UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Aplicación de técnicas de aprendizaje automático sobre juegos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Guillem Aguado Sarrió

Tutor: César Ferri Ramírez

Curso Académico: 2014-2015

Resumen

En el presente proyecto se pretende utilizar y ampliar la plataforma RL-GGP (Reinforcement Learning General Game Playing), para poder disponer en ella de una herramienta con la que realizar todo tipo de experimentos de algoritmos de Aprendizaje por Refuerzo con agentes jugando juntos a un juego por turnos especificado en lenguaje GDL (Game Description Language).

La plataforma usa el GGP-Server, para mediante un juego en GDL, realizar partidas con diversos agentes creados con la herramienta de jugadores de juego genéricos Jocular, con la interfaz RL-Glue (Reinforcement Learning Glue), que hace de puente entre Jocular y los algoritmos.

Estos algoritmos están adaptados de la librería de Hado Van Hasselt en C++, pero migrados a java. Había tres algoritmos en la implementación, Q-Learning, SARSA y QV-Learning. Se han introducido otros muy conocidos como Acla, Expected Sarsa y Cacla, y se ha adaptado la implementación de RL-GGP a ellos, y se ha modificado hasta cierto punto para hacer más fácil la experimentación general con varios agentes y algoritmos.

Se han probado cinco de los seis algoritmos en el juego Tictactoe, que presenta un árbol de estados no demasiado grande, y en Clobber, que presenta un árbol mucho más grande. Estas pruebas se han hecho realizando series de partidas con diferentes algoritmos, en diferentes juegos, y con diferentes parámetros. Se ha excluido a Cacla de las pruebas, pero no de la implementación, porque a día de hoy la plataforma RL-GGP no maneja acciones continuas, y los juegos para test tienen acciones de naturaleza puramente discreta, pero se ha implementado por si en el futuro alguien lo deseara usar para realizar nuevos experimentos, implementando el manejo de acciones continuas en RL-GGP.

Posteriormente a los experimentos, se han recogido, compilado y mostrado los datos y las conclusiones extraídas.

Palabras clave: agente, algoritmo, Aprendizaje por Refuerzo, RL-GGP, Hado Van Hasselt, RL-Glue, GGP, GGP-Server, Q-Learning, QV-Learning, Sarsa, Acla, Expected Sarsa.

Tabla de contenidos

1. Introducción

En esta sección se define la finalidad y motivación del presente proyecto.

1.1. Motivación

Este Proyecto fue ideado y creado con la intención de poder aplicar diferentes algoritmos de aprendizaje por refuerzo en agentes que juegan a juegos por turnos de diversa índole, de modo que se puedan extraer conclusiones de que se debe hacer si uno pretende crear un agente que aprenda de su entorno y de su propia experiencia en el contexto de los juegos por turnos.

Dicha tarea requiere herramientas capaces de integrar agentes software que integren los algoritmos y comportamientos que se quieran modelar, con tecnologías que permitan la ejecución de dichos agentes en un entorno donde puedan jugar a juegos genéricos con otros agentes que, pueden tener también implementaciones de algoritmos de aprendizaje o no.

Como se ha podido disponer de una herramienta de ejecución de agentes con algoritmos de aprendizaje por refuerzo en juegos por turnos como es RL-GGP, se ha decidido ampliar esta tecnología con más algoritmos (ya que solo dispone de SARSA, Q-Learning y QV-Learning), para posteriormente realizar el estudio utilizando diversos algoritmos, juegos, parámetros de los algoritmos, etc.

1.2. Contexto

Inicialmente, el proyecto fue concebido para aplicar técnicas de aprendizaje en sentido amplio a todo tipo de juegos, no obstante se optó por la rama del aprendizaje por refuerzo, ya que es una buena forma de crear agentes que se desenvuelvan y aprendan en un entorno de juegos, en el que deben aprender las estrategias y políticas necesarias para ganar, siendo tal el objetivo principal del aprendizaje por refuerzo, en el cual los agentes aprenden de su entorno la forma de optimizar las recompensas del mismo al ejecutar sus acciones, es decir, aprenden las estrategias para seguir el juego y ganar.

Mediante el uso de la plataforma RL-GGP y diversos algoritmos de aprendizaje, tanto los que tenía la propia plataforma como otros nuevos, se ha podido realizar experimentos sobre juegos por turnos descritos en lenguaje GDL para probar las características de cada algoritmo e intentar determinar cuando usarlos, con que parámetros como el factor de aprendizaje y qué tipo de elección se realiza de las acciones en base a los conocimientos del agente (acción avariciosa, exploración, mixta, etc.).

1.3. Objetivos

Con el presente proyecto lo que se persigue es, en definitiva, intentar sacar el mayor número de conclusiones útiles sobre qué se debe utilizar, cómo y cuándo para lograr que los agentes software aprendan y actúen de la forma más inteligente posible en los juegos por turnos, como es el caso de estudio.

También se pretende mejorar la plataforma RL-GGP para que en el futuro sea más útil para la gente que desee realizar más experimentos próximamente con ella, y de este modo se avance más en el campo.

2. Contexto teórico

En esta sección se van a exponer los conocimientos básicos sobre agentes, entornos y aprendizaje por refuerzo, necesarios para las siguientes secciones.

2.1. Juegos y Agentes

Un juego es una actividad divertida y que provoca disfrute en su participación, sujeto a una serie de reglas que se deben seguir, en muchos casos tiene fines educativos.

Se suele diferenciar de otras actividades sujetas a reglas como el trabajo por la finalidad de divertirse y disfrutar justamente, aunque no siempre es así.

En el contexto de la informática, más concretamente en los sistemas multiagente, un juego consta de un entorno software con otros programas que son los agentes, y que se desenvuelven el entorno, el cual hace de nexo de unión entre ellos, realizando las acciones que han sido programados para hacer de modo que puedan ganar o interaccionar con otros agentes.

Los entornos multiagente, o sistemas multiagente (SMA), es un sistema o entorno compuesto por múltiples agentes inteligentes que interaccionan entre ellos.

Los agentes software son programas, o bien fragmentos de ellos que realizan acciones en el entorno multiagente.

Tienen una función específica en el marco del sistema, y la realizan con independencia de los demás agentes, aunque pueden estar influidos por ellos.

Existen infinidad de tipos de agente, y también de lenguajes con los que programarlos, estos agentes pueden ser tanto elementos de un enjambre (bandadas de pájaros, peces, etc.), como partes de un mismo robot que interaccionan entre ellas y con el entorno para realizar las acciones del mismo.

Estos entornos que son el juego en el que se desenvuelven los agentes, son sistemas multiagente que suelen ser o bien programas que ejecutan a los agentes por sí mismos como sub procesos, o bien un servidor que recibe información de otros procesos que son los agentes, y gestiona el juego en sí.

Los agentes pueden comunicarse entre ellos, razonar, realizar todo tipo de tareas sociales que se les hayan asignado y que estén programados para hacer. Sus procesos de razonamiento pueden estar guiados por diversos factores, como por ejemplo:

- Aprendizaje de su entorno (Aprendizaje por refuerzo)
- Razonamiento lógico o inferencia derivada de lo que perciben, saben, desean, etc.
- Cualquier otro factor que pueda estar presente en el agente.

2.2. Entornos

Un juego se desarrolla en un entorno. Los entornos son, en definitiva, el proceso o programa que acoge a los diferentes agentes que interaccionan entre si, ya sean agentes locales, remotos, propios del mismo entorno, etc.

Para definir un entorno se utilizan diversas tecnologías, en el caso de estudio se ha usado GGP-Server, integrado dentro de RL-GGP, que actúa como servidor para desarrollar el juego y recibir/enviar datos a los agentes o jugadores, haciendo de puente entre ellos y el propio juego, el cual se define mediante reglas lógicas en lenguaje GDL y es interpretado por GGP.

Hay otras muchas tecnologías para gestión de sistemas multiagente, como por ejemplo JADE.

El entorno en el que se mueven los agentes puede ser de muchos tipos, en nuestro caso va a ser un sistema multiagente que sera donde se gestionará una partida entre uno o varios agentes y que recibirá las acciones de los agentes para devolverles una recompensa conforme a lo que han hecho, positiva o negativa (dependiendo de si es una buena acción en las circunstancias en que se da o no para el propio juego).

Para que los jugadores puedan realizar partidas sin problemas, el servidor o programa que gestiona el juego debe implementar protocolos de comunicación que permitan la sincronización en tiempo real de los agentes, dado que las partidas se realizan en orden y en tiempo real, y todas las acciones deben suceder en su momento, estando listos todos los agentes al mismo tiempo para recibir la información del nuevo estado de la partida.

Los entornos pueden ser de dos tipos:

- Naturaleza discreta: Son los que se suelen usar en juegos por turnos descritos en GDL como los del caso de estudio, los estados y acciones son valores discretos que se pasan como argumentos entre el entorno y los agentes.

- Naturaleza continua: Los estados y las acciones son continuos, puede que solo las acciones lo sean, o solo los estados, pero estos pueden ademas tener varias dimensiones.

Un entorno describe las características exactas de lo que los agentes pueden percibir durante su estancia en el, generalmente son estados del mismo entorno o juego.

2.3. Aprendizaje por Refuerzo (RL)

El Aprendizaje por Refuerzo es una rama del Aprendizaje Automático, que a su vez es una rama de la Inteligencia Artificial.

A diferencia de las técnicas clásicas del Aprendizaje Automático, lo que se busca en Aprendizaje por Refuerzo no es dar la salida adecuada a nuevos datos de entrada después de un largo entrenamiento con muchos datos de entrada, sino mas bien saber que acción realizar en un entorno y situación dadas para maximizar la acumulación de recompensas dadas por el entorno a estas acciones en el futuro (tras muchas ejecuciones).

El problema ha sido estudiado en muchos otros campos debido a su generalidad, como son por ejemplo la teoría de juegos, teoría de control, investigación operativa, teoría de la información, la optimización basada en la simulación, estadísticas y algoritmos genéticos.

En el Aprendizaje Automático en general, el entorno se suele modelar como un proceso de decisión de Markov (MDP). Un MDP es un proceso de control estocástico discreto en el tiempo, en el que un agente se encuentra en un entorno cambiante y debe tomar una acción en un estado, para que el entorno cambie a otro estado al ejecutar la acción y le de una recompensa de acuerdo a ella.

Los algoritmos de Aprendizaje por Refuerzo no necesitan conocer el MDP, por tanto pueden utilizarse en cualquier MDP que se les presente, a diferencia de otros métodos exactos como los de la programación dinámica.

En el problema del Aprendizaje por Refuerzo, el agente o el sujeto que debe aprender, se sitúa en un MDP y recibe una señal del entorno que representa el estado actual, de modo que a través de sus conocimientos adquiridos debe tomar una decisión de acuerdo con cierta política y actualizar sus conocimientos en consonancia a los resultados que se observen en el entorno a causa de su acción (una recompensa y la señal de un nuevo estado), siguiendo cierto algoritmo de Aprendizaje por Refuerzo. La política a la hora de elegir una acción también debe ser escogida, además del algoritmo de aprendizaje y sus parámetros, de modo que se garanticen buenos resultados. Comúnmente suele escogerse una política que explote los conocimientos adquiridos del agente pero que a la vez también permita la exploración de nuevas acciones, que conlleven más aprendizaje. Este es el llamado equilibrio entre explotación y exploración, que siempre debe tenerse en cuenta a la hora de crear un agente que se pretenda que aprenda que acciones realizar en un entorno cambiante.

Como se ha comentado, los métodos exactos no son de utilidad en un MDP, ya que el entorno cambiante no permite que estos métodos cumplan todas sus asunciones, de modo que los algoritmos que ofrece el Aprendizaje por Refuerzo son más generales, y no realizan tantas asunciones estrictas que se deban cumplir.

2.4. Algoritmos

Nota: esta sección ha sido adaptada de
http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Notation.html

Los algoritmos son la base del aprendizaje de los agentes en su entorno, los utilizan en su función de actualización tras cada acción en el juego para aprender de sus errores y de los logros.

En este caso de estudio, nos interesan los principales algoritmos de aprendizaje por refuerzo, en concreto se presentarán Q-Learning, SARSA, QV-Learning, Expected Sarsa, Acla y Cacla, que son los que están en la implementación.

Primero explicaremos la notación a usar:

Cuando un valor o función es actualizado, usaremos la siguiente notación para indicarlo:

$$A_{t+1}(x) \xleftarrow{\alpha} B_t$$

Esto significa que, el valor de A, que depende de cierto parámetro de entrada x, se actualiza hacia el valor B. Los sub índices significan instantes en el tiempo, haciendo de esta una formulación discreta en el tiempo. α es simplemente un ratio o parámetro de aprendizaje: 0<=α<=1, que indica en que medida A se acerca a B.

También se ha utilizado el parámetro rt, que es la recompensa en un instante de tiempo, y γ es el denominado factor de descuento.

Q-Learning

Q-Learning es un algoritmo de los mas conocidos de Aprendizaje por Refuerzo, además de ser uno de los mas viejos que existen, su ecuación de actualización es la siguiente:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma \max_a Q_t(s_{t+1}, a)$$

Q-Learning actualiza el valor de ejecutar una acción en un estado mediante la recompensa y el valor estado-acción de la acción optima en el estado siguiente, de modo que sin importar la acción tomada en el estado St+1, el valor de tomar la acción que nos ha llevado al estado actual, estando en St se actualiza positivamente si la suma de la recompensa con el valor de ejecutar la acción optima en el estado St+1 es positiva también.

Esto repercute en que se actualiza la matriz Q de valores sin tener en cuenta la política actual de toma de acciones, asumiendo que se realiza una elección optima siempre.

Esto puede ser útil quizás en algunos casos en los que la naturaleza del experimento haga que la política de elección de acciones no sera relevante, por ejemplo, pero también es peligroso dado que si esta política a veces nos lleva a estados en los que podríamos ir a otros en los que se pierde la partida, como se asume política optima, si existe otro camino bueno se considerará que ir al dicho estado era una buena acción, dado que ahora seguiríamos el camino útil (siguiendo política optima), pero si no tenemos en cambio una política que

siempre toma la acción optima podríamos perder la partida, esto es, tomar la acción que nos lleve al estado en el que perdemos.

El algoritmo es el siguiente:

Initialize $Q_0(s, a)$, for all s, a
Select s_0
For each step $t = 0, 1, 2, \ldots$:
  Derive a policy π_t from Q_t (i.e. with exploration)
  Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t$
    Select new s_{t+1} (starting point for next episode)
  else:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma \max_a Q_t(s_{t+1}, a)$

Fig. 1. Esquema algoritmo Q-Learning. Tomado de *http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Q.html*

Como se puede ver, el algoritmo inicializa una matriz de valores estado-acción, para todo estado y acción, posteriormente entra en su bucle de funcionamiento en el que, primero se deriva una política, con exploración o explotación de conocimientos, mixta, etc., luego se escoge una acción de acuerdo con tal política, y posteriormente se observan el estado siguiente y la recompensa del entorno. Con todo esto, se procede a actualizar la matriz Q (valores estado-acción) conforme a lo visto anteriormente en la ecuación de actualización de Q-Learning, teniendo siempre en cuenta que si nos encontramos al final del episodio, esto es, al final de la partida en nuestro caso, simplemente se actualiza teniendo en cuenta la recompensa, ya que no hay estado siguiente.

SARSA

Sarsa es muy parecido a Q-Learning, la función de actualización cambia simplemente en que se escoge la acción actual tomada para tenerla, en vez de la acción optima:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma Q_t(s_{t+1}, a_{t+1})$$

En este caso, dado que sí tenemos en cuenta la política de elección de acciones, el algoritmo o el agente aprenderá que por ciertos caminos puede haber ciertos peligros, dado que en un estado puede haber caminos siguientes buenos y malos, pero si siempre tenemos en cuenta la acción optima no vemos los caminos malos, como era el caso de Q-Learning, en cambio en Sarsa como se puede ver no existe este problema.

El algoritmo es el que sigue:

Initialize $Q(s, a)$, for all s, a
Select s_0
Derive a policy π_0 from the Q-values (i.e. with exploration)
Select a_0 according to π_0 (i.e. probability of selecting a is $\pi_0(s_0, a)$)
For each step $t = 0, 1, 2, \ldots$:
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    Select new s_{t+1} (starting point for next episode)
  Derive a policy π_t from the Q-values (i.e. with exploration)
  Select a_{t+1} according to π_t (i.e. probability of selecting a is $\pi_t(s_{t+1}, a)$)
  If s_{t+1} is terminal:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t$
  else:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma Q_t(s_{t+1}, a_{t+1})$

Fig. 2. Esquema algoritmo Sarsa. Tomado de
http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Sarsa.html

Podemos apreciar que es muy parecido a Q-Learning, simplemente teniendo en cuenta la diferencia con la actualización, y que al necesitar la acción actual para la actualización, tenemos que replantear el esquema para primero tomar una acción, luego ya entrar en un bucle en que vemos el estado y recompensa, tomamos otra acción y actualizamos, hasta el fin del episodio/partida.

QV-Learning

QV-Learning es un algoritmo que tiene características similares a los anteriores, pero difiere en su fórmula de actualización porque utiliza el valor de los estados, valor que se calcula él mismo al mismo tiempo que actualiza Q, en vez de los valores de Q de tomar la acción at+1 en el estado St+1 (Sarsa), o bien la acción optima en St+1 (Q-Learning).

La función de actualización es la que sigue:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma V_t(s_{t+1})$$

Estos valores de estado se calculan con el valor del estado siguiente al que hemos llegado al realizar una acción, y con la recompensa, de modo similar a como se hacía con las actualizaciones de Q en los algoritmos anteriores.

El algoritmo es el siguiente:

Initialize $Q_0(s, a)$, for all s, a
Initialize $V_0(s)$, for all s
Select s_0
For each step $t = 0, 1, 2, \ldots$:
  Derive a policy π_t from Q_t (i.e. with exploration)
  Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t$
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$
    Select new s_{t+1} (starting point for next episode)
  else:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma V_t(s_{t+1})$
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1})$

Fig. 3. Esquema algoritmo QV-Learning. Tomado de *http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/QV.html*

Podemos decir que el uso de valores de estado hace decrecer la varianza con respecto a los anteriores algoritmos.

El uso de estos valores de estado puede hacer más rápido también el aprendizaje.

Expected Sarsa

Este algoritmo es el mismo que Sarsa, solo que simplemente cambiamos el valor para la actualización del valor Q anterior por un valor suma de todos los valores de ejecutar cada acción de la política actual en St+1, ponderado con su probabilidad de ser elegida. Esta política se puede derivar dos veces (antes de escoger la acción y antes de actualizar Q) o solo antes de elegir acción.

Su función de actualización:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma \sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a)$$

El sumatorio nos muestra como se suman los valores de cada acción posible en St+1 según la política, ponderados con su probabilidad, y este es el valor, junto con la recompensa actual, que se utiliza para la actualización de la matriz Q.

He aquí el algoritmo:

Initialize $Q_0(s, a)$, for all s, a
Select s_0
For each step $t = 0, 1, 2, \ldots$:
  Derive a policy π_t from Q_t (i.e. with exploration)
  Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t$
    Select new s_{t+1} (starting point for next episode)
  else:
    Derive a policy π_t from Q_t (i.e. with exploration)
    $Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma \sum_a \pi_t(s_{t+1}, a) Q_t(s_{t+1}, a)$

Fig. 4. Esquema algoritmo Expected Sarsa. Tomado de http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/ESarsa.html

Se puede considerar una mejora de Sarsa, dado que el concepto es el mismo, salvo que al realizar esa suma ponderada en vez de la acción actual, reduce la varianza.

Acla

Acla actualiza valores estado-acción conforme a la mejora del valor del nuevo estado conforme al viejo, es decir, de St+1 conforme a St. Si esta mejora es positiva, la acción fue una buena acción, por tanto se actualiza hacia 1:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 1$$

Si la mejora es negativa, la acción no fue buena idea por tanto se actualiza el valor estado-acción hacia 0:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 0$$

Nótese que es un algoritmo similar a QV-Learning, solo que esta vez no actualizamos siempre el valor estado-acción sumándole la diferencia entre valores de estado, sino que mejoramos hacia la unidad o retrocedemos hacia 0 dependiendo de si esta diferencia es positiva o negativa, esto es, si nos hallamos en un estado mejor que antes o no.

El algoritmo:

Initialize $P_0(s, a)$, for all s, a
Initialize $V_0(s)$, for all s
Select s_0
For each step $t = 0, 1, 2, \ldots$:
  Derive a policy π_t from P_t (i.e. with exploration)
  Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$
    Select new s_{t+1} (starting point for next episode)
  else:
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1})$
  If $V_{t+1}(s_t) > V_t(s_t)$:
    $P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 1$
  else:
    $P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 0$

Fig. 5. Esquema algoritmo Acla. Tomado de *http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Acla.html*

Este algoritmo puede ser mejor en el caso en que se necesite ir a estados en que hay habitualmente buenas acciones a partir de ellos.

Cacla

Cacla es parecido a Acla, pero cambia la actualización a la acción tomada en St en vez de un valor estado-acción, es decir, solo guarda la acción, no ningún valor. Ademas solo actualiza si el valor del estado mejora con respecto al anterior:

$$A_{t+1}(s_t) \xleftarrow{\alpha_t} a_t$$

Es posiblemente menos aplicable en problemas con espacios de acciones discretos, por tanto es un algoritmo más aplicable a espacios de acciones continuas.

Puede verse como el Acla para espacios de acciones continuas.

El algoritmo:

Initialize $A_0(s)$, for all s
Initialize $V_0(s)$, for all s
Select s_0
For each step $t = 0, 1, 2, \ldots$:
  Select a_t with exploration around $A_t(s_t)$ (i.e. gaussian exploration)
  Perform a_t, observe r_t, s_{t+1}
  If s_{t+1} is terminal:
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$
    Select new s_{t+1} (starting point for next episode)
  else:
    $V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1})$
  If $V_{t+1}(s_t) > V_t(s_t)$:
    $A_{t+1}(s_t) \xleftarrow{\alpha_t} a_t$

Fig. 6. Esquema algoritmo Cacla. Tomado de http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Cacla.html

Este algoritmo puede no ser una buena opción cuando la acción tomada, aunque nos conduzca a un estado con valor mayor, no sea la acción ideal.

3. Herramientas y tecnologías

En esta sección se va a exponer y analizar las diferentes herramientas y tecnologías usadas en el desarrollo del presente proyecto, y se hará una comparativa con otras tecnologías.

3.1. RL-GGP y sus componentes

RL-GGP es una plataforma para usar y evaluar algoritmos de Aprendizaje por Refuerzo en diferentes tipos de juegos por turnos. Estos juegos están definidos mediante el estándar de definición de juegos GDL (Game Description Languaje), y son juegos tales como el tres en raya, conecta cuatro, damas, etc.

El esquema de la arquitectura de RL-GPP es el siguiente que se muestra en la imagen:

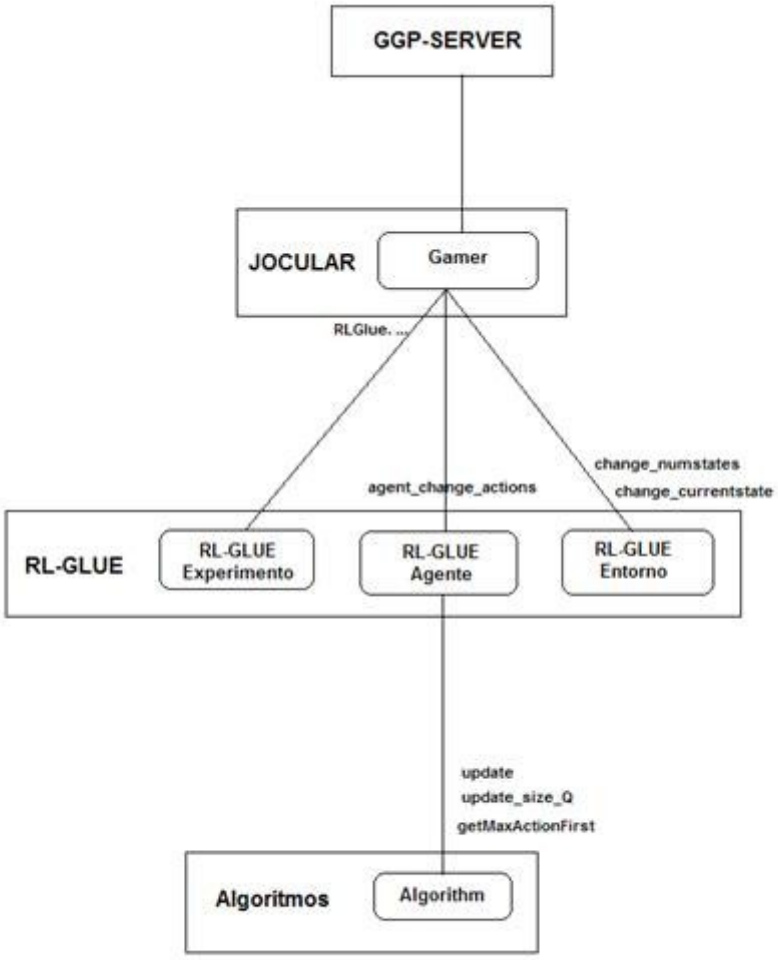


Fig.7.Esquema arquitectura de RL-GGP. Tomado de http://users.dsic.upv.es/~flip/RLGGP/

El servidor GGP-Server ejecuta partidas, cargando un juego en GDL. Uno, dos, cero o los jugadores que se desee, que son jugadores Jocular en la implementación de RL-GGP, se conectan al servidor y juegan las partidas que quieran.

El jugador Jocular crea un agente de la clase SkeletonAgent, que es la clase RLGlue que implementa el agente.

SkeletonAgent a su vez, maneja un algoritmo de Aprendizaje por Refuerzo de la librería de Hado Van Hasselt o uno cualquiera que se implemente, y lo aplica en todas las fases de juego, generando los datos del aprendizaje en ficheros que se generan en la carpeta donde está el agente compilado ejecutándose.

También existe la clase SkeletonEnvironment, que modela el entorno en que se mueve el agente.

RLGue no es mas que un códec que se utiliza para la implementación de un agente de Aprendizaje por Refuerzo sobre Jocular.

3.2. Eclipse y entornos

Para el desarrollo del agente y los algoritmos, así como de la clase Gamer que contiene el experimento, se ha utilizado el entorno Eclipse.

Eclipse es un programa compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar aplicaciones.

Es un entorno de programación y está desarrollado en Java, y su última versión estable es la 4,5 a 24 de Junio de 2015.

Admite el uso de “Plugins”, que permiten por ejemplo añadir control de versiones con Subversion e integración con Hibernate. Además tiene pruebas unitarias con JUnit.

También se ha utilizado Microsoft Visual Studio 2012, el cual resultaba también útil para examinar las librerías en C++ de Hado Van Hasselt.

Existe en la actualidad muchos entornos de programación, pero se ha escogido Eclipse para la codificación y compilación de agentes por su comodidad y porque la implementación de RL-GGP también esta realizada en Eclipse.

3.3. Librerías

Las librerías son muy útiles para la programación de programas avanzados, en concreto para desarrollar el agente que se va a usar se han utilizado las librerías de RLGue.

RLGlue proporciona una el interfaz AgentInterface y EnvironmentInterface, que son implementados por SkeletonAgent y SkeletonEnvironment respectivamente, las cuales son las clases que representan o modelan el agente y el entorno.

Los algoritmos de Aprendizaje por Refuerzo se han migrado de C++ a Java de la librería de algoritmos de Hado Van Hasselt en C++, otros algoritmos que se han utilizado ya estaban en la implementación.

3.4. Comparativa con otras tecnologías

Existen muchas otras herramientas para el desarrollo de agentes y de algoritmos de Aprendizaje por Refuerzo, así como entornos para la ejecución de varios agentes en un sistema multiagente (SMA).

Para el desarrollo de agentes existe JASON, por ejemplo.

JASON es un intérprete basado en Java de una versión extendida de AgentSpeak. AgentSpeak, a su vez, es un lenguaje de programación basado en agentes.

Con esta tecnología se puede realizar todo tipo de agentes que implementen los algoritmos o estrategias que se quiera, y estos agentes pueden integrarse en un sistema multiagente sin problemas.

En el caso del servidor para manejar el SMA, se puede usar JADE, que es una plataforma “open source” para mantener aplicaciones peer-to-peer basadas en agentes. De modo que uno puede conectar varios agentes a la plataforma jade y que desarrollen sus actividades en el SMA que gestiona JADE, claro esta, estas actividades pueden ser juegos.

La plataforma JADE soporta la coordinación de múltiples agentes FIPA, y proporciona una implementación estándar del lenguaje de comunicación FIPA-ACL, que facilita la comunicación entre agentes y permite la detección de servicios que se proporcionan en el sistema.

En el caso de entornos de programación, existen muchos actualmente en el mercado, como por ejemplo Emacs, Vim, Gvim, Med, jEdit, Eclipse, Microsoft Visual Studio, etc.

Se ha utilizado Ecplipse para el desarrollo y modificación por lo ya mencionado antes, y Microsoft Visual Studio por su comodidad para ver cierto código de la librería de Hado Van Hasselt en C++.

En el caso de RLGlue, se ha utilizado porque proporciona una buena interfaz y una buena base en las clases SkeletonAgent y SkeletonEnvironment para el desarrollo de agentes con algoritmos de aprendizaje, en concreto es muy fácil desarrollar agentes con algoritmos de Aprendizaje por Refuerzo, por el modelado del entorno y del agente que proporcionan estas clases.

Se ha utilizado la tecnología de Jocular y GGP-Server por ser la primera en Java y muy fácilmente adaptable al agente RLGlue, y la segunda por ser cómoda en la ejecución de partidas sencillas de juegos por turnos, como es el caso de estudio. No se necesita una interfaz gráfica que muestre el desarrollo de las partidas, además, RL-GGP que es el sistema base que se ha ampliado y utilizado ya utiliza GGP-Server, Jocular y RLGlue.

4. Diseño del sistema

En esta sección se va a definir cómo se ha realizado el diseño y la implementación del sistema final a partir de la base de RL-GGP.

4.1. Esquema de diseño

RL-GGP ya dispone de tres algoritmos básicos de Aprendizaje por Refuerzo, que son Q-Learning, SARSA y QV-Learning. Se ha ampliado este repositorio de algoritmos, de modo que ahora son seis, estos seis algoritmos se usan mediante el fichero config_agent.cgf, que se encuentra al lado del agente compilado (fichero.jar), que solo contiene un número, y es el número del algoritmo que usará el agente. Los valores a usar son:

SARSA	Q-Learning	QV-Learning
0	1	2
Acla	Cacla	Expected Sarsa
3	4	5

Además, también se han Implementado los métodos de selección de acción para acciones continuas y se ha modificado SkeletonAgent para manejar acciones continuas.

También se ha adaptado un poco SkeletonAgent para hacer más cómodas las pruebas, como por ejemplo poniendo el parámetro de ratio de aprendizaje como variable global, de forma que solo haga falta modificarlo una vez para cambiarlo.

Se ha creado funciones para la persistencia y carga de un vector V de valores de estado, además de las de la matriz Q que ya existían.

4.2. Implementación del sistema

El sistema se ha implementado siguiendo las anteriores pautas de la siguiente forma:

Se ha extendido la clase Algorithm para crear los tres algoritmos nuevos, de modo que a partir de ahí se han programado, realizando una migración de C++ de la librería del profesor Hado Van Hasselt.

Los métodos de actualización, inicialización, valores del algoritmo, actualización de estructuras (como Q o V), han sido programados para los tres algoritmos.

También se ha modificado la clase Algorithm para incluir estructuras de datos necesarias para los nuevos algoritmos.

Se ha adaptado SkeletonAgent, además de para hacerlo más cómodo para la realización de experimentos, para poder manejar los nuevos algoritmos y al menos, poder manejar los datos de acciones continuas que da o recibe Cacla.

Se ha implementado también la persistencia del vector de valores de estados V, tanto el método para su guardado como el de su carga.

El algoritmo QV-Learning no poseía persistencia para su vector V, y se ha decidido no implementarla para él, dado que al ser un algoritmo con características no iguales pero sí similares a Acla, el cual si que tiene persistencia del vector V, se podrá probar la influencia de implementar o no persistencia en el vector V, midiendo el rendimiento de estos dos algoritmos en el test.

Se han implementado los métodos que generan acciones de tipo continuo para Cacla, tanto el método getMaxActionC, que proporciona la acción voraz o avariciosa, como el método getActionRandomC, que proporciona una acción aleatoria y el método egreedyC, que da la funcionalidad de política de elección de acciones pseudo-voraz o pseudo-avariciosa.

El algoritmo Cacla ha sido totalmente implementado, pero no sujeto a test, dado que la plataforma no maneja acciones y estados continuos, y además, los juegos por turnos que se usan para test son de naturaleza puramente discreta en sus estados y acciones. Cacla no tiene demasiado sentido aplicado a entornos de acciones discretas.

4.3. Posibles ampliaciones

En el futuro se podría realizar varias ampliaciones que harían que la plataforma RL-GGP fuera mucho más útil y ampliaría horizontes para la prueba de diferentes algentes con diferentes algoritmos de Aprendizaje por Refuerzo.

Por ejemplo, sería muy interesante conseguir ajustar las recompensas del entorno de algún modo de forma genérica para todos los juegos que se pudieran presentar, de modo que estas recompensas fueran en relación con los objetivos del juego. Este es un cuello de botella del entorno RL-GGP, que no tiene una solución fácil, dado que cada juego tiene sus singularidades y no hay un patrón genérico para todos los juegos que se pueda aplicar de forma general a las recompensas, y no se debe modificar la plataforma para un juego en concreto, dado que ésta perdería su generalidad.

Si se ajustaran mejor de algún modo las recompensas del entorno, los agentes podrían aprender mejor las estrategias de victoria de los juegos.

También es interesante dotar a la plataforma de manejo de acciones y estados discretos, ya que el agente SkeletonAgent actualmente solo le proporciona estados discretos y acciones discretas (y los recibe), y los algoritmos no tienen una implementación para el caso de estados continuos.

En el campo de los algoritmos, ya se han implementado la mayoría de los principales algoritmos de Aprendizaje por Refuerzo, incluso Cacla, que es para el caso de acciones continuas, aunque se podrían añadir variantes.

5. Experimento

En esta sección se va a exponer el experimento realizado con las herramientas descritas, para analizar su realización y resultados y posteriormente poder sacar conclusiones.

5.1. Juegos para los test

Los dos juegos que se van a utilizar en los test son Tictactoe y Clobber.

Tictactoe: Es el clásico tres en raya, con un tablero de 3x3. Este juego se ha escogido, entre otros motivos, porque no tiene un número de acciones y estados posibles y se pretende ver el rendimiento de los algoritmos en un entorno como ése.

Clobber: Este es un juego que mezcla azar con profundos conocimientos estratégicos, se sitúan un número igual de piezas de un color y de otro en un tablero de forma que unas ocupan la mitad de las celdas del tablero y las otras la otra, de modo que una ficha de un color esta rodeada por las enemigas o los bordes y fichas enemigas, entonces los jugadores toman turnos moviendo horizontal o verticalmente una de sus fichas hasta una celda adyacente con una ficha enemiga y la comen, luego el rival hace lo mismo, y el juego termina cuando ya no se puede alcanzar ninguna ficha enemiga por parte de ningún jugador, dándole la victoria al último en mover. Este juego será útil para comprobar el efecto de tener un gran árbol de estados en los test, como es el caso de Clobber.

5.2. Especificación global

El experimento en sí, se ha realizado de acuerdo a las siguientes pautas:

Se escogen los algoritmos que serán sometidos a test, y posteriormente se establece en qué juego y de qué forma se realizará el test.

Para cada test se enfrentarán los algoritmos en juicio mediante la ejecución de una serie de partidas seguidas, con una cierta configuración de parámetros y en un juego determinado, variando luego los parámetros para realizar otro experimento en el mismo juego.

Se van a realizar varios test, en los cuales participarán 5 de los algoritmos de la implementación (todos menos CACLA por lo ya comentado anteriormente), en concreto se harán tres test:

- Test 1: Q-Learning, SARSA y QV-Learning, enfrentados con una política de elección avariciosa para la elección de la acción a tomar.
- Test 2: Q-Learning, SARSA y QV-Learning, enfrentados con políticas que favorezcan la exploración y la explotación de conocimientos a la vez.
- Test 3: Acla y Expected Sarsa, enfrentados con políticas que favorezcan la exploración y la explotación de conocimientos a la vez.

A partir de los resultados obtenidos experimentalmente, se extraerán las conclusiones pertinentes.

5.3. Realización del experimento

El experimento se ha realizado de la siguiente forma:

Se ha compilado un jugador específico para cada algoritmo de cada experimento de cada test, y se han depositado en carpetas diferentes, de modo que estén separados los archivos que se generen para cada agente al ejecutar el experimento.

Posteriormente, se ejecuta el servidor y la partida con dos agentes enfrentados en cada experimento, para luego extraer datos y conclusiones.

También cabe decir que para el test 1 y 2, en los cuales se han decidido confrontar a tres algoritmos a la vez, hará falta realizar dos series de partidas en cada experimento, de modo que en la primera se enfrentan Sarsa y Q-Learning, y luego se enfrenta uno de estos dos a QV-Learning, esto se ha hecho así porque SARSA y Q-Learning son bastante parecidos, mientras que QV-Learning, que usa valores para los estados en la actualización de Q, es diferente a ellos dos (aunque conserva cierta similaridad aun así).

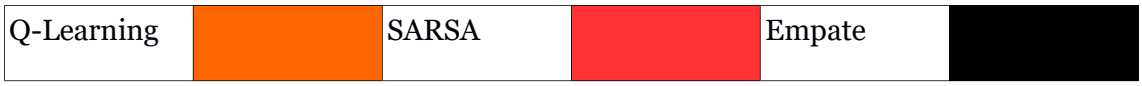
Cabe destacar que el número de acciones exploradas en todos los casos es la misma en todos los algoritmos que se enfrentan, probablemente el máximo de acciones posibles. En Tictactoe es seguro que es el máximo. El número de acciones y estados explorados de cada algoritmo en cada prueba realizada en los experimentos se muestra en los resultados, para así comprobar también que rendimiento han tenido los algoritmos en materia de exploración y analizar este hecho.

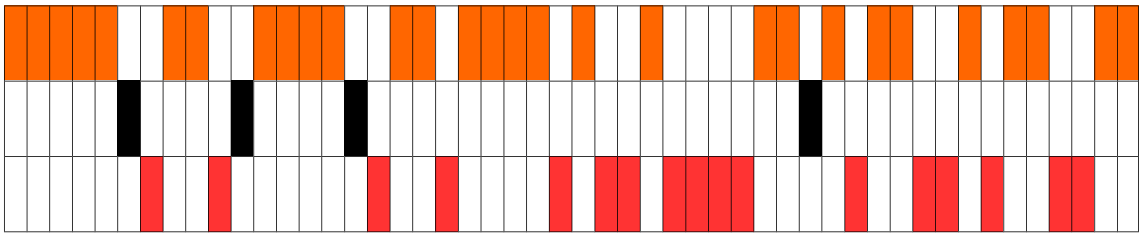
Por último, añadir que como el factor de empezar o no ha resultado ser importante para determinar la victoria en los juegos de test (Tictactoe y Clobber), y esto además ha estado patente en los propios experimentos, se ha decidido eliminar al máximo la influencia de este factor alternando los turnos para empezar la partida, de modo que unas veces empieza un agente con un algoritmo durante una serie de partidas y otras el otro.

5.4. Resultados

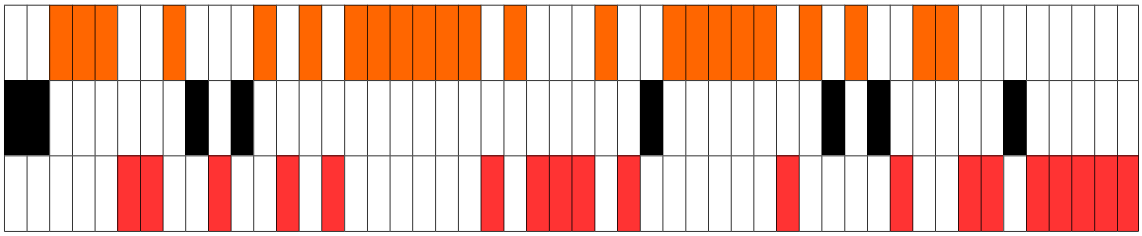
Primer test: Q-Learning, SARSA y QV-Learning, enfrentados con políticas de elección avariciosa:

Experimento 1: Q-Learning y SARSA con elección avariciosa y factor de aprendizaje 0,1, y QV-Learning y SARSA con elección avariciosa y factor de aprendizaje 0,1 también, en el juego Tictactoe o tres en raya.





En las primeras 50 partidas, no acaba de estar claro que algoritmo da mejores resultados, ambos ganan o pierden de forma algo dispersa, aunque Q-Learning muestra cierta predominancia.



Finalmente, en las últimas 50 partidas, se muestra una tendencia a ganar cuando se empieza, esto viene a significar que los algoritmos han aprendido como aprovechar la ventaja de empezar, que es una estrategia para ganar, por lo demás nada demasiado relevante, Q-Learning sigue predominando ligeramente sobre SARSA.

Estados explorados por Q-Learning: 215

Acciones exploradas por Q-Learning: 10

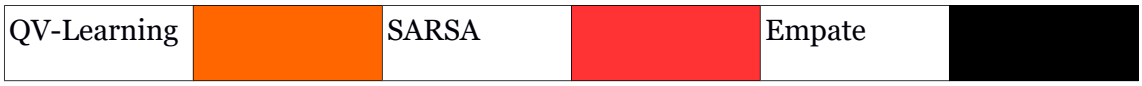
Estados explorados por Sarsa: 204

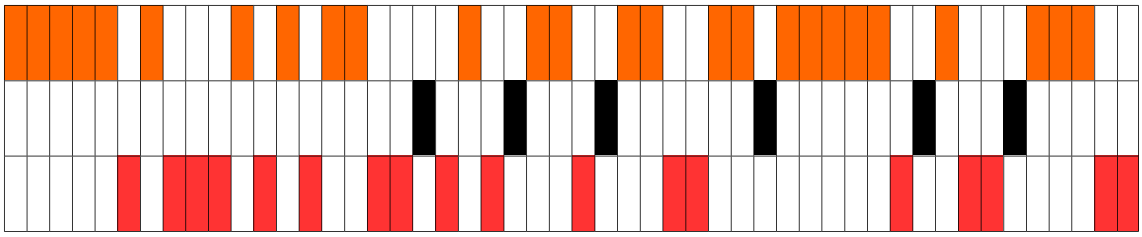
Acciones exploradas por Sarsa: 10

Podemos concluir que no hay una clara tendencia de un algoritmo sobre otro, debido a que estos dos algoritmos son básicamente iguales cuando la política de elección de acciones es avariciosa o elegir la mejor acción conocida siempre.

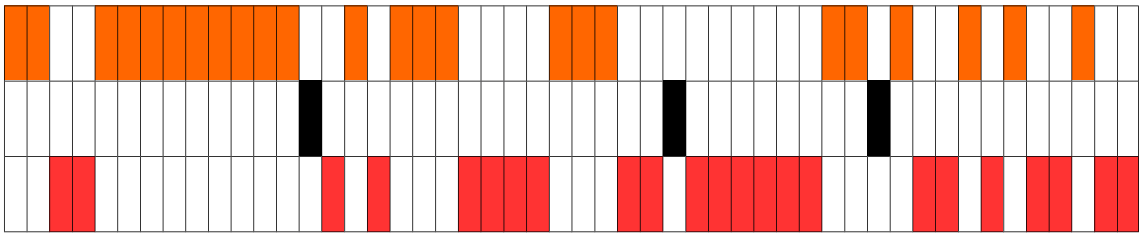
La ligera tendencia de victoria superior de Q-Learning probablemente sea debida a la variabilidad del proceso.

En cuanto a la exploración, no hay demasiada debido a la elección avariciosa de ambos algoritmos, y es muy similar en ambos.





En las primeras 50 partidas, no hay aun convergencia de los algoritmos y por tanto los resultados son muy dispersos. Quizás haya una ligera tendencia a ganar por parte de QV-Learning.



En las últimas 50 partidas, y aunque al principio empieza bien, se nota que el algoritmo QV-Learning pierde muchas mas partidas al final que Sarsa, además de que los algoritmos empiezan a convergir y aprenden la estrategia de aprovechar la ventaja de empezar.

Estados explorados por QV-Learning: 230

Acciones exploradas por QV-Learning: 10

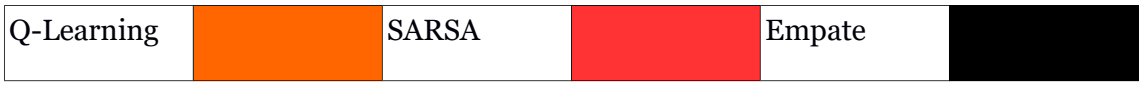
Estados explorados por SARSA: 230

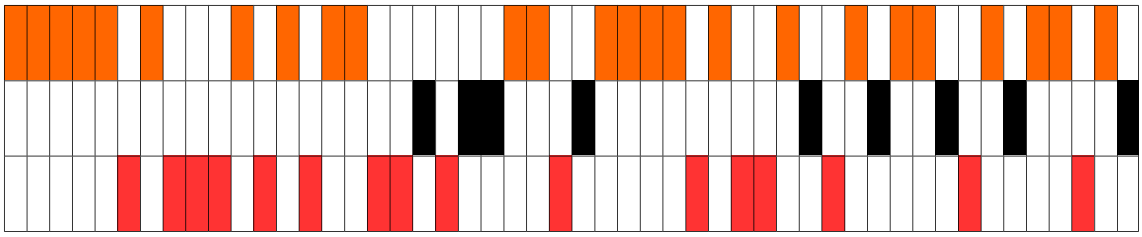
Acciones exploradas por SARSA: 10

Concluimos que, como esta implementación del algoritmo QV-Learning no dispone de persistencia para el vector de valores de estado V, puede aprovechar de forma limitada la información de los estados, ya que esta es cero al inicio de cada partida, de modo que Sarsa le empieza a sacar ventaja conforme empieza a convergir.

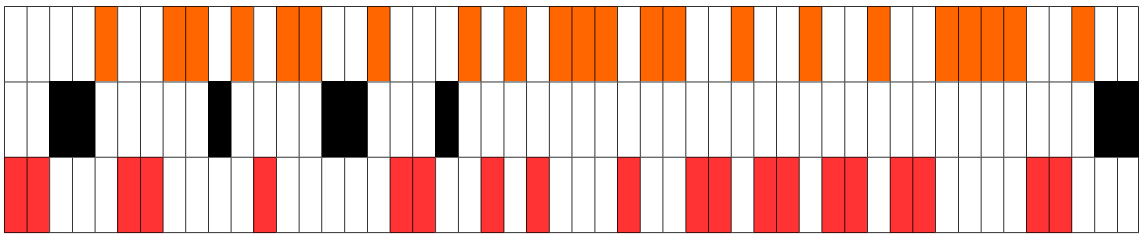
Se detecta algo más de exploración de estados.

Experimento 2: Q-Learning y SARSA con elección avariciosa y factor de aprendizaje 0,3, y QV-Learning y SARSA con elección avariciosa y factor de aprendizaje 0,3 también, en el juego Tictactoe o tres en raya.





No existe una clara tendencia, ambos ganan y pierden bastantes partidas, se percibe ligeramente un patrón de aprendizaje.



Ambos algoritmos consiguen tanto ganar partidas al empezar como arrebatárselas al rival cuando no empiezan.

Estados explorados por Q-Learning: 218

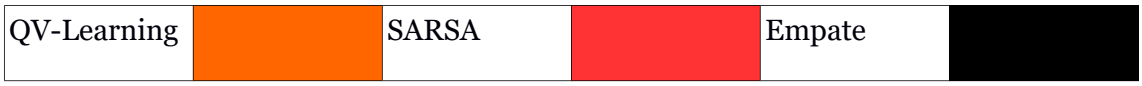
Acciones exploradas por Q-Learning: 10

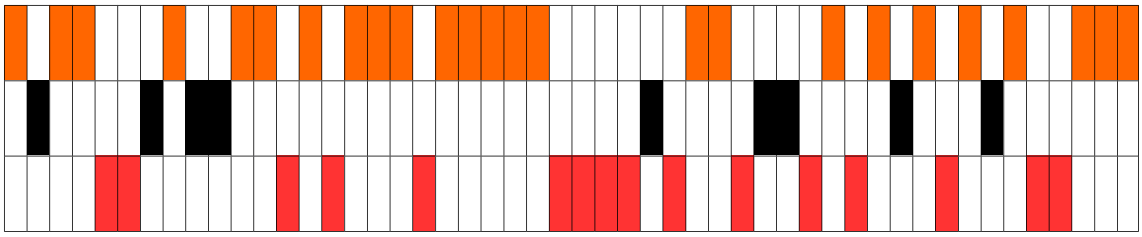
Estados explorados por SARSA: 214

Acciones exploradas por SARSA: 10

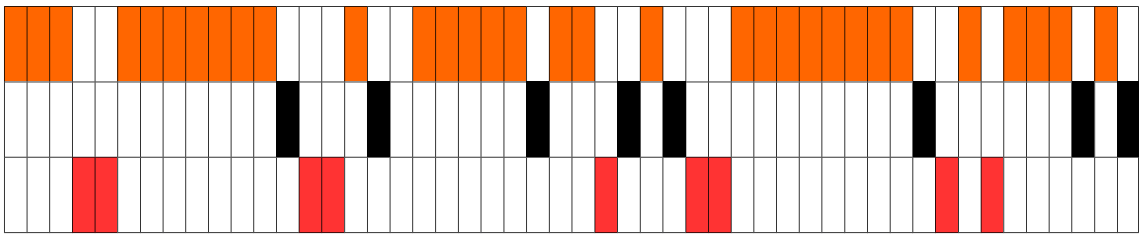
El ratio de aprendizaje más grande hace al parecer que ambos algoritmos converjan antes en este tipo de juego, pero como aparentemente con mas variabilidad, y como son casi el mismo algoritmo por la estrategia de elección avariciosa, no consiguen destacar el uno sobre el otro.

La exploración de estados continúa baja como en el caso anterior en que se enfrentaron estos algoritmos.





No hay clara tendencia de victoria, eso si, los algoritmos parecen aprender antes por el ratio de aprendizaje más grande.



Puede que sea por la convergencia prematura, pero esta vez QV-Learning vence a SARSA, pero además de forma muy visible, es posible que esto sea debido simplemente a la variabilidad existente en el proceso. Se observa la tendencia a ganar cuando se empieza.

Estados explorados por QV-Learning: 211

Acciones exploradas por QV-Learning: 10

Estados explorados por SARSA: 197

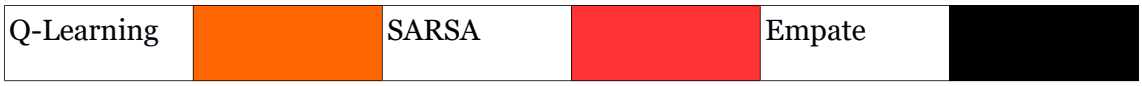
Acciones exploradas por SARSA: 10

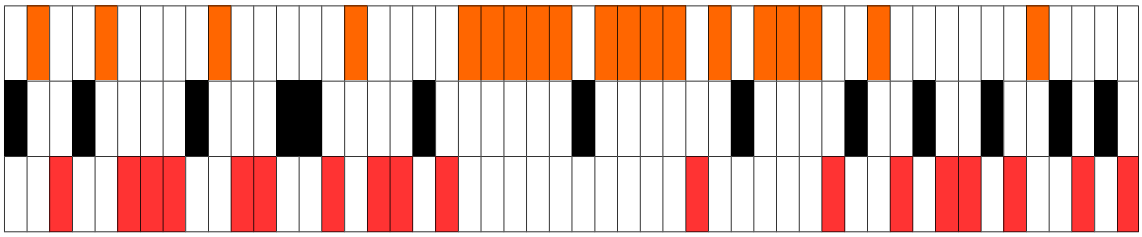
No hay mucho que añadir, simplemente apuntar que QV-Learning parece desenvolverse mejor con factor de aprendizaje más grande que con uno pequeño.

También QV-Learning explora algo más de estados.

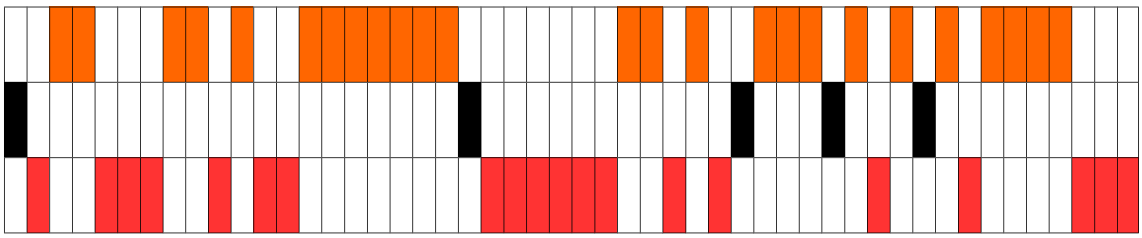
Segundo test: Q-Learning, SARSA y QV-Learning, enfrentados con políticas de elección que favorece tanto la exploración como la explotación de conocimientos:

Experimento 1: Q-Learning y SARSA con elección pseudo-avariciosa y factor de aprendizaje 0,1, y QV-Learning y Q-Learning con elección pseudo-avariciosa y factor de aprendizaje 0,1 también, en el juego Tictactoe o tres en raya.





Ahora si que vemos más dominación de Q-Learning frente a SARSA, aun en etapas tempranas en que los algoritmos no han convergido.



Ahora los algoritmos convergen, y vemos algo de dominación también de Q-Learning frente a SARSA.

Estados explorados por Q-Learning: 249

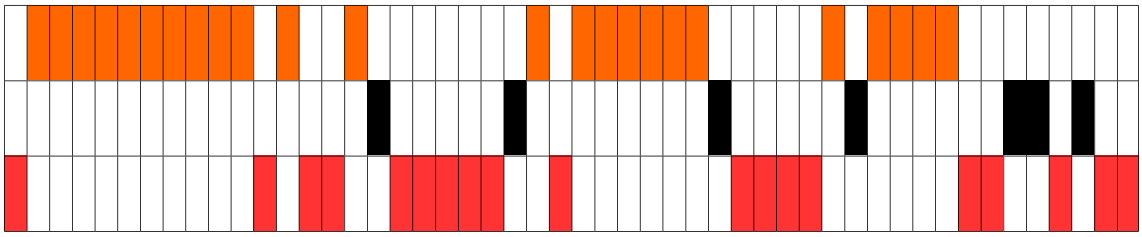
Acciones exploradas por Q-Learning: 10

Estados explorados por SARSA: 255

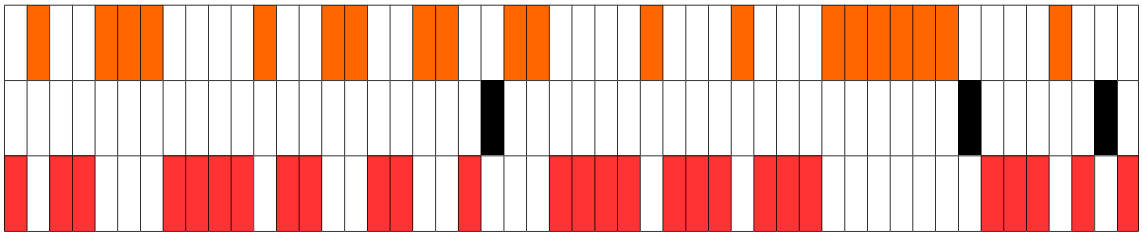
Acciones exploradas por SARSA: 10

En el caso de elección pseudo-aleatoria, Q-Learning detecta las acciones que nos llevan a buenas acciones, sin importar que en el estado al que nos llevan haya también posibles malas acciones, mientras que Sarsa realiza una actualización con la acción tomada, no con la mejor, en el nuevo estado, por tanto añade ruido al valor que le damos a las acciones, dado que las malas acciones simplemente nos hacen volver a elegir, no las realizamos en el juego, y los estados con una muy buena acción óptima son una buena elección normalmente.

Aquí se ve que hay mayor exploración de estados, dado que estamos usando una política que también apoya a la exploración.



En este caso, con la elección pseudo-avariciosa en la que ambos algoritmos exploran, si que se percibe una dominación de Q-Learning sobre QV-Learning, cuyas capacidades son limitadas debido a no disponer de persistencia para el vector V de valores de estado.



Se estabiliza el numero de victorias en ambos bandos, sin un claro vencedor.

Estados explorados por Q-Learning: 224

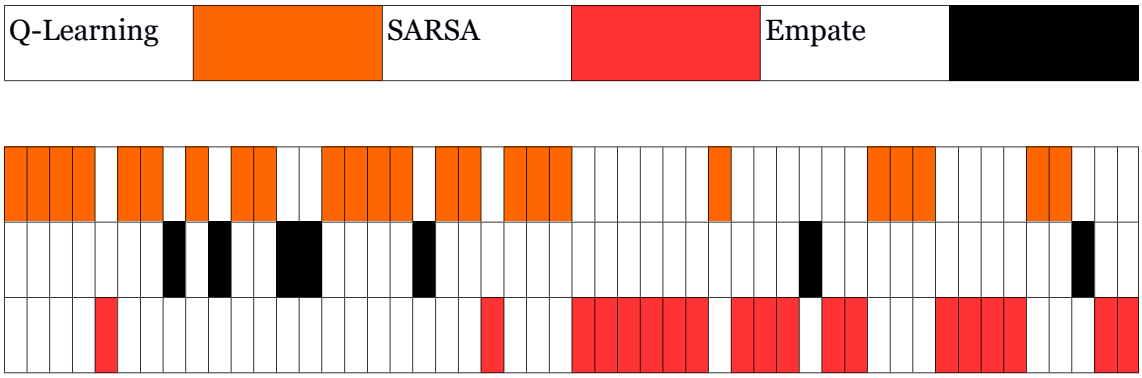
Acciones exploradas por Q-Learning: 10

Estados explorados por QV-Learning: 220

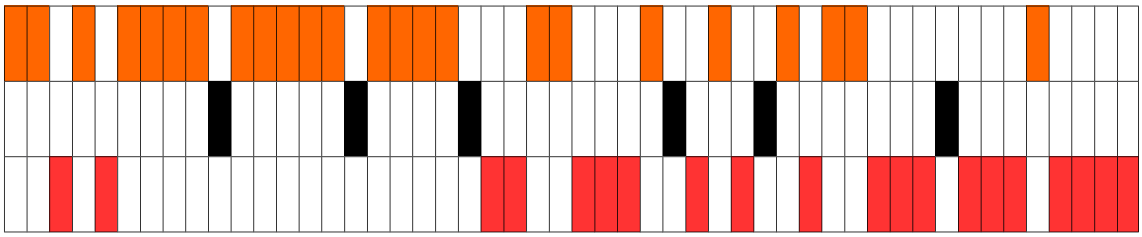
Acciones exploradas por QV-Learning: 10

El algoritmo Q-Learning consigue batir a QV-Learning en etapas tempranas, pero luego el segundo empieza a mostrar que iguala el marcador, el número de estados explorados ahora es algo mas elevado que en el caso de elección avariciosa, posiblemente porque se dan algo más de rivalidad que en ese caso.

Experimento 2: Q-Learning y SARSA con elección pseudo-avariciosa y factor de aprendizaje 0,3, y QV-Learning y Q-Learning con elección pseudo-avariciosa y factor de aprendizaje 0,3 también, en el juego Tictactoe o tres en raya.



Vemos cierta dominación de Q-Learning, además de aprendizaje prematuro de ambos algoritmos, debido al más elevado ratio de aprendizaje. Esta dominación es debida a que la estrategia Q-Learning parece más adecuada a este tipo de juegos. Q-Learning gana en gran medida cuando empieza, y luego le consigue arrebatar varias partidas a SARSA cuando este empieza.



Comprobamos que ahora se estabiliza con el paso del tiempo el numero de victorias a partes iguales, con cierta dominación de Q-Learning.

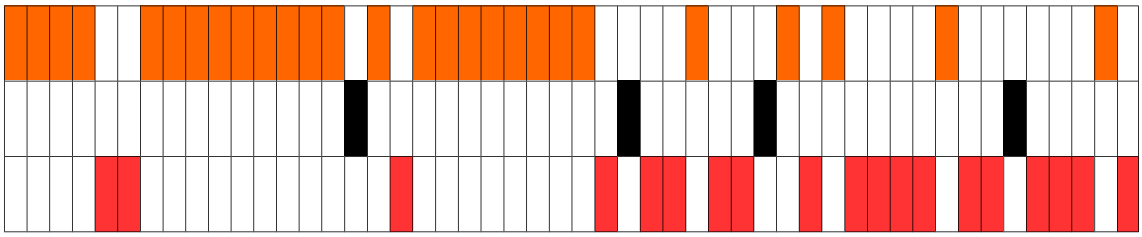
Estados explorados por Q-Learning: 254

Acciones exploradas por Q-Learning: 10

Estados explorados por SARSA: 250

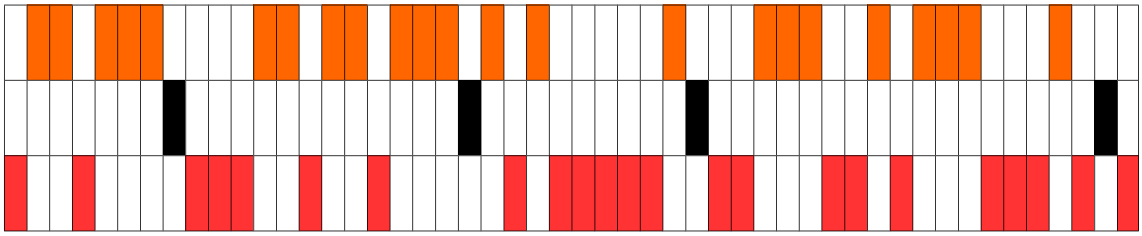
Acciones exploradas por SARSA: 10

Nada más que añadir, solo que no hay una clara diferencia en el número de estados explorados respecto al caso anterior con ratio de aprendizaje 0,1. Son el mismo número de estados básicamente. Puede ser debido a que es un caso con resultados de victorias muy similar, de modo que los algoritmos se han comportado probablemente de forma parecida.



Q-Learning demuestra ganar más y de forma visible en etapas tempranas cuando empieza él, y arrebatarle algunas partidas a QV-Learning cuando no empieza.

Vemos que es un caso similar al anteriormente estudiado con menor ratio de aprendizaje pero con una convergencia mayor en etapas tempranas.



Observamos de nuevo como se estabilizan las victorias a partes iguales.

Estados explorados por Q-Learning: 240

Acciones exploradas por Q-Learning: 10

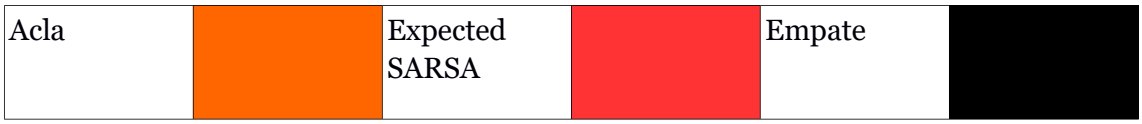
Estados explorados por QV-Learning: 242

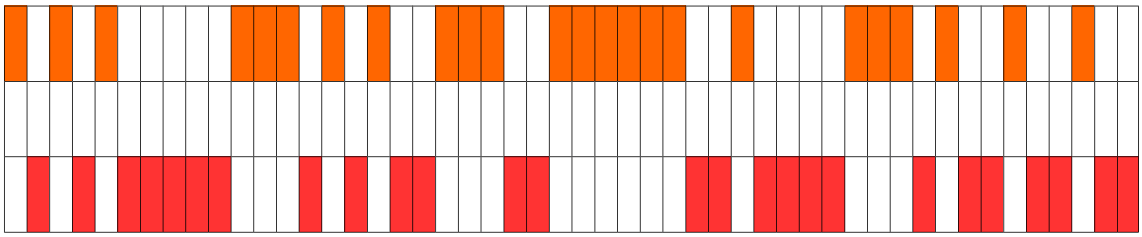
Acciones exploradas por QV-Learning: 10

Solo añadir que se han explorado mas estados en este caso que en el de menor ratio de aprendizaje. Esto podría deberse a que los algoritmos muestran jugar bastante bien y por tanto nos vemos obligados a explorar mas.

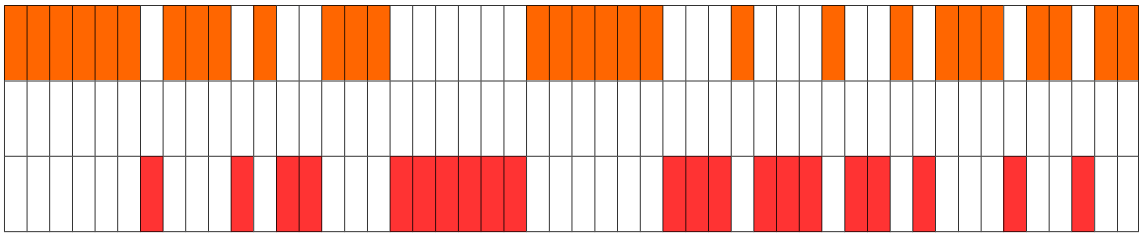
Tercer test: Acla y Expected Sarsa, enfrentados con políticas de elección que favorece tanto la exploración como la explotación de conocimientos:

Experimento 1: Acla y Expected SARSA con elección semi-avariciosa y factor de aprendizaje 0,1, y Acla y Expected SARSA con elección semi-avariciosa y factor de aprendizaje 0,1 también, en el juego Clobber.





Hay resultados bastante dispersos en esta primer tanda de partidas, se aprecia una tendencia de victoria dependiendo de si se empieza o no (en este juego gana el que no empieza si no se consigue aislar las piezas, o si solo se aísla el mismo numero par de cara jugador), pero si que podemos apreciar una ligera ventaja a favor de Acla.



Finalmente los algoritmos convergen bien, aprenden la estrategia para ganar y a aprovechar sus ventajas, no obstante se aprecia de nuevo una tendencia, esta vez más marcada a ganar Acla.

Estados explorados por Acla: 455

Acciones exploradas por Acla: 63

Estados explorados por Expected Sarsa: 437

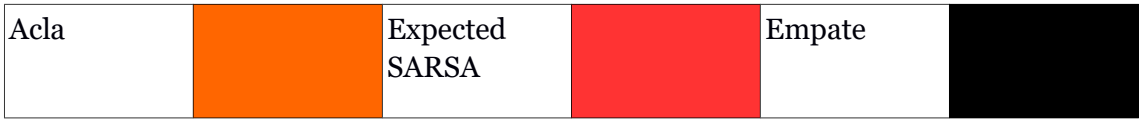
Acciones exploradas por Expected Sarsa: 63

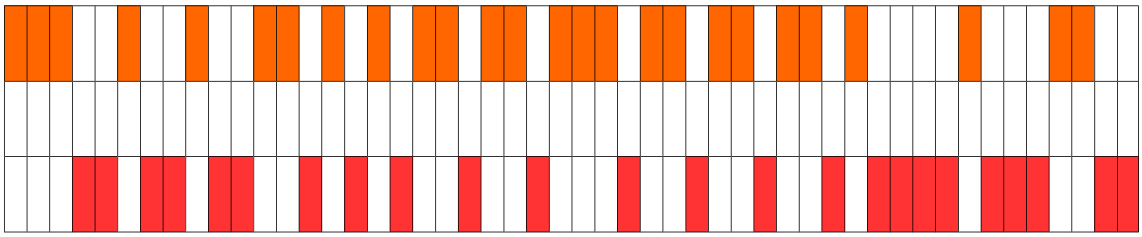
Podemos concluir que los algoritmos convergen correctamente con los actuales parámetros en estos tipos de juegos con un árbol de estados mas grande, un ratio de aprendizaje pequeño ayuda a no perdernos en un mar de estados nuevos cada vez que nos dirigimos hacia una zona nueva del árbol de estados, dado que no se le da una importancia demasiado grande a los nuevos estados que dan buenos resultados (o acciones que conducen a nuevos estados con buenas acciones en el caso de Expected Sarsa), tanto como si utilizáramos un ratio mas grande.

Acla parece demostrar cierta dominación sobre Expected SARSA, parece que almacenar un vector de valores de estados con persistencia y usarlo para determinar si se actualiza Q hacia valores altos o bajos es una idea buena, que consigue batir a un buen algoritmo como es Expected SARSA, sobretodo en juegos con arboles de estados tan grandes, donde ir hacia

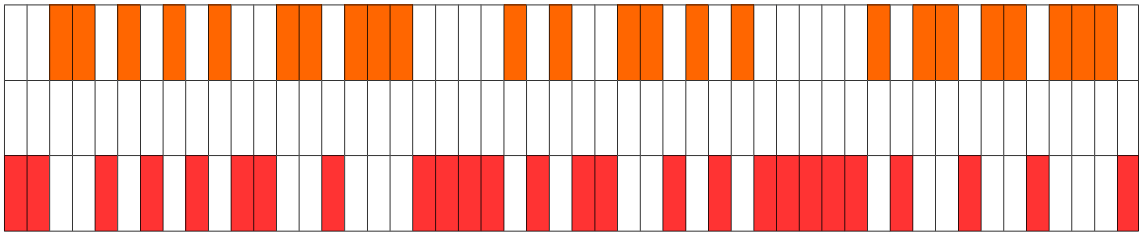
estados que demuestran llevarnos a buenas recompensas futuras es buena idea para no perdernos en el mar de estados.

Experimento 2: Acla y Expected SARSA con elección semi-avariciosa y factor de aprendizaje 0,3, y Acla y Expected SARSA con elección semi-avariciosa y factor de aprendizaje 0,3 también, en el juego Clobber.





No se observa una tendencia a ganar un algoritmo frente al otro, los resultados son altamente divergentes.



Se puede apreciar ahora como los algoritmos han aprendido estrategias, pero no vemos a un claro dominador, el hecho de aumentar bastante el ratio de aprendizaje provoca que los algoritmos creen caminos en el árbol de estados, y esto hace que se pierdan, y no lleguen satisfactoriamente siempre a la victoria.

De modo que esta vez, como no hay clara convergencia, tampoco hay un claro vencedor.

Estados explorados por Acla: 496

Acciones exploradas por Acla: 63

Estados explorados por Expected SARSA: 481

Acciones exploradas por Expected SARSA: 63

Se observa una gran exploración de estados, mayor al caso anterior, cosa que es lógica debido a que los algoritmos crean largos caminos en el árbol de estados.

6. Conclusiones

Podemos concluir que la implementación de RL-GGP tiene buenas características para hacer test diversos de algoritmos de Aprendizaje por Refuerzo en juegos por turnos como los vistos, ya que uno solo tiene que cargar el juego en lenguaje GDL y los agentes con el algoritmo que se quiera poner a prueba, compilado con los parámetros que se desee.

Además, ahora que se han añadido varios algoritmos de Aprendizaje por Refuerzo que faltaban, la plataforma es más versátil y permite más posibilidades.

No obstante, sigue sin manejar acciones ni estados de tipo continuo, y los algoritmos no han sido programados para manejar estados continuos, esto sería una buena posible ampliación que ampliaría horizontes, y también seria deseable que las recompensas fueran más ajustadas para poder aprender mejor las estrategias vencedoras, aunque esto es un problema recurrente, ya que no hay un arquetipo general que sirva como modelo para todos los juegos, cada uno tiene sus singularidades, de modo que no podemos ajustar estas recompensas sin programar la plataforma o los algoritmos específicamente para un juego en particular.

Por supuesto, a pesar de lo implementado y realizado siempre se pueden añadir nuevos algoritmos, dado que el campo de investigación no esta cerrado, y hacer una evaluación experimental más amplia.

Respecto al experimento, podemos decir que Q-Learning y Sarsa son algoritmos muy similares o iguales cuando elegimos una política de elección de acciones avariciosa, esto no es así cuando tenemos una política pseudo-avariciosa, donde Q-Learning parece tomar ventaja sobre SARSA en juegos por turnos con un árbol de estados no demasiado grande.

En cuanto a QV-Learning, parece tener cierto potencial por aprovechar el valor de ciertos estados que llevan a buenas acciones en juegos como el de test sin muchos estados posibles, pero como no dispone de persistencia en el vector V de valor de estados se ha comprobado que esta ciertamente limitado y tanto SARSA como Q-Learning consiguen batirlo o empatar en casi todos los casos de estudio, siendo mejor Q-Learning con política pseudo-avariciosa, ya que la exploración mejora el rendimiento del algoritmo.

El algoritmo Acla, por otra parte, demuestra un buen rendimiento en juegos con un gran espacio de estados, ganando frente a Expected Sarsa en el caso en que ambos tienen un ratio de aprendizaje pequeño.

El uso de un ratio de aprendizaje grande como es 0,3, ha demostrado ser más útil para el aprendizaje en juegos con un árbol de estados más pequeño, donde los algoritmos aprendían mejor y antes las estrategias vencedoras, mientras que en otros juegos con un gran árbol de estados se ha demostrado que era mejor un ratio más pequeño como 0,1, ya que cuando este era usado los algoritmos alcanzaban una buena convergencia, mientras que con 0,3 los resultados mostraban ser bastante aleatorios y dispares entre ellos, de modo que ninguno de los algoritmos alcanzaba una buena convergencia.

Se ha percibido una mejora de las victorias con la estrategia del uso de valores de estados cuando se usaba persistencia del vector de valores de estados V, ya que Acla, que es una

estrategia con cierta similaridad que a QV-Learning que también usa valores de estados, ha demostrado un mayor rendimiento general que QV-Learning.

APÉNDICES

A. Bibliografía

Benacloch-Ayuso, José Luis (2012) “RL-GGP: una plataforma de integración de algoritmos de aprendizaje por refuerzo en el sistema de juegos GDL”, http://users.dsic.upv.es/~flip/RLGGP/RL-GGP.pdf, Agosto 2015.

Benacloch-Ayuso, José Luis (2012) “RL-GGP”, http://users.dsic.upv.es/~flip/RLGGP/, Agosto 2015.

Hado van Hasselt (2011) “A Short Introduction To Acla”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Acla.html, Junio 2015.

Hado van Hasselt (2011) “A Short Introduction To Cacla”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Cacla.html, Junio 2015.

Hado van Hasselt (2011) “A Short Introduction To Expected Sarsa”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/ESarsa.html, Junio 2015.

Hado van Hasselt (2011) “A Short Introduction To Q-learning”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Q.html, Junio 2015.

Hado van Hasselt (2011) “A Short Introduction To QV-learning”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/QV.html, Junio 2015.

Hado van Hasselt (2011) “A Short Introduction To Sarsa”, http://webdocs.cs.ualberta.ca/~vanhasse/rl_algs/Sarsa.html, Junio 2015.

Richard S. Sutton y Andrew G. Barto, “Rreinforcement Learning”. Londres. The MIT Press,1998.

Wikipedia (2015) “Aprendizaje por refuerzo”, https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo, Agosto 2015.

Wikipedia (2015) “Eclipse(software)”, https://es.wikipedia.org/wiki/Eclipse_(software), Agosto 2015.

Wikipedia (2015) “Java Agent Development Framework”, https://es.wikipedia.org/wiki/Java_Agent_Development_Framework, Agosto 2015.

Wikipedia (2015) “Markov decision process”, https://en.wikipedia.org/wiki/Markov_decision_process, Agosto 2015.

B. Implementación de los algoritmos

Acla, Cacla y Expected Sarsa se han implementado extendiendo la clase Algorithm y sobrecargando ciertos métodos necesarios, como son el constructor, el Update y el update_size_Q, que en el caso de Cacla es update_size_A ya que no hay matriz Q sino A.

También se han añadido los métodos de persistencia de Q, V y A en el algoritmo donde se usan, y algunos métodos para el cálculo de acciones continuas en Cacla, además de los métodos que devuelven datos del algoritmo como su nombre.

Ejemplo de Acla y su constructor:

```
public class Acla_ extends Algorithm {

        public Acla_( World_ w ) {

            discreteStates      = w.getDiscreteStates() ;

            stateDimension      = w.getStateDimension() ;


            if ( ! w.getDiscreteActions() ) {

          System.out.println( "Acla does not support continuous
actions." ) ;

          System.out.println( "Please check which MDP you are using it
on." ) ;

          System.exit(0) ;

          } else {

           numberOfActions = w.getNumberOfActions() ;

        }

        if ( discreteStates ) {

            numberOfStates = w.getNumberOfStates() ;

            Q = new double[ numberOfStates ][] ;

             V = new double[ numberOfStates ] ;

            for ( int s = 0 ; s < numberOfStates ; s++ ) {

                Q[s] = new double[ numberOfActions ] ;

                       V[s] = 0.0 ;

                for ( int a = 0 ; a < numberOfActions ; a++ ) {

                    Q[s][a] = 0.0 ;

                }
```

```
                }
            }
            loadValueFunction(getName()+".csv");
            loadStateFunction(getName()+"S.csv");
            QTarget = new double[ 1 ] ;
            VTarget = new double[ 1 ] ;
            policy = new double[ numberOfActions ] ;
        }
...
```

C. Implementación del Agente

Para la implementación del agente se ha usado la clase SkeletonAgent, la cual ha sido adaptada para poder manejar los nuevos algoritmos y características incluídas en la implementación de RL-GGP.

Ejemplo de fragmento del método agent_step, donde se añade código asociado a Acla en el switch:

```
…
QVL.update(lastS, actions, reward, newS, false, learning_rat, 1);


            break;


            case 3:
                  acla.update_size_Q(observation.getInt(1), num_action);
                  //acla.getMaxActionFirst(newS, actions[0]);
                  //acla.explore(newS, actions[0], 1.0, "boltzmann",
false);
                  acla.egreedy(newS, actions[0], 0.6);
                  theIntAction = actions[0].discreteAction;


                  actions[0].discreteAction = lastActionInt;
                  actions[1].discreteAction = theIntAction;


                  learning_rat = new
double[acla.getNumberOfLearningRates()];
                 for(int i=0;i<acla.getNumberOfLearningRates();i++)
learning_rat[i] = learning_r;
                  acla.update(lastS, actions[0], reward, newS, false,
learning_rat, 1);
                  break;


            case 4:
…
```

D. Ejecución de las partidas

Primero que nada, se debe disponer de todos los componentes de RL-GGP en el sistema, son necesarios GGP-Server, juegos GDL, el códec RLGlue instalado y tener la implementación de RL-GGP, más algún entorno en el que poder compilar los jugadores de RL-GGP si se quiere modificar el básico.

Para conseguir todas las componentes de RL-GGP mencionadas y saber como instalarlas, ir a la página de RL-GGP:

Una vez se tenga GGP-Server, RL-GGP, el códec instalado y algún juego en GDL (se pueden encontrar aquí http://130.208.241.192/ggpserver/public/show_games.jsp), ya se puede proceder a ejecutar una partida o una serie de ellas de la siguiente forma:

Ejecutar el servidor GGP-Server pinchando en \GGP-Server-exec\start_ggp_server.bat partiendo de la carpeta donde se encuentra la implementación de RL-GGP, escoger un juego en GDL y elegir los jugadores, si se quiere que sean remotos o jugadores aleatorios por ejemplo. Elegir remoto, localhost y puertos 4001 y 4002 para dos jugadores de nuestro equipo por ejemplo.

Luego dirigirse desde el intérprete de comandos al directorio donde se encuentra el jugador compilado en un .jar con un archivo config_agent.cfg con el valor del algoritmo de Aprendizaje por Refuerzo que se quiera usar (este archivo solo contiene ese valor en un número entero, véase la parte de diseño de la presente memoria para más información), y ejecutar java -jar nombreDelJugador.jar –port=4001 para el caso del jugador que se conecte al puerto 4001 de GGP-Server, hacer lo mismo para el otro jugador y posteriormente dar a start en GGP-Server. La partida comenzará y se mostrará la información relativa a ella en GGP-Server, más información de cada agente en el intérprete de comandos en que se haya ejecutado.

Si se quiere compilar un jugador, simplemente generar un .jar de la implementación RL-GGP con el main de Jocular como clase de inicio de la aplicación con cualquier entorno de programación, aunque se recomienda Eclipse.

Para más información, visitar la página de RL-GGP:
http://users.dsic.upv.es/~flip/RLGGP/