UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Aprendiendo políticas de exploración generales para escalar la síntesis de controladores discretos

Tesis de Licenciatura en Ciencias de la Computación

Tomás Delgado

Director: Sebastián Uchitel

Codirector: Víctor Braberman

Buenos Aires, 2023

LEARNING GENERAL EXPLORATION POLICIES TO SCALE DISCRETE CONTROLLER SYNTHESIS

Controller Synthesis studies the automated construction of behavior strategies with guaranteed correctness, for systems that can be formally described as automatons. The applicability of these techniques is usually limited by the curse of dimensionality, which makes automatons rapidly become intractably large as problems become more interesting. An *on-the-fly* synthesis algorithm can avoid the state explosion by building the automaton partially, starting from the initial state and adding one transition at a time, attempting to only explore the subgraph needed for the winning strategy or to show that such strategy does not exist.

In this thesis, we develop a first method for learning a heuristic that guides the exploration from experience. First, we define a Reinforcement Learning task where the agent represents an exploration policy. Then, Q-Learning is used while abstracting both states and actions with a set of features. This abstraction makes learning and generalization possible, but causes a form of partial observability that we call Very Partially Observable MDP. Empirical evaluation shows that, despite the lack of theoretical guarantees, it is possible to learn competitive policies consistently in the training instances. Furthermore, the policies induced in the larger versions of the problems outperform the best human-designed heuristic overall, pushing the frontier of solvable problems for a subset of the benchmark problems.

Keywords: Controller Synthesis, Reinforcement Learning, Neural Networks, Generalization, Heuristic Search.

APRENDIENDO POLÍTICAS DE EXPLORACIÓN GENERALES PARA ESCALAR LA SÍNTESIS DE CONTROLADORES DISCRETOS

El área de síntesis de controladores discretos estudia la construcción automática de estrategias de comportamiento con garantías de correctitud, para sistemas descriptos formalmente por autómatas. La limitación de estas técnicas viene dada por la maldición de la dimensionalidad, que hace que el tamaño de los autómatas crezca muy velozmente y limita la aplicabilidad. La síntesis *on-the-fly* busca eludir esta problemática construyendo el espacio de posibles estados parcialmente, agregando una transición a la vez desde el estado inicial e intentando explorar solo lo necesario para la estrategia ganadora, o para mostrar que tal estrategia no existe.

En esta tesis desarrollamos un primer método para aprender una heurística que guíe la exploración a partir de la experiencia. En primer lugar, definimos una tarea de aprendizaje por refuerzo para la cual el agente representa una política de exploración. Luego, mostramos una forma de usar Q-Learning abstrayendo tanto estados como acciones en un conjunto de features. Esta abstracción hace posible el aprendizaje y la generalización, pero genera un alto grado de observabilidad parcial. La evaluación empírica muestra que, a pesar de la falta de garantías teóricas, es posible aprender consistentemente políticas competitivas en las instancias de entrenamiento. Más aún, las políticas inducidas en instancias grandes superan en promedio a la mejor heurística desarrollada por humanos, empujando la frontera de problemas resueltos en algunos de los dominios del benchmark.

Palabras claves: Síntesis de controladores, Aprendizaje por Refuerzo, Redes neuronales, Generalización, Heurísticas de búsqueda.

AGRADECIMIENTOS

A mis directores Sebas y Víctor, por darme la oportunidad de arrancar este proyecto medio loco y por estar siempre presentes para charlar cualquier idea o problema. Sin su habilidad para guiarme y su contención cuando las cosas no salían bien nada de esto hubiera sido posible.

A mi mamá y papá y a toda mi familia, por apoyarme desde chiquito con su amor incondicional y por siempre confiar en mí.

A mis amigos del secundario y de la vida (ya saben quiénes son) por alegrarme siempre y estar en los momentos más difíciles.

A mis amigos de la facu, Migue, Santi, Mati G, Mati S, Nico, Nacho, Manu, Elías, Ale y muchos más, por hacer que la carrera sea mucho más linda.

A toda la gente de LaFHIS y del Laboratorio de Neurociencia por darme hermosos espacios para iniciarme en el mundo de la investigación, y en particular a la gente con la que trabajé más de cerca, Juli G, Ceci, Ari, Hernán, Juli B, Marco, Flopy, Mati, Nico.

Al Departamento de Computación, por el programa de pasantías y por la calidad y la calidez que lo caracterizan.

Finalmente, a todos los docentes que me fueron dejando su huella y me enseñaron a aprender.

Índice general

1. INTRODUCCIÓN

El mundo está cada vez más poblado de sistemas críticos en los que cualquier error de programación puede tener costos enormes. Algunos ejemplos de esto son el control de tráfico aéreo, la manufacturación automática, los protocolos de comunicación a través de internet y muchas tareas robóticas. En este contexto, la idea de construir en forma automática estrategias de control con garantías de correctitud (a veces llamada *síntesis de programas*) es muy atractiva pero computacionalmente muy costosa. Muy rápidamente el tiempo de ejecución y la memoria requerida se vuelve imprácticos cuando los problemas a resolver automáticamente se complejizan.

Control de Eventos Discretos (Wonham y Ramadge, 1987), Planning Automático (Nau et al., 2004) y Síntesis Reactiva (Pnueli y Rosner, 1989) son campos que abordan de distintas formas este problema. Si bien tienen diferencias en cuanto a las representaciones utilizadas, la expresividad y los algoritmos, las tres comparten el problema generado por el veloz crecimiento de la cantidad de estados posibles. En todos los casos los sistemas en los que se busca encontrar una estrategia ganadora (pueden pensarse intuitivamente como juegos) se especifican de forma compacta pero con estructuras cuya semántica induce una explosión combinatoria en los escenarios que es necesario analizar.

Por otro lado, Aprendizaje por Refuerzo (RL) (Sutton y Barto, 1998) tuvo un éxito empírico muy destacable como técnica general para problemas de tomas de decisiones secuenciales. Con el trabajo seminal en el que se propuso DQN para los juegos de Atari (Mnih et al., 2013) y los resultados excepcionales para el Go, el Ajedrez y el Shogi (Silver et al., 2017), RL mostró grandes capacidades para aprender desde cero en espacios de estados grandes y complejos. Sin embargo, la ineficiencia de RL a la hora de usar la experiencia hace que el aprendizaje de los agentes requieran grandes cantidades de entrenamiento. Más aún, un problema clave que limita su aplicabilidad en muchos contextos en la falta de garantías sobre el comportamiento de los agentes entrenados. Es imposible asegurar que un agente entrenado para manejar un auto no va a chocar tontamente tras confundir algo que vieron sus sensores, o que un robot no va a tomar acciones autodestructivas al encontrase en un estado que no había visto durante su etapa de aprendizaje.

En esta tesis proponemos una forma de usar RL para aprender estrategias de exploración con el objetivo de acelerar la búsqueda de estrategias ganadoras en problemas de control, preservando las garantías de correctitud e intentando mitigar la explosión de estados.

En particular, el problema que abordamos es una versión de control de Sistemas de

Eventos Discretos (DES) en la que una *planta* (autómata) a ser controlada es especificada modularmente como la composición paralela de un conjunto de autómatas finitos que se comunican entre sí (Wonham y Ramadge, 1988), y el objetivo es construir una estrategia de control *safe* y *nonblocking*. A grandes rasgos, lo que queremos es encontrar un controlador que garantice que siempre se puede alcanzar un estado marcado de la planta, asegurando a su vez que estados inseguros de la planta nunca son alcanzados. En este contexto, nos centramos en encontrar un *director*, que es un un controlador que elige a lo sumo un evento controlable para habilitar en cada estado (Huang y Kumar, 2008), en contraste con un *supervisor*, que es un controlador maximalmente permisivo.

Componer los autómatas de la planta puede resultar en una explosión exponencial de estados dado que, salvo posibles restricciones de la sincronización, los estados de la planta compuesta son la combinación (producto cartesiano) de los estados de los autómatas individuales. Por este motivo las técnicas de síntesis clásicas que empiezan por construir la planta completa y luego buscan un controlador escalan pobremente. En particular, pueden fallar (quedándose sin tiempo o memoria) incluso cuando existe un controlador que mantiene a la planta en una porción pequeña de los estados, para el cual no sería necesario analizar la planta completa. El algoritmo de síntesis dirgida on-the-fly (OTF-DCS) (Ciolek et al., 2023) intenta evitar la explosión de estados *explorando* la planta compuesta incrementalmente. En cada paso agrega una transición entre dos estados y analiza si con esa nueva transición existe una estrategia ganadora para el controlador, o si esa transición le permite asegurar que no existe tal estrategia.

Junto con OTF-DCS se propusieron diversas heurísticas que guían la exploración intentando minimizar la cantidad de pasos necesaria para llegar a un veredicto. Estas heurísticas son independientes del dominio (es decir, su comportamiento no varía según el ambiente en el que se las use) y están diseñadas por seres humanos (los investigadores de LaFHIS que me precedieron y acompañaron). Definir este tipo de heurísticas es una tarea difícil y es mucha la información disponible durante el algoritmo que podría ser valiosa. En particular, aparecen muchos trade-offs cuya respuesta es poco clara y podría fácilmente depender de la planta que se esté considerando.

En este trabajo analizamos la posibilidad de reemplazar estas heurísticas por políticas de exploración aprendidas con RL en *versiones chicas* de los problemas de interés. Si bien RL tradicionalmente se focalizó en tareas en los que los agentes son evaluados en los mismos ambientes en los que aprenden, en nuestro caso esto no es posible porque aprender requiere sintentizar completamente un problema muchas veces, y nuestro objetivo es aprender a sintetizar eficientemente problemas que actualmente no pueden ser resueltos en tiempos razonables. Buena parte de este trabajo entonces está dedicado a encontrar

políticas que *generalizan* a instancias desconocidas.

Lo primero que haremos es enmarcar a OTF-DCS como una tarea de RL (más formalmente, un Proceso de Decisión de Markov) en la cual los agentes (políticas de exploración) reciben recompensas negativas cada vez que expanden una transición de la planta. Este ambiente de aprendizaje es desafiante por varios motivos. En primer lugar, los agentes obtienen información sobre la calidad de sus decisiones muy poco frecuentemente (solo una vez que logran tiene una exploración parcial que permite responder positiva o negativamente a la existencia de un controlador). En términos de RL decimos que la recompensa de la tarea es *esparsa*.

En segundo lugar, aunque quizás más importante, las señales de estado y acciones son muy complejas. El conjunto de acciones es muy grande y tanto la cantidad como la semántica de las acciones disponibles cambia constantemente, por ser las transiciones en la frontera de exploración, que cambia cada vez que expandimos una transición. El estado de la tarea es un grafo con tamaño variable (dado que es la exploración parcial a la que le vamos agregando transiciones), haciendo que sea muy difícil analizarlo con una red neuronal tradicional.

Una restricción adicional no menor en el aprendizaje es que queremos que las políticas aprendidas sean de algún modo aplicables en instancias más grandes, que tienen grafos distintos con distintos nodos y transiciones.

Para solucionar estos desafíos proponemos una modificación de DQN (Mnih et al., 2013) que permite su uso con conjuntos de acciones no acotados y variables, y proponemos *abstraer* tanto estados como acciones a un espacio de features que es único para todas las instancias de un dominio. Observamos que esto permite resolver los últimos dos problemas mencionados de nuestra tarea, y además hace posible la generalización. Sin embargo, la abstracción pierde información generando observabilidad parcial y entrando en conflicto con las hipótesis teóricas del algoritmo de aprendizaje.

Nuestro procedimiento consiste en entrenar en una instancia chica de un dominio dado (resolviendo parcialmente el problema de las recompensas esparsas) por un período corto de tiempo y con una red neuronal chica, guardándonos los pesos de la red neuronal cada cierta cantidad de tiempo. Luego, cada uno de estos conjuntos de pesos (que representa un agente) es evaluado en un conjunto de instancias ligeramente más grandes que las de entrenamiento. Finalmente, de ellos seleccionamos el que aparenta generalizar mejor y analizamos su performance, buscando resolver la mayor cantidad de instancias posibles con un budget de tiempo limitado.

Nuestros resultados muestran que, en primer lugar, incluso con la falta de garantías teóricas de convergencia, con nuestra técnica es posible aprender desde cero políticas

de exploración competitivas en las instancias de entrenamiento. Más aún, muestran que estas políticas son efectivas al ser usadas en instancias más grandes. Nuestros agentes son evaluados tanto en transiciones expandidas como en instancias resueltas y, en promedio, superan a la mejor heurística de Ciolek et al. (2023), empujando la frontera de instancias resueltas en varios de los problemas del benchmark utilizado.

En los capítulos 2 y 3 exponemos el marco teórico utilizado, intentando ser autocontenidos, y en el capítulo 4 desarrollamos nuestro método. En el capítulo 5 nos proponemos dar los primeros pasos hacia proveer un marco teórico para nuestra tarea de aprendizaje. Luego, en el capítulo 6 presentamos los resultados de la evaluación empírica, analizando algunos de los comportamientos observados en nuestros agentes y comparándolos con las heurísticas existentes. Finalmente, en el capítulo 7 presentamos el trabajo relacionado, las conclusiones y posibles líneas de trabajo futuro.

2. SÍNTESIS DE CONTROLADORES

2.1. El problema de control

El ambiente en el que nuestro controlador toma decisiones, que llamamos *planta* o sistema de eventos discretos (DES), es un autómata definido como una tupla $E = (S_E, A_E, \rightarrow_E, \bar{s}, M_E)$, donde

- S_E es un conjunto finito de estados;
- A_E es un conjunto finito de etiquetas de eventos, particionado en eventos controlables y no controlables $A_E^C \cup A_E^U$;
- $\rightarrow_E$: $S_E \times A_E \mapsto S_E$ es una función parcial que representa las transiciones; y
- $\bar{s} \in S_E$ es el estado inicial; y
- $M_E \subseteq S_E$ es un conjunto de estados marcados.

La idea aquí es que los eventos no controlables representan sucesos externos, mientras que el controlador tiene poder de decisión sobre los controlables. Cuando el sistema se encuentra en un estado pueden suceder no determinísticamente tanto los eventos no controlables como los controlables que haya seleccionado el controlador. La relación $\rightarrow_E$ determina qué eventos son factibles en cada estado (y a qué nuevos estados llevan). Los estados marcados pueden representar la terminación de una tarea y son estados deseables para el controlador. El objetivo es encontrar un controlador que asegure que el sistema puede pasar infinitamente por los estados marcados (cosa que definiremos con precisión un poco más adelante en esta sección).

Este autómata define un lenguaje $\mathcal{L}(E) \subseteq A_E^*$, donde * denota la clausura de Kleene de la forma usual (el conjunto de trazas posibles de eventos). Una palabra $w = l_0 \dots l_t \in A_E^*$ pertenece a $\mathcal{L}(E)$ si existe una secuencia $s_0 l_0 s_1 \dots s_t l_t s_{t+1}$ con $(s_i, l_i, s_{i+1}) \in \rightarrow_E$ para $0 \leq i \leq t$, con $\bar{s} = s_0$. Es decir, si es la secuencia de etiquetas de un camino en el grafo determinado por S_E y $\rightarrow_E$. En tal caso notamos $\bar{s} \overset{w}{\dashrightarrow}_E s_t$.

Luego, un controlador es una función que basada en la traza actual observada decide qué eventos controlables habilitar. Dado un DES E, un controlador es una función $\sigma : A_E^* \mapsto \mathcal{P}(A_E^C)$. A partir de un controlador queda definido el conjunto de trazas que permite, que llamamos el lenguaje generado por σ y notamos $\mathcal{L}^\sigma(E)$. Una palabra $w = l_0 \dots l_t \in \mathcal{L}(E)$

pertenece a $\mathcal{L}^\sigma(E)$ si cada evento $l_i \in w$ es o bien no controlable o bien pertenece a $\sigma(l_0 \dots l_{i-1})$. Como comentario, es fácil ver que los controladores también se podrían definir con dominio S_E en lugar de A_E^* porque existe una biyección entre estos dos conjuntos dado que el autómata es determinístico. Teniendo en cuenta esto, puede resultar intuitivo pensar a un controlador como una selección de un subconjunto de las acciones controlables para cada estado de la planta.

Un controlador es un *director* si habilita a lo sumo una controlable en cada estado. Es decir, si $|\sigma(w)| \leq 1$ para todo $w \in A_E^*$. Esta definición constrasta con la noción más clásica de *supervisor* (Ramadge y Wonham, 1987), que es un controlador que habilita la mayor cantidad de eventos posibles.

En este trabajo tratamos con el problema de encontrar directores que satisfagan la propiedad *non-blocking*. La idea es que un controlador debe asegurar que desde todo estado alcanzable se puede alcanzar un estado marcado (incluso desde los estados marcados). Formalmente, un controlador σ para un DES dado E es non-blocking si para toda traza $w \in \mathcal{L}^\sigma(E)$, existe una palabra no vacía $w' \in A_E^*$ tal que la concatenación ww' pertenece a $L^\sigma(E)$, y además $\bar{s} \xrightarrow{ww'}_E s_m$ para algún $s_m \in M_E$.

Notar que con esta definición, un controlador non-blocking debe ser también *safe*, en el sentido de que no puede permitir que un estado deadlock sea alcanzable (i.e. un estado sin transiciones salientes). Los estados inseguros (también llamados ilegales o de error) pueden ser modelados como estados deadlock.

Si bien con esto ya está definido el problema que nos interesa, falta un concepto adicional importante a la hora de modelar la planta. Los DES con los que tratamos en este trabajo se definen modularmente como el producto sincrónico de sistemas más pequeños (Ramadge y Wonham, 1989), que llamamos composición paralela. Formalmente, la composición paralela ($\|$) de dos DES T y Q es el DES

$$T\|Q = (S_T \times S_Q, A_T \cup A_Q, \rightarrow_{T\|Q}, \langle \bar{t}, \bar{q} \rangle, M_T \times M_Q),$$

donde $A^C_{T\|Q} = A^C_T \cup A^C_Q$, y $\rightarrow_{T\|Q}$ es la relación más chica que satisface las siguientes reglas:

(I) si $t \xrightarrow{\ell}_T t'$ y $\ell \in A_T \backslash A_Q$ entonces $\langle t, q \rangle \xrightarrow{\ell}_{T\|Q} \langle t', q \rangle$,

(II) si $q \xrightarrow{\ell}_Q q'$ y $\ell \in A_Q \backslash A_T$ entonces $\langle t, q \rangle \xrightarrow{\ell}_{T\|Q} \langle t, q' \rangle$,

(III) si $t \xrightarrow{\ell}_T t'$, $q \xrightarrow{\ell}_Q q'$, y $\ell \in A_T \cap A_Q$ entonces $\langle t, q \rangle \xrightarrow{\ell}_{T\|Q} \langle t', q' \rangle$.

Destacamos tres características importantes de la composición paralela. En primer lugar, la regla (III) se ocupa de la sincronización entre componentes, asegurando que los

eventos de distintas componentes con la misma etiqueta se ejecuten a la vez, mientras que los de distintas etiquetas son completamente independientes. En segundo lugar, notar la cantidad de estados en $T_0 \| \ldots \| T_n$ crece proporcionalmente al producto de los tamaños de los autómatas (el peor caso es cuando no hay etiquetas compartidas). Finalmente, el lenguaje aceptado por la composición contiene las trazas que alcanzan estados marcados en todas las componentes simultáneamente. Es fácil ver que la composición paralela es asociativa y conmutativa, con lo cual podemos considerar la composición entre conjuntos de autómatas.

Un problema de control modular dirigido, que llamaremos simplemente *problema de control* en este trabajo, está definido por un conjunto de DES $E = \{E_1, \ldots, E_n\}$, y consiste en hallar un director non-blocking para la composición $E_1 \| \ldots \| E_n$. A su vez, los problemas de control que intentamos resolver son paramétricos. Un *dominio de problemas de control* es un conjunto de instancias $\Pi = \{E^p : p \in \mathcal{C}\}$, donde cada E^p es un problema de control (definido modularmente por un tuplas E_i^p) y $\mathcal{C}$ es un conjunto de posibles parámetros. En nuestro caso, todos los problemas de control de cada dominio son generados por una misma especificación que toma los parámetros p como entrada y está escrita en el lenguaje FSP (Magee y Kramer, 2014).

2.2. Síntesis de Controladores *on-the-fly*

La método tradicional para resolver el problema de control definido en la sección anterior consiste en construir la planta compuesta completa y ejecutar sobre ella un algoritmo (polinomial) como el presentado por Huang y Kumar (2008). Mientras que esto rápidamente se vuelve intratable por el tamaño de la planta, hay problemas para los cuales la explosión de estados puede ser retrasada construyendo un subconjunto pequeño de la planta que es suficiente para determinar una estrategia de control ganadora (o para concluir que no existe ninguna). La técnica de síntesis on-the-fly (OTF-DCS), presentada en detalle por Ciolek et al. (2023), está resumida brevemente en el Algoritmo 1. Consiste en partir del subgrafo que tiene solo al estado inicial y agregar una transición a la vez a una estructura de exploración parcial, tomada de la frontera de exploración. A medida que se construye la planta se mantiene una clasificación de estados en ganadores y perdedores y el algoritmo termina cuando hay suficiente información para clasificar al estado inicial. Una vez hecho esto, la construcción del controlador es inmediata.

Formalmente, dado $E = (S_E, A_E, \rightarrow_E, \bar{s}, M_E)$, un problema de control, y $h = \{a_0, \ldots, a_t\} \subseteq \rightarrow_E$, una secuencia de transiciones, la *frontera de exploración* de E luego de expandir h es $\mathcal{F}(E, h)$, el conjunto de transiciones $(s, \ell, s') \in (\rightarrow_E \setminus h)$ tal que $s = \bar{s}$

Algoritmo 1 Algoritmo de exploración on-the-fly.

Input: Un problema de control E y una heurística H para E.

$\bar{s} \leftarrow (\bar{s}^1, \ldots, \bar{s}^n)$
$h \leftarrow$ Lista vacía.
$ES \leftarrow (\{\bar{s}\}, A_E, \emptyset, \bar{s}, M_E \cap \{\bar{s}\})$
$WinningStates \leftarrow \emptyset$
$LosingStates \leftarrow \emptyset$
while $\bar{s} \notin WinningStates \cup LosingStates$ **do**
  $a \leftarrow$ acción seleccionada de $\mathcal{F}(E, h)$ usando H.
  $expandAndPropagate(a, ES, WinningStates, LosingStates)$
  Agregar a a h al final.
if $\bar{s} \in WinningStates$ **then**
  retornar $buildController(h, WinningStates)$
else
  retornar UNREALIZABLE

o $(s'', \ell, s) \in h$ para algún s''. Es decir, las transiciones que van desde un estado explorado hacia uno no explorado. Una *secuencia de exploración* para E es $\{a_0, \ldots, a_t\} \subseteq \rightarrow_E$ tal que $a_i \in \mathcal{F}(E, \{a_0, \ldots, a_{i-1}\})$ for $0 \leq i \leq t$. Es decir, una secuencia de transiciones tal que en cada paso se seleccionó una transición de la frontera.

La función $expandAndPropagate$ es la encargada de clasificar a los estados expandidos en ganadores, perdedores, o ninguno de los dos (si aún se desconoce). Dada una planta E, decimos que un estado $s \in E$ es *ganador* (resp. *perdedor*) en una planta E si existe (resp. no existe) una solución para E_s, donde E_s es el resultado de cambiar el estado inicial de E a s (donde una solución es un director nonblocking). Esencialmente, un estado será ganador si es parte de un circuito que contiene un estado maracado y no tiene eventos no controlables hacia afuera del circuito, o si puede controlablemente llegar a un estado ganador. Un estado será perdedor si no tiene un camino a un estado marcado, si puede ser forzado por eventos no controlables hacia un estado perdedor, o si todos sus eventos son controlables pero llevan a estados perdedores. Para una secuencia de exploración incompleta un estado se define como ganador (resp. perdedor) si lo es al asumir que toda transición en la frontera de exploración va hacia un estado perdedor (resp. ganador).

Es importante destacar que la clasificación del algoritmo es correcta y completa: nunca marca como ganador o perdedor a un estado que no lo es (falsos positivos) y nunca deja sin marcar un estado que es ganador o perdedor en la exploración incompleta (falsos negativos). En consecuencia, necesariamente se llega a un veredicto (correcto) sobre el estado inicial sin importar el orden en el que se expandan las transiciones, en peor caso luego de agregar la última transición de la planta. Las heurísticas entonces intentarán

minimizar la cantidad de transiciones exploradas, pero incluso con las peores decisiones la correctitud del algoritmo de síntesis se preserva.

2.2.1. Ready Abstraction

En conjunto con OTF-DCS, Ciolek et al. (2023) propusieron y evaluaron múltiples heurísticas independientes del dominio diseñadas manualmente. Una de ellas, la que obtuvo mejores resultados y contra la que compararemos nuestra técnica, es la *Ready Abtraction* (RA). Su idea principal es estimar (de forma conservativa) la distancia a los estados marcados usando un grafo de dependencias. Este grafo se construye a partir de las componentes individuales de la planta (los autómatas E_i antes de la composición paralela) y se basa fuertemente en las restricciones de sincronización de la composición paralela. Si el problema fuera de alcanzabilidad (el sistema termina cuando se llega a un estado marcado) y no hubiera transiciones no controlables, esta idea por si sola sería muy razonable.

Como no es ese el caso, es necesario además considerar de alguna forma los eventos no controlables y los estados marcados que ya fueron visitados. Para esto, la decisión de RA es priorizar las no controlables por sobre las controlables, tomando entre ellas primero las que más se alejan de los estados marcados. Además, siempre que mide distancias prioriza las distancias hacia los estados marcados ya visitados. Finalmente, tiene un mecanismo para manejar la frontera de exploración en el que no siempre todos los estados de la frontera están disponibles, para priorizar avanzar en profunidad en ciertos casos.

Si bien este conjunto de decisiones posibilita sus muy buenos resultados en varios dominios, el comportamiento es altamente complejo y difícil de entender y mejorar. En este sentido, es interesante la idea de brindarle toda la información que creemos útil a un algoritmo y desligarnos de la responsabilidad de tomar decisiones heurísticas al respecto.

3. APRENDIZAJE POR REFUERZO

3.1. Interacción agente-ambiente

Mientras que las formas de aprendizaje supervisada y no supervisada toman como entrada conjuntos de datos, aprendizaje por refuerzo es un subparadigma de Aprendizaje Automático que considera un agente que interactúa iterativamente con su ambiente con el objetivo de acumular la máxima cantidad posible de recompensa. En cada paso el agente observa (quizás parcialmente) el estado en el que se encuentra y toma una acción, a lo cual el ambiente responde con la recompensa obtenida y una nueva observación, como lo muestra el diagrama de la figura 3.1.

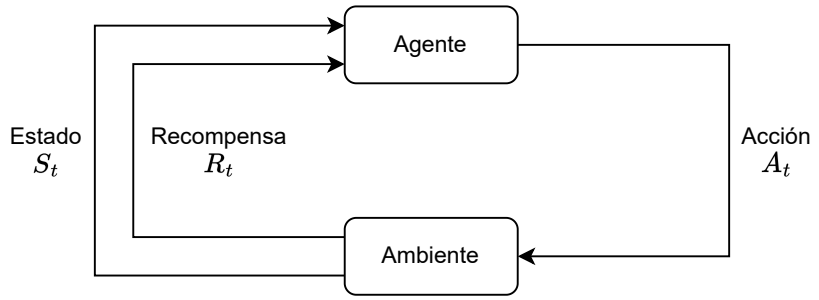

Figura 3.1: Diagrama de un MDP

Una tarea de RL episódica[1] puede ser formalizada como un *Proceso de Decisión de Markov* (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, S_0)$ donde

- $\mathcal{S}$ es un conjunto de estados;
- $\mathcal{A}$ es un conjunto de acciones;
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ codifica la probabilidad $Pr\{s'|a, s\}$ de observar el estado s' luego de seleccionar la acción a en el estado s;
- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ es una función de recompensa y
- s_0 es el estado inicial.

[1] En contraposición con las tareas continuas con un único episodio infinito, de las que no hablaremos.

Además, el conjunto de acciones disponibles del estado $s \in \mathcal{S}$ es denotado por $\mathcal{A}(s)$. En este contexto, el comportamiento de un agente está determinado por su *política*, que es una función $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ que indica la probabilidad de tomar la acción a en el estado s, usualmente notada $\pi(a|s)$, con la restricción obvia de que las probabilidades para cada estado deben sumar 1. Si fijamos una política π tenemos definidas variables aleatorias R_t, S_t, A_t y T que denotan la recompensa obtenida, el estado y la acción elegida en tiempo t, y la cantidad total de pasos del episodio, respectivamente. Además, dado un estado s queda definido el valor esperado de la recompensa total al partir de s y utilizar π. Lo llamamos *valor* del estado s y lo notamos $v_\pi(s)$, y es exactamente

$$v_\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{T} R_t | S_0 = s].$$

Si bien podríamos contentarnos con esto y decir que el objetivo de los algoritmos de RL es hallar una política π^* que maximice $v_\pi(s_0)$, una propiedad sutil pero no menor de los MDP es que existe (al menos) una política que es óptima para todos los estados a la vez. Llamamos π^* a esta política, que cumple $v_{\pi^*}(s) \geq v_\pi(s) \forall \pi, s$. La prueba de esto es sencilla, basta tomar la política que elige en cada estado la distribución de acciones elegida por la mejor política de ese estado.

3.2. Q-Learning

Múltiples métodos fueron desarrollados a lo largo de los años para abordar el problema de la sección anterior. Q-Learning (Watkins y Dayan, 1992) se basa fundamentalmente en la idea de estimar una función de valor de pares estado-acción. En forma similar a los valores de estados, dada una política π podemos definir el valor de un par estado-acción con la función $Q_\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, donde $Q_\pi(s, a)$ indica la recompensa total esperada si se parte del estado s tomando la acción a y se utiliza π a partir de allí. Es decir,

$$Q_\pi(s, a) = \mathbb{E}_\pi[\sum_{t'=t}^{T} R_{t'} | S_t = s, A_t = a].$$

Luego, podemos definir el valor bajo juego óptimo como $Q_*(s, a) = \max_\pi Q_\pi(s, a)$. Si conociéramos esta función para un MDP, sabríamos cómo jugar óptimamente porque la política que en todo estado toma la acción que maximiza q_* con probabilidad 1 es óptima. Notar que esta función es ligeramente más útil que la función de valor de estado porque permite elegir una acción iterando sobre las acciones, sin necesidad de simular un paso del

ambiente.

Lo que caracteriza a Q-Learning no es el uso de la función Q, pues otros algoritmos también la usan, sino su regla para aprenderla. La idea es aproximar Q_* aprovechando la relación entre el valor de un estado y el de los estados a los que se puede llegar en un paso:

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a) | S_t = s, A_t = a]. \tag{3.1}$$

Q-Learning en su versión tabular no es más que aplicar el punto fijo asociado a esta relación sobre una tabla, con el detalle de que en vez de sumar sobre los posibles S_{t+1} (como lo indicaría la definición de esperanza), se utiliza el estado concreto al que llegó el agente interactuando con el ambiente. La regla resultante a aplicar en cada paso t es

$$Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha(R_{t+1} + \max_{a \in A_{S_{t+1}}} Q_t(S_{t+1}, a) - Q_t(S_t, A_t)), \tag{3.2}$$

donde α es un parámetro que determina el tamaño del salto. Si todos los pares estado-acción son visitados infinitas veces y α, el parámetro de longitud del salto, se decrementa adecuadamente, la convergencia a Q_* está garantizada.

Esta regla de aprendizaje es *off-policy* porque no requiere samplear estados y acciones con ninguna política en particular. Si bien iterar repetidamente sobre la tabla completa es posible, lo más común es utilizar una política ε-greedy, que toma la mejor acción según la Q actual con probabilidad $1 - \varepsilon$ y una al azar de las disponibles si no. Esto garantiza la exploración eventual de todo el espacio de estados pero la sesga hacia los estados de interés para la política actual para acelerar el aprendizaje. Una observación importante es que al hacer esto nuestra regla de aprendizaje modifica a la vez la función de valor y la política con la que se obtiene la experiencia. En este sentido se va dando un proceso iterativo en el que la función de valor se acerca a la evaluación de la política actual Q_π y esto modifica a π haciendo que Q se tenga que adaptar de vuelta, y así sucesivamente.

Por otro lado, se dice que Q-Learning hace *bootstrapping*, porque actualiza la función de valor en base a sí misma, en lugar de métodos alternativos como Monte Carlo que toman el promedio de los resultados finales de cada episodio.

3.2.1. Aproximación de funciones

Cuando el espacio de pares estado-acción es grande la versión tabular de Q-Learning se vuelve impráctica, y es usual reemplazar la tabla por un aproximador $\hat{Q}_w(s,a)$ que depende de un vector de pesos w. Es decir, se toma una familia de funciones $\{\hat{Q}_w\}_w$ y se busca dentro de ese conjunto la más cercana a ser un punto fijo para la regla 3.2, midiendo

la distancia por ejemplo con el error cuadrático medio:

$$(R_{t+1} + \max_{a \in A_{S_{t+1}}} \hat{Q}_w(S_{t+1}, a) - \hat{Q}_w(S_t, A_t))^2.$$

Si $\hat{Q}_w$ es diferenciable, esta búsqueda se puede realizar con descenso por gradiente estocástico sobre los pesos w. Notar que el uso de la palabra estocástico se debe a que usamos una estimación del error con el S_{t+1} que sucedió efectivamente, en lugar de calcular el valor esperado, que sería el target exacto. Las familias de funciones típicamente usadas son o bien funciones lineales o bien redes neuronales, con particular éxito para las redes profundas en los últimos años (en lo que se llama Deep RL). En este trabajo utilizaremos perceptrones multicapa relativamente pequeños.

Es importante observar que en este contexto el entrenamiento de la red neuronal es muy similar al de aprendizaje supervisado: tenemos un conjunto de asociaciones entre pares estado-acción (s, a) y sus valores esperados dados por el lado derecho de 3.1. Por lo tanto, muchas de las herramientas desarrolladas para aprendizaje supervisado aplican aquí también. Aunque no menor, la única diferencia es que las asociaciones están fuertemente correlacionadas entre sí por venir dadas por la experiencia secuencial del agente interactuando con el ambiente, violando la hipótesis usual de sampleo i.i.d.

Esto último, en conjunto con algunos ejemplos encontrados, hace que a la combinación de aproximación de funciones, reglas off-policy y bootstrapping se le atribuya el nombre de *deadly triad*, por su inestabilidad y su capacidad para causar divergencia (e.g. ver el Capítulo 11 de Sutton y Barto (2018)). Dos técnicas que fueron propuestas para restaurar la estabilidad son *Experience Replay* y *Fixed Q-Targets* (Lin, 1992; Mnih et al., 2013). La idea de la primera es, en vez de actualizar en forma directa a partir de cada observación, se actualiza usando pequeños conjuntos de experiencias (tuplas (s, a, r, s')) sampleados al azar de las últimas B experiencias. La idea de la segunda es mantener una función *target* que se actualiza periódicamente como copia de $\hat{Q}_w$ y usarla en reemplazo de Q en el lado derecho de la ecuación 3.1, con el objetivo de estabilizar el target hacia el que se mueve la función $\hat{Q}$.

4. OTF-DCS CON APRENDIZAJE POR REFUERZO

4.1. Enmarcando OTF-DCS como una tarea de RL

En esta sección definiremos un MDP que representa exactamente el problema de minimizar la exploración. Dadas las similitudes entre MDP y DES, es tentador definir un MDP como una traducción directa entre los modelos, donde los estados y acciones del DES corresponden a estados y acciones en el MDP, y la recompensa en el MDP se otorga cuando se visita un estado marcado en el DES. Un problema con esto es que no es obvio cómo modela los eventos no controlables. Es necesario distinguirlas de algún modo para modelar el hecho de que son aversarias. Una forma quizás natural de hacerlo es definir que, en lugar de ser acciones como las controlables, se codifican con cierta probabilidad de suceder en la dinámica del ambiente. Sin embargo, esto no modela exactamente la tarea de control porque un controlador no tiene que maximizar la probabilidad de llegar a estados marcados, sino asegurarla bajo cualquier ejecución posible de no controlables.

Un problema quizás más fundamental de este MDP es que lo que buscamos no es modelar la tarea de control sino la tarea de exploración de la planta. De hecho, las políticas de este MDP permitirían seleccionar transiciones para un estado dado de la planta, y esto no es suficiente para elegir una transición en la frontera de exploración. Si bien la idea de usar una política como esta como parte de una heurística es interesante, en este trabajo presentamos un MDP que resuelve en forma exacta nuestro problema.

En nuestro MDP, los estados se definen como estados del proceso de exploración (dados por la secuencia de transiciones expandidas), y una acción representa la expansión de una transición (un par estado-evento del DES). La dinámica (P) es la dada por agregar transiciones de la frontera a la secuencia hasta llegar a un estado terminal y las recompensas son siempre -1. Un estado terminal es un estado de exploración en el que el estado inicial del DES fue marcado como ganador o perdedor por OTF-DCS.

Más formalmente, dado $E = (E_1, \ldots, E_n)$, un problema de control, definimos el MDP asociado como $(\mathcal{S}, \mathcal{A}, P, r, S_0)$, donde

- $\mathcal{S} = \{h : h$ es una secuencia de exploración $E\}$;
- $\mathcal{A} = S_E \times A_E$, $\mathcal{A}(s) = \{(s, \ell) : \exists (s, \ell, s') \in F(E, S_E)\}$;
- $P(s'|s, a) = 1$ si $a \in \mathcal{F}(E, s) \wedge s' = sa$, y 0 si no;

- $r(s, a, s') = -1 \; \forall s, a, s'$;
- $(s, E) \in \mathcal{S}$ es un estado terminal si el estado inicial es marcado como ganador o perdedor luego de expandir la secuencia s en E;
- $S_0 = (\emptyset, E)$.

Una propiedad positiva de este MDP es que es una representación exacta de nuestro problema: una política con recompensa $-R$ mapea directamente a una heurística que expande R transiciones en OTF-DCS. Notar que evitamos recompensar por ejemplo cada vez que se llega a un estado marcado ya que, si bien podría facilitar el aprendizaje inicialmente, induciría un sesgo hacia políticas subóptimas.

El problema de este MDP es que las señales de estado y acción son completamente imprácticas. En primer lugar, el estado es una secuencia de transiciones exploradas que conforman un grafo, que no puede ser procesada por redes neuronales tradicionales que toman como entrada un vector de dimensión fija. En segundo luegar, el espacio de acciones es grande y solo un subconjunto de tamaño variable de acciones está disponible en cada paso (la frontera). Notar además que las acciones en este MDP se toman a lo sumo una vez en cada episodio.

Incluso más desafíos aparecen dado que, como vamos a discutir en las próximas secciones, queremos que la política de exploración aprendida esté bien definida en instancias más grandes de un dominio. Los estados de la planta en distintas instancias son diferentes (porque las plantas son composiciones de distintos conjuntos de autómatas) y las etiquetas de los eventos usualmente también cambian, por ejemplo porque pueden referenciar a autómatas individuales. Más aún, la cantidad de acciones crece no acotadamente con el tamaño de las instancias.

4.2. Abstracción del estado de la exploración

Para resolver los problemas mencionados, proponemos abstraer el subgrafo explorado de la planta y las transicciones disponibles en la frontera, describiéndolos con un conjunto de features $\phi(s, a) = (\phi_1(s, a), \ldots, \phi_{d_E}(s, a)) \in \mathbb{R}^{d_E}$. Luego, los agentes observan en cada estado una lista de vectores de features, uno por cada transición disponible en la frontera (las features que describen el estado de la exploración se replican por cada transición).

Esta featurización tiene múltiples ventajas. En primer lugar, el espacio de features tiene un tamaño fijo (d_E), permitiendo usar una red neuronal tradicional. En segundo

lugar, facilita el aprendizaje al simplificar la señal de estado y enriquecer las acciones; en particular, permite generalizar a través de estados y acciones con características similares, cosa que no sería posible con una identificación granular sin estructura. En tercer lugar, y quizás más importante, si los agentes aprenden en un espacio de features que es único para un conjunto de instancias, la política aprendida en una instancia *induce* políticas en todas las otras.

Por otro lado, la featurización podría (y en nuestro caso lo hará) introducir observabilidad parcial de los estados y de las acciones. Tener estados parcialmente observables remueve la hipótesis de Markovianidad de la tarea (el futuro no depende únicamente de la observación actual) y es una complicación importante tanto para el aprendizaje (dado que Q-Learning fue pensado para MDPs) como para las políticas resultantes. Incluso si aprender es posible, la calidad de las políticas de exploración aprendidas claramente dependerá de la habilidad de las features de separar buenos pares estado-acción de malos pares. Más aún, la calidad de las políticas inducidas solo será preservada en otras instancias si los pares estado-acción de diferentes instancias con features similares son similarmente buenos (en términos del número de transiciones que puede ser expandido al usarlos). Discutiremos los problemas teóricos generados por la abstracción en el capítulo 5.

4.3. Algoritmo de Aprendizaje

En esta sección describiremos cómo puede ser usado Q-Learning con una red neuronal para resolver nuestra tarea de RL. El algoritmo usado es esencialmente DQN (Mnih et al., 2013). Sin embargo, un detalle que no permite aplicarlo en forma directa es que DQN asume un conjunto fijo de acciones (relativamente chico) para todos los estados que, como fue discutido en la subsección previa, no es el caso en nuestra tarea. Esta limitación se debe a que las Deep Q-Networks fueron planteadas con una neurona de salida por cada acción. En su lugar, nosotros evaluamos cada acción por separado usando una red con una única salida, que toma como entrada las características de la acción a evaluar. Más precisamente, la entrada de la red es el vector de features $\phi(s, a) \in \mathbb{R}^{d_E}$ para cada par estado-acción $(s, a) \in \mathcal{S} \times \mathcal{A}$. La red estima la función de valor $Q_*(s, a)$ a través de $Q_w(\phi(s, a))$, donde w es un conjunto de pesos, y la regla de actualización de Q-Learning es utilizada. Formalmente, esto puede verse como Q-Learning con aproximación de funciones, usando a la composición $Q \circ \phi$ como aproximador, con lo cual no tenemos garantías teóricas de convergencia.

Notar que como por cada transición expandida se recibe una recompensa de -1 y Q es la suma esperada de las recompensas, los valores de Q serán estimaciones de la cantidad de

Algoritmo 2 Q-Learning con aproximación de funciones para la tarea de control on-the-fly.

Input: Un problema de control E y un presupuesto T.

$Env \leftarrow$ ambiente OTF-DCS para E.
Inicializar una red neuronal Q con pesos al azar y dimensión de entrada d_E.
Inicializar Q' como copia de Q.
Inicializar buffer B con observaciones de la política aleatoria.
$S_0 \leftarrow reset(Env)$.
for $t = 0$ **to** T **do**

$$a_t \leftarrow \begin{cases} \text{a random action} & \text{with probability } \varepsilon \\ arg \max\limits_{a \in \mathcal{A}(S_t)} Q(\phi(S_t, a)) & \text{otherwise} \end{cases}$$

$S_{t+1} \leftarrow$ Expandir y propagar a_t en Env.
Agregar $(\phi(S_t, a_t), S_{t+1})$ a B.
Samplear transiciones $(\phi(S_j, a_j), S_{j+1})$ uniformemente de B

$$\delta_j \leftarrow -1 + \begin{cases} 0 & \text{if } S_{j+1} \text{ is terminal} \\ \max\limits_{a \in \mathcal{A}(S_{j+1})} Q'(\phi(S_{j+1}, a)) & \text{otherwise} \end{cases}$$

Realizar un paso de descenso por gradiente en Q con minibatch $(\phi(S_j, a_j), \delta_j)$.
$Q' \leftarrow Q$ cada vez que se realiza cierta cantidad de pasos.
$S_{t+1} \leftarrow reset(Env)$ si S_{t+1} es terminal

transiciones que se espera expandir hasta terminar la tarea luego de expandir la transición a en el estado s (con un signo menos adelante).

En el Algoritmo 2 se muestra un pseudocódigo del algoritmo de aprendizaje. En primer lugar, Q se inicializa con dimensión de entrada igual a d_E y pesos aleatorios y una red target Q' se crea como copia. El buffer de replay se incializa usando observaciones de una política que elige una transición al azar en cada paso. Luego, el agente sintetiza el mismo problema repetidamente hasta que se termina el presupuesto (medido en cantidad de expansiones). En el paso t el vector de featuras $\phi(S_t, a)$ para cada transición a en la frontera de exploración es evaluado usando Q y una acción ε-greedy es elegida. El ambiente propaga los veredictos de estados ganadores y perdedores en la planta explorada acordemente y la nueva experiencia se agrega al buffer (removiendo la experiencia más antigua de ser necesario). A nivel implementativo, un vector de vectores de features, uno por cada transición $a \in \mathcal{A}(S_{t+1})$ es guardado en lugar de S_{t+1}. Luego de cada paso se actualiza Q con un minibatch sampleado uniformemente del buffer. El valor objetivo δ_j para la experiencia $(\phi(S_j, a_j), S_{j+1})$ es, si S_{j+1} no es terminal, el valor del mejor vector de features en S_{j+1}, según Q', menos uno (la recompensa). Una vez cada cierta cantidad de time steps la red target se actualiza con una nueva copia de Q. Finalmente, cada vez que se llega a un estado terminal el proceso de síntesis se reinicia.

Asintóticamente, la evaluación de la red no induce un overhead dado que la complejidad de cada iteración de OTF-DCS (expandir y propagar los veredictos) es $O(|S_{ES}|^2 \times |A_E|)$ y la cantidad de transiciones en la frontera está acotada por $|S_{ES}|^2$. En la práctica sin embargo, podría pasar que la cota de peor caso para el procedimiento de propagación sea raramente alcanzada y la evaluación de una red neuronal grande podría generar un overhead temporal significativo.

4.4. Generalizando a instancias más grandes

Este trabajo tiene como objetivo mejorar la escalabilidad del algoritmo de síntesis hacia sistemas más grandes. Específicamente, dado un dominio de control Π, queremos encontrar políticas de exploración que permitan resolver instancias que actualmente no pueden ser resueltas con cantidades razonables de recursos (tiempo o memoria). En este sentido, dado que entrenar un agente de RL requiere jugar episodios repetidamente, nuestro diseño punta a punta tiene una restricción importante: *el entrenamiento no puede ser realizado en las instancias que queremos resolver*. Para resolver esto, proponemos una metodología que aprovecha lo que puede ser aprendido en instancias relativamente chicas de un problema e intenta extrapolar lo aprendido hacia las versiones más grandes.

Claramente, para que esto sea posible, las instancias grandes tienen que estar relacionadas de algún modo a las instancias de entrenamiento. Esta *hipótesis de homogeneidad* en nuestro caso se basa en el hecho de que todas las instancias $E_p \in \Pi$ se definen usando la misma especificación paramétrica en el lenguaje FSP.

Dado que, presumiblemente, nuestro algoritmo de aprendizaje es capaz de producir buenas políticas de exploración para una instancia dada, y que la función Q que produce *induce* políticas en todas las instancias del mismo dominio, nuestro enfoque para resolver las instancias más grandes posibles para un dominio dado Π consiste en los siguientes tres pasos:

(P1) Entrenar en una instancia E_{p_0} (tal como es descripto en la Sección 4.3), guardando N agentes sampleados uniformemente de distintos puntos del entrenamiento.

(P2) Testear las políticas obtenidas en **(P1)** en cada instancia $E_p \in \Pi$ con un budget pequeño de transiciones. La política que generaliza mejor (i.e. resuelve la mayor cantidad de instancias, desempatando según transiciones expandidas totales) es elegida.

(P3) La política elegida en **(P2)** es utilizada con un budget completo para resolver tantas instancias como sea posible de Π.

Si bien nuestra hipótesis de homogeneidad sugiere que buenas performances en la instancia de entrenamiento correlacionan en algún sentido con buenas performances en instancias más grandes, es claramente posible para un conjunto de pesos dado ser una excepción para esta idea. Una política podría estar overfitteada a la instancia de entrenamiento de dos maneras. En primer lugar, podría tomar buenas decisiones únicamente en su trayectoria determinística de expansiones, obteniendo malos resultados en caso de ser forzada a jugar desde algún otro estado de la misma instancia, como fue ejemplificado en el Arcade Learning Environment Machado et al. (2018). En segundo lugar, y quizás más importante en nuestro caso, un agente podría aprender una estrategia robusta que depende de características específicas de la instancia de entrenamiento y no generaliza bien. El paso **(P2)** es importante para abordar la potencial diversidad en capacidades de generalización de los conjuntos de pesos obtenidos durante el entrenamiento.

Otra idea que podría tener un impacto positivo en la generalización es detener el entrenamiento relativamente rápido. Esto podría ser útil siguiendo la idea común de aprendizaje supervisado de frenar el aprendizaje cuando la performance en el conjunto de test empieza a decrecer. Sin embargo, la performance en nuestro caso es bastante ruidosa y no tenemos evidencia fuerte de que ese fenómeno se replique completamente. Otro motivo similar y un poco más sencillo para hacer esto es que las políticas podrían ser más diversas durante las primeras etapas del entrenamiento, antes de converger, haciendo más alta la probabilidad de encontrar una buena estrategia general.

4.5. Definición de un vector de features

La definición del conjunto de features que componen la señal de estado-acción, describiendo las transiciones de la frontera y el estado general de exploración es una parte clave de nuestro enfoque. La función de features $\phi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}^{d_{E_p}}$ debería ser suficientemente informativa para permitir buenas políticas. Sin embargo, está considerablemente restringida por el objetivo de generalización. En primer lugar, la cantidad de features (d_{E_p}) debe ser la misma para todas las instancias de un dominio, dado que es la dimensión de entrada de la red neuronal. En segundo lugar, la semántica de cada elemento del vector de features se debería mantener y su distribución debería alterarse lo menos posible al cambiar de instancias. Una restricción más filosófica es que las features tienen que poder extraerse automáticamente de E, no pueden ser definidas ad-hoc para cada dominio. Por último, considerando que se calculan y evalúan todos los vectores de features en cada expansión, no pueden ser demasiado costosas computacionalmente.

Si bien es perfectamente posible usar features con valores reales, y de hecho hasta

Feature (tamaño)	Descripción
Etiqueta de evento ($\|A_{E_p}\|$)	Determina la etiqueta ℓ en A_{E_p}.
Etiquetas de estado ($\|A_{E_p}\|$)	Determina las etiquetas en A_{E_p} de las transiciones exploradas que llegan a s.
Controlable (1)	Indica si $\ell \in A^C_{E_p}$ o $\ell \in A^U_{E_p}$.
Estado marcado (2)	Indica si s y $s' \in M_{E_p}$.
Fases (3)	Si (en algún punto del episodio) un estado marcado fue econtrado, un estado ganador fue determinado, o un ciclo que contiene un estado marcado fue cerrado.
Estado de llegada (3)	Indica si s' es ganador, perdedor, si todavía no se sabe, o si no fue explorado.
No controlables (4)	Indica si s y s' tienen eventos no controlables salientes y si fueron explorados.
Explorados (2)	Indica si ya se exploró algún evento desde s o s'.
Última expansión (2)	Indica si s es el último estado expandido en h (saliente o entrante).

Tabla 4.1: Features que describen el par estado-acción $(h, (s, \ell, s'))$ para un problema de control E_p. El tamaño refiere a la cantidad de booleanos usados para cada feature.

cierto punto en el proyecto lo hicimos, su rango tiende a variar entre distintas instancias, complejizando la generalización. Para evitar que la red encuentre valores nuevos en instancias más grandes, el vector utilizado está compuesto únicamente por features booleanas. El conjunto específico utilizado se muestra en la tabla 4.1. Notar que todas las features son o bien muy baratas de calcular o bien ya son calculadas como parte de OTF-DCS.

Un problema con las primeras dos features es que el conjunto A_{E_p} usualmente depende de los parámetros de instancia p. Por ejemplo, en Air Traffic, uno de los problemas del benchmark utilizado (Ciolek et al., 2019), hay una etiqueta de acción `land.i` por cada avión `i`, y el número de aviones es una de las dimensiones de p. Estos índices tienen que ser removidos para mantener fija d_{E_p}. En el ejemplo usamos una única etiqueta `land` para el cálculo de features. Esta es una limitación significativa para las políticas exploración aprendidas dado que no son capaces de distinguir distintas componentes del mismo tipo (aviones en este caso).

Otra limitación que vale la pena destacar en este punto es que las features no dan información sobre la posición de las transiciones con respecto a la planta. Las etiquetas del estado intentan dar información sobre el estado de salida pero no lo caracterizan unívocamente (usar el id de estado es imposible porque generaría confusión al cambiar de instancias). Tampoco describen qué eventos sucedieron antes de alcanzarlo más allá los últimos. En este sentido, a nuestros agentes les falta mucha información de la topología de

la planta.

La idea de la feature para las fases fue tomada de la Tesis de Licenciatura de Hernán Gagliardi, en donde se mostró la importancia que puede tener cambiar la estrategia de exploración al alcanzar cada subtarea de la exploración.

Un detalle a mencionar es que no indicamos si el estado de salida (s) es ganador o perdedor porque se puede probar que nunca conviene explorar transiciones salientes de estados que ya tengan un veredicto, esencialmente porque no aportan información útil.

5. Q-LEARNING EN VERY PARTIALLY OBSERVABLE MDPS

Este capítulo estudia un formalismo que denominamos Very Partially Observable MDP, con el que buscamos enmarcar formalmente a la tarea de RL que resolvemos. Nuestro desarrollo está muy fuertemente basado en el trabajo de Singh et al. (1994) para Partially Observable MDPs. Aunque en estos temas tenemos muchas más preguntas abiertas que cerradas, este análisis conforma los primeros pasos hacia la comprensión teórica de nuestra técnica y permite ejemplificar sus limitaciones.

En la sección 3.2 Q-Learning quedó bien definido para MDPs. Sin embargo, el salto a utilizarlo con observaciones parciales del estado no debería ser subestimado. Como mencionamos en el capítulo 4, un motivo para este salto es que el par (s, a) no necesariamente tiene un formato adecuado para ser tomado como entrada de las funciones aproximadoras, y en el cambio de formato es posible perder información. Más allá de esta necesidad técnica, es poco común en las aplicaciones que el agente observe completamente el estado en el que se encuentra: los seres humanos solo percibimos lo que nos muestran los sentidos, no observamos todas las variables del universo, ni siquiera vemos lo que está detrás del objeto que tenemos en frente. Esta asunción clásica de RL es en muchos casos demasiado fuerte y está perdiendo valor (e.g. Sutton et al., 2022).

Como mencionamos en la sección 4.3, el problema de observabilidad parcial es un caso particular del de aproximación de funciones, para el cual hay escasos resultados teóricos. Sin embargo, vale la pena analizarlo en forma concreta y aislada. Una forma de hacer esto es cambiar el modelo de interacción de la figura 3.1 por uno más representativo de nuestra situación. Un modelo que se acerca bastante a nuestras necesidades es el de Partially Observable MDP (POMDP). Un POMDP es $(\mathcal{M}, \mathcal{O}, Z)$ con $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, S_0)$ un MDP, $\mathcal{O}$ un conjunto de observaciones posibles y $Z : \mathcal{S} \mapsto \Delta(\mathcal{O})$ una función de observación (donde $\Delta(\mathcal{O})$ es el conjunto de distribuciones sobre $\mathcal{O}$).

Si quisiéramos usar un POMDP para modelar nuestra tarea, podríamos. El MDP subyacente ya lo tenemos y las observaciones serían vectores de vectores de features. Las acciones serían un índice indicando el vector de features elegido. Sin embargo, esto no es del todo satisfactorio porque nuestro agente evalúa un par estado-acción a la vez, no todos juntos, y esa partición del estado en las descripciones de las acciones no está explícita en el modelo. Dado un POMDP, la función de valor que tiene sentido definir es $q(o, a)$. Sin embargo, nuestra función de valor depende de una observación parcial del estado y de la acción a la vez: $q(\phi(s, a))$. Esto motiva a definir la siguiente variante de MDP, que

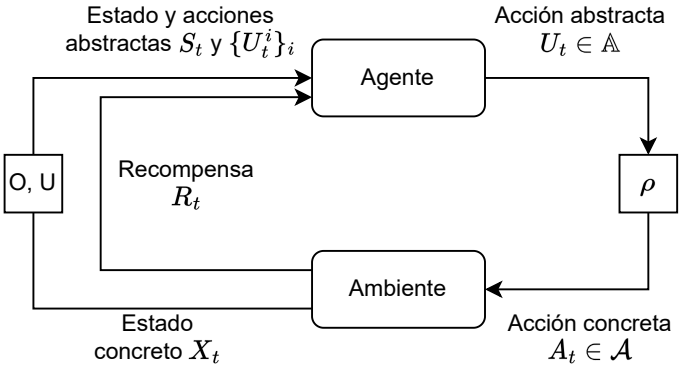

Figura 5.1: Diagrama de un VPMDP

llamamos Very Partially Observable MDP.

Definición (VPOMDP)**.** Un VPOMDP es $(\mathcal{M}, \mathbb{S}, \mathbb{A}, O, U, \rho)$ con $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, S_0)$ un MDP, $\mathbb{S}$ y $\mathbb{A}$ los espacios de estados y acciones abstractas, respectivamente, y $O : \mathcal{S} \mapsto \mathbb{S}$, $U : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{A}$ las funciones observación de estados y acciones, respectivamente. $\rho : \mathcal{S} \times \mathbb{A} \mapsto \mathcal{A}$ es el criterio que define qué acción concreta de $\mathcal{A}$ se toma cuando una acción abstracta de $\mathbb{A}$ es elegida en un estado.

En un VPOMDP, en cada paso t el agente se encuentra en un estado subyacente $S_t \in \mathcal{S}$ con acciones disponibles $\mathcal{A}(S_t) \in \mathcal{A}$ y observa $O_t = O(S_t) \in \mathbb{S}$, un resumen del estado, y $\{U(S_t, a) : a \in \mathcal{A}(S_t)\}$, un conjunto de descripciones abstractas de las acciones disponibles, y elige una de ellas. El diagrama de interacción actualizado es el de la figura 5.1. Notar que así como el agente no sabe en qué estado subyacente está, tampoco sabe qué acciones tiene disponibles, solo conoce un resumen de sus características. La necesidad de utilizar a ρ viene del caso en el dos acciones concretas tienen igual imagen por U, se utiliza para desempatar. Lo natural es que $\rho(s, u) \in U^{-1}(s, \rho(s, u))$. Es decir, que la acción concreta tomada tenga la descripción elegida. Además, obviamente debería ser una de las acciones disponibles ($\mathcal{A}(S_t)$). Como nota de color, en nuestro problema elegimos siempre la acción que primero entró a la frontera de exploración.

Una posible definición de política (determinística) para un VPOMDP es una función $\pi : \mathbb{S} \times \mathbb{A} \mapsto \mathcal{R}$ que, dada una observación del estado y una descripción de una acción disponible devuelve un número real. Los valores en sí no nos interesan, una definición también válida es retornar un orden total sobre las descripciones de acciones dado $\mathbb{S}$.

Una política π en el VPOMDP define una política $\bar{\pi}$ en el MDP subyacente de la forma natural: $\bar{\pi}(s)$ toma la acción disponible cuya descripción es elegida por π dada la observación $O(s)$, usando ρ para desemptar si hay más de una acción disponible con

la misma descripción. Bajo esta definición, podemos definir para VPOMDPs el retorno esperado de un agente con política π en el estado s en términos de la función v del MDP subyacente: $v_\pi(s) = v_{\bar{\pi}}(s)$. Similarmente, $q_\pi(s,a) = q_{\bar{\pi}}(s,a)$. También quedan definidos $v_*(s) = \max_\pi v_{\bar{\pi}}(s)$ y $q_*(s,a) = \max_\pi q_{\bar{\pi}}(s,a)$.

Lo que aún no está definido es la función de valor que sería razonable estimar en este contexto: $q_\pi(o,u)$, el retorno esperado de tomar la acción con descripción u dada la observación parcial o y luego seguir la política π. Una posible definición es:

$$\begin{aligned} q_\pi(o,u) &:= \mathbb{E}^\pi[R_1 + R_2 + \ldots | O_0 = o, U_0 = u] \\ &= \sum_{s \in \mathcal{S}} P^\pi(S_0 = s | O_0 = o) \mathbb{E}^\pi[R_1 + R_2 + \ldots | S_0 = s, U_0 = u] \\ &= \sum_{s \in \mathcal{S}} P^\pi(S_0 = s | O_0 = o) \mathbb{E}^\pi[R_1 + R_2 + \ldots | S_0 = s, A_0 = \rho(s,u)] \\ &= \sum_{s \in \mathcal{S}} P^\pi(S_0 = s | O_0 = o) q_{\bar{\pi}}(s, \rho(s,u)). \end{aligned}$$

Aquí, intuitivamente la esperanza promedia sobre todos los estados subyacentes en los que podríamos estar. Luego, en cada uno de ellos sabemos qué acción se tomaría según el criterio ρ (si ρ fuera estocástico habría que tomar la suma pesada por las probabilidades). El problema es que a priori $P^\pi(S_0 = s | O_0 = o)$ no está definida porque no dijimos cómo se distribuyen S_0 y O_0. Usando Bayes, tenemos

$$P_\pi(S_0 = s | O_0 = o) = \frac{P^\pi(O_0 = o | S_0 = s) P^\pi(S_0 = s)}{P^\pi(O_0 = o)},$$

donde $P^\pi(O_0 = o | S_0 = s)$ es 1 para o tal que $O(o) = s$ y 0 si no y $P^\pi(O_0 = 0)$ es simplemente lo necesario para normalizar el numerador ($\sum_{s' \in \mathcal{S}} P^\pi(O_0 = o | S_0 = s') P^\pi(S_0 = s')$). Con esto, $P^\pi(S_0 = s)$ es la distribución que nos queda por definir. Esto es muy natural, para evaluar la recompensa esperada en base a una observación parcial necesitamos saber la probabilidad de cada estado.

Singh et al. (1994) definen a $P^\pi(S_0 = s)$ como la distribución asintótica de la cadena de Markov inducida por π. Es decir, la probabilidad en el infinito de estar en el estado s si se sigue la política π. Hay un detalle aquí que es que como nuestra cadena termina en finitos pasos eso mismo no funciona, pero definimos entonces a $P^\pi(S_0 = s)$ como la distribución asintótica de la cadena de Markov que es igual a la inducida por π, pero vuelve a S_0 con probabilidad 1 cuando llega a un estado terminal.

Con esto definido, tenemos también definidas $q_*(o,u) = \max_\pi q_\pi(o,u)$ y $v_*(o) =$

$\max_\pi v_\pi(o)$, la funciones de valor bajo juego óptimo. Aquí sin embargo aparecen varias diferencias algo antintuitivas con el caso de MDPs.

1. Aquí π afecta q_π no solo por el efecto que tiene sobre los pasos siguientes (como en MDPs) sino también porque afecta la probabilidad de estar en un estado subyacente u otro. La propiedad de Markov hacía que no importara qué política se había seguido hasta llegar a un estado, ahora sí nos importa.

2. Para MDPs vimos en la sección 3.1 que existían políticas globalmente óptimas. Para VPOMDPs (o POMDPs) esto no es cierto. Es decir, no existe una política que realice el máximo de $v_*(o)$ en todas las observaciones o a la vez. Intuitivamente, el problema es que fijar una política en base a una observación parcial no es una buena idea porque no conocés fielmente el estado en el que estás, y al avanzar hacia nuevos estados podrías ganar información que te haga elegir una política u otra. El ejemplo de la figura 3 de Singh et al. (1994) para POMDPs es suficiente para VPOMDPs también, pero mostramos en la figura 5.2 un ejemplo que además ilustra el tipo de problemas que pueden aparecer al abstraer también las acciones. Esto es bastante problemático, porque en consecuencia q_* no indica el máximo valor que se puede obtener a partir la observación actual sino solamente el máximo que se puede obtener si se usa una política fija durante todo el episodio.

3. En la línea del item anterior, tanto para POMDPs como para VPOMDPs la mejor política con memoria puede ser arbitrariamente mejor que la mejor política sin memoria, donde una política con memoria es una que puede depender del historial de observaciones y acciones abstractas tomadas hasta el momento (y una sin memoria es lo que definimos nosotros como política).

4. Tanto para POMDPs como VPOMDPs se puede ver que la mejor política estocástica puede ser arbitrariamente mejor que la mejor política determinística (a diferencia de MDPs donde la política greedy con respecto a q_* es óptima y determinística). La intuición de un ejemplo de esto es que si estás buscando algo con los ojos tapados y no tenés memoria, es mejor moverte al azar que moverte siempre en la misma dirección.

5. Para VPOMDPs, la mejor política que usa la información de qué conjunto de acciones abstractas está disponible puede ser arbitrariamente mejor que una que no lo usa (como las que definimos nosotros). Para obtener un ejemplo de esto basta agregar una transición desde s_3 hacia sí mismo en la figura 5.2, que permitiría

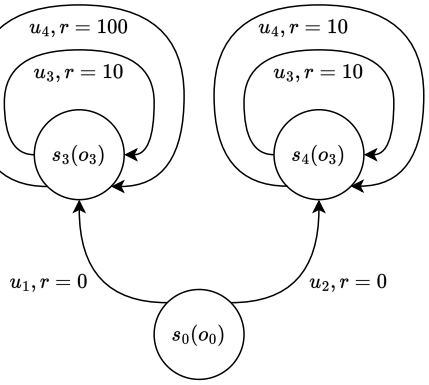

Figura 5.2: Ejemplo de que no hay una política globalmente óptima. En la figura, la política que maximiza $v_\pi(o_1)$ es la que en o_3 elige u_4 y la que maximiza $v_\pi(o_2)$ es la que en o_3 elige u_3.

distinguir a s_3 de s_4 si la política usara esta información. Nuestra definición de política deliberadamente esconde esta información, porque nos interesan políticas que solo se basan en evaluar transiciones individuales como las que genera nuestro algoritmo de aprendizaje.

Ejemplos de los items 2 y 3 fueron dados por Singh et al. (1994). Como corolario de los items 2, 3 y 4, sabemos es posible que nuestra técnica sea incapaz de encontrar políticas de exploración óptimas (las que seleccionan la menor cantidad de transiciones de la planta posible). Sin embargo, no queda claro en la práctica (o en nuestros casos de estudio) qué tan frecuentes serán estos casos patológicos. Más en general, la diferencia entre la mejor política que podamos encontrar y la política óptima dependerá de la capacidad de nuestro conjunto de features de distinguir las transiciones óptimas del resto.

En nuestro algoritmo de aprendizaje intentamos encontrar una función $q(o, u)$ buena en algún sentido y luego seguimos la política greedy con respecto a ella. La regla en versión tabular que usamos para aproximar este valor es:

$$q_{t+1}(O_t, U_t) = q_t(O_t, U_t) + \alpha(R_{t+1} + \max_{u \in \bar{U}_{t+1}} q_t(O_{t+1}, u) - q_t(O_t, U_t))$$

donde O_t, U_t, $\bar{U}_t$, R_t son variables aleatorias que representan la observación, la descripción de la acción tomada, el conjunto de descripciones de las acciones disponibles y la recompensa del tiempo t, respectivamente.

En este punto tiene sentido preguntarnos a qué esperamos que converja esta regla y bajo qué hipótesis. Esto escapa al alcance de esta tesis y, a nuestro entender, es una pregunta abierta, pero a continuación damos nuestra intuición. La primera observación es

que estamos usando un O_{t+1} que no es parte de la aridad de la función sino una muestra que sampleamos de la variable aleatoria asociada, cuya distribución va a depender de la política π que usemos durante los episodios de entrenamiento. En nuestro caso, π es una política ε-greedy con respecto a q, la notamos π_q. Luego, el límite debería ser solución de la ecuación:

$$\begin{aligned} q(o,u) &= E_{\pi_q}[R_{t+1} + \max_{u' \in \bar{U}_{t+1}} q(O_{t+1}, u') | O_t = o, U_t = u] \\ &= \sum_{s,s' \in \mathcal{S}} P^{\pi_q}(S_t = s | O_t = o) P(s', \rho(s,u), s)(r(s, \rho(s,u), s') + \max_{u' \in \bar{U}(s)} q(O(s'), u')) \end{aligned}$$

donde $\bar{U}(s)$ es el conjunto de descripciones de acciones del estado s y P y r son la dinámica y la señal de recompensa del MDP subyacente. Singh et al. (1994) prueban la convergencia de Q-Learning (tabular) sobre POMDPs *fijando* una política de exploración π, no permitiendo que dependa de q. El problema de esto es que al hacer eso el valor de $q(o,u)$ calcula la probabilidad de cada estado subyacente en base a una política que no es greedy con respecto a sí misma, con lo cual al usar la política greedy con respecto a esa q tenemos un sesgo errado hacia esa distribución de estados.

Para terminar esta sección, destacamos que una importante ventaja y motivación para estudiar los VPOMDPs es que permiten analizar el comportamiento de un agente en múltiples ambientes con MDPs subyacentes completamente distintos gracias a que la interfaz de interacción del agente se mantiene. En particular, si la abstracción es buena y la política es general, podríamos obtener agentes con buen retorno esperado en ambientes completamente nuevos aprovechando lo aprendido en otros ambientes. Es interesante como trabajo futuro precisar formalmente propiedades que permitan esta forma de generalización, relacionándolas con el algoritmo de aprendizaje.

6. EVALUACIÓN EXPERIMENTAL

En este capútlo presentamos resultados empíricos de nuestra técnica. Utilizamos una implementación de OTF-DCS extendida con el cálculo de features dentro de MTSA (D'Ippolito et al., 2008). El algoritmo de aprendizaje fue implementado en Python y está disponible aquí[1]. La computadora utilizada para los experimentos utiliza un Intel i7-7700 y 16GB de RAM, y no tiene GPU. En todos los casos, nuestros resultados son comparados con una política de exploración que siempre elige una transición al azar de la frontera (RANDOM) y con la Ready Abstraction (RA).

Las evaluaciones son realizadas sobre el benchmark introducido por Ciolek et al. (2019), que contiene 6 dominios de problemas de control: Air Traffic (AT), Bidding Workflow (BW), Travel Agency (TA), Transfer Line (TL), Dinning Philosophers (DP) and Cat and Mouse (CM). Todos los problemas escalan en dos dimensiones. El número de componentes intervinientes crece con el parámetro n y el número de estados por componente crece con el parámetro k. Es decir, las instancias son de la forma E_p con $p = (n, k) \in \mathbb{N} \times \mathbb{N}$.

En todos los casos las palabras performance, transiciones expandidas y recompensa acumulada son usadas equivalentemente, con el detalle de que las máximas performances y recompensas acumuladas se obtienen al minimizar las transiciones expandidas.

6.1. Aprendizaje en instancias chicas

El primer paso de nuestra técnica consiste en entrenar (para cada dominio) un agente en una instancia relativamente chica, usando el algoritmo descripto en la sección 4.3. Para todos los dominios entrenamos en la instancia $(2, 2)$ por un mínimo de 500000 pasos, y luego cortando el entrenamiento si en el último tercio de los pasos no se encontró una nueva máxima performance. Las instancias $(2, 2)$ van desde 91 transiciones totales en AT hasta 5044 en CM, por lo que el número de episodios puede variar significativamente.

La arquitectura de la red neuronal utilizada es un perceptrón con una única capa oculta de 20 neuronas y función de activación ReLU. Nuestros experimentos informales no mostraron mejoras usando redes más profundas o anchas, pero creemos que podrían ser útiles si el conjunto de features fuera más grande. El optimizador usado fue SGD (Stochastic Gradient Descent) con 1e-5 como tasa de aprendizaje y 1e-4 como weight decay. La tasa de exploración (ε) fue decrementada linealmente desde $1{,}0$ hasta $0{,}01$ en

[1] https://github.com/tdelgado00/Learning-Synthesis

los primeros 250000 pasos del entrenamiento. El tamaño del buffer de Experience Replay es 10000, con minibatches de tamaño 10 y la red target fue reseteada cada 10000 pasos.

En esta sección buscamos responder dos preguntas:

(Q1) Aprenden nuestros agentes a reducir la porción expandida de la planta?

(Q2) Son las políticas aprendidas competitivas en las instancias de entrenamiento?

Observabilidad parcial, recompensas esparsas, y la *deadly triad* son individualmente motivios suficientes para que el aprendizaje falle completamente, con lo cual no es obvio que la recompensa obtenida vaya a aumentar con el tiempo de entrenamiento. Las curvas de la figura 6.1 muestran la evolución de la recompensa durante los episodios de entrenamiento y la obtenida por RANDOM en cada problema, respondiendo la pregunta **(Q1)**.

Lo primero que vale la pena mencionar es que en ningún caso observamos divergencia del modelo (pesos tendiendo a infinito), cosa que sí había ocurrido, aunque muy raramente, antes de empezar a utilizar Experience Replay y Fixed-Q Target. El aprendizaje además alcanza en todos los casos performances promedio no aleatorias. Las curvas de aprendizaje muestran mejoras consistentes para AT, BW y TL. En CM, solo vemos una pequeña mejora con respecto a RANDOM. En DP y TA los agentes parecen alcanzar un pico de performance que luego pierden y no pueden recuperar. Si bien esta pérdida podría ser meramente explicada por la inestabilidad de la regla de aprendizaje, observamos empíricamente que remover el uso de momentum del optimizador elimina completamente la bajada. Sin embargo, elegimos mantener el momentum dado que eliminarlo reduce ligeramente la performance los mejores agentes encontrados.

Incluso si el aprendizaje converge, no es claro inicialmente si las features elegidas son suficientemente informativas para codificar buenas políticas, ni si nuestros agentes serán capaces de encontrarlas. Las líneas horizontales rojas en la figura 6.1 muestran la performance de RA, respondiendo **(Q2)**. Nuestros agentes rápidamente superan a RA en AT, BW y TA y la performance promedio en DP y TL la aproxima cercanamente. CM muestra ser desafiante para nuestros agentes, que se mantienen lejos de los resultados de RA en la instancia de entrenamiento.

Las dificultades en CM podrían deberse a que es el dominio con episodios más largos y por lo tanto recompensas más esparsas. Sin embargo, creemos que la causa principal es que nuestras features no son suficientemente informativas para codificar una buena política de exploración. En particular, en CM (Cat and Mouse) hay un tablero con gatos y ratones como piezas y se usan índices para hacer referencia tanto a las casillas del tablero como a los gatos y ratones individuales. Al eliminar estos índices (ver sección 4.4) nuestros agentes no tienen forma de, por ejemplo, decidir qué ratón mover ni hacia dónde.

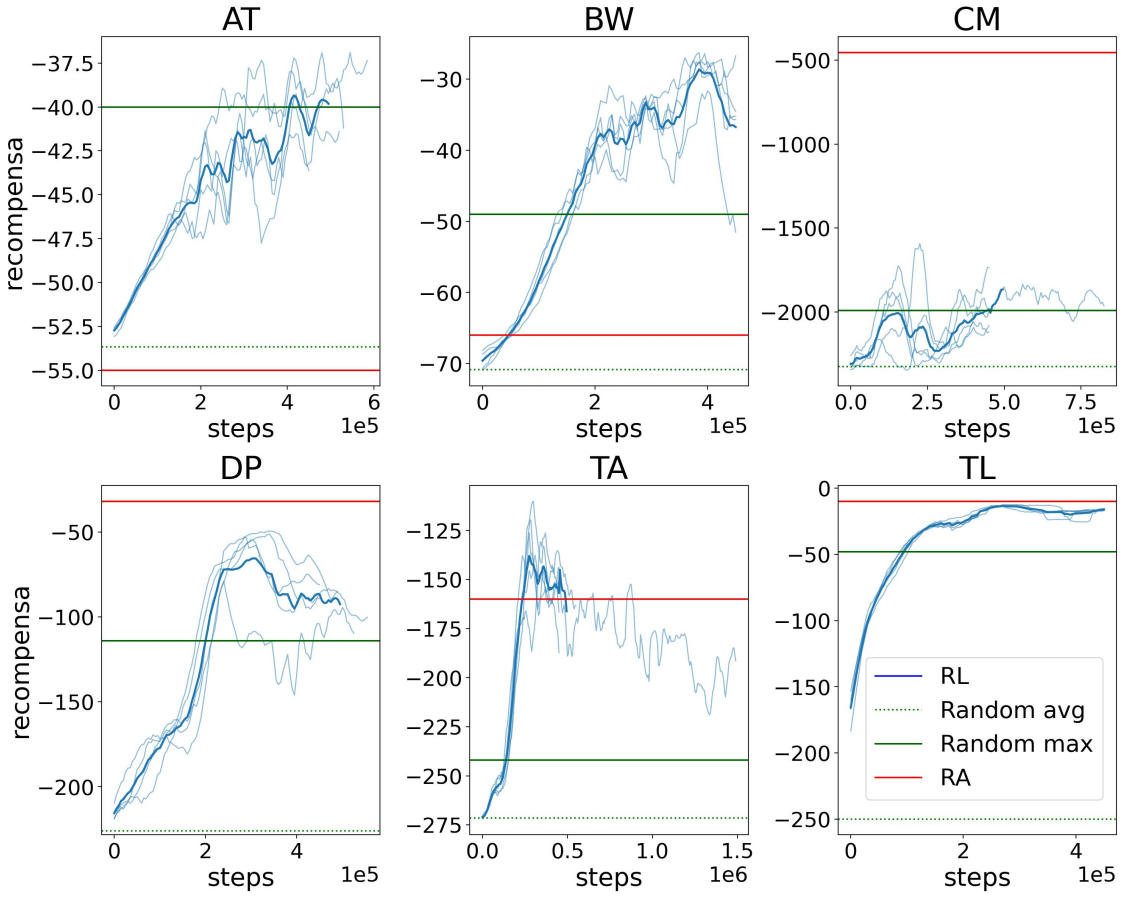

Figura 6.1: Evolución de la recompensa (cantidad de transiciones expandidas con un signo menos adelante) durante los episodios de entrenamiento, con una política ε-greedy. Se muestran los resultados para 5 semillas aleatorias, con el promedio resaltado en azul. Las curvas están suavizadas usando buckets de 5000 pasos y una ventana deslizante de tamaño 10 para mejorar la legibilidad. La performance de RA y RANDOM se muestra en rojo y verde, respectivamente. Para RANDOM se muestra la media y máximo de 100 ejecuciones.

Por otro lado, en el resto de los dominios queda claro que, al menos para el caso $(2, 2)$, nuestras features son suficientes para codificar políticas interesantes. De hecho, conjeturamos que en AT y BW las transiciones expandidas por nuestros mejores modelos son la mínima cantidad posible, y en TA, DP y TL están muy cerca de serlo, con lo cual hay poco espacio para mejoras en $(2, 2)$ en la mayoría de los dominios.

6.2. Generalización a instancias grandes

En esta sección buscamos responder la siguiente pregunta:

(Q3) Son las políticas inducidas competitivas en instancias más grandes?

Durante el entrenamiento se guardan los pesos de la red neuronal cada 5000 pasos para luego elegir 100 al azar de esas políticas y testearlas. El testeo (paso **(P2)**) se realiza con todos los valores de n y k hasta $(15, 15)$ con un budget de 5000 transiciones, y solo testeando las instancias cuando tanto $(n-1, k)$ como $(k-1, n)$ fueron resueltas dentro del budget. Es decir, si para una instancia se están expandiendo más de 5000 transiciones se detiene el algoritmo de síntesis y se considera no resuelta, y no se evalúan instancias más grandes ($n' \geq n$ y $k' \geq k$). Luego, seleccionamos la red neuronal que maximiza la cantidad de instancias resueltas desempatando según la mínima suma de transiciones expandidas. Los resultados de esta sección corresponden a estas políticas ganadoras, evaluadas nuevamente con el mismo formato pero con un budget de 15000 transiciones.

Como fue discutido en la sección 4.4, buenos resultados en las instancias de entrenamiento no necesariamente se trasladan a buenos resultados en las versiones más grandes. Para evaluar las capacidades de generalización de nuestros agentes, la figura 6.2 muestra para cada dominio las transiciones expandidas por la política seleccionada en el paso **(P2)** y por RANDOM y RA, a medida que crece el tamaño de la planta (los valores particulares de n y k son ignorados).

Nuestros agentes expanden significativamente menos transiciones que RANDOM en todos los dominios, reduciendo considerablemente la velocidad de crecimiento de la porción explorada de la planta (notar que la escala es logarítmica). Más aún, en AT, BW y TA, los problemas en los que nuestros agentes superaron a RA durante el entrenamiento, los victoria se preservó significativamente. Análogamente, en CM, DP y TL, nuestros agentes no superaron a RA durante la generalización tampoco, pero en todos los casos la performance se mantuvo en las instancias más grandes, y en DP y TL la distancia con RA no es demasiado grande.

Una observación no menor de estos resultados es la continuidad de las transiciones expandidas en función de los tamaños de instancias para la mayoría de los problemas. Esto es sorprendente dado que cada instancia es un autómata con distintos estados y etiquetas, y parece mostrar que tanto el conjunto de features como las políticas aprendidas son capaces de capturar ciertas similitudes entre las instancias.

AT se distingue del resto de los dominios en este aspecto, con un gráfico que parece ser mucho más ruidoso. Sin embargo, esto se debe a que todas las instancias con $n > k$ en AT son no realizables (en el resto de los dominios para $n > 1$ y $k > 1$ son siempre realizables) y son más fáciles de resolver con pocas expansiones. La figura 6.3 se pueden ver los resultados de AT separando en instancias realizables y no realizables, que muestran que el comportamiento de AT no es distinto al de otros dominios en las instancias realizables. Más allá de esto, ignorando la distinción de realizabilidad, los agentes de RL en AT

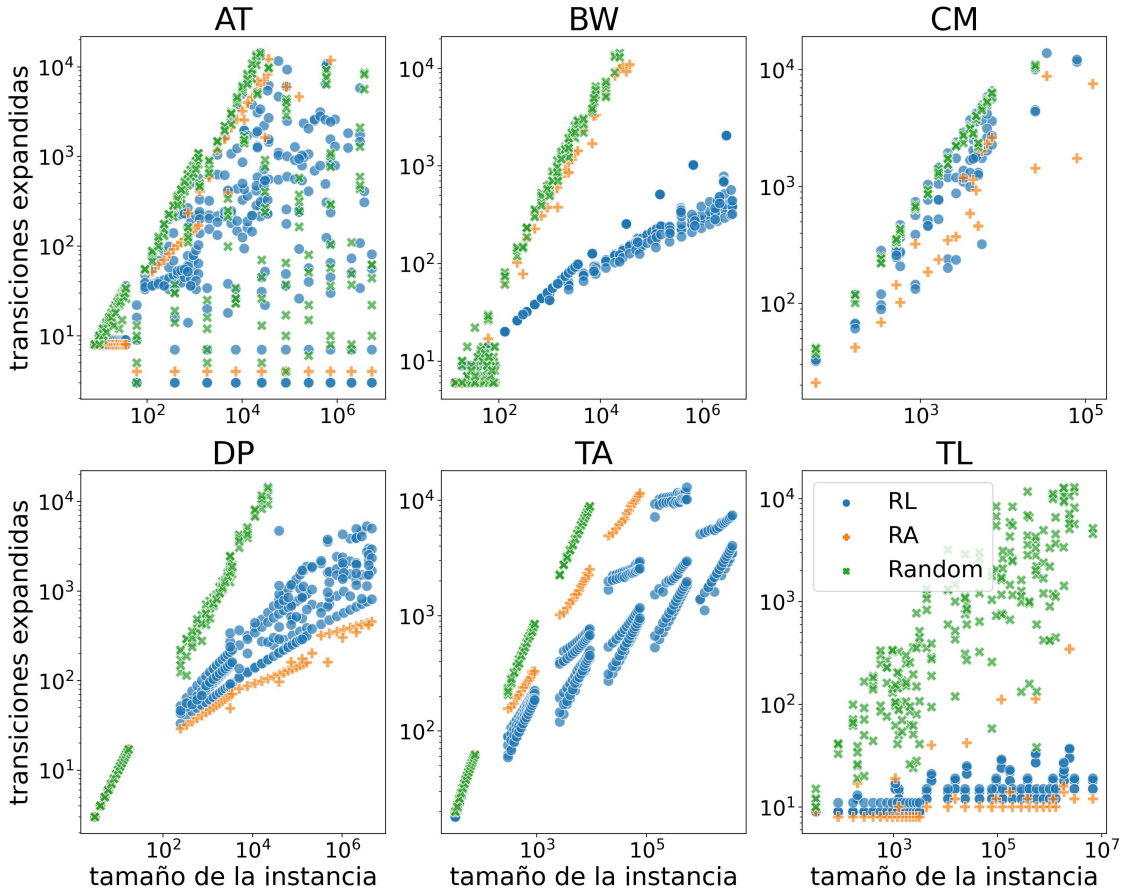

Figura 6.2: Transiciones expandidas de RA y de cada semilla aleatoria de RL y RANDOM, para todas las instancias para las cuales el tamaño de la planta (x-axis) pudo ser computado y la cantidad de transiciones expandidas es menor a 15000. El segundo gráfico de AT es igual al primero, pero con las instancias no realizables removidas.

son significativamente mejores que RA y RANDOM en $51\,\%$ y $71\,\%$ de las instancias, respectivamente, mientras que RA y RANDOM son ambos solo mejores en aproximadamente $1\,\%$ de las instancias (usando una desviación estandar como medida de significatividad).

6.3. Cantidad de instancias resueltas en un tiempo dado

En esta sección buscamos responder las siguientes preguntas:

(Q4) Es nuestra técnica competitiva para un presupuesto de tiempo de ejecución fijo?

(Q5) Ablación: Cuál es el impacto del paso de selección **(P2)**?

Como fue mencionado en la sección 4.3, nuestra técnica conlleva el overhead de computar las features y evaluar el modelo para cada transición en cada paso. Esto podría

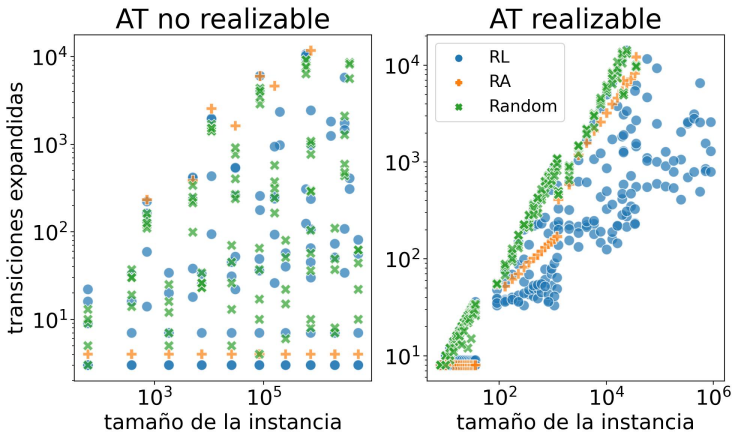

Figura 6.3: Resultados de AT de la figura 6.2 separando en instancias realizables (i.e. existe un controlador) y no realizables.

ser problemático en el objetivo final de empujar la frontera de instancias resueltas dentro de un budget de tiempo de ejecución. Sin embargo, RA también tiene un overhead significativo, que viene de construir un grafo de dependencias y calcular las distancias a estados marcados allí.

La tabla 6.1 muestra la cantidad de instancias resueltas en promedio por los modelos seleccionados y las baselines con un presupuesto de 10 minutos, respondiendo **(Q4)**. Nuestra técnica resuelve significativamente más instancias que RA en tres de los seis dominios (AT, BW, y TA), no muestra diferencia significativas en dos (CM y TL) y resuelve significativamente menos instancias en DP. Finalmente, la cantidad de instancias resueltas es significativamente mayor a la de RA.

Notamos además que si bien en TL ambas políticas resuelven el mismo número de instancias, y es cierto que en la mayoría de los casos RA lo hace con menos transiciones, al evaluar instancias más grandes RA falla primero porque obtiene malos resultados en los casos $k = 1$ (ver las cruces amarillas que crecen más rápido en la figura 6.2).

Finalmente, considerando que el paso de selección **(P2)** implica un costo considerable en nuestro pipeline completo (dado que requiere evaluar muchas políticas), vale la pena preguntarse qué tan necesario es. Para responder **(Q5)**, la segunda columna de la tabla 6.1 (RLNS) muestra la cantidad de instancias resueltas al reemplazar **(P2)** por un método de selección más barato y quizás más obvio, que no focaliza en la generalización: seleccionar el agente con la mejor recompensa en la instancia de entrenamiento (tomando el último agente en caso de empate). Los resultados muestran que para AT, BW y DP **(P2)** permitió resolver significativamente más instancias (en promedio 20,6, 68,8 y 23,8 instancias,

	RL	RLNS	RANDOM	RA
AT	**99.2 ± 8.16**	79.0 ± 3.79	82.6 ± 0.8	88
BW	**159.4 ± 4.59**	92.2 ± 25.91	47.2 ± 0.4	47
CM	22.8 ± 0.98	23.6 ± 0.8	22.0	24
DP	89.4 ± 11.41	62.4 ± 6.62	46.6 ± 0.49	**137**
TA	**99.2 ± 21.71**	100.6 ± 22.84	51.8 ± 0.4	65
TL	225.0	225.0	60.6 ± 4.45	225
All	**695.0 ± 23.32**	582.8 ± 38.01	310.8 ± 5.04	586

Tabla 6.1: Cantidad de instancias resueltas por las distintas heurísticas. Se muestra la desviación estandar solo cuando es distinta de cero.

respectivamente) mientras que para CM, TA y TL la diferencia no fue significativa. En el total, este paso fue necesario para resolver más instancias que RA, aunque no para resolver más instancias que RANDOM. En particular, estos resultados se condicen empíricamente con las ideas mencionadas en la sección 4.4 para algunos dominios, pero muestran que en otros no hay necesidad de preocuparse por el overfitting.

6.4. Otras cosas que miramos y probamos

En esta sección listamos otras métricas y gráficos que analizamos durante la experimentación y algunas ideas que probamos y no terminaron de funcionar.

Cantidad de instancias resueltas en función del tiempo. La cantidad de instancias resueltas de cada uno de los modelos evaluados en **(P2)** función del tiempo de entrenamiento. Este gráfico es escencial porque es lo que utilizamos para seleccionar el modelo. Sin embargo, es muy ruidoso y, salvo quizás en BW, en ningún dominio se ve una mejora consistente a lo largo del entrenamiento. De todos modos, en ese ruido justamente se explica la utilidad de nuestro método de selección.

Análisis del valor esperado por los modelos. Cualquier modelo de los obtenidos es una función de $\mathbb{R}^{d_E}$ en $\mathbb{R}$ (o $\{0,1\}^{d_E}$) que puede ser analizada. Para esto, tomamos un conjunto de vectores de features sampleados usando una política aleatoria y evaluamos su salida para cada uno de ellos. Un primer gráfico importante con esto es el valor promedio de esta salida (Q) en función del tiempo de entrenamiento, propuesto por Mnih et al. (2013). Aquí lo esperable sería encontrar que con el avance del entrenamiento el Q promedio de los modelos sea cada vez mejor, porque expanden en promedio cada vez menos transiciones. Sin embargo lo que vemos es que tiende a "empeorar" durante la mayor parte del entrenamiento porque la inicialización de los pesos de la red hace que

la cantidad de transiciones expandidas empiece en 0 (muy optimista). Solo en TL, el dominio que se termina resolviendo con menor cantidad de transiciones se ve un claro decrecimiento en la cantidad de transiciones esperadas luego de llegar al mayor pico.

Análisis del impacto de las features. Un segundo análisis que se puede hacer a partir de los valores de un modelo Q para un conjunto de pares estado acción dado, es intentar evaluar estadísticamente el impacto de cada una de las features en esa función. Para esto, la mejor opción que encontramos fue usar SHAP (Lundberg y Lee, 2017), un framework de interpretabilidad que resuelve exactamente este problema. Alternativas como analizar los pesos de un modelo lineal o usar un análisis de varianza no terminan de capturar lo que nos interesa, que es encontrar la utilidad de una feature para determinar el valor final, considerando también cómo impacta en el valor que se le da a otras features. Del análisis de SHAP sobre los mejores modelos que encontramos para cada dominio lo que observamos es que, en primer lugar, ninguna feature es completamente irrelevante en todos los dominios, siendo la menos relevante la que indica si ya se exploró algún evento de los estados de entrada y salida de la transición. En segundo lugar, observamos que son considerablemente distintas las features que cobran mayor relevancia en cada uno de los dominios.

Entrenar en instancias más grandes. Entrenar en $(3, 3)$ ni siquiera es posible para CM y TL porque es tan grande la planta que 16GB de RAM no son suficientes para almacenar toda la frontera. Para el resto de los problemas es posible y en particular para AT, el problema con las plantas más chicas, da mejores resultados que para $(2, 2)$ (la única ejecución competa que hicimos dio mejor en todas las métricas que las 5 semillas de $(2, 2)$). Para BW, DP y TA resulta mejor entrenar en $(2, 2)$. De hecho, para estos problemas se obtienen mejores resultados en $(3, 3)$ entrenando en $(2, 2)$ que entrenando en $(3, 3)$.

Entrenar en múltiples instancias. Agregar diversidad en el entrenamiento mostró en algunos casos ser una buena idea para reducir el overfitting (Zhang et al., 2018). Probamos tomar por ejemplo todas las instancias con menos de 10000 transiciones totales y entrenar con un esquema round-robin o incremental, cambiando de instancia cada cierta cantidad de pasos. Sin embargo los resultados en ningún caso superaron a entrenar solo en $(2, 2)$. Creemos que el problema es que nuestra función Q estima el número de transiciones a ser expandidas y la escala de este número varía significativamente entre instancias. Es decir, si bien las diferencias de escala en cantidad de transiciones no hacen que Q sea mala para tomar decisiones en distintas instancias (porque se toma el máximo), sí hacen que la Q sea numéricamente una muy mala estimación en otras instancias. Minimizar el error de estimación sobre múltiples instancias simultáneamente daría un promedio que podría ser una mala política para ambas. Para resolver esto se podría intentar reescalar la recompensa

o cambiar la arquitectura de la red agregando una capa que normalice, entre otras ideas, pero todas quedan como trabajo futuro. Otra solución podría ser agregar los valores de n y k como features. En tal caso nuestros agentes tendrían que encontrar un patrón en cómo usar la información del tamaño de la instancia que generalice a tamaños más grandes. En la prueba que hicimos esto no fue mejor que entrenar en múltiples instancias sin los valores de n y k. El problema podría ser la capacidad limitada (en diversos sentidos) de los modelos que usamos para hacer esto, y la poca diversidad en instancias de tamaño chico que tiene el benchmark utilizado.

Otras features. Varias features quedaron descartadas durante el proceso de experimentación. Una de ellas fue la feature de RA, que escencialmente intentaba que fuera posible para nuestros agentes codificar completamente a RA a partir de las features. Si bien esta feature (al menos en el momento en el la probamos) tenía un impacto positivo, no es posible codificar completamente a RA en forma de vector de tamaño fijo, ni los agentes fueron capaces de encontrar exactamente la mejor forma de usarlas. Las descartamos principalmente para que nuestra técnica fuera estrictamente comparable con RA. Otras features que fueron descartadas son la distancia al estado inicial y la relación entre el la cantidad de transiciones en la frontera y en la planta exporada, por ser números reales. Finalmente, descartamos la posibilidad de usar historiales de más de un evento en la feature *Etiquetas de estado* porque agregaban un costo importante y no proporcionaban mejoras significativas.

7. DISCUSIÓN

7.1. Trabajo relacionado

Guiar la exlporación de la planta en el algoritmo OTF-DCS es un problema de Búsqueda Heurística en el que el objetivo es encontrar un subgrafo de la planta que contenga una estrategia de control ganadora (o que pruebe que no existe ninguna). Un enfoque similar fue propuesto recientemente para Planning clásico (Gehring et al., 2022), donde se entrena un agente de RL para guiar la búsqueda de planes (secuencias de acciones para alcanzar un estado objetivo) y se estudia la generalización a instancias más grandes. Una diferencia clave es que ellos resuelven un problema determinístico mientras que nostoros resolvemos uno en el que el controlador no tiene completo control del ambiente, algunos eventos son no controlables. Un marco más comparable a Control de Eventos Discretos sería Fully Observable Non-Deterministic planning. Además, Gehring et al. (2022) abordan propiedades de *reachability* que generan ejecuciones finitas, mientras que non-blocking trata con ejecuciones infinitas. En términos de la tarea aprendida, su recompensa estimada es la distancia a un estado objetivo en un punto dado de la ejecución de un plan, mientras que nosotros estimamos el número de transiciones adicionales que tienen que ser agregadas a la planta para que OTF-DCS termine. Otra diferencia es que ellos aprenden residuos con respecto a heurísticas existentes, usando *reward shaping*, para acelerar el proceso de aprendizaje en lo que de otro modo sería un ambiente con recompensa esparsa; nosotros entrenamos desde cero en un ambiente con recompensa esparsa. Finalmente, ellos usan Neural Logic Machines para representar la función de valor, tomando fórmulas lógicas que describen el estado del problema como entrada, mientras que nosotros usamos un perceptrón multicapa que toma un conjunto de features general como entrada.

Nuestra hipótesis de homogeneidad es similar a la asunción subyacente de patrones comunes en soluciones de Planning Generalizado, donde diferentes formas de aprendizaje fueron aplicadas Groshev et al. (2018); Karia y Srivastava (2021); Ståhlberg et al. (2022); Toyer et al. (2020). Los planes o políticas están representados de tal forma que pueden ser aplicados para resolver problemas de planning clásico en cualquier instancia de un dominio dado Martín y Geffner (2004); Srivastava et al. (2011). En nuestro escenario, ni las políticas de exploración son representaciones algorítmicas(e.g. Srivastava et al., 2011) ni se definen features específicas para cada dominio o dominios abstractos (e.g. Ståhlberg et al., 2022; Toyer et al., 2020). En contraste, nuestro enfoque utiliza features independientes

del dominio y esto es suficiente para ser aplicado directamente en cualquier instancia de cualquier dominio, incluso cuando el número de componentes que intractúa cambia.

Un esfuerzo reciente en el contexto de Búsqueda Heurística es el de Sudry y Karpas (2022). Más allá de las diferencias entre control y planning clásico, comparten con nuestro trabajo la idea clave de estimar el progreso de la búsqueda. Sin embargo, utilizan aprendizaje supervisado con LSTMs (Hochreiter y Schmidhuber, 1997) para estimar el progreso de búsqueda relativo de una heurística fija, mientras que usar RL le permite a nuestra técnica actualizar la estimación del progreso de búsqueda (cantidad de transiciones que se espera expandir) y la política de exploración simultáneamente.

Si bien fue considerada en múltiples ocasiones, generalización en RL es un tópico emergente y poco estudiado. Kirk et al. (2022) desarrollan categorizaciones para tareas y métodos para abordarla. Siguiendo sus definiciones, nuestro problema es un MDP contextual, donde cada MDP es determinado por un conjunto de parámetros que afecta el tamaño de la instancia. Luego, realizamos *zero-shot policy transfer* porque trasladamos las políticas aprendidas sin reentrenar en las instancias objetivo. Además, hacemos generalización *out-of-distribution* porque los ambientes de evaluación no son sampleados de la misma distribución que los del entrenamiento. La relación entre los MDPs en nuestro caso es sutil dado que no comparten ni estados ni acciones, y en nuestra técnica, esto es resuelto a través de la abstracción. Generealizar a un conjunto distinto de acciones no es considerado en su revisión y es, a nuestro entender, poco común.

Finalmente, hasta donde sabemos Control Supervisado y RL solo fueron estudiados en conjunto por Ushio y Yamasaki (2003) usando RL tabular; pero el foco fue puesto en la extensión a ambientes de control parcialmente observables y en maximizar la permisividad, en lugar de escalar a plantas más grandes. Su enfoque requiere resolver el problema objetivo repetidamente, mientras que el nuestro no lo usa durante el entrenamiento.

7.2. Conclusiones y trabajo futuro

En esta tesis presentamos una forma de combinar Aprendizaje por Refuerzo y Control de Eventos Discretos, usando RL como una heurística para acelerar un algoritmo de síntesis correcto y completo, guiando la exploración del espacio de estados. Propusimos una forma de enmarcar al algoritmo de síntesis on-the-fly como una tarea de RL y abstrajimos tanto estados como acciones con un conjunto de features para hacer factible el entrenamiento y la generalización. Fue necesario además modificar DQN para posibilitar el aprendizaje con un conjunto de acciones que varía en todos los estados. Adicionalmente, presentamos los primeros pasos hacia un marco teórico para nuestra técnica, analizando el Very Partially

Observable MDP, un formalismo que permite abordar la observabilidad parcial de estados y acciones.

Ante la ausencia total de garantías teóricas, nuestros resultados muestran empíricamente que aprender en instancias chicas es posible; que las funciones de valor aprendidas pueden inducir, en instancias grandes, políticas que reducen la porción explorada de la planta con respecto a heurísticas existentes; y que, en promedio, las políticas aprendidas permiten resolver más problemas de control en un tiempo dado.

Además, destacamos un conjunto de componentes de nuestra técnica que pueden ser útiles para mejorar la generalización. A saber, seleccionar políticas de exploración según sus resultados en instancias ligeramente más grandes, frenar el entrenamiento tempranamente para evaluar un conjunto diverso de políticas, y definir un conjunto de features que apunta más a la generalidad que a la completitud.

Como primera iteración de la idea, hay mucho espacio para mejoras y trabajo futuro. Un aspecto que podría ser mejorado es nuestro abordaje de la observabilidad parcial. Nuestro vector de features pierde información con respecto a la historia y al estado general de la exploración, y usar una red neuronal recurrente podría permitir a nuestros agentes recordar los aspectos importantes de la planta explorada y las acciones que fueron tomadas, a su vez reduciendo la necesidad de usar features diseñadas a mano. Esta idea fue tanto propuesta para aplicar DQN en POMDPs (Hausknecht y Stone, 2015) como para estimar el progreso de la búsqueda para una heurística de planning fija (Sudry y Karpas, 2022). Sin embargo, no es inmediata la aplicación en nuestro caso, donde no solo estados sino también las acciones son parcialmente observables, y todas las acciones son evaluadas individualmente en cada paso.

Otra idea que podría valer la pena explorar es la de explotar las heurísticas existentes, como proponen Gehring et al. (2022) para el caso de planning, para facilitar el aprendizaje en las tareas más desafiantes para nuestros agentes. Sin embargo, *reward shaping* implica utilizar las estimaciones de recompensa de las heurísticas existentes en la función de recompensa, y no es posible en nuestro caso porque las evaluaciones de RA no son valores en $\mathbb{R}$ sino listas de tuplas. Otras ideas en esta línea serían utilizar el controlador existente o las transiciones elegidas por RA durante el entrenamiento para guiar a nuestros agentes, ya sea con la función de recompensa o aprovechando que Q-Learning es off-policy para tomar directamente esas transiciones con cierta probabilidad.

Por otro lado, nuestra técnica busca aprender y escalar dentro de un dominio parametrizado dado, pero estudiar las ideas presentes en este trabajo en el contexto de generalización entre distintos dominios es de interés. Además de entrenar en un dominio del benchmark y trasladar a otros, se podrían expandir las instancias de entrenamiento con instancias

generadas al azar.

Un objetivo importante dentro del trabajo futuro es aplicar esta técnica a $GR(1)$ (Bloem et al., 2012), donde esencialmente se reemplaza la propiedad non-blocking por un subconjunto de LTL, y ya se encuentra desarrollada una versión del algoritmo de síntesis on-the-fly adaptada. Finalmente, creemos que las ideas reportadas en este trabajo podrían ser útiles para otros problemas de las comunidades de Automated Planning y Reactive Synthesis.

REFERENCIAS

Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., y Saar, Y. (2012). Synthesis of Reactive(1) Designs. *J. of Computer Systems Sci.*, *78*(3), 911–938.

Ciolek, D., Duran, M., Zanollo, F., Pazos, N., Braier, J., Braberman, V., D'Ippolito, N., y Uchitel, S. (2023). On-the-fly informed search of non-blocking directed controllers. *Automatica*, *147*, 110731.

Ciolek, D. A., Braberman, V., D'Ippolito, N., Sardiña, S., y Uchitel, S. (2019). Compositional supervisory control via reactive synthesis and automated planning. *IEEE Transactions on Automatic Control*.

D'Ippolito, N., Fischbein, D., Chechik, M., y Uchitel, S. (2008). MTSA: The Modal Transition System Analyser. En *Proc. of the int. conf. on automated software eng.* `https://bitbucket.org/lnahabedian/mtsa/src/master/`.

Gehring, C., Asai, M., Chitnis, R., Silver, T., Kaelbling, L., Sohrabi, S., y Katz, M. (2022). Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. En *Proc. of the intl. conference on automated planning and scheduling* (Vol. 32, pp. 588–596).

Groshev, E., Goldstein, M., Tamar, A., Srivastava, S., y Abbeel, P. (2018, Jun.). Learning generalized reactive policies using deep neural networks. *Proc. of the Intl. Conference on Automated Planning and Scheduling*, *28*(1), 408-416.

Hausknecht, M. J., y Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. En *Aaai fall symposia.*

Hochreiter, S., y Schmidhuber, J. (1997, 12). Long short-term memory. *Neural computation*, *9*, 1735-80.

Huang, J., y Kumar, R. (2008). Directed Control of Discrete Event Systems for Safety and Nonblocking. *IEEE Trans. Automation Science & Engineering*, *5*.

Karia, R., y Srivastava, S. (2021, May). Learning generalized relational heuristic networks for model-agnostic planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(9), 8064-8073.

Kirk, R., Zhang, A., Grefenstette, E., y Rocktäschel, T. (2022). *A survey of generalisation in deep reinforcement learning.*

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks*. Carnegie Mellon University.

Lundberg, S. M., y Lee, S.-I. (2017). A unified approach to interpreting model predictions.

En I. Guyon et al. (Eds.), *Advances in neural information processing systems 30* (pp. 4765–4774). Curran Associates, Inc.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., y Bowling, M. (2018, jan). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *J. Artif. Int. Res.*, *61*(1), 523–562.

Magee, J., y Kramer, J. (2014). *Concurrency: State models and java programs*. Wiley.

Martín, M., y Geffner, H. (2004). Learning generalized policies from planning examples using concept languages. *Appl. Intell.*, *20*(1), 9–19.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., y Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Nau, D., Ghallab, M., y Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers.

Pnueli, A., y Rosner, R. (1989). On the Synthesis of a Reactive Module. En *Proc. of the symp. on principles of programming languages* (pp. 179–190).

Ramadge, P. J., y Wonham, W. M. (1987). Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, *25*.

Ramadge, P. J., y Wonham, W. M. (1989). The control of discrete event systems. *Proc. of the IEEE*, *77*.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., y Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, *abs/1712.01815*.

Singh, S. P., Jaakkola, T., y Jordan, M. I. (1994). Learning without state-estimation in partially observable markovian decision processes. En *Machine learning proceedings 1994* (pp. 284–292). Elsevier.

Srivastava, S., Immerman, N., y Zilberstein, S. (2011). A new representation and associated algorithms for generalized planning. *Artif. Intell.*, *175*(2), 615–647.

Ståhlberg, S., Bonet, B., y Geffner, H. (2022). Learning generalized policies without supervision using gnns. En *Procs of the 19th intl. conference on principles of knowledge representation and reasoning, KR.*

Sudry, M., y Karpas, E. (2022, Jun.). Learning to estimate search progress using sequence of states. *Proceedings of the International Conference on Automated Planning and Scheduling*, *32*(1), 362-370.

Sutton, R. S., y Barto, A. G. (1998). Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, *9*(5), 1054-1054.

Sutton, R. S., y Barto, A. G. (2018). *Reinforcement learning: An introduction*. Cambridge, MA, USA: A Bradford Book.

Sutton, R. S., Bowling, M., y Pilarski, P. M. (2022). *The alberta plan for ai research.* arXiv.

Toyer, S., Thiébaux, S., Trevizan, F. W., y Xie, L. (2020). Asnets: Deep learning for generalised planning. *J. Artif. Intell. Res.*, *68*, 1–68.

Ushio, T., y Yamasaki, T. (2003). Supervisory control of partially observed discrete event systems based on a reinforcement learning. En *Ieee international conference on systems, man and cybernetics.* (Vol. 3, pp. 2956–2961).

Watkins, C. J. C. H., y Dayan, P. (1992, may). Technical note: Q -learning. *Mach. Learn.*, *8*(3–4), 279–292.

Wonham, W. M., y Ramadge, P. J. (1987). On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, *25*(3).

Wonham, W. M., y Ramadge, P. J. (1988). Modular Supervisory Control of Discrete-Event Systems. *Mathematics of Control, Signals and Systems*, *1*(1), 13–30.

Zhang, C., Vinyals, O., Munos, R., y Bengio, S. (2018). *A study on overfitting in deep reinforcement learning.* arXiv.