrosos.

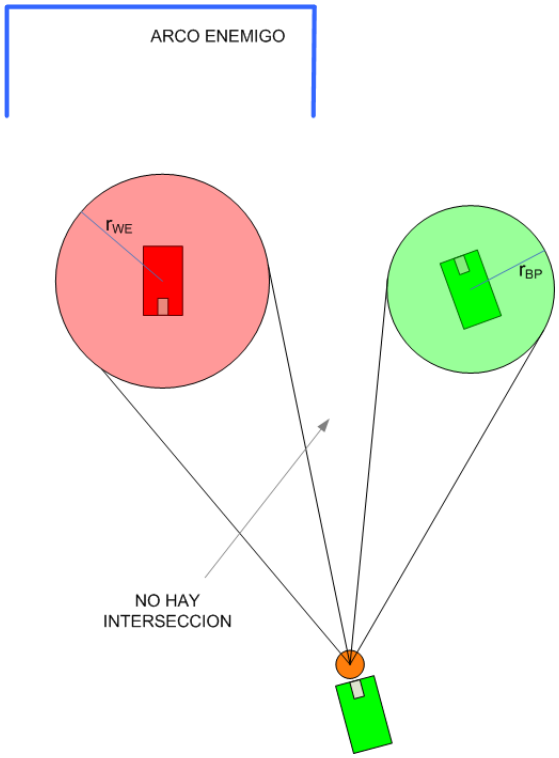

Figura 3.4: Segundo criterio para la asignación del peor oponente

3.1.2. Definición del Espacio de Acciones

Como espacio de acciones se escoge un set de golpes que puede ejecutar el robot cuando está en posesión de la pelota. Como se explicó anteriormente, la limitación del algoritmo al aprendizaje de los golpes se hace con motivos de simplicidad. Cada golpe queda representado por una media y una varianza, ambas en dos dimensiones[7]. Intuitivamente, es deseable que el espectro de posibilidades ofrecidos por los golpes sea amplio, es decir, que ofrezca un rango de distintos lugares donde puede llegar la pelota tras ejecutarse cada golpe, de modo de enriquecer el juego y las posibilidades de hacer jugadas colectivas o goles. Este requisito es fácil de entender si se toma cómo ejemplo la técnica de un jugador de fútbol: un jugador que es capaz de posicionar la pelota en distintos lugares de la cancha, está mejor calificado técnicamente que uno que no puede hacerlo, y evidentemente, es deseable para un equipo un jugador que puede ejecutar diversos golpes.

Por otra parte, el conjunto de acciones no puede ser demasiado grande, pues se debe recordar que el tiempo de aprendizaje depende de la cantidad de pares estado-acción que exista, por lo que al aumentar el número de posibles acciones, aumenta considerablemente el tiempo de entrenamiento. Es por esto que se ha decidido limitar el número de golpes a seis. En la figura 3.5 se observa una representación de estos golpes con sus medias y varianzas que determinan áreas de probabilidad en las que puede caer la pelota.

3.1.3. Definición de la Recompensa

Para elegir la recompensa, se debe tener en cuenta el objetivo a lograr con el aprendizaje. En este caso, se trata de ganar los partidos. Es natural entonces recompensar resultados, sin embargo, a modo de experimentación se hace una diferencia en la implementación de la recompensa para cada esquema. Por una parte, para el algoritmo que utiliza *Q-Learning* se decide otorgar sólo una recompensa numérica positiva cuando hay un gol a favor y una recompensa numérica negativa cuando hay un gol en contra, como se observa en la ecuación 3.2.

[7]Las dos dimensiones de la cancha, x e y

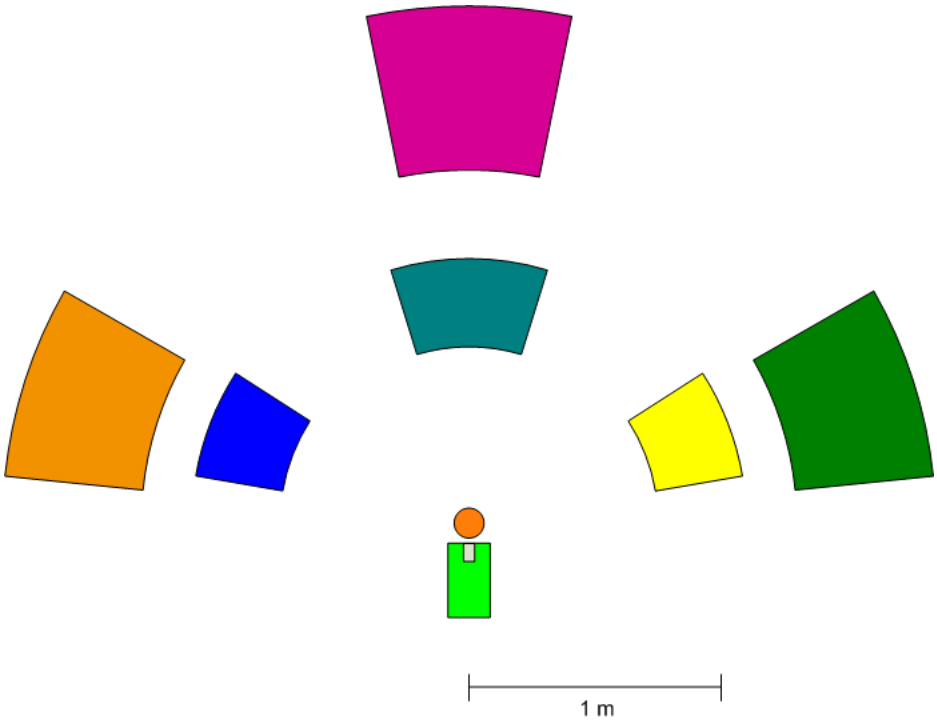

Figura 3.5: Posibles golpes de la pelota a ser ejecutados por el robot. Cada golpe es representado por un área de incerteza en la cual puede caer la pelota tras ser ejecutado el golpe. Una media determina el centro de cada área y dos varianzas determinan la dimensión de ésta

$$R_t = \begin{cases} +1000, & \text{si en } t \text{ hay gol a favor} \\ -1000, & \text{si en } t \text{ hay gol en contra} \end{cases} \tag{3.2}$$

Si bien la definición de la recompensa podría complejizarse mucho más —considerando otros casos como cuando la pelota sale del campo o se da un mal pase, se cree que esto no es necesario e incluso podría llegar a ser perjudicial para el aprendizaje, debido a que se estaría imponiendo preconcepciones que no necesariamente tienen que ver con el objetivo deseado.

Por otra parte, para el algoritmo basado en *SARSA* se ocupa como referencia lo realizado por Peter Stone para solucionar el *Keepaway* [9], que consiste en otorgar recompensas por tiempos de retención de la pelota. Esta solución se adapta al caso que aquí se trata de resolver. Es por esto que se decide agregar recompensas por pases bien realizados, quedando la función

de recompensas como se observa en la ecuación 3.3.

$$R_t = \begin{cases} +1000, & \text{si en } t \text{ hay gol a favor} \\ -1000, & \text{si en } t \text{ hay gol en contra} \\ t_i - t_{i-1} & \text{si en } t_i \text{ se mantiene la posesión de la pelota con respecto a } t_{i-1} \end{cases} \tag{3.3}$$

La idea detrás de esto es que el equipo mantenga posesión de la pelota, por lo que la recompensa se hace igual a la diferencia de tiempo (en milisegundos) que existe entre la posesión ininterrumpida de la pelota por jugadores del mismo equipo. De esta forma, no sólo se está incentivando que se hagan goles, sino que además se incita el juego colectivo.

La definición anterior presenta una grave falencia: a simple vista se puede observar que mantener la posesión de la pelota por un segundo tendría la misma recompensa que hacer un gol, por lo que se decide modificar la definición de la ecuación 3.3 de modo de dejar absolutamente claro la importancia que se le está dando a cada recompensa. Se decide entonces en una segunda etapa, modificar la función de recompensas por la que se observa en la ecuación 3.4. Por supuesto, se fija una recompensa mucho menor para los pases realizados que para cuando hay un gol, a modo de incentivar que el juego sea orientado hacia el arco oponente con el objetivo último de ganar el partido.

$$R_t = \begin{cases} +1000, & \text{si en } t \text{ hay gol a favor} \\ -1000, & \text{si en } t \text{ hay gol en contra} \\ 10, & \text{si en } t_i \text{ se mantiene la posesión de la pelota con respecto a } t_{i-1} \end{cases} \tag{3.4}$$

Además, a diferencia de lo hecho con *Q-Learning*, en donde las diferencias temporales que habían era entre un paso y el paso anterior, se implementa una propagación de las recompensas en el tiempo. Esto se logra mediante el uso de un vector de trazos de elegibilidad, que almacena y pondera por un factor λ la información de los pares estado-acción visitados en el tiempo. Esto se describió con más detalle en la sección 2.2.2.2.

3.2. Implementación de la Solución

En esta sección se explica cómo el modelo presentado en la sección 3.1 es llevado a la práctica. Para esto, se presenta el pseudocódigo utilizado para cada acercamiento. Posteriormente, en las secciones 3.2.1 y 3.2.2, se incluye un detalle de las funciones y clases más importantes que fueron implementadas y utilizadas para los algoritmos de aprendizaje reforzado y para *tile coding*, respectivamente.

En la figura 3.6 se puede observar las etapas más notorias de la implementación del algoritmo que utiliza *Q-Learning*.

Para el comienzo del episodio:
1. inicializarPesos($\vec{w}$)
2. s = calcularEstadoActual()
3. $a = \arg\max_{a'} Q(s, a', \vec{w})$, con prob. $(1 - \epsilon)$ o acción aleatoria, con prob. ϵ

En cada paso temporal:
4. $s_{Anterior} = s$
5. s = calcularEstadoActual()
6. r = obtenerRecompensa()
7. si hay gol:
8. $\quad Q(s_{Anterior}, a, \vec{w}) = (1 - \alpha)Q(s_{Anterior}, a, \vec{w}) + \alpha \cdot (r)$
9. en caso contrario:
10. $\quad Q(s_{Anterior}, a, \vec{w}) = (1 - \alpha)Q(s_{Anterior}, a, \vec{w}) + \alpha(r + \gamma \max_a Q(s, a, \vec{w}))$
11. actualizarPesos($\vec{w}$, $Q(s_{Anterior}, a, \vec{w})$)
12. a = $\arg\max_{a'} Q(s, a', \vec{w})$, con prob. $(1 - \epsilon)$ o acción aleatoria, con prob. ϵ
13. volver al paso 4.

Figura 3.6: Esquema general de la implementación de *Q-Learning*. Notar que la función Q se ha definido para el par (s, a) y además para el vector que almacena los pesos, w. Esto es necesario pues para poder calcular el valor de un estado cualquiera, se debe calcular los *tiles* activados por éste, y luego realizar una suma de los pesos correspondientes a dichos *tiles*.

En la figura 3.7 se muestra las etapas más notorias de la implementación del algoritmo que utiliza *SARSA*. Se puede observar diferencias sustanciales con el algoritmo presentado en la figura 3.6. En primer lugar, se utiliza el vector $\vec{e}$ para llevar un registro de los trazos de elegbilidad. Esto se traduce en que cuando se hace la actualización de los pesos (linea 20, fig.

3.7), se hace para todo el vector, pero sólo tiene efecto en los pesos que están "marcados"por $\vec{e}$. Además, los trazos de elegibilidad deben ponderarse en el tiempo[8] (línea 22) y reemplazarse en caso de que se visite un estado ya marcado (líneas 23–27).

El esquema presentado para *SARSA* corresponde al del enfoque presentado en la ecuación 3.3. Para obtener el enfoque de la ecuación 3.4, solo se debe modificar el cálculo de la recompensa por un número fijo.

3.2.1. Clases y Funciones Bases

A continuación se describe las principales clases y funciones implementadas para el funcionamiento de los algoritmos desarrollados en el presente trabajo. Como se verá, algunas clases fueron creadas exclusivamente para este proyecto, mientras que otras clases ya eran parte del simulador, caso en que sólo se les agregó las funciones necesarias para el proyecto.

Por motivos de claridad, las descripciones realizadas en esta sección tienen simplificaciones respecto del código original. Para mayor detalle, consultar el anexo C , en donde puede encontrarse el código completo correspondiente a estas y otras clases.

- **Clase Q_Strategy**

  Es la clase que implementa todas las funciones y variables de *Q-Learning* y de *SARSA*. Cada robot llama a una de estas clases al momento de su creación, e invoca a sus funciones cada vez que ejecuta un paso temporal del algoritmo, es decir, cuando está en posesión de la pelota. Algunas de las funciones más importantes implementadas en esta clase son:

  Q_Strategy::selectQObjective(): Escoge y configura la acción a ejecutar de acuerdo a la política definida.

  Q_Strategy::get_state(): Es responsable de calcular el estado actual para el robot que la llama.

[8]Para dar mayor importancia a los eventos más recientes

Para el comienzo del episodio
1. inicializarPesos($\vec{w}$)
2. s = calcularEstadoActual()
3. para todo $a \in A$ (espacio de acciones):
4. $\quad Fa$ = set de *tiles* activados por (s, a)
5. $\quad Qa = \sum_{i \in Fa} w(i, a)$
6. $a = \arg \max_{a'} Q(s, a', \vec{w})$, con prob. $(1 - \epsilon)$ o acción aleatoria, con prob. ϵ
7. $tiempoAccion = tiempoActual()$
8. $\vec{e} = \vec{0}$
9. para todo $i \in Fa$:
10. $e(i) = 1$

En cada paso temporal 11. $r = tiempoActual() - tiempoAccion$
12. $\delta = r - Q(s, a, \vec{w})$
13. s = calcularEstadoActual()
14. para todo $a \in A$:
15. $\quad Fa$ = set de *tiles* activados por (s, a)
16. $\quad Qa = \sum_{i \in Fa} w(i, a)$
17. $a = \arg \max_{a'} Q(s, a', \vec{w})$, con prob. $(1 - \epsilon)$ o acción aleatoria, con prob. ϵ
18. $tiempoAccion = tiempoActual()$
19. $\delta = \delta + \gamma Q(s, a, \vec{w})$
20. $\vec{w} = \vec{w} + \alpha \delta \vec{e}$
21. $Qa = \sum_{i \in Fa} w(i, a)$
22. $\vec{e} = \lambda \vec{e}$
23. para todo $a' \in A$ s.a. $a \neq a$:
24. $\quad$ para todo $i \in Fa'$:
25. $\quad\quad e(i) = 0$
26. $\quad$ para todo $i \in Fa$:
27. $e(i) = 1$

Para el final del episodio
28. si hay gol:
29. $\quad r = 1000$, si es a favor ó $r = -1000$, si es en contra
30. en caso contrario (la toma un oponente):
31. $\quad r = tiempoActual() - tiempoAccion$
32. $\delta = r - Qa$
33. $\vec{w} = \vec{w} + \alpha \delta \vec{e}$

Figura 3.7: Esquema general de la implementación de *SARSA*.

Q_Strategy::refreshTable(): Función implementada para *Q-Learning*, se encarga de llamar a la función que actualiza la tabla (esto ocurre al final de cada instante de tiempo y al final de cada episodio).

Q_Strategy::set_Reward(int r): Le dice al jugador la recompensa del estado actual[9], fijando la variable `reward` de la clase `Q_Strategy` como `reward = r`, para que sea posteriormente leída al momento de actualizar la función de valor .

Q_Strategy::get_best_partners(bool m_comp[]): Calcula los mejores compañeros para el robot que llama a esta función y llena el arreglo de booleanos `m_comp[]` con verdadero o falso, dependiendo de si el índice del arreglo corresponde o no a uno de los mejores compañeros calculados.

Q_Strategy::get_worse_enemies(bool p_opon[]): Calcula los peores oponentes para el robot que llama a esta función y llena el arreglo de booleanos `p_opon[]` con verdadero o falso, dependiendo de si el índice del arreglo corresponde o no a uno de los peores oponentes calculados.

- **Clases B_go_to_ball_and_kick y B_ball_kick**

  Ambas clases son parte del código de los robots, y es usada por todas las estrategias. Son parte de un esquema jerarquizado de comportamiento para cada robot, en donde existen diversos estados y transiciones entre éstos.

  Para poder utilizar estas clases con *Q-Learning* y *SARSA*, se agregó una nueva estrategia. Esta estrategia, es llamada sólo cuando el robot esta en posesión de la pelota, por lo que se procura intervenir estas clases de modo que los algoritmos implementados se llamen adecuadamente. Las principales funciones son:

  B_go_to_ball_and_kick::selectKickObjective(): Esta función es llamada cuando el robot está en posesión de la pelota y debe decidir qué golpe realizar. Es entonces cuando debe ser llamada la función que calcula el estado, la función que decide la

[9]Esto se hace al revés de lo conceptual —en donde el agente reconoce su recompensa del estado— por motivos de simplicidad, debido a que el simulador maneja el ambiente en el cual se mueve el robot.

próxima acción a ejecutar y la función que actualiza la tabla. Este procedimiento varía un poco entre la implementación de *Q-Learning* y la implementación de *SARSA*: en primer lugar, una diferencia de forma, es que en *SARSA* la mayor parte del algoritmo se implementa en esta función, mientras que en *Q-Learning* se llama a funciones de la clase `Q_Strategy` para que hagan el trabajo. Lo segundo, es que como se mencionó, para *SARSA* un episodio termina cuando hay gol (al igual que en *Q-Learning*), pero además cuando la pelota entra en posesión de un oponente. La forma más simple de implementar esto último es incluyéndolo en esta función, para el caso en que se llame a la estrategia del oponente[10].

Por último, esta función es utilizada para realizar un entrenamiento previo basado en otra estrategia. Para esto, se incluye en el código de la estrategia de la cual se desea aprender, las funciones que calculen el estado, y actualicen la tabla. De esta forma, al existir en la tabla información con cierto grado de coherencia, se puede lograr un entrenamiento mucho más rápido que el tiempo que tomaría un entrenamiento "desde cero".

B_ball_kick::evaluateState(): Esta función es la que mediante una máquina de estados se encarga de realizar algunas de las transiciones entre los distintos comportamientos que puede tener el robot (ej: Buscar la pelota → Ir hacia la pelota → Patear la pelota → ...). Para poder implementar los algoritmos de este trabajo, fue necesario modificar las posibles transiciones a ocurrir, eliminando la posibilidad de driblar o regatear. Esto se hizo para limitar las acciones posibles y por ende, la dimensionalidad del problema.

- **Clase Actions**

  En esta clase se hace el mapeo entre la acción escogida por el robot y las seis acciones posibles definidas en el modelo.

  Actions::selectKick(): Permite escoger un golpe de acuerdo a la estrategia que se

[10]Se debe recordar que esta función es llamada cuando el robot esta en posesión de la pelota y debe decidir qué golpe realizar, por lo que si el oponente llama a esta función, significa que es el fin de un episodio.

esté utilizando. En el caso de los algoritmos implementados, simplemente lee la acción fijada por `Q_Strategy::selectQObjective()`. Además, para la etapa antes mencionada de entrenamiento utilizando otra estrategia, es necesario pasar la acción seleccionada a la clase `Q_Strategy`.

- **Clases Engine y B_BehaviorEAN**

  En estas clases se revisa la condición terminal de cuando hay gol. Ambas clases son ejecutadas una por cada robot. Dentro de las siguientes funciones se llama a las funciones de *Q-Learning* o *SARSA* correspondientes al fin del episodio. (Se utiliza un indicador de modo de que estas funciones sean llamadas una sola vez por cada evento.)

  Engine::UpdateGameControl(): Se revisa si hay gol y a qué equipo favorece. En función de esto, se fija la recompensa en 1000 o en -1000 según corresponda, utilizando la función `Q_Strategy::set_Reward(int r)`. Además, en caso de que se trate de un gol en contra, se llama a la función que actualiza la tabla.

  B_BehaviorEAN::selectBehavior(): Si se trata de un gol a favor, aquí es donde se llama a la función que actualiza a la tabla.

- **Otras Funciones**

  Las siguientes funciones no son parte de una clase, y fueron implementadas para integrar las librerías de *Tile Coding* con la aproximación de la función de valor. Para esto, se creó el archivo "TileCoding.cpp", en el que se encuentra las siguientes funciones:

  Q(float s[],int a, float weights[][]): Calcula el valor de Q para un par (s, a) y un arreglo de pesos `weigths[][]`.

  init_weights(float weights[][]): Inicializa los pesos en cero mediante la creación de un archivo de texto.

  save_weights(float weights[][]): Guarda los pesos en el archivo de texto. Esta función es llamada al final de cada episodio.

fIndex(float e[][], float s[], int a, float v): Calcula los *tiles* —o índices— activados por el par (s, a) y llena el vector e[][] (vector de elegibilidad) para dichos índices con el valor v. Esta función sólo se utiliza para la implementación de *SARSA*, debido a que sólo en este caso se implementan trazos de eligibilidad.

refreshTiles(float $\mathbf{s}_t$[], int a, float $\mathbf{s}_{t-1}$[], int $\mathbf{r}_t$, float weights[][]): Se encarga de actualizar la tabla con la nueva estimación de $Q(s, a)$ y el máximo valor de Q en el instante siguiente.

3.2.2. Integración con *Tile Coding*

La implementación de *Tile Coding* se basa en el uso de la función GetTilesWrap() provista por Richard S. Sutton, de la Universidad de Alberta[11].

Estas funciones generan particiones rectangulares y uniformemente espaciadas sobre los estados, por lo cual para generar otro tipo de particiones se debe escalar las variables de estado de una forma conveniente, o bien, redefinirlas[12].

Además, las funciones traen *hashing* incorporado, y dado cierto estado, retornan los índices de los *tiles* activados por dicho estado. Su funcionamiento puede explicarse revisando el encabezado:

```
void GetTilesWrap(
   int tiles[],
   int num_tilings,
   int memory_size,
   float floats[],
   int num_floats,
   int wrap_widths[],
   int ints[],
```

[11] http://www.cs.ualberta.ca/~sutton/tiles2.html

[12] Por ejemplo, si las variables de estado son X e Y, se puede cambiar esto por $X - Y$ y $X + Y$ para producir una rotación.

`int num_ints)`

Dado un cierto estado, la función `GetTilesWrap()` se encarga de calcular qué *tiles* son activados por éste. Para esto, las variables de estado son pasadas a la función mediante el arreglo `floats[]` —que debe tener la dimensionalidad del estado, y los índices de los *tiles* activados son llenados en el arreglo `tiles[]`. La variable `num_tilings` corresponde al número total de particiones a realizar y se define como 32. La variable `memory_size` corresponde al tamaño del arreglo de pesos y se define como 51200[13]. A medida que el parámetro `memory_size` sea mayor, menores colisiones existirán debido al *hashing*.

Además, esta función permite la generalización en conjuntos acotados, por ejemplo, intervalos angulares en donde es deseable que ejemplos presentados cercanos a 2π, generalicen a los *tiles* cercanos a cero. Para esto, se debe pasar a la función el ancho del intervalo de cada dimensión mediante el arreglo `wrap_widths[]`.

Para implementar estas funciones adecuadamente, en primer lugar, se debe definir cómo se particiona el espacio de estados descrito en la sección 3.1.1. A continuación se describen dichas particiones.

En un principio se decide utilizar 32 *tilings* que particionen las variables cada un décimo (de 2π) para los ángulos y un quinto (de la diagonal de la cancha) para las distancias. En términos sencillos, esto significa que las distancias pueden tomar 5 valores posibles y los ángulos 10.

Para el caso con 2 mejores compañeros y 3 peores oponentes, se tiene un total de 13 variables de estado (6 angulares y 7 de distancia). Esto corresponde a un espacio de estados de tamaño:

$$5^6 \cdot 10^7 = 1,5625 \cdot 10^{11}$$

en donde la mínima distancia que puede ser representada es $d_{max}/(5 \cdot 32)$[14] y el mínimo

[13]Notar que este número es considerablemente menor al tamaño total del espacio de estados que se calcula más adelante, por lo que existe un ahorro significativo en memoria.

[14]d_{max} corresponde a la máxima distancia que puede haber entre dos elementos en la cancha, y es igual a la diagonal de la cancha que mide 533 cm.

ángulo, $2\pi(10 \cdot 32)$. Posteriormente, se reduce el tamaño del espacio de estados cambiando el tamaño de los *tiles*. Para los ángulos, se deja en un octavo (de 2π) y para la distancia en un cuarto (de la diagonal de la cancha). En este caso, el espacio de estados queda de tamaño:

$$4^6 \cdot 8^7 = 8,5899 \cdot 10^9$$

la mínima distancia que puede ser representada es $d_{max}/(4 \cdot 32)$ y el mínimo ángulo es $2\pi(8 \cdot 32)$.

Para implementar estos esquemas, se debe multiplicar las variables que se pasan a la función `GetTilesWrap()` por un factor que corresponda a la generalización deseada. Por ejemplo, si se tiene una variable que puede tomar valores entre 0 y 1, y se desea particionarla en *tiles* de tamaño 0,25, se deberá multiplicar la variable por un factor igual a 4. A continuación se muestra como se hace esto en términos de código, para una variable angular y una de distancia correspondientes al valor del estado del segundo peor oponente[15]:

```
float_array[DIST_WE_2] = estado[DIST_WE_2]/(D_MAX*0.25f);
float_array[ANG_WE_2] = estado[ANG_WE_2]/(2*3.14159f*0.125f);
```

Se puede observar un resumen del sistema implementado (Aprendizaje Reforzado con *Tile Coding*) en el diagrama de la figura 3.8.

[15]Este procedimiento es idéntico para todo el resto de las variables angulares y de distancia definidas para el espacio de estados.

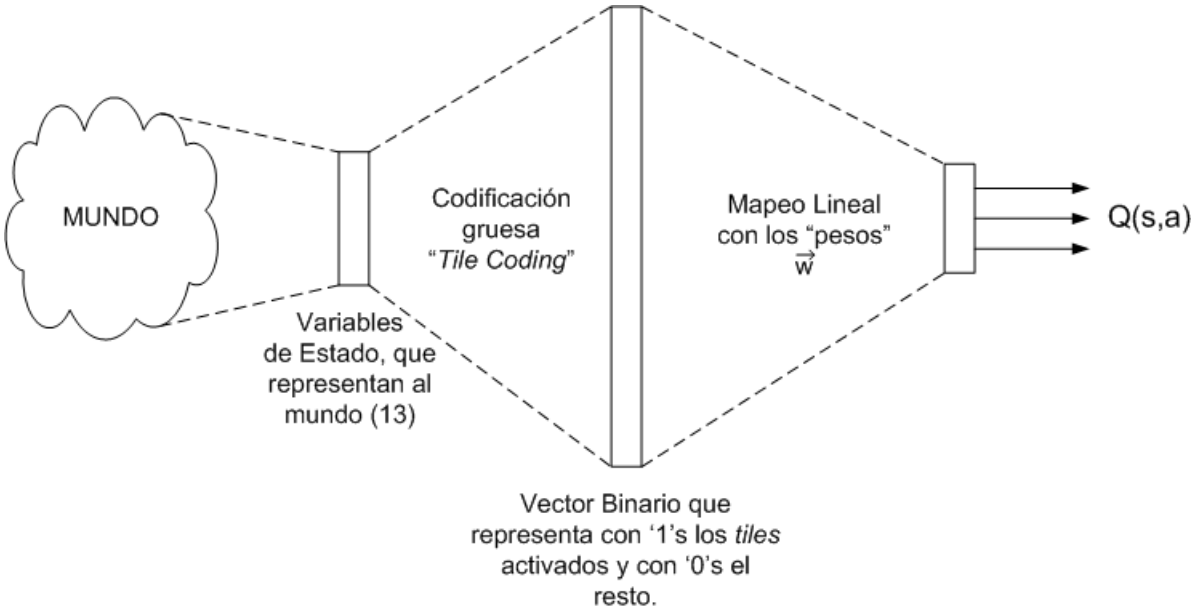

Figura 3.8: Diagrama resumen de la representación utilizada. El espacio de estados se representa sólo por algunas variables, que luego se particionan en características binarias para por último, ser combinadas y obtener el valor de $Q(s, a)$.

Capítulo 4

Análisis de los Resultados

En este capítulo se observa los resultados obtenidos para la solución propuesta. En primer lugar, se analiza brevemente el comportamiento del algoritmo que escoge a los mejores compañeros y a los peores oponentes. Posteriormente, se muestra y analiza los resultados obtenidos para las dos alternativas de aprendizaje reforzado (TD) implementadas: *Q-Learning* y *SARSA*. Finalmente se realiza una comparación de los resultados obtenidos con ambos métodos.

4.1. Algoritmo de Selección de Mejores y Peores compañeros

En esta sección se presenta los resultados de los algoritmos descritos en las secciones 3.1.1.1 y 3.1.1.2, que corresponden a la elección de los mejores compañeros y la elección de los peores oponentes respectivamente.

Se realiza una serie de pruebas que tienen como objetivo comprobar que la elección de los mejores compañeros y peores oponentes que se realiza, está en concordancia con las reglas que se definieron. Además, se busca comprobar que esta elección coincide –al menos en la mayoría de los casos– con lo que la intuición dice. Cada prueba consiste en el análisis de una serie de situaciones particulares, con pequeñas modificaciones, en donde los compañeros y oponentes escogidos se representan por medio de líneas que los unen con el agente.

Primero, se realizan pruebas donde el jugador que está en posesión de la pelota se en-

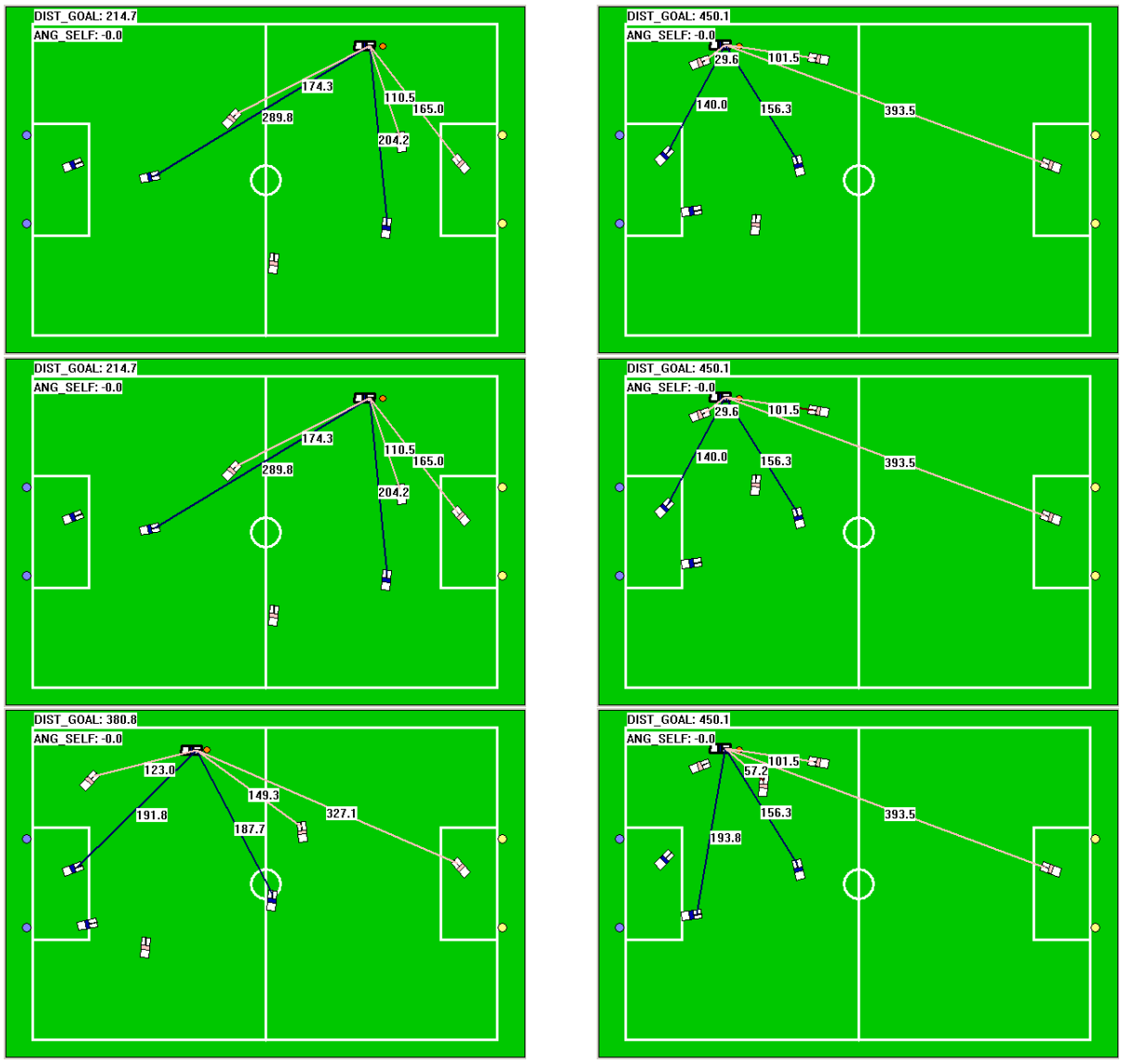

Figura 4.1: Distintas configuraciones para una situación que transcurre por el lado izquierdo de la cancha. Los mejores compañeros y peores oponentes escogidos por el algoritmo se muestran mediante líneas que los unen con el jugador que toma la decisión. Además, asociadas a estas líneas se muestra algunas de las variables de estado, de modo de comprobar que el estado se esté llenando correctamente.

cuentra en un costado de la cancha, a distintas distancias del arco oponente y a distintas distancias de los demás jugadores. Esto se puede observar en la figura 4.1, en donde el resultado de estas pruebas es completamente coherente con como se pensó la heurística (y además coincide con la intuición acerca de qué jugadores se deberían escoger). Desde el punto de vista

de la elección de los mejores compañeros (que es la primera que se hace en el algoritmo), en los 6 casos presentados se puede observar cómo la elección de éstos se ve reflejada por las variables "distancia entre el compañero y el agente", "distancia a los dos oponentes mas cercanos"y "distancia al arco". Por ejemplo, en las dos imágenes inferiores de la figura 4.1 se aprecia cómo afecta la posición de un oponente en la elección de los mejores compañeros. Desde el punto de vista de la elección de los peores oponentes, se debe recordar que esta depende tanto de la obstaculización del arco al agente, como la interferencia que tengan con los mejores compañeros escogidos. Observando la figura, se aprecia la relevancia que tiene estas variables, y el correcto funcionamiento del modelo. Por ejemplo, en las dos imágenes inferiores del lado izquierdo de la figura 4.1 se puede apreciar como sólo variando la posición de un oponente, se incluye o no éste en la representación del estado.

En la figura 4.2 se muestra otro tipo de situación, la que transcurre en el centro de la cancha. Se representan distintas configuraciones con variaciones pequeñas, de modo de observar cómo va cambiando la representación del estado en función de estas variaciones. Del mismo modo que lo que ocurre en la figura anterior, se está conforme con la elección de mejores compañeros y peores oponentes por parte del método.

Por último, en la figura 4.3 se muestra una serie de casos "límite", en donde la elección de los mejores compañeros y peores oponentes no resulta completamente intuitiva, pero al analizarlo desde el punto de vista de las reglas decididas, se comprueba que es la representación correcta. Por ejemplo, en tres de las imágenes pertenecientes a las figura 4.3, se da el caso que hay un oponente que si bien esta bastante cercano al agente, no es representado en el estado. Esto se debe a que no obstruye el arco oponente ni interfiere con los mejores compañeros, por lo que esta elección es correcta.

En particular, los parámetros que pueden calibrarse son los respectivos radios asignados a los compañeros y oponentes, y la forma de función ponderadora de la figura 3.3. Además, se puede realizar pruebas tanto para 2 mejores compañeros y 3 peores oponentes, como para 1 mejor compañero y 2 peores oponentes.

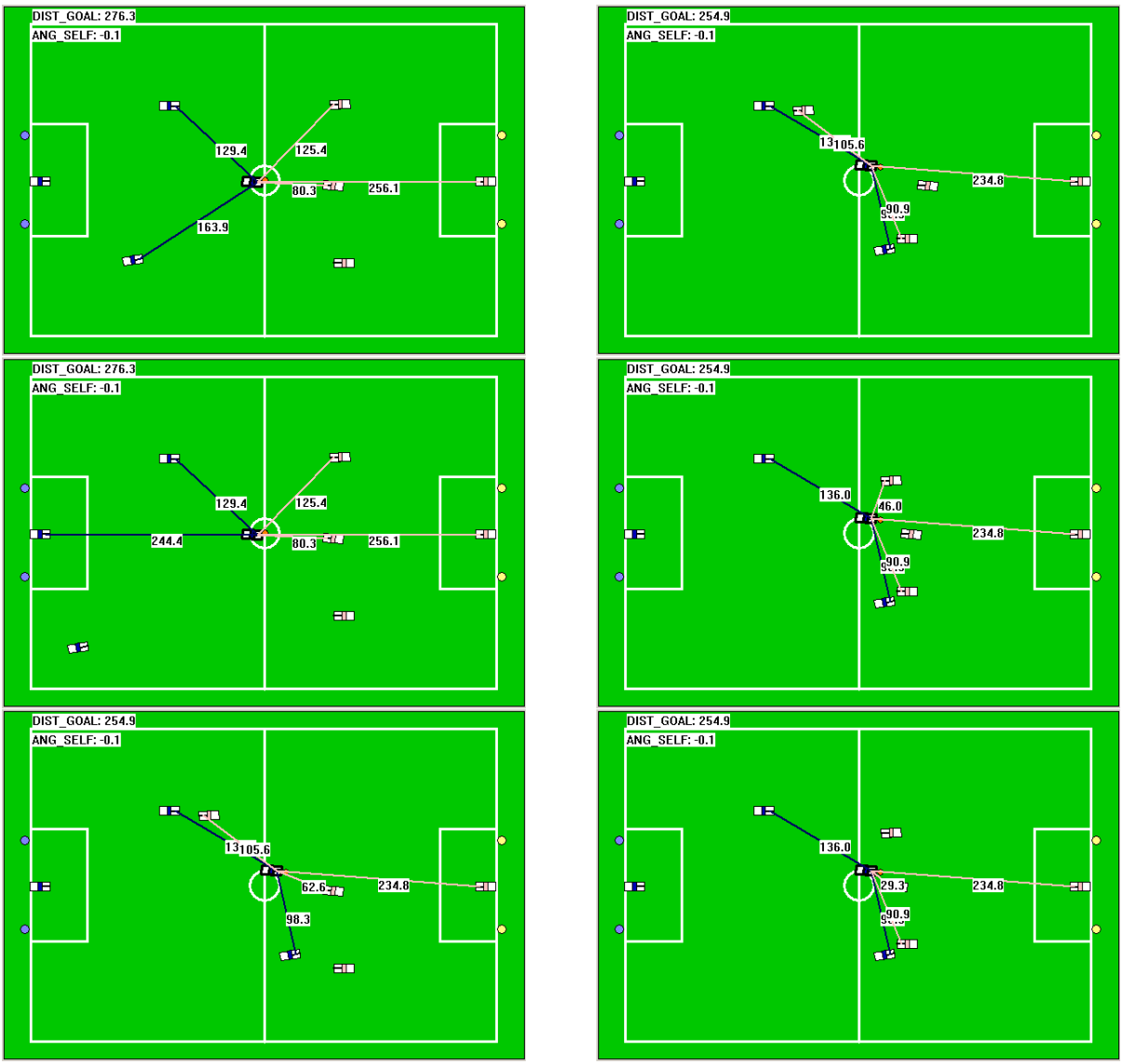

Figura 4.2: Distintas configuraciones para una situación en donde el jugador está en el centro de la cancha. Si se observa al jugador del equipo oponente situado en el lado superior derecho de la cancha, se aprecia que sólo es representado cuando queda en una posición en la que interfiere el arco al agente.

4.2. Resultados Obtenidos con *Q-Learning*

La primera prueba se realiza para asegurarse de que el sistema está actuando de forma coherente. Ésta consiste en una situación en la que hay sólo dos jugadores del mismo equipo. Se escoge con dos jugadores, pues de la manera en que está implementado el simulador,

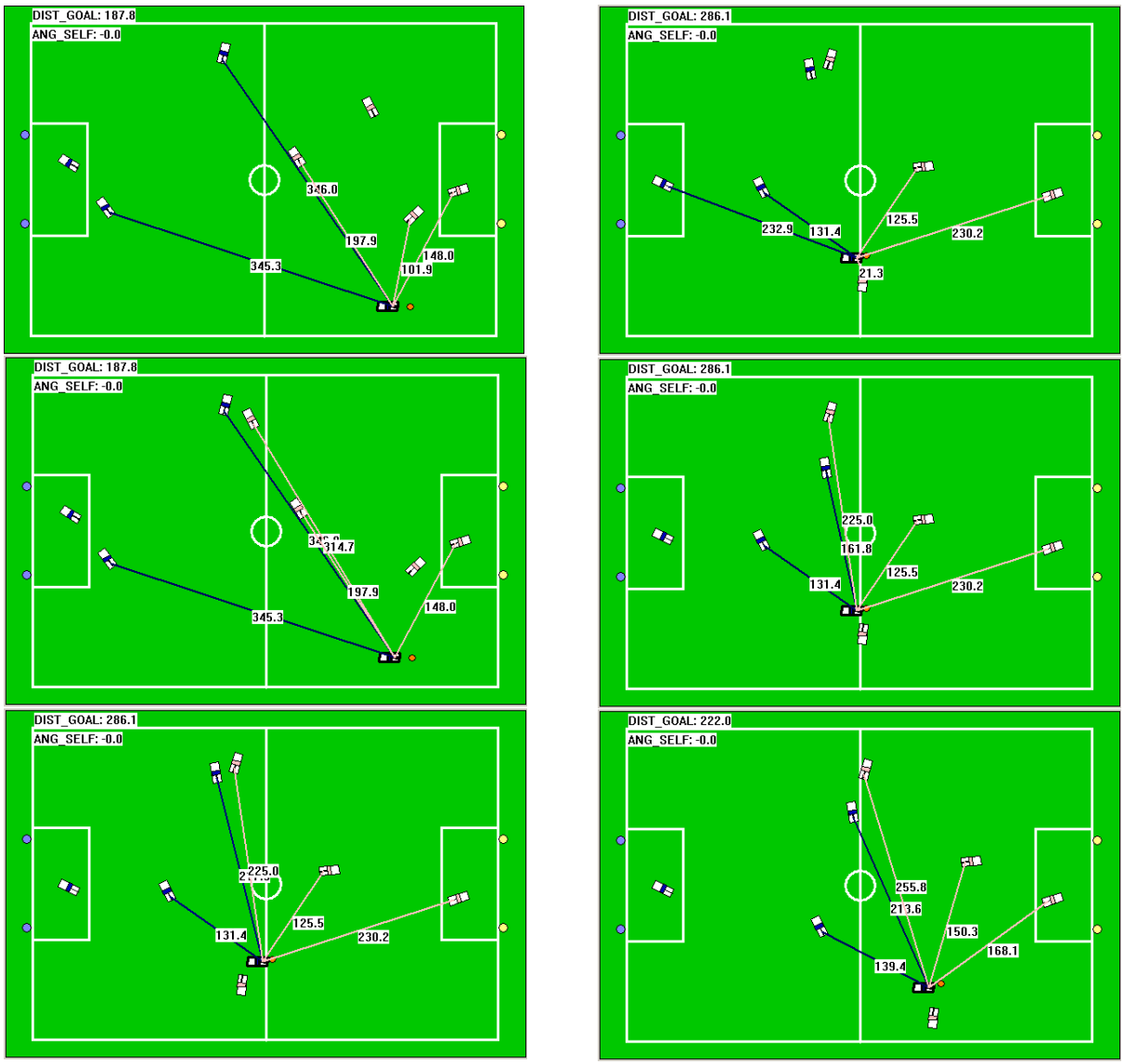

Figura 4.3: Casos en donde la elección de los mejores compañeros y peores oponentes no resulta tan intuitiva a primera vista, pero que al revisarlos se comprueba que corresponden con las reglas diseñadas y tienen sentido.

el primer jugador que se agregue va a asumir el rol de arquero y permanecerá cerca de su área. El segundo jugador asumirá entonces el rol de jugador de campo, o en otras palabras, tendrá movilidad por toda la cancha. Esto permite una prueba real del algoritmo que no sería posible jugando sólo con un arquero.

En un principio, la tabla Q está vacía, y los jugadores comienzan a experimentar el mundo

sin información previa. La política a seguir es una política ϵ-*greedy*[1], como la que se muestra en la ecuación 4.1. Como la tabla está vacía, los primeros partidos son jugados con un ϵ grande, es decir, prácticamente sólo con decisiones al azar. A medida que pasa el tiempo ϵ se va disminuyendo, dejando que la política sea cada vez más ambiciosa —o *greedy*.

$$\pi(s) = \begin{cases} arg\,\text{máx}_a\, Q^\pi(s,a), & \text{con probabilidad } 1-\epsilon \\ acción\ aleatoria, & \text{con probabilidad } \epsilon \end{cases} \tag{4.1}$$

Se juegan partidos de 10 minutos simulados, en los que se registra los goles a favor y los goles en contra (en este caso autogoles). Además, se registra una estadística de las acciones realizadas, a modo de asegurarse que los robots están jugando con la política deseada.

Es importante mencionar que para jugar un partido de 10 minutos simulados, el tiempo real requerido es de 23 minutos y 30 segundos, aproximadamente. Además, la simulación es "semiasistida" pues debe reiniciarse después de 3 partidos jugados debido a que el simulador presenta un problema de memoria. Esto implica que se hace difícil dejar el sistema entrenando por períodos largos de tiempo de manera desasistida.

En el gráfico de la figura 4.4 se observa la diferencia de goles a medida que los jugadores van ganado experiencia, para una situación en la que sólo el jugador de campo —y no el arquero— utiliza *Q-Learning*, respectivamente. Esta diferencia se hace porque en un principio se pensó que los intereses del arquero podrían entrar en conflicto con los del otro jugador debido a que en general el arquero tiene un comportamiento distinto que el resto de los jugadores. De este modo, al realizar el experimento con un sólo jugador utilizando *Q-Learning*, se está observando la configuración más simple posible, en la que se pueden detectar anomalías fácilmente.

Después de algunas configuraciones (ajuste de los *tiles*, de las recompensas), se logra contar con entrenamientos que demuestran que el sistema está funcionando. Como se observa en la figuras 4.4, en un principio existen autogoles y la diferencia está centrada en torno a cero, lo cual es lógico, debido a que el robot está jugando con una conducta completamente aleatoria.

[1]En inglés, *greed* se refiere a codicia o ambición. De ahí el nombre de la política, en donde la variable ϵ define cuán ambicioso es el robot en su juego.

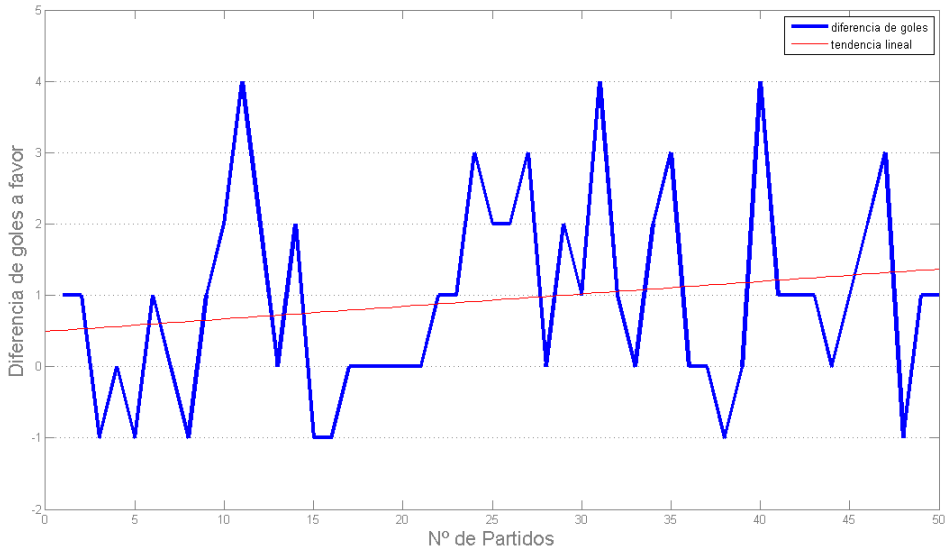

Figura 4.4: Diferencia de goles obtenida en una cancha con sólo 2 jugadores del mismo equipo. Se jugó un total de 50 partidos, lo que es equivalente a 500 minutos, tiempo suficiente para observar una tendencia creciente.

A medida que el robot lleva más partidos jugados, la diferencia comienza a ser positiva, lo que quiere decir que el robot aprende a hacer goles en el arco contrario y a no hacerse autogoles.

Posteriormente, se procedió a probar el esquema completo de 4 contra 4 jugadores, tanto para el caso de 1 mejor compañero y 2 peores oponentes, como para el caso con 2 mejores compañeros y 3 peores oponentes. Para los jugadores de campo del equipo sobre el que se prueba el algoritmo, se selecciona, naturalmente, la estrategia de *Q-Learning*. Para el arquero, y los jugadores del otro equipo, se selecciona una estrategia estándar, conocida como *Objetivo Generalizado*.

Se realiza un primer experimento para el caso con 2 mejores compañeros y 3 peores oponentes. A medida que se juegan partidos y los robots van ganando experiencia, se fija un ϵ cada vez menor, de modo que los jugadores adopten una política cada vez más *greedy* y se pueda comprobar si el método esta funcionando. Los resultados se observan en la figura 4.5, en donde no se logra entrar en un escenario favorable, sino que por el contrario, la diferencia se hace más negativa a medida que se juegan más partidos.

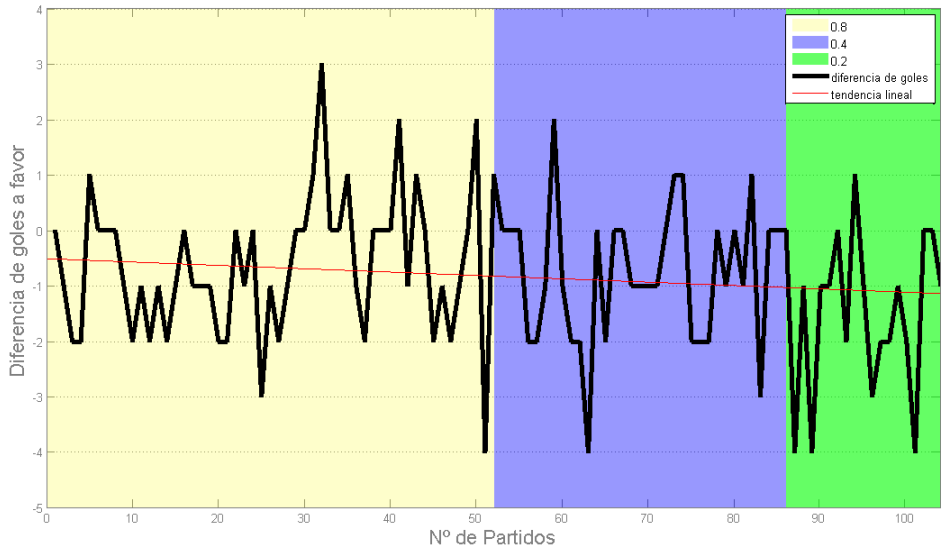

Figura 4.5: Diferencia de goles obtenida para el primer experimento realizado en un partido de 4 contra 4 jugadores. Se jugó un total de 104 partidos, lo que equivale a 1040 minutos o a 17 horas con 20 minutos de juego simulado. La zona denotada por la franja de la izquierda fue entrenada con $\epsilon = 0{,}8$, la zona del medio con $\epsilon = 0{,}4$ y la de la derecha con $\epsilon = 0{,}2$.

Para los siguientes experimentos, se cambia el esquema de ϵ-*greedy* por un escenario alternante, en el cual se combina etapas de entrenamiento con un ϵ mayor que cero, con etapas de prueba en donde ϵ es igual a cero. De esta forma, se puede apreciar más claramente si el algoritmo está produciendo los beneficios esperados mediante la sóla observación de las zonas del gráfico en donde $\epsilon = 0$, que corresponden a un juego absolutamente "ambicioso" o *greedy*.

El segundo experimento que se realiza con este esquema se observa en la figura 4.6. Además de cambiar el esquema de ϵ-*greedy*, se modifica la discretización hecha por *tile coding* por una más gruesa, de modo de disminuir el tamaño del espacio de estados (ver sección 3.2.2). Pese a esto, y a que el número de partidos se duplica para este experimento, se puede observar que los resultados no mejoran con respecto a los de la figura 4.5, sino que presentan un comportamiento similar. Concretamente, se puede observar cómo en las zonas de la figura marcadas por franjas oscuras —que corresponden a partidos jugados con $\epsilon = 0$, se obtienen

resultados negativos, que no mejoran en el tiempo[2].

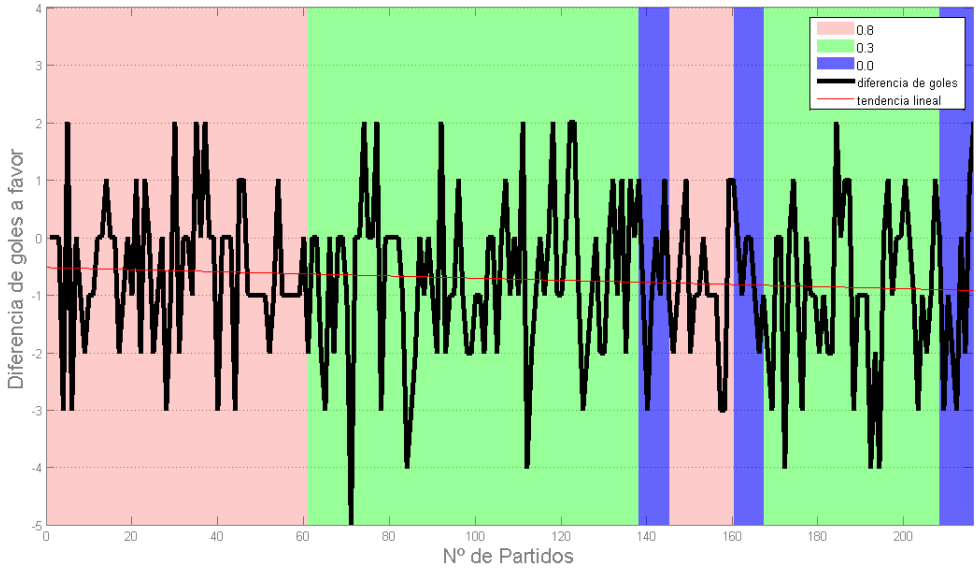

Figura 4.6: Diferencia de goles obtenida para el segundo experimento realizado en un partido de 4 contra 4 jugadores. Se jugó un total de 216 partidos, lo que equivale a 2160 minutos o a 36 horas de juego simulado. La zona denotada por la franja más clara (la de la izquierda) fue entrenada con $\epsilon = 0{,}8$, la zona del intermedia con $\epsilon = 0{,}3$ y la zona oscura con $\epsilon = 0{,}0$.

Hasta ahora se ha trabajado empezando el entrenamiento con una tabla vacía:

$$Q(s, a) = 0, \quad \forall (s, a)$$

es decir, aprendiendo desde cero. Bajo el supuesto de que ésta sea la razón por la cual los experimentos realizados no han funcionado, y que podrían requerir un tiempo demasiado grande para mostrar algún resultado, se decide utilizar en adelante una estrategia ya "probada", de modo de aprender a partir de ésta valores coherentes de $Q(s, a)$. Para esto, se juega una serie de partidos utilizando una estrategia[3] basada en MDP (Markov Decision Process)[4],

[2]El comportamiento para los resultados en esta zona es completamente opuesto al deseado.

[3]que corresponde a la mejor de las estrategias que se ha programado en el laboratorio para los robots y utiliza programación dinámica para la estimación de la función de valor.

[4]En español, proceso de decisión Markoviano.

y se realizan actualizaciones de $Q(s, a)$ de la misma forma en que se han venido haciendo. Esto equivale a partir la etapa descrita como "evaluación de política'útilizando una política que se sabe buena (porque presenta buenos resultados en la práctica), por lo que la convergencia hacia una política óptima debería ser más rápida.

En el siguiente experimento, se juega 13 partidos con la estrategia MDP para llenar la tabla y posteriormente se realiza un entrenamiento como el que se ha venido haciendo. Este experimento se realiza para el esquema en donde se elige 1 mejor compañero y 2 peores oponentes, de modo de probar si esta configuración es capaz de generalizar más, y por lo tanto, aprender más rápidamente. Se simula un total de 131 partidos, lo que equivale a 1310 minutos o a 21 horas con 50 minutos. Los resultados se pueden observar en la figura 4.7, en donde se aprecia la misma tendencia de los experimentos anteriores. Nuevamente se puede observar cómo en las zonas que corresponden a partidos jugados con $\epsilon = 0$, se obtienen resultados negativos y que no mejoran en el tiempo.

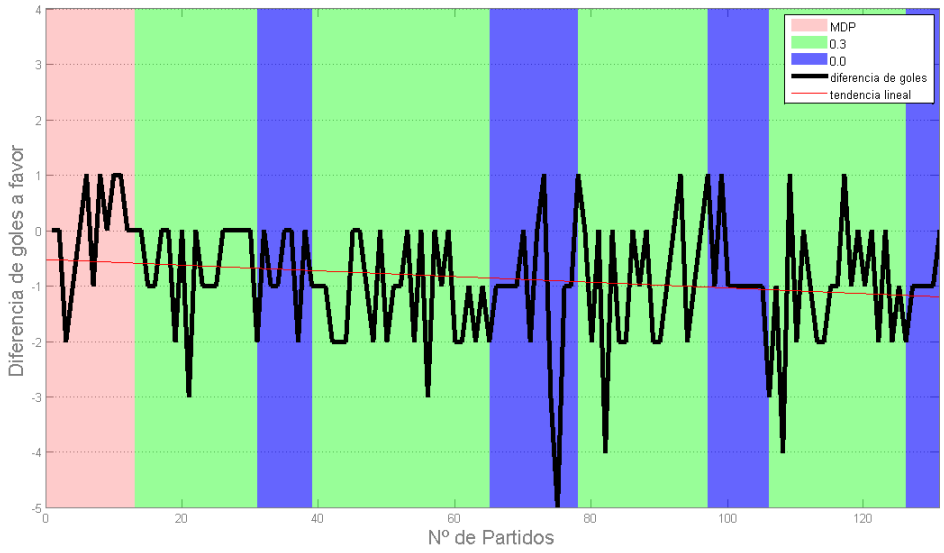

Figura 4.7: Diferencia de goles obtenida para partidos de 4 contra 4 jugadores, utilizando un entrenamiento previo con una estrategia MDP.

Finalmente, se repite el experimento, pero esta vez para dos mejores compañeros y tres

peores oponentes. Además, se simula un total de 366 partidos, lo que equivale a 3660 minutos, o a 61 horas. De esta forma, se busca cubrir las posibilidades no exploradas en los experimentos anteriores. Los resultados no son satisfactorios, sino que repiten el comportamiento observado para los experimentos anteriores. Estos pueden observarse en la figura 4.8.

Si bien el esquema de dos mejores compañeros y tres peores oponentes se traduce en un estado de dimensionalidad bastante mayor que el del esquema con un mejor compañero y dos peores oponentes, la situación de lo que está ocurriendo en la cancha está representada de manera más fiel por el primero de éstos. Como no se tiene un algoritmo que funcione de manera comprobada, se debería probar primero si funciona para la configuración con dos mejores compañeros y tres peores oponentes, y luego probar con un espacio de estados más reducido.

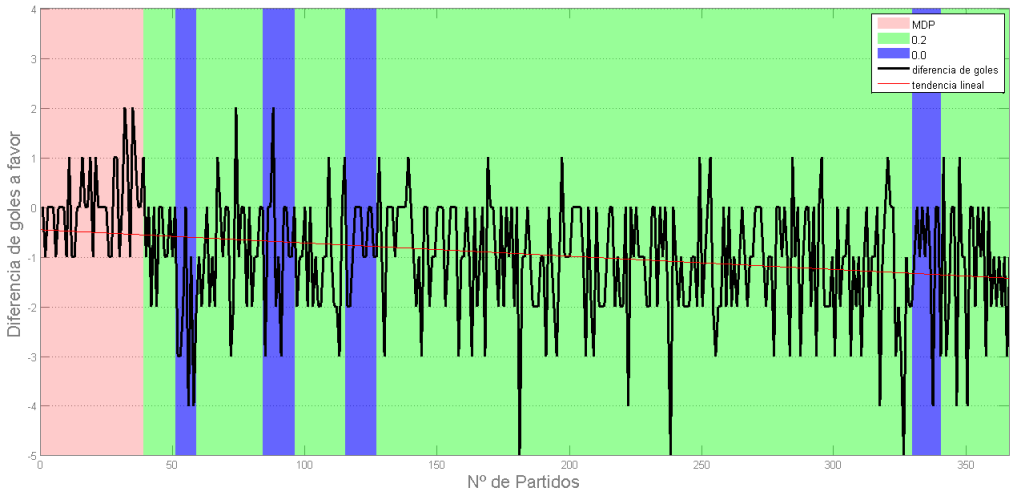

Figura 4.8: Diferencia de goles obtenida para partidos de 4 contra 4 jugadores, con un estado en el que se representan 2 mejores compañeros y 3 peores oponentes.

4.3. Resultados Obtenidos con *SARSA*

Los experimentos realizados con *SARSA* parten desde la base de lo que se hizo para los experimentos realizados con *Q-Learning*, es decir, se trabaja desde un comienzo con una

etapa de entrenamiento mediante el uso de una estrategia MDP, y se trabaja todo el tiempo con 2 mejores compañeros y 3 peores oponentes.

El primer experimento se realiza con el primer esquema de recompensas propuesto para *SARSA*, en donde se recompensa además de las situaciones en que hay gol, el tiempo que se mantiene posesión de la pelota por parte del equipo (ver ecuación 3.3). Los resultados se observan en la figura 4.9. Pese a que se juegan sólo 103 partidos, se aprecia una tendencia claramente decreciente, lo que es una motivación adicional para cambiar el esquema de las recompensas.

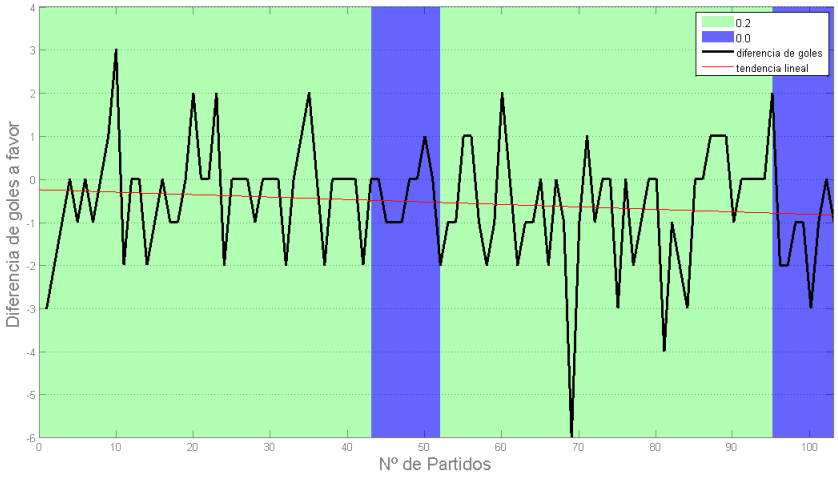

Figura 4.9: Diferencia de goles para partidos de 4 contra 4 jugadores. Se simuló un total de 103 partidos, lo que equivale a 1030 minutos o a 17 horas y 10 minutos, en los que no se logra una tendencia creciente.

El segundo experimento se realiza con el segundo esquema de recompensas propuesto para *SARSA*, que es muy similar al primero, pero con una recompensa constante otorgada por la posesión de la pelota (ver ecuación 3.4). Los resultados se observan en la figura 4.10.

Si se compara estos resultados con los últimos resultados obtenidos con *Q-Learning* para un número igual de partidos y bajo condiciones similares, se obtiene la figura 4.11. De la simple observación de la figura resulta difícil concluir si alguna estrategia es superior a la otra. Al realizar las interpolaciones lineales, se obtiene una pendiente prácticamente igual a

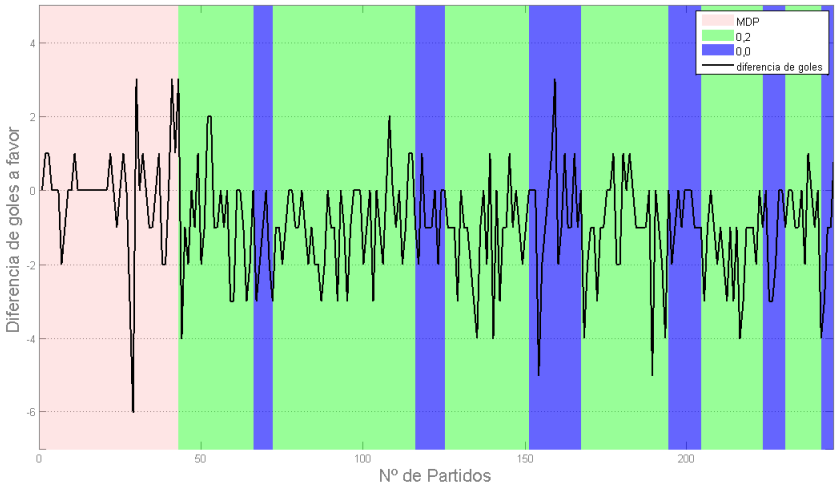

Figura 4.10: Diferencia de goles obtenidas para partidos de 4 contra 4 jugadores, con un esquema de recompensas fijas.

cero para *SARSA* y una pequeña pendiente negativa para *Q-Learning*, lo que significa que por alguna razón *Q-Learning* en promedio está empeorando sus resultados (recordar que este análisis incluye la etapa de entrenamiento). Esto es natural debido a que *Q-Learning* es un algoritmo *off-policy* y por lo tanto proclive al riesgo.

Por otra parte, en términos de promedio, la diferencia de goles obtenida con *Q-Learning* promedia $-0{,}945$, mientras que para *SARSA* promedia $-0{,}576$, lo que reafirma el análisis anterior.

Finalmente, se decide realizar una prueba con *SARSA* utilizando una codificación mas gruesa, en donde las particiones tengan *tiles* de tamaño 1/6 de la distancia máxima para las variables de distancia, y de tamaño 1/8 de una circunferencia para las variables angulares. Además, el número de tiles se reduce a 8, con lo que la resolución final de la distancia queda en:

$$d_{max}/6 \cdot 1/8 = 533/48 = 11,1[cm]$$

mientras que la resolución angular queda en:

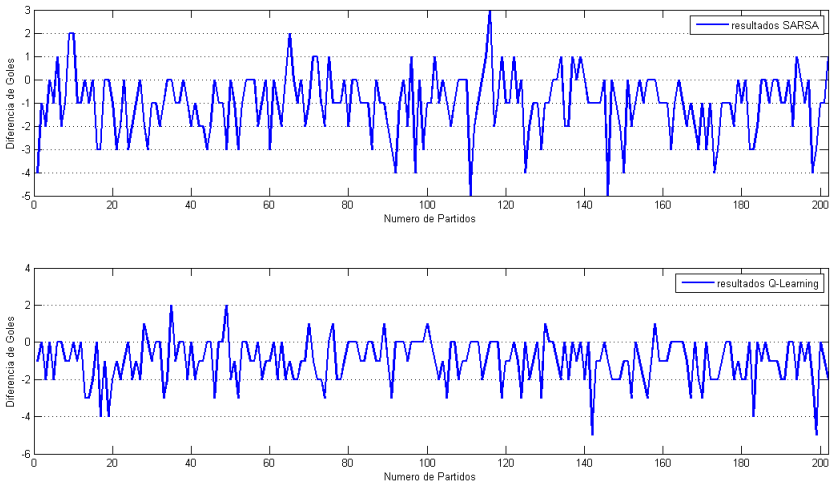

Figura 4.11: Diferencia de goles para ambos algoritmos. Si se interpola linealmente, para *SARSA* se obtiene la ecuación: $y_{sarsa} = 0,0004855 \cdot x - 0,97547$ y para *Q-Learning* la ecuación: $y_{qlearning} = 0,00131 \cdot x - 0,85335$.

$$2\pi/8 \cdot 1/8 = 2\pi/64 = 0,09817[rad] = 5,625[^o]$$

ambas resoluciones completamente aceptables. Los resultados de esta configuración se observan en la figura 4.12, en donde el total de partidos simulados es 577, lo que equivale a 5770 minutos o a 92 horas y 50 minutos. Cabe mencionar que en tiempo real, esta simulación tomó más de 218 horas, las cuales no transcurrieron de manera continua (debido a las limitaciones del simulador antes mencionadas).

Un análisis que puede entregar información más relevante, es comparar el comportamiento de *SARSA* y de *Q-Learning* sólo para los partidos que fueron jugados con $\epsilon = 0$ (es decir, cuando la política a seguir fue escoger siempre la acción que maximizara la función de valor). Esto se puede observar en las figuras 4.13, 4.14 y 4.15, en donde la primera corresponde a *Q-Learning* y las dos últimas a *SARSA* (la primera corresponde al entrenamiento referido por la figura 4.10 y la segunda al referido por la figura 4.12. Los tres casos presentan una tendencia creciente, aunque para *Q-Learning* esta es levemente más pronunciada que para *SARSA*. Esta comparación no es absolutamente directa, pues los esquemas alternantes de

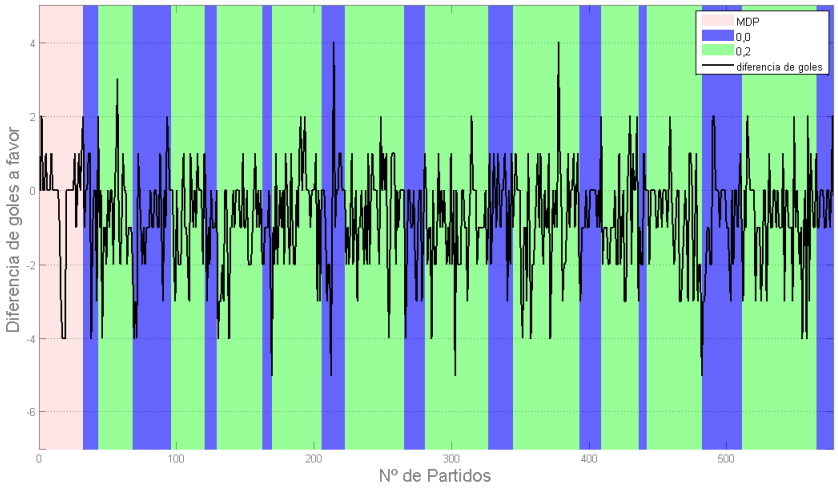

Figura 4.12: Diferencia de goles obtenidas para partidos de 4 contra 4 jugadores, para un total de 577 partidos.

políticas no calzan completamente para cada ejemplo. En cualquier caso, si esta tendencia se mantiene, en todos los casos se tendrá resultados positivos tras la continuación de los experimentos por —al menos— el mismo tiempo que se ha entrenado hasta ahora.

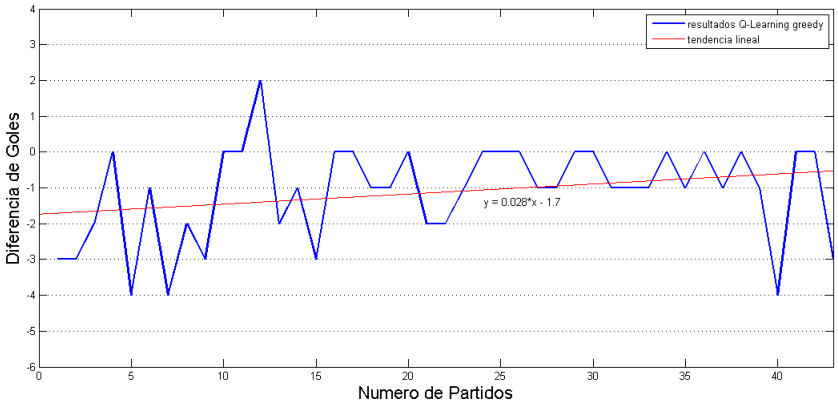

Figura 4.13: Diferencia de goles de *Q-Learning*. (Sólo para los partidos jugados con $\epsilon = 0$).

Se debe realizar entonces un análisis en profundidad que indique porqué los resultados no son satisfactorios.

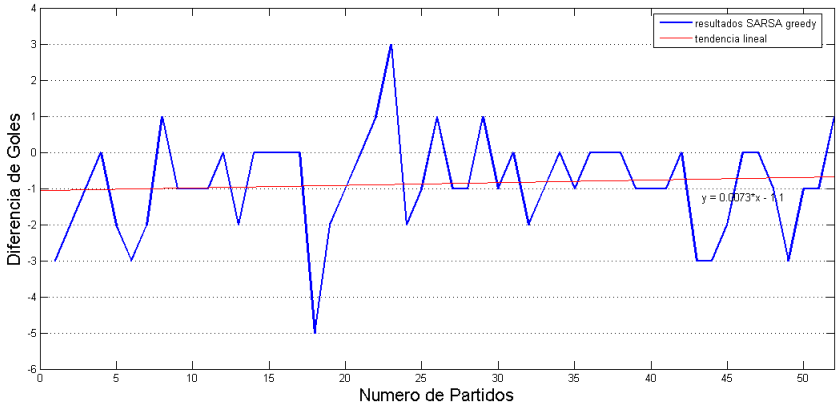

Figura 4.14: Diferencia de goles de la segunda implementación de *SARSA*.(Sólo para los partidos jugados con $\epsilon = 0$).

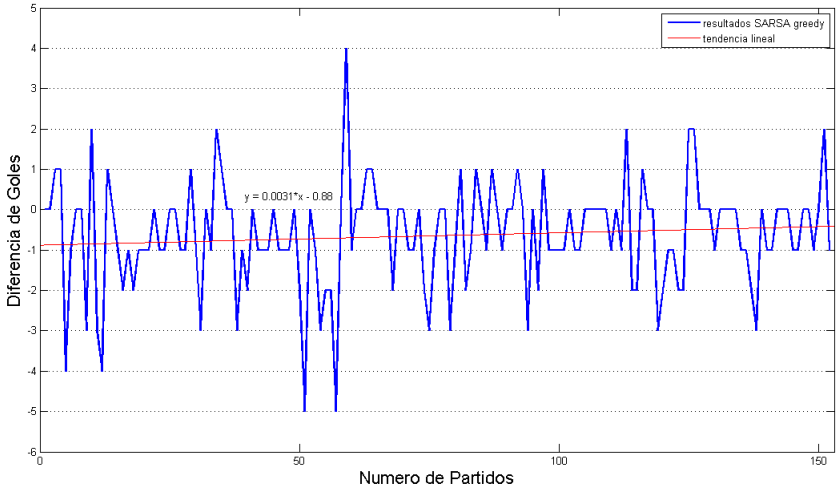

Figura 4.15: Diferencia de goles de la tercera implementación de *SARSA*. (Sólo para los partidos jugados con $\epsilon = 0$). Se utiliza una codificación más gruesa que en el caso anterior.

En primer lugar, se revisa nuevamente que el algoritmo funcione correctamente. Para esto se hace un *debugueo* extenso y además se realizan diversas pruebas de modo de comprobar que el aprendizaje está siendo coherente. Se evalúa la función de valor para distintas configuraciones, utilizando los pesos correspondientes al último partido jugado con *SARSA*. Primero, se compara para situaciones fijas el valor que tiene cada acción disponible. En las figuras 4.16

y 4.17 se observan ejemplos de cómo frente a situaciones básicas el agente tomaría la decisión que nos parecería correcta (esto es, la de mayor valor).

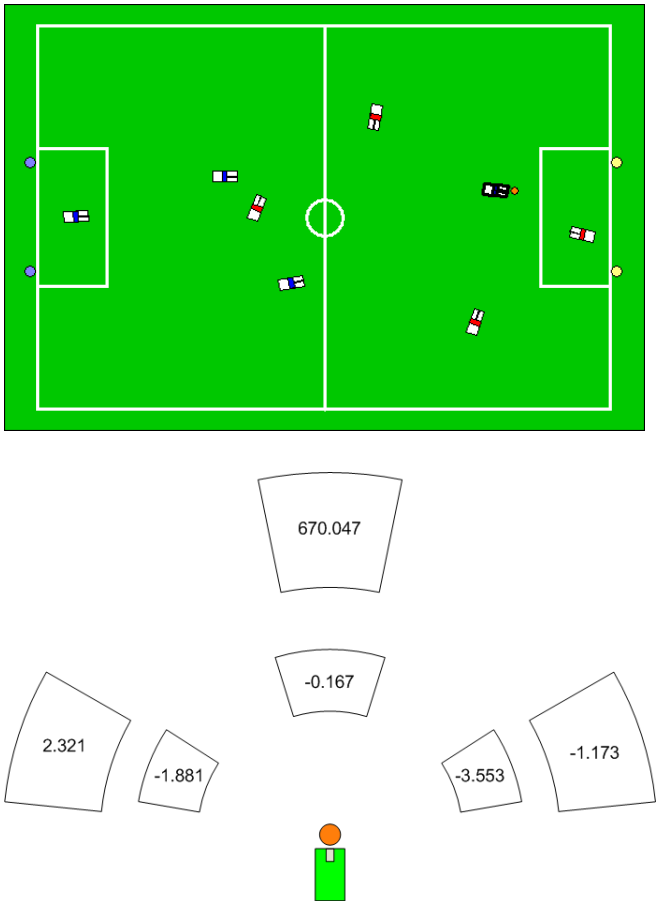

Figura 4.16: Evaluación de las distintas acciones del agente para una situación específica. En la imagen superior se observa la situación para la cual se realiza la evaluación, mientras que en la imagen inferior se puede observar el espacio de acciones al agente, donde cada número indica el valor que tiene para el agente realizar dicha acción. Nótese que la acción con mayor valor corresponde a la acción que convierte el gol.

En segundo lugar, se prueba para distintas situaciones el valor que tiene la realización de una sola acción, modificando sólo una variable de estado. En la figura 4.18 se observa un ejemplo en el cual sólo se modifica la distancia del agente al arco oponente y se evalúa la función $Q(s, a)$ para un estado arbitrario s correspondiente a una configuración de los jugadores fija alrededor del agente, y a la acción número uno, que corresponde al golpe frontal corto. Es esperable que en general, la función de valor sea mayor en las cercanías del

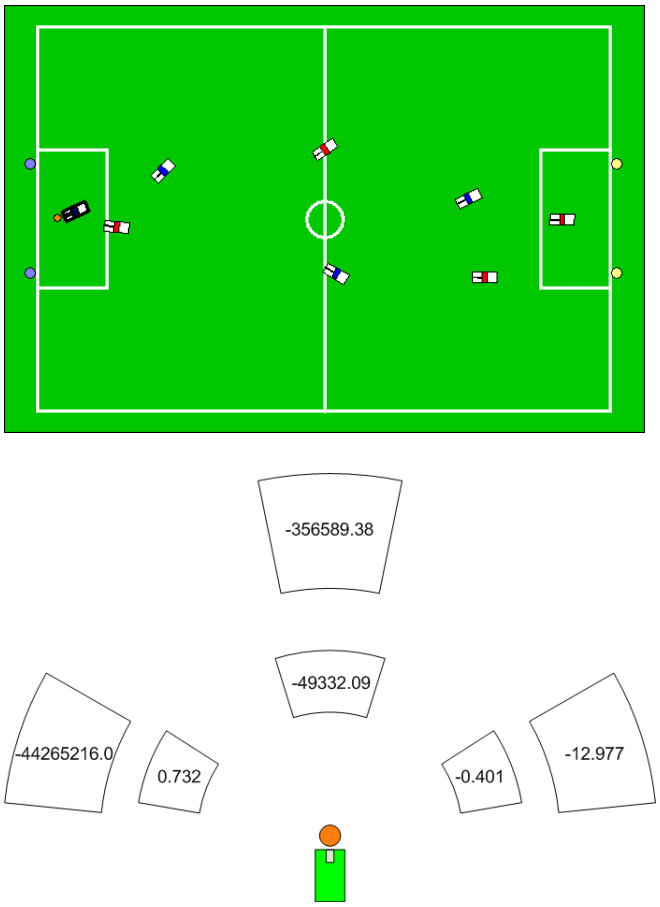

Figura 4.17: Evaluación de las distintas acciones del agente para una situación específica. En la imagen superior se observa la situación para la cual se realiza la evaluación, mientras que en la imagen inferior se puede observar el espacio de acciones al agente, donde cada número indica el valor que tiene para el agente realizar dicha acción. Nótese que la acción con mayor valor corresponde a la acción que evita el autogol..

arco oponente.

Otro ejemplo se puede observar en la figura 4.19, donde se sitúa el agente en el área oponente, y sólo se modifica el ángulo absoluto de éste, es decir, se rota en 360 grados. Nuevamente se evalúa la función de valor para la primera acción. Es esperable que cuando el robot esté de frente al arco (ángulos cercanos a cero), la función presente valores mayores.

Si bien las funciones representadas en las figuras 4.18 y 4.19 dan confianza en que el agente está aprendiendo de manera correcta, se cree que la presencia de discontinuidades en éstas (y en otras funciones observadas que no se incluyen aquí) es un indicador de que todavía falta

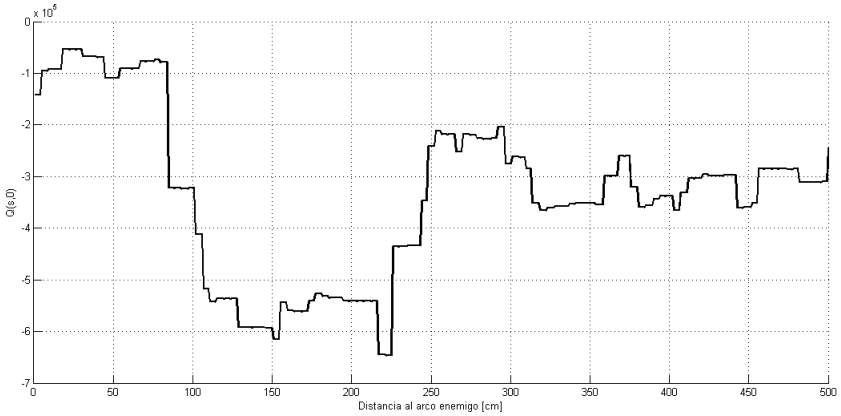

Figura 4.18: Evaluación de la función de valor para un estado al que sólo se le modifica la variable de distancia al arco, para la primera acción (acción cero). Si bien la función de valor resulta negativa en todo el espacio en que se evalúa, tiene un valor mayor en las cercanías del arco oponente (distancias cercanas a cero), lo que es completamente intuitivo.

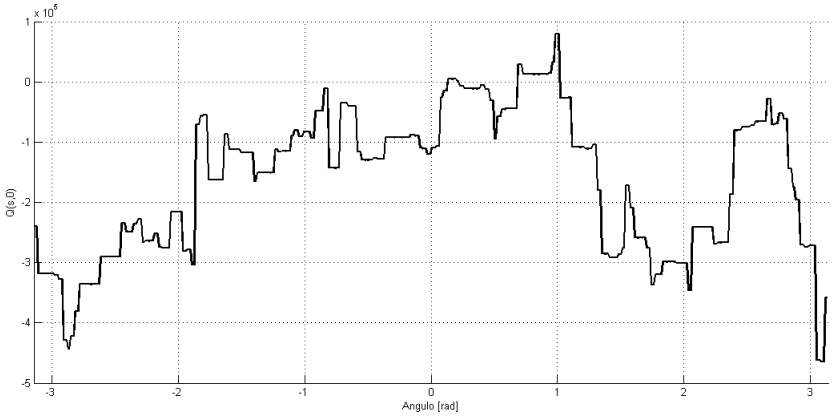

Figura 4.19: Evaluación de la función de valor para un estado al que sólo se le modifica la variable de angulo absoluto, para la primera acción (acción cero). Se comprueba cómo el valor es mayor para cuando el robot mira de frente al arco oponente y este disminuye cuando le da la espalda.

entrenamiento. Esto bajo el supuesto de que la función de valor es una función suave.

Una vez que se tiene seguridad de que la implementación es correcta, se compara con los resultados obtenidos en problemas similares, y es ahí donde se puede vislumbrar porqué no se alcanzan escenarios favorables:

En el caso del *Keepaway* [9], la elección del espacio de estados involucró un total de 13 variables de estado (11 variables de distancia y 2 variables angulares). Las variables angulares fueron discretizadas en intervalos de 10 [^{o}] mientras que las variables de distancia fueron discretizadas cada 3 [mt]. Como la cancha utilizada para el Keepaway es de 20x20 [mt^2] (para la configuración de 3 keepers contra 2 takers) , la máxima distancia a ser representada es de 28,6 [mt], por lo que la dimensión del espacio de estados se puede estimar como:

$$\frac{28,6[mt]}{3[mt]} \cdot 11 \cdot \frac{360[^o]}{10[^o]} \cdot 2 = 7551$$

mientras que para el caso de 4 keepers contra 3 takers la cancha es de 30x30 [mt^2] por lo que la máxima distancia a ser representada es de 42,3 [mt]. Con esto, la dimensión del espacio de estados queda:

$$\frac{42,3[mt]}{3[mt]} \cdot 11 \cdot \frac{360[^o]}{10[^o]} \cdot 2 = 11168$$

Para este problema, que fue abordado con *SARSA*, los tiempos de entrenamiento necesarios para alcanzar la convergencia de la función de valor fueron entre 15 a 20 horas de tiempo simulado (y para una prueba que se hizo con *Q-Learning* fueron entre 40 a 50 horas). Además, comprueban como los tiempos de convergencia aumentan considerablemente al aumentar el tamaño del estado (al pasar de 3v2 a 4v3, estos se duplican).

Algo similar ocurre en el caso del *Half Field*, en donde si bien el espacio de estados se define de un tamaño menor al del *Keepaway*, por tratarse de un problema más complejo, los tiempos de convergencia son aún mayores [4].

En comparación, este algoritmo trabaja con espacio de estados del orden de 10^{11}, es decir, 7 órdenes mayores a lo utilizado en los trabajos recién citados. Esto permite suponer que las horas de entrenamiento necesarias para la convergencia de la función de valor son al menos 7 órdenes mayor[5] a las horas requeridas para solucionar el *Keepaway* o el *Half Field*. De esta forma, si para lograr una convergencia del Keepaway fueron necesarias 15 horas, para lograr

[5]Es sabido que en este tipo de problemas los tiempos de convergencia y procesamiento crecen en general exponencialmente con el número de variables, por lo que suponer un crecimiento lineal es ponerse en una situación lo más favorable posible.

la convergencia en el problema aquí planteado, podrían llegar a necesitarse 150000000 horas. Esto es un hecho insoslayable, que induce a cuestionarse la aplicabilidad del modelo a un problema como éste.

Por último, de los gráficos de las figuras 4.13, 4.14 y 4.15 también puede obtenerse una conclusión similar, pues si bien estos no son categóricos, todos presentan una tendencia creciente muy lenta. Esto hace pensar que la tendencia creciente sí refleja lo que pasa (y no sólo corresponde a un ruido), sobre todo porque para el gráfico de la figura 4.15 se considera más de 150 partidos (más de 25 horas de juego). Si se extrapola la tendencia lineal de dicha figura, se obtiene que se entraría en una zona de resultados positivos (en promedio) recién después de 283 partidos. Sin embargo, se debe considerar que estos gráficos no son continuos temporalmente, pues se está representando sólo los partidos jugados con una política absolutamente ambiciosa. De esta forma, extrapolando nuevamente, esta vez en base a la tasa de partidos jugados con la política *greedy* (154) versus el total de partidos (557), se puede estimar un total de 1027 partidos (171 horas aproximadamente) para que los resultados comiencen a ser positivos en promedio. Este análisis no permite predecir cuánto tiempo se demorará en converger, pero con seguridad será mayor que eso.

Capítulo 5

Conclusiones

5.1. Conclusiones y análisis finales

Los resultados obtenidos en los experimentos no fueron satisfactorios. Si bien con el esquema basado en *SARSA* se logró mejores resultados para toda la etapa de entrenamiento que con la implementación de *Q-Learning* originalmente implementada, y se obtuvo tendencias crecientes para ambos algoritmos, no se logró llegar a un régimen permanente en el cual se ganaran los partidos. Se cree que la principal razón de esto —aplicable a ambos algoritmos—, es el gran tamaño del espacio de estados, requiriéndose una cantidad enorme de horas de entrenamiento para alcanzar la convergencia de la función de valor. (Mediante un análisis comparativo se vio que la cantidad de horas necesarias podía llegar a ser del orden de 10^7 lo cual resulta completamente impracticable, al menos para los alcances de este trabajo o de la competencia de fútbol robótico).

Cabe preguntarse porqué con *SARSA* se lograron resultados levemente mejores que con *Q-Learning*. ¿Se puede simplemente decir que *SARSA* es mejor que *Q-Learning* para un problema de este tipo? Se cree que no: en primer lugar porque si bien las condiciones bajo las que se realiza la comparación son idénticas en términos de variables de estado, etapas de entrenamiento, tipo de juego y otros aspectos; el algoritmo implementado con *SARSA* utiliza los trazos de elegibilidad —por lo que realiza actualizaciones correspondientes a múltiples instantes temporales, mientras que el algoritmo implementado con *Q-Learning*, sólo actualiza la función de valor para diferencias de un instante temporal. En segundo lugar, las pruebas

observadas en donde se jugó con una política absolutamente ambiciosa (*greedy*) mostraron una tendencia creciente en ambos casos, pero mayor para *Q-Learning*, lo que sugiere que de seguir entrenando *Q-Learning* convergería más rápido. Sin embargo, esto no es categórico, debido a que los datos presentan una dispersión muy alta.

Sí se puede decir en cambio, que para un problema con una alta dimensionalidad del espacio de estados como éste, conviene usar trazos de elegibilidad de modo de converger más rápido. Esto es un hecho reconocido en la literatura que fue comprobado en este trabajo de manera parcial.

Otra razón por la que *SARSA* puede haber resultado más efectivo para esta aplicación, es el hecho de que se trata de un algoritmo *on-policy*. Esto significa que es un algoritmo averso al riesgo (en comparación a *Q-Learning*), pues los pares estado acción que se llenan durante el entrenamiento son sólo los visitados, y por lo general, estos son los que tienen un valor mayor. El hecho de que *SARSA* sea averso al riesgo no es suficiente para decir que puede ser más o menos efectivo, pero si a esto se suma el hecho de que el sistema es un Sistema Estocástico Multiagente, resulta natural ver porqué un algoritmo *on-policy* puede ser más efectivo —o demorar menos en producir resultados positivos: en este tipo de sistemas resulta mucho más difícil aprender de las acciones del oponente que en sistemas Monoagente. Por esta razón, un sistema proclive al riesgo puede pasar mucho tiempo cometiendo "errores" y por lo tanto tomará más tiempo en aprender.

Una alternativa para abordar el problema Estocástico Multiagente en cuestión es agregando a la tabla $Q(s, a)$ las acciones del enemigo. Si bien este método podría haber convergido más rápido, se trata de un esquema ficticio, difícil de aplicar a situaciones con robots reales. En este sentido, la solución aquí planteada cuenta con diversos elementos que agregan incerteza al juego —como por ejemplo la representación de acciones, compañeros y enemigos como áreas de incerteza—, haciendo más realistas los resultados obtenidos. Además, el simulador en sí provee un alto grado de incerteza, principalmente debido a acciones que en algunos casos son saltadas porque cambia la configuración del juego (ocurre de manera muy parecida con los robots reales).

Claramente las pruebas que se hicieron para comprobar el funcionamiento de los métodos fueron muy alejadas de las pruebas finales que se realizaron. Hubiera sido útil realizar pruebas de una dificultad intermedia, por ejemplo, disminuyendo los grados de incerteza mencionados en el párrafo anterior lo más posible. De esta forma, se podría haber comprobado de manera sencilla y rápida que el algoritmo está funcionando para luego pasar a las pruebas más complejas. En este contexto se debe recordar que uno de los principales problemas asociados a la experimentación, es que las pruebas toman mucho tiempo de realizar y esta es la razón principal por la que no se hicieron más experimentos.

Se comprueba la importancia de escoger una recompensa que se alinee con el objetivo global. En particular, se observa que el esquema original de recompensas propuesto para *SARSA* no es tan útil como el que se implementa después. El hecho de que las recompensas sean fijas permite una calibración más exacta del sistema, mientras que asociar el tiempo de posesión de la pelota a las recompensas resulta más útil para tareas como el *Keepaway*, donde no se mezclan con recompensas fijas. Nótese que esto es independiente del algoritmo que se esté usando, ya que sólo tiene que ver con el modelo de aprendizaje reforzado que se está usando y de cómo la recompensa escogida se orienta hacia el objetivo.

En cuanto a la reducción del espacio de estados, el solo hecho de obtener resultados crecientes indica que es una idea que da resultado. Si bien no se cuenta con la cantidad de pruebas suficientes como para poder comparar qué reducción es más efectiva, o hasta qué punto se puede reducir el espacio de estados, se puede decir con propiedad que el algoritmo utilizado para escoger a los mejores compañeros y peores oponentes funciona sin problemas. En cualquier caso, observando lo que ocurre en los algoritmos desarrollados para el *Keepaway* y el *Half Field*, se cree que es posible (y necesario) reducir el espacio de estados mucho más para lograr buenos resultados.

Definitivamente el tamaño o resolución de las particiones es un factor relevante. Tal como se observó en la sección 3.2.2—en donde se obtuvo una disminución de dos órdenes de magnitud para el tamaño del espacio de estados tras aumentar el tamaño de los *tiles*, la variación de estos parámetros puede incidir considerablemente en la dimensión del espacio de estados.

Por otra parte, existe un límite superior sobre el cual no conviene seguir aumentando el tamaño de los *tiles*: tener particiones cada vez más gruesas equivale a discretizar el espacio de estados de una forma cada vez más gruesa, con lo que el agente va perdiendo paulatinamente la capacidad de distinguir entre estados distintos. Como ya se ha mencionado, si bien esto puede ser visto como una ventaja que permite una mayor generalización, existe un punto en el cual el agente se vuelve "ciego" a la diferencia entre dos estados que convendría distinguir. Esto significa que la posibilidad de funcionamiento de un algoritmo es muy sensible a la discretización que se haga: un algoritmo que converge para una discretización dada podría converger en un tiempo infinito para una discretización más fina, o podría no converger nunca si la discretización se hace muy gruesa.

Se logra apreciar también que se está muy lejos de llegar a un entrenamiento en línea (*online*) efectivo, pues los tiempos de convergencia para este problema se estiman tan grandes, que resultaría muy poco conveniente jugar contra un equipo y tratar de entrenar a medida que se juega. Además, se debe resolver el problema de los tiempos de procesamiento (que si bien en *SARSA* son teóricamente mayores por el uso del vector de elegibilidad, esto no se comprobó en la práctica pues resultaron muy similares), ya que es deseable que estos robots jueguen en tiempo real.

Se debe destacar que la propuesta de esta memoria logra entrenar un sistema para un espacio de estados muy grande, reduciendo considerablemente los requerimientos de memoria. Además, los requerimientos de procesamiento son naturalmente menores a los de una estrategia basada en cálculos probabilísticos, debido a que la mayoría de las operaciones asociadas a los algoritmos aquí presentados tiene que ver con accesos a memoria y operaciones simples.

Finalmente, cabe mencionar que el acercamiento propuesto por esta memoria va en la línea de lo que se ha venido haciendo en materia de Aprendizaje Reforzado, pero es un paso nuevo, pues se aplica a una situación más compleja de las que se han resuelto anteriormente. Si bien no se logra solucionar el problema, se obtienen directrices para la elección de las variables importantes a ser representadas, para el tipo de entrenamiento a realizar y para las recompensas a fijar. Queda abierto el problema de cuán grande puede ser el espacio de

estados, y de qué forma se puede minimizar la etapa de entrenamiento, de modo de decidir qué problemas —sino todos— son factibles de resolver utilizando aprendizaje reforzado.

5.2. Trabajo Futuro

A continuación se presenta algunas de las mejoras o complementos posibles de realizar para este trabajo.

En primer lugar, se puede implementar una función $s' = \texttt{mirror}(s)$, que aproveche las simetrías existentes en una cancha de fútbol para reducir el espacio de estados a la mitad. De esta forma, el estado s y el estado s' serían situaciones idénticas desde el punto de vista del aprendizaje. Esta situación se puede observar en al figura 5.1. Además, esta función puede llegar a ser de utilidad para optimizar otras estrategias ya implementadas o que se deseen implementar.

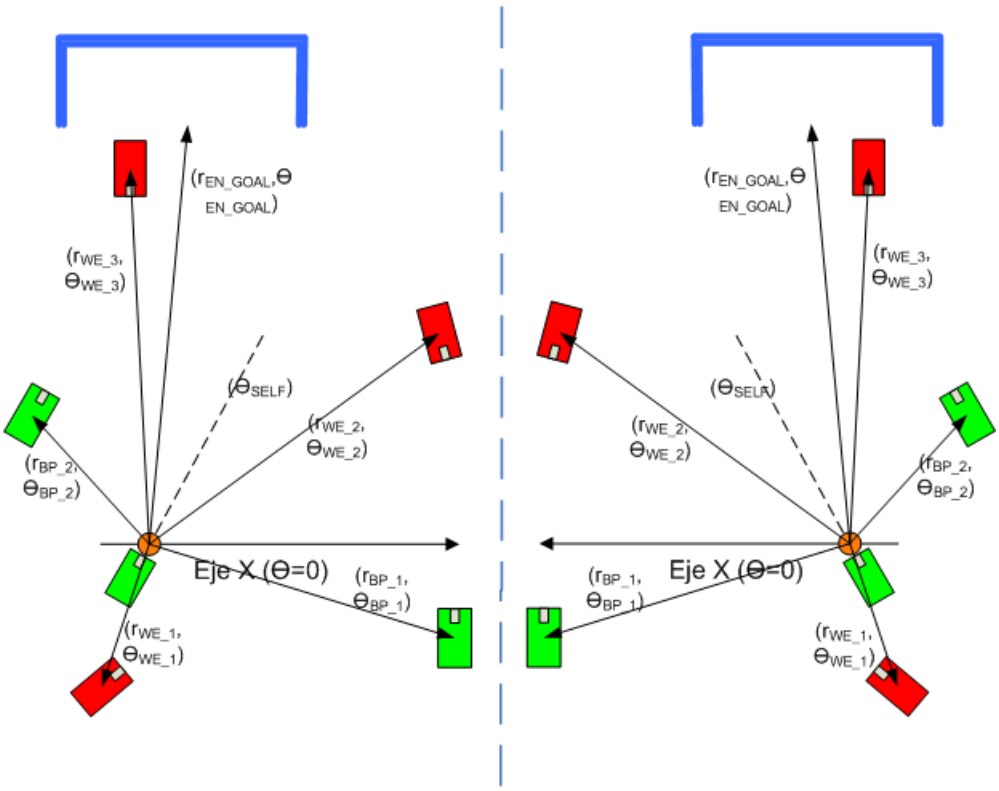

Figura 5.1: Dos situaciones diferentes que tienen una simetría de espejo.

Otra tarea interesante de realizar es entrenar el algoritmo de *Q-Learning* por más tiempo de modo de ver cuánto tiempo demorará en producir resultados, y si estos no se logran, implementar una versión de *Q-Learning* con trazos de elegibilidad, de modo de poder comparar ésta con el algoritmo de *SARSA* aquí implementado. Si esto tampoco funciona, se debe simplificar el problema a ser resuelto por *Q-Learning*.

Si se desea continuar con experimentos para partidos de este tipo, es imprescindible cambiar o reparar el simulador de alto nivel, pues tiene muchas pérdidas de memoria. Estas pérdidas de memoria se traducen en que cada 3 partidos hay que reiniciar el simulador, por lo que es muy poco eficiente si se desea dejar entrenando por largos períodos de tiempo.

Por último, resultaría interesante también optimizar la resolución del problema aquí planteado de modo de tener el menor tiempo de convergencia a un escenario ganador. Para esto se debe experimentar variando los parámetros del algoritmo[1] y realizar muchas pruebas de entrenamiento hasta encontrar un ajuste óptimo. Una vez realizado esto, se puede convertir el problema en un problema estándar —o *benchmark*— y probar sobre éste distintos algoritmos de aprendizaje reforzado.

[1]Como por ejemplo realizar pruebas variando la cantidad de compañeros y oponentes representados en el estado.

Referencias

[1] Alexander A.Sherstov and Peter Stone. Function approximation via tile coding: Automating parameter choice. In *Lecture Notes in Computer Science: Abstraction, Reformulation and Approximation*, pages 194–205. Springer Berlin / Heidelberg, 2005.

[2] Avraham Bab and Ronen I. Brafman. Multi-agent reinforcement learning in common interest and fixed sum stochastic games: An experimental study. *Journal of Machine Learning Research*, 9:2635–2675, Dic 2008.

[3] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[4] Shivaram Kalyanakrishnan, Yaxin Liu, and Peter Stone. Half field offense in robocup soccer: A multiagent reinforcement learning case study. In *RoboCup 2006: Robot Soccer World Cup X*, pages 72–85. Springer-Verlag, 2006.

[5] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370. Morgan Kaufmann, 1995.

[6] Jing Peng and Ronald J. Williams. Incremental multi-step q-learning. In *Machine Learning*, pages 226–232. Morgan Kaufmann, 1994.

[7] Javier Ruiz del Solar, Pablo Guerrero, Paul Vallejos, Patricio Loncomilla, Rodrigo Palma-Amestoy, Pablo Astudillo, Ricardo Dodds, Javier Testart, David Monasterio, and Andres Marinkovic. Uchile1 strikes back, 2006 team description paper. In *Robotics Sym-*

posium, 2006. LARS '06. IEEE 3rd Latin American, pages 200–2007, Santiago, Chile, Oct 2006.

[8] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. In *Machine Learning*, pages 287–308, 2000.

[9] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[11] Keiki Takadama1 and Hironori Fujita. Toward guidelines for modeling learning agents in multiagent-based simulation: Implications from q-learning and sarsa agents. In *Lecture Notes in Computer Science: Multi-Agent and Multi-Agent-Based Simulation*, pages 159–172. Springer Berlin / Heidelberg, 2005.

[12] Joaquin Victoriano. Aprendizaje reforzado en sistemas multiagente. una aproximación desde teoría de juegos, memoria para optar al título de ingeniero civil eléctrico. Universidad de Chile, Oct 2006.

[13] Christopher J. C. H. Watkins. *Learning From Delayed Rewards*. PhD thesis, Cambridge University, King's College, 1989.

Apéndices

C . Código de las Clases y Funciones Principales Implementadas

A continuación se anexa con motivos referenciales el código de las clases más represetnativas. (Para no extenderse demasiado, sólo se incluye aquí para la implementación basad en *SARSA*, que es suficientemente ilustrativa).

Listing 5.1: Q_Strategy.h

```
#ifndef _Q_STRATEGY_
#define _Q_STRATEGY_

#include "../common/kickObjective.h"
#include "../common/UCHSoccerSituation.h"
#include "../common/TileCoding.h"
#include "../common/cancha.h"


/*definiciones para la eleccion de los mejores compañeros
y los peores enemigos */
#define N_PARTNERS 2
#define N_ENEMIES 3
#define R_ENEMY 20 //revisar
#define R_PARTNER 15 //revisar tambien
#define W_PARTNER_DIST (1.0/40.0)
#define W_ENEMY_DIST (1.0/100.0)
#define W_PARTNER2GOAL_DIST (1.0/60.0)
#define W_DISTANCIA(d) ((d)>LI/3)?0:(-(3/LI)*(d)+1.0)

class StrategyState;
class UCHSoccerSituation;

class Q_Strategy
        : public kickObjective
{
public:

        Q_Strategy();
        ~Q_Strategy(){};

        // Variables estaticas correspondientes a quien realiza la accion, pero comunes a todos:

        // Guarda la informacion de el estado del mundo acutal discretizado.
        static float estado[N_ESTADOS];
        // Guarda la informacion del estado anterior al actual.
        static float estadoAnterior[N_ESTADOS];

        // En este arreglo esta guardado mediante un hash() la tabla de Qlearning, Q(s,a)
        static float weights[M_SIZE][N_ACCIONES];

        //Arreglo para los Eligibility traces
        static float e_vector[M_SIZE][N_ACCIONES];

        /* Flag que determina el primer estado para no actualizar la tabla ahi. Debe ser comun          *
         * a todos los robots para no llamar a algunas funciones dos veces (ej: gol en contra en engine) */
        static bool firstTime;
```

```
        static bool episodeStart;

        static double delta;
        static double qLast;
        static int tot_accions[N_ACCIONES];
        static int tot_egreedy;

        static unsigned long lastActionTime;

        bool actionSelected;
        bool movSelected;
        bool comunicate;

        int nEnemies;
        int nPartners;

        int ownID;
        void set_world_info();

        void get_best_partners(bool mejores_compañeros[N_ROBOT]);
        void get_worst_enemies(bool *worsts_enemies, coordenada2d source);
        void get_state(float estado[N_ESTADOS]);
        void inline setStrategyState(StrategyState *ss) { strategyState = ss; };
        StrategyState *getStrategyState()const;
        double probabilidad_interseccion(const areaang& area,double *bestAngle)const;

        void selectQObjective(bool golpe);

        int refreshTable(bool hayGol);
        void pruebaTabla();
        void set_reward(double a);
        double get_reward();
        void set_kicker(const StrategyState *pstrat);
        void reset_kicker();
        bool get_kicker(unsigned long &time_to_wait);
        void set_Timestamp(unsigned long time);
        int get_id_accion();
        void inline set_id_accion(int acc) {  id_accion=acc;    return;  };

        void fillIndex(float vector[M_SIZE][N_ACCIONES], float estado[N_ESTADOS], int accion, float valor);

        Desplazamiento get_Desplazamiento();
        void set_Desplazmiento(double x,double y, double theta);
        void set_partner_accion(int accion_amigo);
        ACCION get_accion();
        void set_id_last_partner_ball(int id, unsigned long partner_time);
        void resetCounters();
        static ACCION accion;
#ifndef IN_AIBO
        void Dibujar(CDC *dc, ubicacion *pose) const;
#endif

private:

        StrategyState *strategyState;
        bool best_partners[N_ROBOT];
        bool worst_enemies[N_ROBOT];
        UChSoccerSituation world_info;
        double reward;
        unsigned long lastKickTime;
        bool kicker;
        int id_last_partner_ball;
        unsigned long last_time_partner_touch_ball;

        unsigned long timestamp;
        int id_accion;
        Desplazamiento mov_accion;

};

#endif
```

Listing 5.2: Q_Strategy.cpp

```
#include "Q_Strategy.h"
#include "../engine/strategystate.h"

#ifndef MIN
#define MIN(x,y) ((x)<(y)?(x):(y))
#endif
#define N_ROBOT_ON_FIELD(x,y) (MIN((x),(y)))

float Q_Strategy::weights[M_SIZE][N_ACCIONES];
float Q_Strategy::e_vector[M_SIZE][N_ACCIONES];
```

```
bool Q_Strategy::firstTime;
unsigned long Q_Strategy::lastActionTime;
bool Q_Strategy::episodeStart;
double Q_Strategy::delta;
double Q_Strategy::qLast;
ACCION Q_Strategy::accion;
float Q_Strategy::estado[N_ESTADOS];
float Q_Strategy::estadoAnterior[N_ESTADOS];
int Q_Strategy::tot_accions[6] = {0, 0, 0, 0, 0, 0};
int Q_Strategy::tot_egreedy = 0;

Q_Strategy::Q_Strategy(void)
{


        for (int i=0;i<4;i++)
        {
                best_partners[i] = false;
                worst_enemies[i] = false;
        }
        kicker = false;
        comunicate = false;
        actionSelected = false;
        timestamp = 0;
        //!esto debiese estar en el constructor de TILES si se hace una clase
        int ptr = init_weights(weights);  // solo la primera vez que se invoca a tiles()
                                          // los pesos se inicializan.
        //int Q_Strategy::e_vector[M_SIZE][N_ACCIONES];


        last_time_partner_touch_ball=0;
        id_last_partner_ball = 0;


        //pruebaTabla();


        //numero total de enemigos
        nEnemies = 0;
        //numero total de compañeros, incluyendo al propio jugador
        nPartners = 0;
        for( int i=0;i<N_ESTADOS;i++)
        {
                estado[i]=0;
        }
        firstTime=true;
        episodeStart=true;
        delta=0;
        reward=0;
        //RAND_MAX
        srand( (unsigned)time( NULL ) );
}

void Q_Strategy::pruebaTabla()
{
        int wrap_array[N_ESTADOS];
        float float_array[N_ESTADOS];
        int tiles_array[N_TILINGS];

        wrap_array[DIST_BP_1]=0;
        wrap_array[ANG_BP_1]=10;
        wrap_array[DIST_BP_2]=0;
        wrap_array[ANG_BP_2]=10;
        wrap_array[DIST_WE_1]=0;
        wrap_array[ANG_WE_1]=10;
        wrap_array[DIST_WE_2]=0;
        wrap_array[ANG_WE_2]=10;
        wrap_array[DIST_WE_3]=0;
        wrap_array[ANG_WE_3]=10;
        wrap_array[DIST_GOAL]=0;
        wrap_array[ANG_GOAL]=10;
        wrap_array[ANG_SELF]=10;


        /* lleno el estado con una generalización de un arbitraria de prueba */
        float_array[DIST_BP_1] = 244/(D_MAX*0.2f);//estado[DIST_BP_1]/(D_MAX*0.2f);
        float_array[ANG_BP_1] = 0.5f/(2*3.14159f*0.1f);
        float_array[DIST_BP_2] = 44/(D_MAX*0.2f);
        float_array[ANG_BP_2] = -0.5f/(2*3.14159f*0.1f);
        float_array[DIST_WE_1] = 44/(D_MAX*0.2f);
        float_array[ANG_WE_1] = 0.5f/(2*3.14159f*0.1f);
        float_array[DIST_WE_2] = 144/(D_MAX*0.2f);
        float_array[ANG_WE_2] = -0.5f/(2*3.14159f*0.1f);
        float_array[DIST_WE_3] = 84/(D_MAX*0.2f);
        float_array[ANG_WE_3] = 0.5f/(2*3.14159f*0.1f);
        float_array[DIST_GOAL] = 30/(D_MAX*0.2f);
```

```
        float_array[ANG_GOAL] = 0/(2*3.14159f*0.1f);
        float_array[ANG_SELF] = 0/(2*3.14159f*0.1f);

        FILE *fd = fopen("../ConfFiles/pruebatabla.txt","w");
        double result;
        for(int i = 0; i< 533; i++)
        {
                float_array[DIST_GOAL] = (float)i/(D_MAX*0.2f);
                GetTilesWrap(tiles_array,N_TILINGS,M_SIZE,float_array,N_ESTADOS,wrap_array);
                result = 0.0;
                for (int j=0;j<N_TILINGS;j++)
                        result += weights[tiles_array[j]][1];
                fprintf(fd,"%d\t%f\n",(i/**2*3.14159f/533.0f - 3.14159f*/),result);

        }
        fclose(fd);
        return;
}

void Q_Strategy::get_worst_enemies(bool peores_enemigos[N_ROBOT], coordenada2d source)
{
        if(nEnemies)//por si se da el caso que no haya enemigos
        {
                coordenada2d myPose = world_info.partners[world_info.ownID-1].getPosition();
                double myOrientation = world_info.partners[world_info.ownID-1].getOrientation();

                UChAngleInterval enemy_cone, partner_cone;
                UChAngleInterval partner_goal, partner_enemy_cone;
                UChAngleInterval goal =
                        UChAngleInterval((world_info.opponentGoal.right-myPose).theta()
                        ,(world_info.opponentGoal.left-myPose).theta());


                double alpha_medio;
                int i, j, w_en_no=-1;
                double max_score=-1.0;
                bool intercepta = false;
                double r_enemy, r_partner;

                /* un arreglo por cada criterio */
                double score[N_ROBOT], score1[N_ROBOT], score2[N_ROBOT], score3[N_ROBOT];

                for(int k=0; k<N_ROBOT; k++)
                {
                        score[k] = 0;
                        score1[k] = 0;
                        score2[k] = 0;
                        score3[k] = 0;

                }

                /*primer criterio: enemigos que obstaculicen el tiro al arco*/
                /*itero sobre el numero de enemigos*/
                for(i = 0; i<N_ROBOT; i++)
                {
                  //!veo si el robot esta realmente en la cancha primero
                        if(strategyState->map.isEnemieOnField(i))
                        {
                                alpha_medio =
                                        atan(R_ENEMY/(world_info.opponents[i].getPosition()-myPose).r());
                                enemy_cone = UChAngleInterval(-alpha_medio +
                                        (world_info.opponents[i].getPosition()-myPose).theta(),
                                        alpha_medio +
                                        (world_info.opponents[i].getPosition()-myPose).theta());

                                //revisar argumentos
                                switch(goal.AngleIntersectionType(enemy_cone))
                                {
                                case UCH_ANGLE_INTERSECTION_TYPE_CONTAINS:
                                        score1[i] += 2.0;
                                        break;
                                case UCH_ANGLE_INTERSECTION_TYPE_IS_CONTAINED:
                                        score1[i] += 2.0;
                                        break;
                                case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_LEFT:
                                        score1[i] += 1.0;
                                        break;
                                case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_RIGHT:
                                        score1[i] += 1.0;
                                        break;
                                case UCH_ANGLE_INTERSECTION_TYPE_NULL_INTERSECTION:
                                        score1[i] += 0.0;
                                        break;

                                };
                                score1[i] *=
```

```
185                                     W_DISTANCIA((world_info.opponents[i].getPosition()-myPose).r());
                }
        }

        /*segundo criterio: enemigos que obstaculicen al mejor
190             compañero en el pase, y en su vision del arco*/
        /*itero sobre el numero de enemigos, solo los que esten en la cancha*/
        for(i = 0; i<N_ROBOT; i++)
        {
                if(strategyState->map.isEnemieOnField(i))
195             {
                        /*itero sobre el numero de compañeros, solo para mejores compañeros*/
                        for(j = 0; j<N_ROBOT; j++)
                        {
                                if(best_partners[j])
200                             {

                                        r_enemy = (world_info.opponents[i].getPosition()
                                                -myPose).r();
                                        alpha_medio = atan(R_ENEMY/r_enemy);
205                                     enemy_cone =
                                                UChAngleInterval(-alpha_medio +
                                                (world_info.opponents[i].getPosition()-
                                                myPose).theta(),
                                                alpha_medio +
210                                             (world_info.opponents[i].getPosition()-
                                                myPose).theta());
                                        r_partner = (world_info.partners[j].getPosition()-
                                                myPose).r();
                                        alpha_medio = atan(R_PARTNER/r_partner);
215                                     partner_cone =
                                                UChAngleInterval(-alpha_medio +
                                                (world_info.partners[j].getPosition()-
                                                myPose).theta(),
                                                alpha_medio +
220                                             (world_info.partners[j].getPosition()-
                                                myPose).theta());
                                        if (r_enemy<r_partner) intercepta = true;
                                        partner_goal =
                                                UChAngleInterval(
225                                             (world_info.opponentGoal.right-
                                                world_info.partners[j].getPosition()).theta(),
                                                (world_info.opponentGoal.left-
                                                world_info.partners[j].getPosition()).theta());
                                        alpha_medio =
230                                             atan(R_ENEMY/
                                                (world_info.opponents[i].getPosition()-
                                                world_info.partners[j].getPosition()).r());
                                        partner_enemy_cone =
                                                UChAngleInterval(-alpha_medio +
235                                                     (world_info.opponents[i].getPosition()-
                                                        world_info.partners[j].
                                                        getPosition()).theta(),
                                                        alpha_medio +
                                                        (world_info.opponents[i].getPosition()-
240                                                     world_info.partners[j].
                                                        getPosition()).theta());

                                        switch(partner_cone.AngleIntersectionType(enemy_cone))
                                        {
245                                     case UCH_ANGLE_INTERSECTION_TYPE_CONTAINS:
                                                if (intercepta)
                                                {
                                                        score2[i] += 2.0;
                                                        intercepta = false;
250                                             }
                                                else score2[i] += 0.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_IS_CONTAINED:
                                                if (intercepta)
255                                             {
                                                        score2[i] += 2.0;
                                                        intercepta = false;
                                                }
                                                else score2[i] += 0.0;
260                                             break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_LEFT:
                                                if (intercepta)
                                                {
                                                        score2[i] += 1.0;
265                                                     intercepta = false;
                                                }
                                                else score2[i] += 0.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_RIGHT:
270                                             if (intercepta)
                                                {
```

```
                                                        score2[i] += 1.0;
                                                        intercepta = false;
                                                }
275                                             else score2[i] += 0.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_NULL_INTERSECTION:
                                                score2[i] += 0.0;
                                                break;
280
                                        };
                                        switch(partner_goal.AngleIntersectionType(
                                                partner_enemy_cone))
                                        {
285                                     case UCH_ANGLE_INTERSECTION_TYPE_CONTAINS:
                                                score3[i] += 2.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_IS_CONTAINED:
                                                score3[i] += 2.0;
290                                             break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_LEFT:
                                                score3[i] += 1.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_INTERSECTS_RIGHT:
295                                             score3[i] += 1.0;
                                                break;
                                        case UCH_ANGLE_INTERSECTION_TYPE_NULL_INTERSECTION:
                                                score3[i] += 0.0;
                                                break;
300
                                        };
                                        score3[i] *=
                                                W_DISTANCIA((world_info.opponents[i].getPosition()-
                                                world_info.partners[j].getPosition()).r());
305                                     }
                                } /*fin for partners */
                        }
                } /* fin for enemies */
                for(i = 0; i<N_ROBOT; i++)
310                     score[i] = score1[i]+score2[i]+score3[i];

                for(i = 0; i<N_ROBOT_ON_FIELD((N_ENEMIES),(nEnemies)); i++)
                {
                        for(j = 0; j<N_ROBOT; j++)
315                     {
                                if(max_score<score[j])
                                {
                                        max_score=score[j];
                                        w_en_no = j;
320                             }

                        }
                        score[w_en_no] = -1;
                        peores_enemigos[w_en_no]=true;
325                     max_score = -1;
                }
        }
        return;
}
330
void Q_Strategy::get_best_partners(bool mejores_compañeros[N_ROBOT])
{
        double score[N_ROBOT];
        double partDist;
335     double oppDist[2];
        double goalDist;
        double maxScore;
        int maxPart=0;

340     for (int i=1;i<N_ROBOT+1;i++)
        {
                if (i!= world_info.ownID && strategyState->map.isPartnerOnField(i-1))
                {
                        /*lleno las distancias de interés.
345                     OJO !!! NO FUNCIONA BIEN PARA EL CASO EN QUE NO HAY OPONENTES.*/
                        partDist = world_info.getPartnerDistance(i);
                        world_info.min2OpponentDistance(i,oppDist);
                        goalDist = world_info.distancePartner2Goal(i);

350                     //calculo el puntaje con ponderaciones dadas.
                        score[i-1] =
                                -W_PARTNER_DIST*partDist+W_ENEMY_DIST*(oppDist[0]+oppDist[1])-
                                W_PARTNER2GOAL_DIST*goalDist;

355             }
                /*ESTE NUMERO LO DEJO EN -25 PORQUE EL PEOR CASO ES CUANDO LA DISTANCIA ES
                IGUAL A LA DIAGONAL INTERNA, 566 CM. */
                else score[i-1]=-25;
```

```
        }
        //idem
        maxScore = -25;
        //itero la cantidad de veces igual al nuemro de best partners a encontrar
        for (int t=0;t<N_ROBOT_ON_FIELD(N_PARTNERS, nPartners-1);t++)
        {
                //busca el mayor score y lo guarda en i.
                for (int i=0;i<N_ROBOT;i++)
                {
                        //no se considera el propio jugador que llama a esta función
                        if (score[i]>maxScore && (i+1)!=ownID)
                        {
                                maxScore = score[i];
                                maxPart = i;
                        }
                }
                //setea ese score en -25 para no volverlo a escoger, resetea maxScore
                score[maxPart]=-25;
                maxScore = -25;

                /*si el maximo puntaje no corresponde al jugador que esta iterando,
                  se setea como mejor compañero */
                if(ownID!=maxPart+1)
                        mejores_compañeros[maxPart]=true;
        }
}

void Q_Strategy::get_state(float estado[N_ESTADOS])
{
        coordenada2d myPose;

        for (int i=0;i<4;i++)
        {
                best_partners[i] = false;
                worst_enemies[i] = false;
        }

        //se guarda el estado anterior
        for( int i=0;i<N_ESTADOS;i++)
        {
                estadoAnterior[i] = estado[i];
                estado[i]=0;
        }
        //init distances debe inicializar los estados de distancia
        //en un numero mayor que el largo de la cancha.
        estado[DIST_BP_1] = estado[DIST_BP_2] =
                estado[DIST_WE_1] = estado[DIST_WE_2] =
                estado[DIST_WE_3] = estado[DIST_GOAL] = (float)D_MAX*4;
        //y los estados angulares en cero. en cualquier numero, algo estandar.
        estado[ANG_BP_1] = estado[ANG_BP_2] =
                estado[ANG_WE_1] = estado[ANG_WE_2] =
                estado[ANG_WE_3] = estado[ANG_GOAL] = estado[ANG_SELF] = 0.0f;

        //se obtiene la información actual del mundo, entre esta se encuentra la posición de los
        //jugadores.
        world_info = strategyState->getSituation();

        myPose = world_info.partners[world_info.ownID-1].getPosition();

        nEnemies = strategyState->map.getNEnemies();
        nPartners = strategyState->map.getNPartner();

        get_best_partners(best_partners);
        get_worst_enemies(worst_enemies, myPose);

        int k = 0;

        for (int i=0;i<N_ROBOT;++i)
        {
                if(best_partners[i] && ownID!=(i+1))
                {
                        //DIST_BP_i
                        estado[k] = (float)(world_info.partners[i].getPosition()-myPose).r();
                        //ANG_BP_i
                        estado[k+1] = (float)(world_info.partners[i].getPosition()-myPose).theta();
                        k+=2;

                }
        }
        k = 4;
        for (int j=0;j<N_ROBOT;++j)
        {
                if(worst_enemies[j])
                {
                        //DIST_WE_j
                        estado[k] = (float)(world_info.opponents[j].getPosition()-myPose).r();
                        //ANG_WE_j
```

```
                        estado[k+1] = (float)(world_info.opponents[j].getPosition()-myPose).theta();
                        k+=2;

                }
        }
        k = 10;
        //estado[k] = (float)(world_info.ball-myPose).r(); //DIST_BALL
        //estado[k+1] = (float)(world_info.ball-myPose).theta(); //ANG_BALL
        //k+=2;

        coordenada2d arco;
        arco.setX((world_info.opponentGoal.left+world_info.opponentGoal.right).x()/2);
        arco.setY((world_info.opponentGoal.left+world_info.opponentGoal.right).y()/2);

        estado[k] = (float)(arco-myPose).r(); //DIST_GOAL
        estado[k+1] = (float)(arco-myPose).theta(); //ANG_GOAL

        estado[k+2] = (float)world_info.partners[world_info.ownID-1].getOrientation(); //ANG_SELF


}

double Q_Strategy::probabilidad_interseccion(const areaang& area,double *bestAngle)const
{
        return 0;
}

void Q_Strategy::selectQObjective(bool golpe)
{
        double e_greedy;
        int rand_acc;
        float eval;


        float result =  Q(estado,(ACCION)0,weights);
        accion = GOLPE1;
        int n_accion = GOLPE1;

        int n_acc = N_ACCIONES;
        if (golpe)
                n_acc = N_GOLPES;

        // Como result ya fue definido como Q para la acción 0, este "for" debería partir desde 1
        for (int i = 1;i<n_acc;i++)
        {
                eval = Q(estado,(ACCION)i,weights);
                if(result<eval)
                {
                        result = eval;
                        n_accion = i;
                        accion = (ACCION)n_accion;
                }
        }
        e_greedy = (double)rand();


        if (e_greedy > (8.0*32767.0/10.0))       //Aca se fija epsilon - greedy
        {
                rand_acc = rand()*n_acc/32767; //RAND_MAX = 32767.0
                accion = (ACCION) rand_acc;
                tot_egreedy++;
        }

        tot_accions[accion]++;

        return;
}

void Q_Strategy::fillIndex(float vector[M_SIZE][N_ACCIONES],
        float estado[N_ESTADOS], int accion, float valor)
{
        fIndex(vector, estado, accion, valor);
}
int Q_Strategy::refreshTable(bool hayGol)
{

        //Esto se hace para actualizar la tabla solo desde la segunda iteracion
        if(firstTime)
        {
                firstTime = false;
        }
        else
        {
                //!nuevo tile la accion fue la que ocurrio entre last_estado y estado
                refreshTiles(estado,accion,estadoAnterior,reward,weights,hayGol);
                reward=0;
```

```
        }
        return 1;
535
}

void Q_Strategy::set_reward(double a)
{
540     reward=a;
}
double Q_Strategy::get_reward()
{
        return reward;
545 }
void Q_Strategy::set_kicker(const StrategyState *pstrat)
{
        kicker=true;
        comunicate = true;
550     lastKickTime = pstrat->timestamp;
        for(int i=0; i<N_ESTADOS; ++i)
        {
                estadoAnterior[i] = estado[i];
        }
555     id_last_partner_ball = world_info.ownID;
        last_time_partner_touch_ball = lastKickTime;
}

void Q_Strategy::reset_kicker()
560 {
        kicker = false;
}

bool Q_Strategy::get_kicker(unsigned long &time_to_wait)
565 {
        time_to_wait = lastKickTime;
        return kicker;
}
void Q_Strategy::Dibujar(CDC *dc, ubicacion *pose) const
570 {


        int bPart[N_PARTNERS], wEne[N_ENEMIES];
        int k=0,j=0;
575     if(nPartners>1) // por ahora solo quiero dibujar a los peores enemigos ->ya no !
        {
                for (int i=0;i<N_ROBOT;i++)
                {
                        if(best_partners[i])
580                     {
                                bPart[k] = i;
                                k++;
                        }
                }
585
                CPen PenAzul(PS_SOLID,2,RGB(0,0,100));
                dc->SelectObject(&PenAzul);

                CPoint BP1[2];
590             BP1[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
                BP1[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                BP1[1].x = (long)world_info.partners[bPart[0]].getPosition().x();
                BP1[1].y = AN-(long)world_info.partners[bPart[0]].getPosition().y();

595             dc->Polyline(BP1, 2);

                CString texBP1;
                texBP1.Format("%.1f",estado[0]);
                dc->TextOut((BP1[0].x+BP1[1].x)/2,(BP1[0].y+BP1[1].y)/2,texBP1);
600
                if(k>1)
                {
                        CPoint BP2[2];
                        BP2[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
605                     BP2[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                        BP2[1].x = (long)world_info.partners[bPart[1]].getPosition().x();
                        BP2[1].y = AN-(long)world_info.partners[bPart[1]].getPosition().y();

                        dc->Polyline(BP2, 2);
610
                        CString texBP2;
                        texBP2.Format("%.1f",estado[2]);
                        dc->TextOut((BP2[0].x+BP2[1].x)/2,(BP2[0].y+BP2[1].y)/2,texBP2);
                }
615     }
        /* else */if(nEnemies)
        {

                for (int i=0;i<N_ROBOT;i++)
```

```
620                     {
                                if(worst_enemies[i])
                                {
                                        wEne[j] = i;
                                        j++;
625                             }
                        }
                        CPen PenRojo(PS_SOLID,2,RGB(100,0,0));
                        dc->SelectObject(&PenRojo);

630                     CPoint WE1[2];
                        WE1[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
                        WE1[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                        WE1[1].x = (long)world_info.opponents[wEne[0]].getPosition().x();
                        WE1[1].y = AN-(long)world_info.opponents[wEne[0]].getPosition().y();
635
                        dc->Polyline(WE1, 2);

                        CString texWE1;
                        texWE1.Format("%.1f",estado[4]);
640                     dc->TextOut((WE1[0].x+WE1[1].x)/2,(WE1[0].y+WE1[1].y)/2,texWE1);

                        if(j>1)
                        {
                                CPoint WE2[2];
645                             WE2[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
                                WE2[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                                WE2[1].x = (long)world_info.opponents[wEne[1]].getPosition().x();
                                WE2[1].y = AN-(long)world_info.opponents[wEne[1]].getPosition().y();
                                dc->Polyline(WE2, 2);
650
                                CString texWE2;
                                texWE2.Format("%.1f",estado[6]);
                                dc->TextOut((WE2[0].x+WE2[1].x)/2,(WE2[0].y+WE2[1].y)/2,texWE2);
                        }
655
                        if(j>2)
                        {
                                CPoint WE3[2];
                                WE3[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
660                             WE3[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                                WE3[1].x = (long)world_info.opponents[wEne[2]].getPosition().x();
                                WE3[1].y = AN-(long)world_info.opponents[wEne[2]].getPosition().y();

                                dc->Polyline(WE3, 2);
665
                                CString texWE3;
                                texWE3.Format("%.1f",estado[8]);
                                dc->TextOut((WE3[0].x+WE3[1].x)/2,(WE3[0].y+WE3[1].y)/2,texWE3);
                        }
670             }
                /*
                CPen PenVerde(PS_SOLID,2,RGB(0,100,0));
                dc->SelectObject(&PenVerde);

675             CPoint DG[2];
                DG[0].x = (long)world_info.partners[world_info.ownID-1].getPosition().x();
                DG[0].y = AN-(long)world_info.partners[world_info.ownID-1].getPosition().y();
                DG[1].x = (long)(world_info.opponentGoal.left+world_info.opponentGoal.right).x()/2;
                DG[1].y = AN-(long)(world_info.opponentGoal.left+world_info.opponentGoal.right).y()/2;
680
                dc->Polyline(DG, 2);
                */

                CString texDIST_GOAL;
685             texDIST_GOAL.Format("DIST_GOAL: %.1f",estado[DIST_GOAL]);
                //dc->TextOut((DG[0].x+DG[1].x)/2,(DG[0].y+DG[1].y)/2,texDIST_GOAL);
                dc->TextOut(AN/12,LA/22-24,texDIST_GOAL);
                CString texANG_SELF;
                texANG_SELF.Format("ANG_SELF: %.1f",estado[ANG_SELF]);
690             dc->TextOut(AN/12,LA/22-2,texANG_SELF);


    }

695 void Q_Strategy::set_world_info()
    {
            world_info = strategyState->getSituation();
    }

700 void Q_Strategy::set_Timestamp(unsigned long time)
    {
            timestamp=time;
    }

705 int Q_Strategy::get_id_accion()
    {
```

```
        return id_accion;
    }

710 Desplazamiento Q_Strategy::get_Desplazamiento()
    {
        return mov_accion;
    }

715 ACCION Q_Strategy::get_accion()
    {
        return accion;
    }

720 void Q_Strategy::set_Desplazmiento(double x,double y, double theta)
    {
        mov_accion = Desplazamiento(x,y,theta);
        return;
    }
725
    void Q_Strategy::set_id_last_partner_ball(int id, unsigned long partner_time)
    {
        id_last_partner_ball=id;
        last_time_partner_touch_ball=partner_time;
730     return;
    }

    StrategyState *Q_Strategy::getStrategyState() const
    {
735     return strategyState;
    }
    void Q_Strategy::resetCounters()
    {
        for(int i=0; i<N_ACCIONES; i++)
740             tot_accions[i] = 0;
        tot_egreedy = 0;
        return;
    }
```

Listing 5.3: B_go_to_ball_and_kick.cpp

```
 1 const kickObjective
        *B_go_to_ball_and_kick::selectKickObjective(const StrategyState *pstrat,
        bool *es_pase, bool calcularPosIdeal) const
   {
        const kickObjective *result;
 6      double min_prob;
   #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
        debugEf.EmpezarTarea("selectKickObjective");
   #endif
        switch (pstrat->objectiveType)
11      {
        case UCH_OBJECTIVE_TYPE_PRIORITIZED:
            min_prob = B_go_to_ball_and_kick_params.minKickSuccessProb;
            result = selectKickObjective(pstrat,es_pase,min_prob);
            #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
16              debugEf.TerminarTarea("selectKickObjective");
            #endif
            if (result!=NULL)
                return result;
            return selectKickObjective(pstrat,es_pase, 0.0);
21      case UCH_OBJECTIVE_TYPE_GENERALIZED:
            if(!pstrat->generalObj.valid)
            {
                #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
                    debugEf.EmpezarTarea("getGeneralizedObjective");
26              #endif
                getGeneralizedObj(pstrat, calcularPosIdeal);
                #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
                    debugEf.TerminarTarea("getGeneralizedObjective");
                #endif
31
                if(!pstrat->QLObj.episodeStart)
                {
                    pstrat->QLObj.set_reward(0.010/*(int)(pstrat->timestamp -
                        pstrat->QLObj.lastActionTime)*/);
36                  pstrat->QLObj.delta = pstrat->QLObj.get_reward()-pstrat->QLObj.qLast;
                    //update weights
                    for (int i=0; i<M_SIZE; i++)
                        for (int j=0; j<N_ACCIONES; j++)
                            pstrat->QLObj.weights[i][j]=pstrat->QLObj.weights[i][j] +
41                              0.4f*(float)pstrat->QLObj.delta*(
                                pstrat->QLObj.e_vector[i][j]);
                                //alpha = 0.4 tasa de aprendizaje

                    pstrat->QLObj.episodeStart= true;
```

```
46                          }
                    }
                    #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
                            debugEf.TerminarTarea("selectKickObjective");
                    #endif
51                  return &pstrat->generalObj;
            case UCH_OBJECTIVE_TYPE_Q_LEARNING:
            {
                    //Si no hay una accion escogida todavia
                    if(!pstrat->QLObj.actionSelected)
56                  {
                            //aca esta la clave de todo el codigo:

                            //Calcula el estado
                            pstrat->QLObj.get_state(pstrat->QLObj.estado);
61                          if(pstrat->QLObj.episodeStart)
                            {
                                    //escoge la accion y almacena el valor de Q(s,a)
                                    getQLObj(pstrat);
                                    pstrat->QLObj.qLast = Q(pstrat->QLObj.estado,
66                                          pstrat->QLObj.accion,pstrat->QLObj.weights);

                                    pstrat->QLObj.lastActionTime = pstrat->timestamp;

                                    for (int i=0; i<M_SIZE; i++)
71                                          for (int j=0; j<N_ACCIONES; j++)
                                                    pstrat->QLObj.e_vector[i][j]=(int)0;

                                    //revisar el hecho de que se esta usando componentes de pstrat
                                    pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
76                                          pstrat->QLObj.estado, pstrat->QLObj.accion, 1.0f);

                                    pstrat->QLObj.episodeStart = false;

                            }
81                          else
                            {
                                    pstrat->QLObj.set_reward(10/*(int)(pstrat->timestamp -
                                            pstrat->QLObj.lastActionTime)*/);

86                                  pstrat->QLObj.delta = pstrat->QLObj.get_reward() - pstrat->QLObj.qLast;

                                    //escoge la accion y almacena el valor de Q(s,a)
                                    getQLObj(pstrat);
                                    pstrat->QLObj.qLast = Q(pstrat->QLObj.estado,
91                                          pstrat->QLObj.accion,pstrat->QLObj.weights);

                                    pstrat->QLObj.lastActionTime = pstrat->timestamp;

                                    pstrat->QLObj.delta  = pstrat->QLObj.delta  +
96                                          pstrat->QLObj.qLast*0.9 ;
                                            //gamma  =  0.9 se usa si hay descuento

                                    //update weights

101                                 for (int i=0; i<M_SIZE; i++)
                                            for (int j=0; j<N_ACCIONES; j++)
                                                    pstrat->QLObj.weights[i][j]=pstrat->QLObj.weights[i][j] +
                                                            0.4f*(float)pstrat->QLObj.delta*(
                                                            pstrat->QLObj.e_vector[i][j]);
106                                                         //alpha = 0.4 tasa de aprendizaje

                                    // se decae el vetor de eligibility traces
                                    for (int i=0; i<M_SIZE; i++)
                                            for (int j=0; j<N_ACCIONES; j++)
111                                                 pstrat->QLObj.e_vector[i][j]=
                                                            0.9f*pstrat->QLObj.e_vector[i][j];
                                                     // lambda = 0.9
                                    // replace traces
                                    for(int i=0; i<N_ACCIONES; i++)
116                                         if(i!=pstrat->QLObj.get_id_accion())
                                                    pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
                                                            pstrat->QLObj.estado, (ACCION)i, 0.0f);

                                    pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
121                                         pstrat->QLObj.estado, pstrat->QLObj.accion, 1.0f);


                            }

126                 }
                    else
                    {
                            return &pstrat->QLObj;
                    }
131
                    return &pstrat->QLObj;
```

```
        }
        case UCH_OBJECTIVE_TYPE_GENERALIZED_MDP:
        case UCH_OBJECTIVE_TYPE_GENERALIZED_MDP_INTERPOLATED:
136     case UCH_OBJECTIVE_TYPE_GENERALIZED_MDP_FORWARD:
        default:
                if(!pstrat->MDPObj.valid)
                {
                        #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
141                             debugEf.EmpezarTarea("getMDPObjective");
                        #endif
                        getMDPObj(pstrat);
                        #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
                                debugEf.TerminarTarea("getGeneralizedObjective");
146                     #endif

                        #ifdef MODO_CARGA_MDP

                                        //Calcula el estado
151                     pstrat->QLObj.get_state(pstrat->QLObj.estado);
                        if(pstrat->QLObj.episodeStart)
                        {
                                //escoge la accion y almacena el valor de Q(s,a)
                                //getQLObj(pstrat);
156                             pstrat->QLObj.qLast = Q(pstrat->QLObj.estado,
                                        pstrat->QLObj.accion,pstrat->QLObj.weights);

                                pstrat->QLObj.lastActionTime = pstrat->timestamp;

161                             for (int i=0; i<M_SIZE; i++)
                                        for (int j=0; j<N_ACCIONES; j++)
                                                pstrat->QLObj.e_vector[i][j]=(int)0;

                                //revisar el hecho de que se esta usando componentes de pstrat
166                             pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
                                        pstrat->QLObj.estado, pstrat->QLObj.accion, 1.0f);

                                pstrat->QLObj.episodeStart = false;

171                     }
                        else
                        {
                                pstrat->QLObj.set_reward(10/*(int)(pstrat->timestamp -
                                        pstrat->QLObj.lastActionTime)*/);
176
                                pstrat->QLObj.delta = pstrat->QLObj.get_reward() - pstrat->QLObj.qLast;

                                //escoge la accion y almacena el valor de Q(s,a)
                                //getQLObj(pstrat);
181                             pstrat->QLObj.qLast = Q(pstrat->QLObj.estado,
                                        pstrat->QLObj.accion,pstrat->QLObj.weights);

                                pstrat->QLObj.lastActionTime = pstrat->timestamp;

186                             pstrat->QLObj.delta  = pstrat->QLObj.delta  +
                                        pstrat->QLObj.qLast*0.9 + 0.01 ;
                                        //gamma  =  0.9 se usa si hay descuento

                                //update weights
191
                                for (int i=0; i<M_SIZE; i++)
                                        for (int j=0; j<N_ACCIONES; j++)
                                                pstrat->QLObj.weights[i][j]=pstrat->QLObj.weights[i][j] +
                                                        0.4f*(float)pstrat->QLObj.delta*(
196                                                     pstrat->QLObj.e_vector[i][j]);
                                                        //alpha = 0.4 tasa de aprendizaje

                                // se decae el vetor de eligibility traces
                                for (int i=0; i<M_SIZE; i++)
201                                     for (int j=0; j<N_ACCIONES; j++)
                                                pstrat->QLObj.e_vector[i][j]=
                                                0.9f*pstrat->QLObj.e_vector[i][j];
                                                // lambda = 0.9
                                // replace traces
206                             for(int i=0; i<N_ACCIONES; i++)
                                        if(i!=pstrat->QLObj.get_id_accion())
                                                pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
                                                        pstrat->QLObj.estado, (ACCION)i, 0.0f);

211                             pstrat->QLObj.fillIndex(pstrat->QLObj.e_vector,
                                        pstrat->QLObj.estado, pstrat->QLObj.accion, 1.0f);


                        }
216                     #endif
```

```
221
                    }
                    #ifdef DEBUG_EFICIENCIA_GO_TO_BALL_AND_KICK
226                         debugEf.TerminarTarea("selectKickObjective");
                    #endif
                    //Calcula el estado
                //pstrat->QLObj.get_state(pstrat->QLObj.estado);

231                 //Actualiza la tabla para el par (s,a) antiguo (con a = Q_Strategy::accion)
                    //pstrat->QLObj.refreshTable(false);
                    return &pstrat->MDPObj;
            }
    }
```