UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

DEPARTAMENTO DE COMPUTACIÓN

Agregando información especifica de dominio para escalar la síntesis de controladores de tipo GR(1)

Tesis de Licenciatura en Ciencias de la Computación

Darío Juan Nicolás Turco

Director: Sebastián Uchitel

Buenos Aires, 2024

Agregando información especifica de dominio para escalar la síntesis de controladores de tipo GR(1)

El área de síntesis de controladores busca construir automáticamente estrategias para resolver problemas bajo ciertas garantías. El algoritmo On-The-Fly Directed Controller Synthesis (OTF-DCS), propuesto por Ciolek en su trabajo de tesis doctoral, resuelve este problema, pero depende de una heurística auxiliar. Además de proponer el algoritmo OTF-DCS, Ciolek también propuso una heurística llamada Ready Abstraction(RA), la cual es la que mejores resultados logra. Luego, Tomas Delgado propuso una heurística basada en aprendizaje por refuerzos, la cual, requiere una función de abstracción que captura las features mas importantes de un estado.

En este trabajo de tesis, exploraremos el impacto de agregar features con información específica del dominio del problema a resolver a la función de abstracción que usa la heurística basada en aprendizaje por refuerzos. Para esto, ofrecemos una metodología para abstraer información referente a las entidades del problema de control. Esto tiene como objetivo mejorar el rendimiento de dicha heurística y por lo tanto, el rendimiento del algoritmo OTF-DCS en su versión para restricciones de tipo GR(1). Estas mejoras serán comparadas de manera empírica contra otras heurísticas comúnmente utilizadas tales como RA, entre otras.

Palabras claves: Síntesis de Controladores, Aprendizaje por Refuerzos, Redes Neuronales, Aprendizaje Automático, OTF-DCS, Reactividad General.

Adding domain specific information to scale the synthesis of GR(1) controllers

The area of controller synthesis aims to automatically construct strategies to solve problems under certain guarantees. The On-The-Fly Directed Controller Synthesis (OTF-DCS) algorithm, proposed by Ciolek in his doctoral thesis, addresses this problem but relies on an auxiliary heuristic. In addition to proposing the OTF-DCS algorithm, Ciolek also proposed a heuristic called Ready Abstraction(RA), which achieves good results. Later, Tomas Delgado proposed a heuristic based on reinforcement learning, which requires an abstraction function that captures the most important features of a state.

In this thesis, we will explore the impact of adding features with specific information of the problem to be solved to the abstraction function used by the reinforcement learning heuristic. To do this, we offer a methodology for abstracting information related to the entities in the control problem. This aims to improve the performance of the heuristic and consequently, the performance of the OTF-DCS algorithm in its version for GR(1) constraints. These improvements will be empirically compared against other commonly used heuristics such as RA, among others.

Keywords: Controller Synthesis, Reinforcement Learning, Neuronal Networks, Machine Learning, OTF-DCS, General Reactivity.

Agradecimientos

Quiero agradecer a toda mi familia, en especial a mamá y papá, que siempre me apoyaron y ayudaron. A todos mis amigos del CBC, con los que compartimos un montón de recuerdos inolvidables. También agradecerles a Franco, Luciano, Lu, Ivo, Sujo, Nico, Mateo y a todas las personas con las que transité este largo camino, sin su compañía, esto no habría sido tan divertido como lo fue. Gracias a Juli por siempre acompañarme y animarme cuando las cosas no salían tan bien. Te amo, mi vida.

Gracias a toda la gente de LaFHIS, que siempre me ayudó y me dio un hermoso espacio para hacer este trabajo, en especial a mi director Sebas, que siempre estuvo para guiarme. También a Floppy y Hernán, de quienes aprendí un montón y siempre tuvieron la mejor de las voluntades.

Agradecerles a todos los docentes que tuve a lo largo de estos años, los cuales me enseñaron y acompañaron a lo largo de toda la carrera, incluso pasando por una pandemia y un escenario incierto, y aun así no bajaron los brazos.

A todo el equipo de divulgación de esta facultad por darme la oportunidad de participar como divulgador. Y al equipo de divulgación de los talleres DOV, que me dieron a conocer la facultad allá por 2016, en especial a Christian y Floppy, que sin saberlo en ese momento, serian personas con las que compartiría mucho en un futuro.

Índice

1. Motivación

Hoy en día es cada vez más común querer automatizar sistemas encargados de resolver importantes tareas, como por ejemplo: automatización de fábricas, sincronización de procesos, coordinación del aterrizaje de aviones, robótica, entre muchas otras. En ese contexto, es de mucha utilidad tener una manera de poder automatizar estas tareas. Para esto existen muchas herramientas, la mayoría del área de *Machine Learning* o *Inteligencia Artificial*, las cuales son áreas que han ganado mucha popularidad en los últimos tiempos.

Si bien estas áreas otorgan una manera de resolver estas tareas de manera automática, por lo general no otorgan ninguna garantía de resolver el problema de manera correcta en todos los casos. Es por esto que para sistemas críticos, donde un error puede ser extremadamente costoso, no siempre se pueden usar ese tipo de métodos. En particular, el problema de síntesis de controladores se trata de obtener de manera automática un plan que cumpla ciertos requisitos.

Bajo este contexto, las áreas de *Control de Eventos Discretos*[1], *Planificación Automática*[2] y *Síntesis Reactiva*[3] estudian el problema de síntesis de controladores, es decir, cómo, dada una representación de un entorno, proporcionar automáticamente un plan para cumplir sistemáticamente ciertos objetivos en el mismo. Si bien las tres áreas se enfocan en resolver el mismo problema, cada una utiliza representaciones y algoritmos distintos. A pesar de esto, algo que todas comparten es que, a medida que la cantidad de componentes que representan el ambiente crecen, la complejidad de los algoritmos aumenta de manera exponencial, haciendo que para algunos problemas no haya soluciones prácticas.

Esto se debe a que la representación de entornos complejos, se compone de una lista de entidades, representadas por distintos autómatas, los cuales se combinan usando composición paralela para dar lugar al autómata sobre el cual resolveremos el problema de síntesis. Este autómata se denomina *planta* y la cantidad de estados crece de manera exponencial, a medida que se aumenta la cantidad de entidades que lo componen. Por lo tanto, si bien es posible de manera teórica, la idea de construir toda la planta y después buscar un controlador sobre ella es impracticable cuando se tienen muchas entidades en el entorno.

En este trabajo usaremos el algoritmo On-The-Fly DCS, propuesto por Ciolek[4], en su adaptación para resolver restricciones de tipo GR(1). Este algoritmo va construyendo la planta resultado de la composición paralela a medida que se va explorando, permitiendo llegar a un controlador sin necesidad de construir la totalidad de la planta. Para ello, DCS va expandiendo una transición de la planta a la vez, hasta llegar a un director compuesto de las transiciones expandidas. Esto se hace dependiendo de una heurística, la cual indica qué transición será la siguiente a expandir. De acuerdo con esta decisión, la exploración On-The-Fly puede terminar muy rápido si se eligen las transiciones correctas, o en el peor de los casos, si se deciden tranciciones incorrectas,

puede que se tenga que explorar una gran parte de la planta o incluso toda, haciendo que sea el rendimiento en memoria sea igual al algoritmo monolítico, el cual compone toda la planta y luego busca el controlador sobre esta. El algoritmo On-The-Fly DCS junto con todas sus heurísticas y el algoritmo monolítico, están implementados en la herramienta MTSA[5]. Esta herramienta es capas de resolver dos tipos de restricciones, las cuales son muy distintas, estas son: Non-Blocking o bien GR(1). En este trabajo nos enfocaremos principalmente en la restricción de tipo GR(1).

En esta tesis propondremos mejoras sobre la heurística desarrollada por Tomas Delgado[6], la cual se basa en usar *Aprendizaje por Refuerzos*(que abreviaremos RL por sus siglas en ingles, *Reinforcement Learning*). Este método tiene un proceso de aprendizaje mediante el cual se obtiene una heurística determinada por una *red neuronal*, capaz de resolver instancias más grandes y complejas de la que se uso para entrenar, siempre y cuando estas instancias mas grandes, tengan una estructura similar a la instancia en la que el agente fue entrenado.

La heurística basada en RL utiliza una función de abstracción que condensa la información de la exploración parcial en un vector binario, el cual es la entrada de la red neuronal del agente. Llamaremos a este vector *Feature Vector*, este contendrá información necesaria para que el agente pueda separar las transiciones que no son buenas para expandir de las que sí lo son y de esta manera decidir cual es la mejor transición a expandir.

En este trabajo propondremos un nuevo método para usar información que antes se descartaba por completo y que podría ser de mucha utilidad para el agente. Esta información se encuentra en los índices de las acciones, los cuales son números que indican que entidad esta hace una acción determinada. Usando esta nueva técnica proponemos agregar, a los feature vectors, información específica del problema que se quiere resolver, con el objetivo de darle más datos al agente y que de esta manera pueda explorar la planta de manera más óptima. Para esto, analizamos cada una de las 6 familias de instancias del benchmark propuesto por Ciolek, con el fin de entender cada uno de estos problemas y pensar, en cada caso, cuales son estos features particulares que le faltan al agente para mejorar su rendimiento. También exploraremos mejoras sobre el proceso de entrenamiento, tales como *curriculum learning*.

2. Síntesis de Controladores

2.1. Sistema de Eventos Discretos

Los Sistema de Eventos Discretos (DES por sus siglas en ingles), son sistemas los cuales pueden cambiar su estado a partir de la ocurrencia de eventos previamente establecidos, los cuales son identificados mediante etiquetas o labels. Estos eventos pueden ser de naturaleza controlada, si es posible elegir cuando realizar ese evento, o bien puede ser no controlada si el evento sucede de manera externa. Estos eventos cambian el estado del sistema de manera instantánea. Tantos los eventos que pueden suceder, como los posibles estados del sistema forman espacio discretos.

Los DES pueden representarse como autómatas finitos deterministicos o como redes de Petri. En este trabajo usaremos la representación de autómatas finitos deterministicos.

Autómata Finito Deterministico(AFD): Denotamos a un AFD como una tupla E = $(S_E, A_E, \rightarrow_E, \bar{e}, M_E)$, donde:

- S_E: Conjunto finito de estados.
- A_E: Conjunto finito de labels que denotan los eventos o acciones.
- $\rightarrow_E$: $S_E \times A_E \mapsto S_E$ es una función que toma un estado e, una acción a y devuelve el estado e' al que se llega al realizar la acción a en e.
- $\bar{e}$: Estado inicial. Este estado debe pertenecer al conjunto de estados ($\bar{e} \in S_E$).
- M_E: Conjunto de estados marcados ($M_E \subseteq S_E$).

Complementariamente, definiremos $e \xrightarrow{a} e'$ como la transición que indica que cuando el sistema este en el estado e y ocurre la acción a, este cambia al estado e'. Adicionalmente, definimos $_F(e)$ como el conjunto de acciones que salientes de e, formalmente: $a \in {}_F(e) \iff \exists \hat{e} / e \xrightarrow{a} \hat{e}$. También definimos el concepto de traza $\pi = \ell_0, \ell_1, \ell_2, ...$ como una sucesión de labels $\ell_i \in A_E$, estas trazas pueden tener una longitud finita o infinita.

Veamos el ejemplo mostrado en la figura 1, el cual esta inspirado en una de las instancias que usaremos mas adelante en este trabajo, el mismo modela un equipo que puede rechazar dos veces un proyecto o simplemente aceptarlo. Este sistema lo representaremos mediante la tupla $(\{1,2,3,4\}, \{\texttt{aceptar}, \texttt{rechazar}, \texttt{siguiente}\}, \rightarrow, 1, 2)$. La función $\rightarrow$ esta dada por las flechas del gráfico.

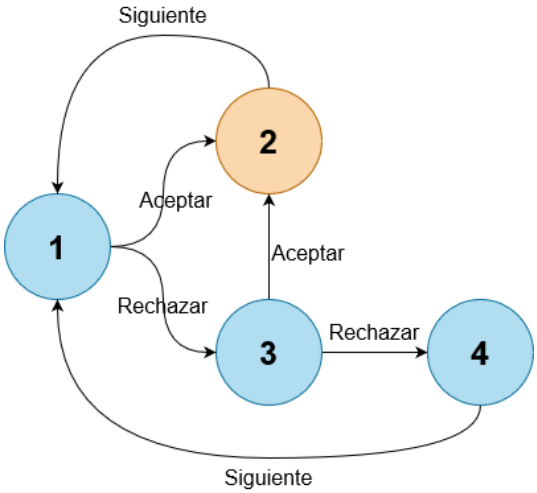

Figura 1: Un DES que modela un equipo el cual puede aceptar un documento o rechazarlo dos veces. Una vez hecho esto, puede pedir el siguiente documento y el proceso comienza de vuelta. El estado naranja representa el único estado marcado.

2.2. Composición Paralela

Si bien mediante los DES podemos modelar un sistema discreto, también es de interés modelar un conjunto de sistemas que interactúan entre si al mismo tiempo. Para eso podemos modelar cada sistema mediante un DES y combinarlos mediante una operación conocida como *Composición Paralela*. Esta operación nos permite combinar dos AFD y obtener uno nuevo, el cual representa la ejecución simultanea de ambos sistemas.

Composición Paralela: Dados dos AFD S $= (S_S, A_S, \rightarrow_S, \bar{e_S}, M_S)$ y T $= (S_T, A_T, \rightarrow_T, \bar{e_T}, M_T)$, se define la composición paralela entre estos dos como S$\|$T $= (S_S \times S_T, A_S \cup A_T, \rightarrow_{S\|T}, < \bar{e_S}, \bar{e_T} >, M_S \times M_T)$. Donde la función $\rightarrow_{S\|T}$ es la mínima relación que satisface simultáneamente las siguientes tres reglas:

(a) $s \overset{l}{\rightarrow}_S s' \wedge l \in A_S \setminus A_T \quad \Rightarrow \quad < s,t > \rightarrow^l_{S\|T} < s',t >$

(b) $t \overset{l}{\rightarrow}_T t' \wedge l \in A_T \setminus A_S \quad \Rightarrow \quad < s,t > \rightarrow^l_{S\|T} < s,t' >$

(c) $s \overset{l}{\rightarrow}_S s' \wedge t \overset{l}{\rightarrow}_T t' \wedge l \in A_S \cap A_T \quad \Rightarrow \quad < s,t > \rightarrow^l_{S\|T} < s',t' >$

La composición paralela es una operación asociativa y conmutativa, esto es importante ya que los AFD con los que trabajaremos serán el resultado de la composición paralela de n AFD. De esta manera usaremos el abuso de notación $E_1 \parallel E_2 \parallel ... \parallel E_n$ para denotar la composición de los AFD $E_1, E_2, ..., E_n$.

2.3. Fluent Linear Temporal Logic

Fluent Linear Temporal Logic (FLTL)[7] es un tipo de lógica modal usada para especificar requerimientos de manera estandarizada. Esta lógica proporciona un marco formal que permite hablar sobre eventos que ocurren en un DES junto con su temporalidad. Para esto se usan *Fluents*, los cuales sirven para representar las condiciones que pueden cumplir o no el sistema, en un instante determinado de tiempo. Estos se definen como una 2-tupla $Fl =< I_{Fl}, T_{Fl} >$ donde:

- I_{Fl}: conjunto de acciones que prenden(inicializa) el fluent Fl.
- T_{Fl}: conjunto de acciones que apagan(termina) el fluent Fl.

Además, si los fluent predican sobre el AFD E, se debe cumplir las siguientes propiedades, $I_{Fl} \subseteq A_E$, $T_{Fl} \subseteq A_E$ y $I_{Fl} \cap T_{Fl} = \emptyset$. O sea, las acciones que prenden y apagan el fluent deben ser acciones posible en el AFD E al cual hacen referencia y ninguna de estas acciones puede prender y apagar el fluent al mismo tiempo.

En este trabajo asumiremos que por defecto, todos los fluents empiezan con el valor de falso(apagado) y dependiendo de los eventos que ocurren en el sistema cambian su valor.

Llamaremos $\mathcal{F}$ al conjunto de todos los posibles Fluents. Con esto en mente, toda formula FLTL φ puede definirse inductivamente utilizando los operadores booleanos tradicionales y los operadores temporales $\mathbf{X}$(*next*) y $\mathbf{U}$(*until*) de la siguiente manera:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$$

Como es normal, se agregan los operadores $\wedge$, $\Diamond$ (*eventually*) y $\Box$ (*always*) como remplazos sintácticos para simplificar expresiones comúnmente usadas.

Luego se definen las valuaciones de la siguiente manera: decimos que la traza $\pi = \ell_0, \ell_1, ...$ satisface el fluent $Fl =< I_{Fl}, T_{Fl} >$ en la posición i si se cumple que: $\exists j \in \mathbb{N}(j \leq i \wedge \ell_j \in I_{Fl} \wedge \forall k \in \mathbb{N}(j < k \leq i \Rightarrow \ell_k \notin T_{Fl}))$. Esto lo denotamos como $\pi, i \vDash Fl$.

Bajo las mismas ideas se definen las valuaciones sobre las formulas FLTL. De esta manera decimos que la traza $\pi = \ell_0, \ell_1, ...$ satisface la formula φ en la posición i si se cumplen las reglas recursivas:

$$\begin{array}{lcl}
\pi, i \vDash Fl & \equiv & \pi, i \vDash Fl \\
\pi, i \vDash \neg\varphi & \equiv & \neg(\pi, i \vDash \varphi) \\
\pi, i \vDash \varphi_1 \vee \varphi_2 & \equiv & (\pi, i \vDash \varphi_1) \vee (\pi, i \vDash \varphi_2) \\
\pi, i \vDash \mathbf{X}\varphi & \equiv & \pi, i+1 \vDash \varphi \\
\pi, i \vDash \varphi_1 \mathbf{U} \varphi_2 & \equiv & \exists j \geq i \quad \pi, j \vDash \varphi_2 \wedge \forall i \leq k < j \quad \pi, k \vDash \varphi_1
\end{array}$$

Denotamos la satisfacibilidad como $\pi, i \vDash Fl$. Además, decimos que π satisface φ si $\pi, 0 \vDash Fl$, esto lo simplificaremos diciendo $\pi \vDash \varphi$. Decimos que el DES E satisface una formula φ, si para toda traza infinita de E se cumple que se satisface φ, esto lo denotamos como $E \vDash \varphi$.

Por ultimo, todo fluent se puede representar como un AFD de dos estados. Cada uno de estos estados representa si el fluent esta apagado o prendido, el estado inicial por defecto es el estado apagado y el AFD cambia al estado prendido si se realiza una acción del conjunto I_{Fl} y cambia al estado apagado si se realiza una acción del conjunto T_{Fl}. Por ejemplo, notemos que el fluent $< \{a, b\}, \{c\} >$ puede ser visto como el AFD $(\{\texttt{on}, \texttt{off}\}, \{\texttt{a}, \texttt{b}, \texttt{c}\}, \rightarrow, \texttt{off}, \{\texttt{on}\})$.

Esto nos va a servir, mas adelante, para componer paralelamente el fluent con otros DES, marcando los estados del DES donde se prende o no el fluent.

2.4. Problema de control

Con todos los conceptos previamente establecidos, ahora podemos definir el problema de control. Dado un conjunto AFD $E_1, ..., E_n$, un conjunto de acciones que llamaremos *controlables* A_C y una conjunto de formulas FLTL H. Denotaremos la *planta* a controlar como $E = E_1 \parallel E_2 \parallel ... \parallel E_n = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$. Las acciones controlables cumple que $A_C \subseteq A_E$ y denotaremos $A_U = A_E \setminus A_C$ a las acciones no controlables.

Queremos encontrar una función $\sigma : A_E^* \rightarrow \mathcal{P}(A_E)$ la cual llamaremos *controlador*, que dado un camino w de la planta E, devuelva el conjunto de de acciones que pueden ser realizadas de manera tal para cualquier acción que se realiza de ese conjunto, existe una camino que cumple la conjunción de formulas en H.

De esta manera, el controlador obtenido determina un nuevo AFD, en el cual estamos seguros de que podemos cumplir los objetivos expresados en H. Aunque un controlador puede dejar habilitadas múltiples acciones controlables, nos enfocaremos únicamente en encontrar controladores los cuales, para cada estado, habiliten como máximo una acción controlable. Este tipo de controladores se llaman **directores** y es el tipo de controlador en el que nos enfocaremos en este trabajo. Esta idea contrasta con la noción clásica de *supervisor*, el cual es un controlador que intenta habilitar la mayor cantidad de acciones controlables en cada estado.

Dado un controlador σ, definimos el lenguaje inducido por σ sobre la planta E, como $\mathcal{L}^\sigma(E)$. Este conjunto contiene todas las trazas posibles en E, que empiezan en el estado inicial de la planta E y que son permitidas por el controlador σ.

2.5. General Reactivity

Ya hemos definido el problema de control, sin embargo, se puede restringir de diversas maneras las fórmulas FLTL H. En este trabajo, abordaremos una restricción del problema de control llamada *General Reactivity* de rango 1, la cual llamaremos por su forma abreviada GR(1)[8]. En esta versión del problema de control, restringimos las fórmulas H a las formulas de la forma $\square\diamond \bigwedge_{i=1}^{n} \phi_i$ y $\square\diamond \bigwedge_{i=1}^{m} \gamma_i$ donde ϕ_i y γ_i son fluents. De esta manera las formulas que necesitamos que el controlador satisfaga, tienen dos partes, *assumptions* y *goals* donde nosotros buscamos que se cumpla que las *assumptions* impliquen las *goals*.

Problema de Control GR(1): Dado un AFD $E = E_1 \parallel E_2 \parallel ... \parallel E_k$ que llamaremos *planta*, dos conjuntos de fluents $As = \phi_1, ..., \phi_n$, $G = \gamma_1, ..., \gamma_m$ y un conjunto de acciones controlables A_C. Queremos obtener un controlador $\sigma : A_E^* \rightarrow \mathcal{P}(A_E)$ que cumpla las siguiente propiedades:

1. *Controlable*: $A_U \subseteq \sigma(w) \quad \forall w \in A_E^*$
2. *Dirigido*: $|A_C \cap \sigma(w)| \leq 1 \quad \forall w \in A_E^*$
3. *General Reactivity*: $\forall w \in \mathcal{L}^\sigma(E) \quad \exists w' \in A_E^* : |w'| > 0 \wedge w.w' \in \mathcal{L}^\sigma(E) \wedge w.w' \vDash \square\diamond \bigwedge_{i=1}^{n} \phi_i \Rightarrow w.w' \vDash \square\diamond \bigwedge_{i=1}^{m} \gamma_i$

Las propiedades *Controlable* y *Dirigido* indican, respectivamente, que el controlador solo debe deshabilitar acciones controlables (las no controlables deben permanecer siempre habilitadas) y junto a eso, debe actuar como un director, es decir, habilitar como máximo una sola acción.

Finalmente, la propiedad de *General Reactivity* asegura, en primer lugar, que la planta controlada por el director no tenga ningún estado deadlock, ya que w' no puede ser una cadena vacía ($|w'| > 0$), esto nos da la idea de que la planta controlada debe poder estar en constante movimiento, debe estar "viva". En segundo lugar, asegura que toda traza que cumple el conjunto de *assumptions* debe cumplir también el conjunto de *goals*.

2.6. OTF-DCS

Para resolver el problema de control GR(1), existen algunos algoritmos tales como el Monolítico [9], el cual consiste en componer la planta completa y resolver el problema analizando la planta en su totalidad. Si bien esta solución tiene una complejidad polinomial en la cantidad de estados de la planta, el tamaño de la planta crece de manera exponencial a medida que se agregan más AFD que la componen. Esto hace que en la práctica sea imposible aplicar esta solución a problemas con muchos AFD a componer o con muchos estados.

Para resolver esto, Ciolek propone el algoritmo On The Fly - Directed Controller Synthesis (OTF-DCS) en su tesis doctoral[4], el cual posteriormente fue revisado, corregido y mejorado por Zanollo y Duran [10]. Esta alternativa construye una representación parcial de la planta, la cual se extiende según sea necesario (de ahí el nombre "On The Fly"), con la esperanza de encontrar un director antes de construir toda la planta entera.

OTF-DCS comienza desde el estado inicial de la planta. A partir de este estado, se decide una transición saliente y se agrega el estado destino a la exploración parcial. Luego, cada estado de la exploración parcial se clasifica en 3 tipos: *Goal* (desde ese estado se conoce una estrategia ganadora para resolver el problema), *Error* (desde ese estado no hay una estrategia ganadora, por lo tanto, existe un controlador que comienza en ese estado) y *None* (aún no se tiene suficiente información para decidir la clase del estado). Luego, este proceso se repite múltiples veces hasta que el estado inicial esté en un estado distinto a *None* o hasta que no existan más transiciones para expandir. En caso de que el estado inicial esté en estado *Goal*, se construye un controlador. En caso contrario, no se puede construir un controlador, ya que no existe.

De esta manera, es necesario en cada paso de expansión decidir una transición a expandir del conjunto de expansiones, el cual llamaremos *frontera de expansión*, ya que delimita la exploración parcial del resto de la planta. Uno podría pensar que la decisión de qué transición expandir no influye en la performance final de OTF-DCS sin embargo, esto no es así, ya que explorar un área de la planta que no estará en el director final resulta en un gasto de tiempo considerable. Por lo tanto, es crucial seleccionar con precisión las transiciones a expandir para maximizar la eficiencia del algoritmo.

Existen múltiples heurísticas que se encargan de decidir la mejor transición a expandir dada una exploración parcial y su frontera de expansión. El algoritmo OTF-DCS es muy sensible a la heurística a usada, esto significa que si la misma decide mal que transiciones expandir, el algoritmo expandirá muchas transiciones de manera innecesaria, causando que se pierda mucho tiempo chequeando el cierre de ciclos y la propagación de los tipos de estados. El caso ideal sería tener una heurística que siempre expanda las transiciones pertenecientes al director con menor cantidad de transiciones, de esta manera se realizarían la mínima cantidad de expansiones, lo que llevaría a un resultado mucho más rápido. Otra cosa a considerar de las heurísticas es que deben ser rápidas a la hora de estimar que transición debe ser expandida, ya que esta consulta se realizara en cada expansión.

En la figura 2 vemos un ejemplo de exploración parcial de una planta. En esta exploración ya se han expandido las transiciones representadas por flechas negras y la frontera de exploración consta de cuatro transiciones, representadas por flechas de color amarillo. En este paso, la heurística debe decidir cuál de esas cuatro transiciones expandirá, añadiendo así un estado de la zona no explorada a la exploración parcial. La zona no explorada de la planta no tiene transiciones, ya que el algoritmo la considera como un único estado que agrupa todos los estados no explorados hasta el momento. De esta manera, no es necesario construir todos los estados desde el inicio, sino que se crean a medida que se encuentran en la zona no explorada.

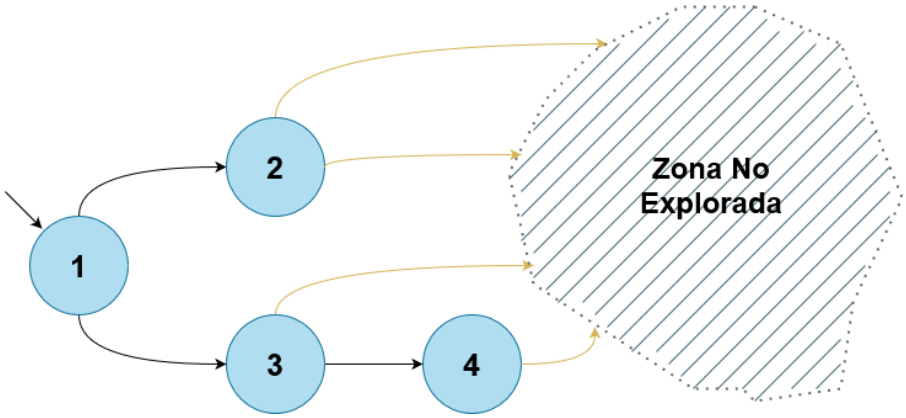

Figura 2: Exploración parcial donde ya se exploraron 4 estados y se expandieron 3 transiciones(negras). La frontera de expansión consta de 4 transiciones(amarillas).

También cabe aclarar que no se sabe a qué estado va cada transición hasta que se expande. Esto hace que algunas transiciones puedan ir al mismo estado, sin embargo, nosotros lo desconocemos. Por este motivo, algunas heurísticas también construyen los estados a los que van las transiciones de la frontera de expansión, con el objetivo de tener mayor información de la planta y poder decidir mejor qué transición expandir.

En este trabajo, nos enfocamos en utilizar una heurística basada en aprendizaje por refuerzos y en cómo podemos mejorarla utilizando información específica del dominio del problema. A continuación, expondremos algunas heurísticas conocidas que más adelante usaremos como base para comparar las mejoras introducidas.

2.7. Heurísticas

2.7.1. Random

Esta heurística es la más simple de todas y la usaremos como la línea de base en la comparación entre distintas heurísticas. Se trata de tomar una transición de la frontera en base a una distribución aleatoria uniforme. Si bien es muy fácil de implementar, no suele ofrecer buenos resultados. Además, al depender de la aleatoriedad de las transiciones elegidas, su comportamiento puede variar.

2.7.2. BFS

La heurística de *Breadth First Search*(BFS) es otra heurística simple de entender y de implementar. Esta se basa en explorar la planta a lo ancho, como si fuera un grafo. Incluimos esta heurística con el objetivo de tener un punto de comparación un poco más avanzado que la heurística random, pero no tan sofisticado como las heurísticas que presentaremos a continuación. Además, esta heurística es totalmente determinista, por lo que es mas fácil usarla como comparación inicial.

2.7.3. Ready Abstraction

Si bien Ciolek propuso el algoritmo OTF-DCS, en su tesis doctoral, también propuso una heurística llamada *Ready Abstraction* (RA) [4], la cual luego fue corregida y mejorada por Pazos [11]. Si bien esta heurística fue pensada originalmente para el problema de control en su versión Non-Blocking, esta puede ser adaptada para resolver el problema de control en su versión GR(1).

La idea principal de esta heurística es estimar la distancia a los estados donde los fluents están activos (estado marcado). Para esto se construye un grafo a partir de los AFD individuales que componen la planta. Con este grafo como estructura auxiliar, se calcula un estimado de la distancia de cada nodo de la frontera de exploración a un estado marcado. Una vez obtenido esta estimación, se puede ordenar las posibles transiciones a expandir según este valor junto con otros criterios que también influyen a la hora de decidir cuál es la mejor transición a expandir. Algunos de esos criterios extra que observa la heurística RA son, darle importancia a las acciones no controlables y priorizar los estados marcados ya recorridos con el objetivo de cerrar ciclos lo antes posible.

Es importante remarcar que originalmente, la heurística RA tenía un mecanismo llamado *Open Queue*, el cual se encargaba de bloquear algunas transiciones de la frontera con el objetivo de guiar la exploración en profundidad en ciertos casos específicos. En su trabajo de tesis Pazos demostró que esto no siempre contribuye a que RA resuelva el problema usando menor cantidad de expansiones y además le saca flexibilidad a la decisión de la heurística. El propuso una serie de mejoras a la hora de encontrar la transición más prometedora, una de ellas fue remover el mecanismo de *Open Queue*.

En este trabajo adaptamos algunas mejoras introducidas por Pazos para la versión del problema GR(1), esto lo hicimos con el objetivo de tener una mejor heurística a la hora de comparar resultados, ya que con estas mejoras, esta heurística aumenta considerablemente su performance y es, de las heurísticas disponibles, la que mejor resultados otorga.

2.7.4. RL

En su trabajo de tesis de licenciatura, Delgado[6] desarrolló una heurística basada en *Reinforcement Learning* (RL). La idea principal de esta heurística es contar con un agente que utiliza una red neuronal profunda, la cual esta enmarcada en el algoritmo de *Deep Q-Learning*, para asignarle un valor a cada transición en frontera de exploración. Luego para decidir la siguiente transición a expandir, elije la transición con el mayor valor(la mejor según la red neuronal).

Este algoritmo requiere un proceso previo de entrenamiento para ajustar los parámetros de la red neuronal y luego se utiliza dicha red para predecir cuál es la mejor transición de la frontera de expansión. Esta red neuronal toma una abstracción de las transiciones de la frontera de exploración, en la cual esta codificada la información que nosotros consideramos mas importante sobre cada transición.

En esta tesis expondremos algunas mejoras sobre la abstracción de los estados que recibe la red neuronal del agente. Para poder explicar estas mejoras propuestas, primero necesitamos entender como funciona esta heurística basada en RL. Por este motivo, en la siguiente sección, explicaremos los fundamentos del área de Reinforcement Learning y posteriormente detallaremos cómo enmarcamos la heurística del DCS-OTF en el esquema explicado de Reinforcement Learning.

3. Heurística Basada en Reinforcement Learning

El área de *Aprendizaje por Refuerzo* o *Reinforcement Learning* (RL) es un área de la inteligencia artificial, la cual se inspira en la forma en que los seres vivos aprenden a través de la interacción con su entorno. A diferencia de otros enfoques de aprendizaje automático, donde se proporcionan conjuntos de datos etiquetados, en RL el agente aprende a través de la prueba y error, tomando decisiones secuenciales para maximizar una recompensa (*reward*).

A diferencia de otras formas de aprendizaje automático, como podrían ser el aprendizaje supervisado o no supervisado, el marco conceptual de RL se basa en la idea de que un agente, que toma decisiones en un entorno específico y observando las consecuencias de las acciones realizadas, el agente puede aprender a través de la experiencia acumulada.

Esto es lo que se ilustra en la figura 3, donde vemos el esquema de interacción entre el ambiente y un agente, los cuales interactúan entre sí. El agente observa el estado en el que se encuentra el ambiente y mediante algún proceso, decide realizar una acción, la cual modifica el estado del ambiente. Además, el agente también observa la recompensa que obtiene al realizar una acción, y en base a esto, puede modificar su comportamiento para maximizar dicha recompensa en las próximas acciones.

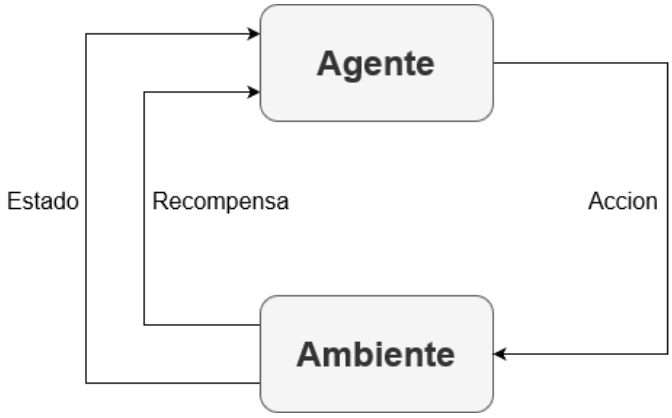

Figura 3: Diagrama de interacción ambiente agente.

A lo largo del tiempo, el agente aprende a mapear estados a acciones de manera que maximice la recompensa acumulada a largo plazo. Este proceso de aprendizaje implica la exploración de diferentes estrategias para descubrir aquellas que conducen a las mayores recompensas en el ambiente específico en el que se encuentra el agente. Si bien la interacción entre el agente y el ambiente podría ser infinita, en este trabajo solo nos enfocaremos en ambientes en los cuales se puede considerar que el mismo ha terminado al cumplirse algunas condiciones específicas. Bajo este contexto, el objetivo del agente será maximizar la sumatoria de recompensas obtenidas desde el primer estado observado hasta que el ambiente finalice.

El ambiente con el que interactúa el agente se puede representar como un proceso de Decisión de Markov (MDP por sus siglas en inglés), el cual cambia de manera estocástica de estado a partir de las acciones que realiza el agente. Este proceso se puede representar como una tupla $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, s_0)$ donde:

- $\mathcal{S}$ es el conjunto de todos los posibles estados.
- $\mathcal{A}$ es el conjunto de las posibles acciones que pueden ser tomadas.
- $\mathcal{P}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ es la función que codifica la probabilidad de, al realizar la acción a en el estado s, terminar en el estado s'. Formalmente $\mathcal{P}(s'|a, s)$.
- $\mathcal{R}$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ es la función que determina la recompensa obtenida.
- $s_0 \in \mathcal{S}$ estado inicial del proceso.

Uno de los conceptos clave en los MDP es la propiedad de Markov, que establece que el estado futuro del sistema depende únicamente del estado actual y no de la secuencia completa de estados previos. Esta propiedad simplifica el modelado y la resolución de problemas de toma de decisiones secuenciales al permitir que los agentes tomen decisiones basadas únicamente en la información actual del entorno.

A continuación, definimos algunos conceptos referentes a MDP, los cuales serán importantes ya que los usaremos frecuentemente en este trabajo:

– Step: Es un paso del MDP. Pasar de un estado s a otro s' mediante la acción a.

– Episodio: Es una secuencia de Steps los cuales empiezan en el estado inicial s_0 y terminan en un estado sin posibles acciones a realizar.

– $\mathcal{A}(s)$: conjunto de acciones de acciones posibles a tomar en el estado s.

– Política o Policy: $\Pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ es la función que determina un agente. Esta indica la probabilidad de que un agente tome una acción a en un estado s dado.

– $V_\Pi(s)$ (Valor esperado): es la sumatoria de recompensas que se espera obtener si se esta en el estado s y se toma la policy Π. Formalmente se define como: $\mathbb{E}_\Pi[\sum_{i \in \{s,\dots,s_n\}} \mathcal{R}(s)]$

– $Q_\Pi(s, a)$ (Q-Function): es la sumatoria de recompensas que se espera obtener si se esta en el estado s y se comienza realizando la acción a.

– $Q^*(s, a)$: es la Q-Function óptima, formalmente $Q^*(s, a) = max_\Pi Q_\Pi(s, a)$.

3.1. Q-Learning

Si bien aprender una política puede ser una tarea muy desafiante si el ambiente es complejo, Q-Learning es un algoritmo que nos permite aprender los valores de la función $Q^*(s, a)$ en base a la exploración que puede hacer el agente sobre el ambiente con el que interactúa. Luego, una vez obtenida la función $Q^*(s, a)$, la política sale de seleccionar la acción con el valor más alto para un estado fijo. Esta es una política greedy, ya que se quiere buscar maximizar el valor inmediato, ignorando las posibles futuras implicaciones que esto podría traer.

Para obtener el valor de la función $Q^*(s, a)$, iterativamente se hace correr al agente en el entorno usando una política ϵ-greedy, la cual se trata de hacer con ϵ probabilidad una acción aleatoria y con $1-\epsilon$ probabilidad una acción en base a los valores de $\hat{Q}(s, a)$, la cual es la aproximación que tenemos de $Q^*(s, a)$. Además, el valor de ϵ se decrementa de manera lineal a medida que se realiza una acción, hasta que este llegue a 0. De esta manera, se delimita una etapa de exploración (donde el agente puede explorar haciendo movimientos aleatorios con cierta probabilidad) y otra donde solo se usan los valores de $\hat{Q}(s, a)$ ya existentes en base a lo que ya se conoce. Además, cuando el episodio finaliza, el ambiente se reinicia volviendo al estado inicial.

Inicialmente, todos los valores de la representación de la función $\hat{Q}(s, a)$ son inicializados aleatoriamente. Cada vez que el agente hace un step, se actualiza un valor de la aproximación de acuerdo a la ecuación de Bellman:

$$\hat{Q}(s, a) = \hat{Q}(s, a) + \alpha(\mathcal{R}(s, a) + \max_{a' \in \mathcal{A}(s')} \hat{Q}(s', a') - \hat{Q}(s, a))$$

Esta fórmula nos da una manera de actualizar el valor para un par *estado-acción* (a, s), este nuevo valor se define a partir del valor previo. Para esto es necesario saber el estado siguiente a s, este estado es representado por s'. Además, esta fórmula depende de un hiperparámetro α, el cual regula el tamaño del salto. La ecuación de Bellman nos garantiza convergencia a $Q^*(s, a)$ para todo *estado-acción* (a, s), siempre y cuando se pase suficientes veces por cada par y el valor de α decremente correctamente.

Para representar la aproximación $\hat{Q}(s, a)$ se suele usar una matriz de tamaño $n \times m$ donde n es la cantidad de posibles estados en los que puede estar el ambiente y m la cantidad de posibles acciones. Esta manera de representar $\hat{Q}(s, a)$ es llamada Tabular Q-Learning.

Si bien existen otras alternativas, siempre que se quieran guardar los valores para cada par *estado-acción*, en el peor de los casos siempre se va a necesitar guardar $n \times m$ valores en memoria, esto hace que en ambientes donde el espacio de estados posibles(n) es muy grande o las posibles acciones(m) son muchas esto se vuelve impracticable.

3.2. Deep Q-Learning

Para solucionar el problema de necesitar una estructura que use una cantidad de memoria inmanejable, podemos aproximar la función $Q^*(s,a)$ usando una red neuronal, la cual depende de un vector de pesos ω. Esta aproximación la llamaremos $\hat{Q}_\omega(s,a)$. La red neuronal, en un principio, es inicializada con pesos aleatorios y de alguna manera se entrena para mejorar la aproximación de la función $\hat{Q}_\omega(s,a)$. Esto nos trae otros beneficios, como facilidad para generalizar, ya que ahora no es necesario haber pasado por un estado para obtener su valor esperado.

Si bien esto resuelve el problema de almacenar el valor esperado para cada par *estado-acción*, también nos obliga a cambiar la forma de mejorar la aproximación, ya que no podemos usar la ecuación de Bellman previamente introducida. Esto se debe a que las redes neuronales no nos permiten actualizar un único valor como previamente hacíamos, sino que, solo podemos modificar el vector de pesos ω, lo cual cambia múltiples valores de la función.

Para encontrar un vector de pesos ω óptimo, usaremos descenso de gradiente para minimizar el error cuadrático medio con la intención de encontrar un punto fijo para la ecuación de Bellman. Es decir, queremos encontrar el ω que optimice la ecuación:

$$(\mathcal{R}(s,a) + \max_{a' \in \mathcal{A}(s')} \hat{Q}_\omega(s',a') - \hat{Q}_\omega(s,a))^2$$

Aunque esto nos brinda una manera de encontrar $Q^*(s,a)$, existen tres características importantes. Primero, al no explorar con una política fija (*off-policy*), segundo el hecho de que la función se actualice en base a sí misma (*bootstrapping*) y por ultimo, que las muestras con las que entrenamos la aproximación están fuertemente correlacionadas (*function approximation*). Estas tres características combinadas se conocen como la *deadly triad*[12] y pueden llevar a la divergencia.

Aunque existen múltiples alternativas para solucionar estos problemas, en este trabajo principalmente utilizaremos dos alternativas, las cuales pueden emplearse simultáneamente. A continuación exploraremos estas alternativas.

3.2.1. Experience Replay

El *Experience Replay Buffer*[13] es la primera técnica que exploraremos para solucionar la *deadly triad*. La idea principal detrás del *Experience Replay Buffer* es almacenar en un búfer una colección de experiencias pasadas del agente. Cada cierto tiempo se eligen aleatoriamente una cantidad fija de experiencias y se entrena la red neuronal. De esta manera, podemos romper la correlación entre los datos con los que entrenamos la red, mejorando su convergencia. Sin embargo, esta solución agrega el problema de tener varios hiperparámetros que ajustar.

3.2.2. Target Network

La segunda alternativa es el uso de una *Target Network*[14]. Esta técnica implica tener otra red neuronal adicional, la cual llamaremos $\hat{Q}_{\hat{\omega}}(s,a)$. Cada cierto tiempo, esta red se copia de la original y no se actualiza. El objetivo es hacer que la red neuronal que entrenamos ($\hat{Q}_{\omega}(s,a)$) se asemeje a esta *Target Network* mientras mantenemos la última congelada, hasta el momento de copiarla nuevamente. De esta manera, queremos obtener los pesos ω que minimicen la expresión:

$$(\mathcal{R}(s,a) + \max_{a' \in \mathcal{A}(s')} \hat{Q}_{\hat{\omega}}(s',a') - \hat{Q}_{\omega}(s,a))^2$$

De esta manera, hacemos que el objetivo a minimizar varíe, lo que ayuda a permitir que la red original converja a un resultado óptimo.

3.3. Aplicando Reinforcement Learning

Ahora que ya definimos de manera general los conceptos mas importantes que usaremos sobre *Reinforcement Learning*, podemos ver cómo se enmarca la heurística usada por OTF-DCS como un proceso resoluble mediante Aprendizaje por Refuerzo. Para esto, se define el MDP con el objetivo de que el agente aprenda a explorar la planta de manera eficiente en cuanto a la cantidad de expansiones a lo largo del episodio.

Dada una planta $E = E_1 \parallel ... \parallel E_n = (S_E, A_E, \rightarrow_E, \bar{e}, M_E)$ se define su MDP asociado de la siguiente manera:

- $\mathcal{S} = h : h$ es una secuencia de exploración de E
- $\mathcal{A} = S_E \times A_E$
- $\mathcal{P}(s'|s,a) = \begin{cases} 1 & \text{si } a \in f(s) \\ 0 & \text{caso contrario} \end{cases}$
- $\mathcal{R} =$ Función constante en -1
- $s_0 = \emptyset$

De esta manera, el agente explora un MDP compuesto por las secuencias de exploración parciales, y las acciones que puede realizar corresponden a todas las posibles transiciones que puede expandir en algún momento de la exploración. Además, este MDP es completamente determinista, ya que todas las transiciones posibles están definidas por la planta, la cual es determinística al salir de la composición paralela de n AFD y al tener el estado inicial como la secuencia vacía, dado que inicialmente el agente no ha realizado ninguna expansión. Finalmente, la función de recompensa es la función constante -1, ya que nuestro objetivo es minimizar la cantidad de expansiones que se realizan. Al elegir esta función, la sumatoria de recompensas representan la du-

ración del episodio, elemento que queremos minimizar, como RL solo puede maximizar cambiamos de signo esta recompensa para lograr esto.

Una anotación a realizar, es que en la sección anterior mencionamos el concepto de *estado-acción*. Ahora como estamos en el contexto de síntesis de controladores, pasaremos a llamar *transición* a los pares *estado-acción*, sin embargo para el agente el concepto que representan es el mismo.

Con el MDP definido, podríamos comenzar a entrenar al agente para resolver este problema. Sin embargo, nuestro objetivo es abordar instancias de gran tamaño, las cuales suelen requerir numerosas expansiones para ser resueltas. Esto implica que el agente de RL puede enfrentar dificultades para aprender en estas instancias, especialmente debido a lo desafiante que resulta completar un episodio utilizando la política ϵ-greedy. Por esta razón, optaremos por entrenar al agente en instancias de menor tamaño, donde los episodios son más breves y manejables.

3.4. Feature Vector

Aunque es importante que las instancias en las que entrenamos al agente sean estructuralmente similares a aquellas que queremos resolver, esto no es suficiente. También necesitamos que los estados en ambas instancias sean idénticos, asegurando así que la red neuronal del agente siempre reciba entradas del mismo tamaño. Si no pudiéramos lograr esto, no podríamos usar la red neuronal aprendida en una instancia chica para resolver instancias grandes, o sea, no podríamos generalizar la resolución de instancias similares.

Para lograr esta homogeneización de las transiciones, utilizaremos *features vectors*. Estos codifican cada transición de la frontera de exploración en un vector que contiene las características más relevantes de dicha transición. A estos vectores los llamaremos *feature vectors* y los denotaremos como $\phi(s,a)$, este vector binario donde cada elemento determine si se cumple o no alguna propiedad de la transición (s,a) en la exploración parcial del momento. De esta manera, la entrada de la red neuronal que determina al agente serán los features vectors de la frontera de expansión.

La propiedad clave de los feature vectors es que, para instancias con el mismo conjunto de acciones, estos tienen el mismo tamaño. De esta manera, mantenemos constante el tamaño de los datos de entrada de la red neuronal, haciendo posible usar el agente entrenado en instancias pequeñas sobre instancias más grandes.

En este trabajo, adaptamos los features utilizados por Delgado[6]. En su tesis de licenciatura, la heurística de RL estaba diseñada para resolver la versión Non-Blocking del problema de control. En nuestro estudio, adaptamos esta heurística para que pueda resolver la versión GR(1) utilizando features vectors equivalentes. A continuación, listamos los grupos de features que se extraen de la transición $e \xrightarrow{a} e'$, entre paréntesis la cantidad de features que tiene cada grupo. Usamos la notación $\#\hat{A}$ para indicar la

cantidad de posibles acciones en la planta si eliminamos los valores numéricos(índices) que estas tienen.

- Etiqueta de evento ($\#\hat{A}$): es un vector donde el único elemento prendido es el que corresponde al label a sin índice.
- Etiqueta de estado ($\#\hat{A}$): es un vector donde los únicos elementos prendidos son los que corresponden los labels sin índice que llegan a e desde estados ya explorados.
- Controlable (1): Indica si a es un label controlable.
- Estado marcado (2): Indica si el estado e cumple el *goal*.
- Fase de Exploración (3): Estos features se activan cuando se ya encontró un estado que cumple el *goal*, ya se cerro algún ciclo o bien, si OTF-DCS ya cambia algún estado de *None* a *Goal*.
- Estado de llegada (3): Cada feature indica si e' es marcado por OTF-DCS como *None*, *Goal* y *Error*.
- Vecindario controlable (4): Indica si tanto e como e', tienen eventos no controlables y en caso de tenerlos también si ya fueron todos explorados.
- Explorados (2): Indica si ya se exploro algún evento saliente de e o de e'
- Ultima expansión (2): Indica si e y e' es el ultimo estado expandido por el egente.

4. Mejoras Propuestas

El principal aporte de este trabajo es demostrar de manera empírica que la heurística basada en RL puede mejorar si se le provee información específica del dominio a resolver.

Para esto, usaremos un benchmark que fue originalmente propuesto por Ciolek[4] y contiene 6 problemas de interés en distintas áreas. Cada una de las familias de instancias tiene dos variables que escalan el tamaño del problema. Estas variables son n que indica la cantidad de entidades que componen el sistema y k que aumenta la cantidad de estados de cada componente. Mientras mas aumenta el valor de estas variables mas grande es la planta a explorar y por lo tanto mas difícil a resolver el problema.

Esto es particularmente útil para nuestro caso, ya que nos permite clasificar las instancias según su dificultad y utilizar las más fáciles para entrenar heurísticas como RL. De esta manera, utilizaremos las instancias de tamaño $n = 2$, $k = 2$ para entrenar, seleccionaremos el mejor agente y evaluaremos el desempeño del mismo en cada posible combinación de $n \in [1, 15]$ y $k \in [1, 15]$. De esta forma, tendremos 225 instancias para cada familia de instancias.

4.1. Familias de Instancias

En este apartado introduciremos cada familia de instancias con el objetivo de entenderlas individualmente y poder explicar cuáles fueron los features que agregamos al feature vector pasado al agente en cada caso. Todas estas instancias están especificadas en formato FSP [15].

- Air-Traffic Management (`AT`):
  Esta familia de instancias[16] modela n aviones que quieren aterrizar a un aeropuerto. La torre de control central controla los n aviones, estos pueden estar en k espacios aéreos. La torre puede decidir si un avión aterriza o realiza maniobras de espera en alguno de los k aeropuertos. El objetivo es que todos los aviones puedan aterrizar de manera segura en alguno de los aeropuertos. Es importante aclarar que esta familia de problemas solo tiene solución si la cantidad de aviones es menor o igual que la cantidad de espacios aéreos, es decir, $n \leq k$. Esto se debe a que si hay más peticiones de aterrizaje que espacios de espera, no hay manera de evitar choques.

- Bidding Workflow (`BW`):
  Esta familia de instancias[17] modela una empresa que debe decidir si aprobar o rechazar el documento de un proyecto. Para que el documento se considere aprobado, debe ser aceptado por n equipos de trabajo. El mismo puede ser reevaluado por un equipo hasta k veces, sin embargo, no se puede reasignar a un equipo que ya lo haya aceptado. Finalmente, si un equipo lo rechaza más de k veces, el proyecto es definitivamente rechazado. El objetivo controlar a

que equipo se le asigna la decisión de aceptar o rechazar el documento, de manera tal de sintetizar un flujo de trabajo que logre llegar a un consenso, ya sea aceptando o rechazando el proyecto. Esta familia de instancias es un caso específico de estudio del dominio de Business Process Management (BPM).

- Cat and Mouse (`CM`):
  Esta familia de instancias[18] modela un juego por turnos de dos jugadores. Hay n gatos y n ratones en un corredor de $2k + 1$ áreas. Todos los ratones comienzan en un extremo del corredor y todos los gatos comienzan en el otro extremo. En el centro del corredor hay un agujero seguro para los ratones, si estos alcanzan esta posición los gatos ya no pueden atraparlos. Tanto gatos como ratones pueden moverse a cualquier casilla adyacente o quedarse en su lugar. El movimiento de los gatos está modelado mediante transiciones no controlables, mientras que nosotros podemos controlar el movimientro de los ratones, los cuales son los primeros en mover. El objetivo es que los ratones puedan alcanzar ese lugar seguro sin cruzarse con ningún gato. Cabe aclarar que esta familia de instancias es la más difícil de resolver en términos de tiempo de ejecución.

- Dinning Philosophers (`DP`):
  Esta familia de instancias[19] modela una variante del problema clásico de los filósofos. En este problema, hay n filósofos sentados en una mesa redonda, donde cada filósofo tiene dos tenedores a su lado, los cuales comparte con los filósofos adyacentes. Cada filósofo debe alternar entre comer y pensar. Para comer, un filósofo necesita tomar los dos tenedores adyacentes a él. Una vez que un filósofo ha tomado un tenedor, necesita pensar durante k instantes de tiempo antes de tomar el otro tenedor. Todas las acciones son no controlables salvo por la decisión de que un filosofo tome un tenedor. El objetivo es que todos los filósofos puedan comer tarde o temprano y que ninguno caiga en un estado de deadlock.

- Travel Agency (`TA`):
  Esta familia de instancias[20] modela un servicio de ventas de paquetes mediante una página web. Un paquete de viaje depende de n servicios, además un cliente puede elegir entre k variantes de un servicio dado. Una vez seleccionadas las variantes, es necesario reservar y pagar el paquete. El objetivo del sistema es orquestar los servicios para obtener un paquete completo, sin que se reserven y paguen paquetes incompletos.

- Transfer Line (`TL`):
  Esta familia de instancias[21] modela una fábrica con n máquinas, las cuales están conectadas de manera secuencial con n buffers. Además, el último buffer está conectado a una máquina especial llamada Test Unit. Cada uno tiene una capacidad de k elementos. La salida de una máquina está conectada al buffer de entrada de la siguiente. El objetivo del sistema es procesar todos los elementos,

pasando por las n máquinas y llegando hasta la Test Unit, sin caer en overflow ni underflow de los buffers. Cabe aclarar que esta familia de instancias es la más sencilla de resolver en términos de tiempo de ejecución.

4.2. Abstrayendo Información de las Entidades

Si bien Delgado[6] probó de manera experimental que la heurística basada en RL funciona muy bien en algunos casos para resolver problemas de Non-Blocking, también hay ciertos casos en los que no logra resolver el problema de manera óptima.

Al comenzar este trabajo, observamos que el comportamiento de la heurística basada en RL es muy similar en las versiones del problema Non-Blocking y GR(1). Con base en eso, hicimos dos observaciones importantes. La primera es que existen muchas transiciones (s, a) para las cuales, al obtener su feature vector $\phi(s, a)$, conseguíamos exactamente el mismo vector. De esta manera, estas transiciones eran indistinguibles para el agente. Una manera de solucionar esto sería agregar más features que nos permitan distinguir transiciones entre sí.

La segunda observación que hicimos es que la información que indica a cuál de las n entidades está haciendo referencia la transición nunca es pasada al agente. Estas etiquetas especiales tienen un número, al cual llamaremos *índice*, que refiere a cuál de las n entidades afecta la acción o cuál de los k pasos se realiza. Por ejemplo, en `DP`, existe la etiqueta `eat.i`, donde el índice i indica que el i-ésimo filósofo es el que come al realizar esta acción. Originalmente, la información referente a este índice i es ignorada en el feature vector $\phi(s, a)$, ya que se eliminan todos los índices. De esta manera, el agente desconoce estos índices, ignorando por completo cual es el filosofo que come al realizar esta acción.

Cuando se piensa en agregar esta información, podemos inclinarnos a considerar el directamente agregar el índice en el como un valor mas en el feature vector $\phi(s, a)$. Sin embargo, esto presenta dos grandes problemas. En primer lugar, $\phi(s, a)$ es un vector binario, mientras que el valor del índice podría tener múltiples valores posibles. En segundo lugar, el problema más fundamental surge durante el entrenamiento: dado que estamos trabajando con una instancia de tamaño pequeño, el agente no ve todo el rango de posibles índices. Sin embargo, al testear el agente en instancias de tamaño grande, sí puede observar transiciones con índices de cualquier valor, valores de los cuales el agente no estaba al tanto en el proceso de entrenamiento.

Por ejemplo, al entrenar el agente para resolver `DP`, solo puede ver transiciones que hagan referencia al filósofo 1 o al 2 (ya que se entrenó en `DP-2-2`). Sin embargo, luego utilizaremos un agente ya entrenado para resolver instancias más grandes como `DP-15-15`, en la que el agente tiene que decidir sobre 15 filósofos, cuando solo aprendió a distinguir entre 2. Lo que pasa es que al ser totalmente nueva esta información, el agente no sabe como lidiar con ella.

Por estos motivos, no podemos pasar directamente el índice al agente. Sin embargo, en este trabajo presentamos una manera de usar esta información de manera indirecta, combinándola con otros features particulares de las entidades. Supongamos que en DP queremos crear un feature que indique para cada uno de los n filósofos si ya ha comido o no. Inicialmente, no podemos hacerlo directamente ya que la cantidad de filósofos varía según la instancia a resolver. Lo que podemos hacer es tener un feature que se active cuando el filósofo al que hace referencia la acción a haya comido.

Esto nos lleva a un nuevo tipo de features, que hablan sobre una propiedad de la entidad a la que hace referencia la transición. Generalmente, estas propiedades son complicadas de generalizar, ya que son dependientes de las entidades individualmente, las cuales tienen un comportamiento que varia mucho de problema en problema. Es por este motivo que en este trabajo analizaremos cómo utilizar este nuevo tipo de features en cada una de las familias de instancias.

Por último, agregar estos features ayuda a solucionar el primer problema que presentamos en esta sección, ya que nos permite distinguir las transiciones que se mapean al mismo feature vector. De esta manera, al separarlas por la entidad a la que hacen referencia, reducimos estas colisiones.

4.3. Features Customs

Con los conceptos previamente establecidos, profundizaremos en los features adicionales de cada familia de instancias en particular, justificando la razón por la que creemos que estos pueden ayudar al agente a mejorar la exploración si son bien utilizados.

Además de los features que cada familia de instancias tendrá, agregaremos dos features extra a todas las familias. El primero indica si la transición tiene un índice referente a alguna de las n entidades y el segundo indica si la transición tiene un índice referente a uno de los k pasos de la instancia. Por como están especificadas las instancias, las acciones solo hablan de uno de los k pasos cuando tienen una entidad a la que hacer referencia, el segundo feature solo está activo si el primero lo está.

Esto nos lleva a distinguir tres tipos de transiciones: aquellas que no tienen índices, las que tienen solo uno (este hace referencia a una entidad) y las que tienen dos índices (estos hacen referencia a la entidad y al número de paso que hace esa entidad). Llamaremos i al índice al que hace referencia la transición (s, a). Supondremos que esta transición tiene un índice i y en caso de no tenerlo, estos features personalizados estarán desactivados.

Algo interesante a considerar sobre estos nuevos features customs, es que surgieron a partir de nuestro conocimiento y nuestra intuición sobre cada problema. De hecho, muchos de ellos fueron fuertemente inspirados al observar cómo RA resuelve eficientemente estos problemas. De esta manera se podría introducir un sesgo significativo, ya

que es posible que exista una solución mejor, diferente a RA, la cual desconocemos y que podríamos estar ignorando.

A continuación, exploraremos detalladamente qué features personalizados agregaremos a cada familia de instancia y por qué pensamos que son una buena adición.

4.3.1. Air-Traffic Management (AT)

En esta familia de instancias, debemos coordinar el aterrizaje de n aviones. Para indicar que un avión ha aterrizado, utilizamos la acción `land.i`, que representa el aterrizaje seguro del avión i. En primer lugar, proponemos agregar un feature que marque si el avión i-ésimo ya ha aterrizado. En segundo lugar, incluimos un feature que indica cuando el avión número i comienza a descender. Por último, añadimos un feature que indica si la transición deja al avión en la última de los k niveles, con el objetivo de señalar al agente qué transición permitirá al avión aterrizar primero.

4.3.2. Bidding Workflow (BW)

Esta familia de instancias tiene como entidades los n equipos, los cuales cada uno puede rechazar el documento hasta k veces o aceptarlo una única vez. Hemos visto que la mejor manera de resolver este problema es tomar un equipo, asignarle el documento y explorar los caminos en los que el equipo lo acepta y los caminos en los que lo rechaza. En caso de aceptarlo, podemos pasar al siguiente equipo; en caso de no aceptarlo, podemos seguir asignándoselo al mismo equipo.

La mayor dificultad de este problema radica en tener varios frentes de exploración causados por la necesidad de explorar tanto los escenarios de aceptación como los de rechazo, y saber cuál expandir. Para ayudar al agente en esta tarea, decidimos agregar un feature que indique si al equipo número i ya se le asignó el documento en algún momento, y otro feature que indique si el equipo i aún no ha aceptado el documento pero ya lo rechazó al menos una vez.

4.3.3. Cat and Mouse (CM)

En CM, las entidades son los n ratones y los n gatos. Dado que los movimientos de los gatos no son controlables, cuando les toca su turno, debemos explorar todas sus posibles movimientos, lo que limita nuestras opciones en esa etapa de la exploración. Esta dificultad es lo que hace que esta familia de instancias sea tan desafiante de resolver. Por otro lado, cuando es el turno de los ratones, cuyos movimientos son controlables, podemos expandir de manera más inteligente. La clave en esta etapa de exploración es expandir solo los movimientos que acerquen a los ratones a la zona segura, ignorando el resto. Ya que si permitimos que un ratón se quede en su lugar o retroceda (alejándose de la zona segura), esto permitiría que los gatos lleguen antes a la zona segura, lo que podría hacer imposible ganar en el futuro.

Por ese motivo, proponemos un nuevo feature para esta instancia: uno que, dado el índice i, se active cuando su acción acerque a un ratón a la zona segura central. Si esto no ocurre, sería una pérdida de tiempo hacer retroceder a un ratón. Nuestra esperanza es que el agente pueda utilizar la información de este nuevo feature para ignorar las transiciones que hagan retroceder a los ratones.

4.3.4. Dinning Philosophers (DP)

En este problema, observamos que la forma más óptima de resolverlo es expandir un filósofo a la vez. De esta manera, la clave es hacer que todos los filósofos piensen, luego seleccionar uno y hacer que tome los dos tenedores que necesita, realice los k pasos, coma y luego deje los tenedores para que otro filósofo pueda realizar ese mismo proceso. Esto lo repetimos hasta que todos los filósofos hayan comido.

Basándonos en esto, los features de este problema están pensados para orientar al agente a marcar al filósofo que se está explorando para seguir explorándolo completamente. Los dos features propuestos son: primero, marcar si el i-ésimo filósofo ya tomó un tenedor y segundo, marcar si el i-ésimo filósofo ya soltó ambos tenedores (esto solo puede hacerse después de comer). De esta manera, distinguimos los filósofos que nos comenzaron a ser explorados, de los que están parcialmente explorados, de los que ya fueron explorados.

Es importante destacar que esta idea de marcar el inicio y el final de la exploración de una entidad (en este caso un filósofo) es posible en esta familia de instancias, solamente porque los filósofos no interactúan entre sí cuando se los explora de esta manera. En otros casos donde las entidades tengan que sincronizarse entre sí, no es posible tomar esta aproximación al problema.

4.3.5. Travel Agency (TA)

Para esta familia de instancias, pensamos que la estrategia de resolución óptima es saber cuándo iniciar la query correspondiente al siguiente servicio. Se debe realizar la acción `query.i` cuando el servicio $i - 1$ ya haya aceptado su query, de manera de sincronizar los servicios. Es por este motivo que el primer feature que agregaremos será uno que indica cuando un servicio está en el estado listo para aceptar su query correspondiente. Esta idea se complementa con otro feature que indica si el servicio anterior (el número $i - 1$, en caso de que exista) también está en el estado listo para aceptar su propia query.

Finalmente, agregamos un feature que indica si un servicio ya fue adquirido (*purchase*) o ya fue rechazado (*fail*). Esto es con el objetivo de indicar si este servicio ya cumplió su misión.

4.3.6. Transfer Line (TL)

Esta familia de instancias tiene la particularidad de que el tamaño del controlador puede no crece si cambia el valor del parámetro k. Esto se debe a que este parámetro modifica la cantidad de elementos que puede almacenar cada buffer, sin embargo, como solo necesitamos que un elemento se transfiera a través de estos, no nos importa la capacidad de los mismos. Esta característica hace que, para las heurísticas, sea poco eficiente expandir múltiples veces la acción `get.i`. De hecho, RA resuelve de manera óptima este problema, explorando esta acción una vez para cada buffer.

Con esta idea en mente, los features que proponemos buscan ayudar al agente a que no expanda múltiples veces la acción `get` de un buffer dado. Además, agregamos un feature que indica cuándo ya se realizó un `get` sobre el último buffer, lo que indica que un elemento llegó a la Test Unit y, por consecuencia, se debería expandir un `accept`.

4.4. Cambios en el Entrenamiento

Otra mejora que propusimos fue cambiar el esquema de entrenamiento de aprendizaje por refuerzos (RL) con el objetivo de mejorar el rendimiento del agente aprendido. Inicialmente, nos preguntamos si entrenar con instancias más grandes ayudaría, ya que esto podría mejorar cómo el agente generaliza a instancias de mayor tamaño, las cuales se asemejan más a las instancias de entrenamiento. Aunque esto parecía una buena idea al principio, encontramos que la larga duración de los episodios en las instancias más grandes hacía difícil entrenar al agente en estas condiciones. De hecho, a partir de tamaños como $n = 5$ y $k = 5$, por cuestiones de tiempo, es impracticable realizar el entrenamiento esto ocurre en cada una de las familias de instancias que tenemos.

Por este motivo, intentamos otro proceso distinto de aprendizaje, llamado *Curriculum Learning*[22] [23]. Esta es una técnica originaria del área de *Machine Learning*, que consiste en ordenar los ejemplos de los que aprende una red neuronal de manera que primero observe los más “fáciles” y luego los más “difíciles”. Este método se inspira en cómo los humanos aprendemos, comenzando con ejemplos fáciles para entender las bases de la tarea a aprender y luego incrementando gradualmente la dificultad de los ejemplos para entender las partes mas complejas de la tarea. En nuestro caso, si este aumento de dificultad se hace de manera escalonada, la red neuronal puede aprender más rápido y esperamos que pueda aprender de todas las instancias.

Intentamos entrenar siguiendo tres posibles esquemas curriculares: el primero avanza de manera diagonal y entrena con las instancias `1-1`, `2-2`, `3-3` el segundo esquema avanza aumentando el parámetro n con las instancias `2-2`, `3-2`, `4-2` y el último esquema aumenta el parámetro k con las instancias `2-2`, `2-3`, `2-4`. Notemos que en los dos primeros esquemas curriculares el tamaño de la planta crece de manera exponencial, ya que aumentamos el parámetro n. Mientras que en el ultimo esquema curricular el tamaño de la planta crece de manera polinómica.

Usando estos distintos esquemas, no pudimos observar mejoras en ninguno de los casos. Pensamos que esto se debe a que no se cumple una de las principales condiciones para que la graduación de la dificultad ayude mejorar el entrenamiento del agente. Esta condición es que la dificultad de los casos a los que el agente es expuesto aumente de manera suave, es decir, que no haya grandes saltos de complejidad en los ejemplos con los que entrenamos. Sin embargo, creemos que esto puede no cumplirse por la simple razón de que, si bien las instancias crecen en dificultad de manera exponencial o polinómica, dependiendo del esquema curricular usado, la red neuronal está entrenada para diferenciar feature vectors "malos" de "buenos". No obstante, no queda claro que esta tarea se vuelva más "difícil" en instancias más grandes, ya que, aunque aumentemos el tamaño de la planta (y por lo tanto la instancia sea más difícil de resolver), los feature vectors que encuentra la red son exactamente los mismos y por lo tanto la dificultad de estimar su valor es igual de difícil. Esto pasa porque la red neuronal solo ve un feature vector individualmente y no toda la planta, la cual es la que determina la dificultad de la instancia. De esta manera concluimos que la dificultad de estimar el valor de feature vectors es independiente de la dificultad de resolver una instancia.

5. Evaluación

A continuación, vamos a comparar empíricamente las distintas heurísticas expuestas previamente. Todo el código con el cual fueron corridos estos experimentos es una extensión del proyecto MTSA[5] y esta publico en un repositorio de github[24]. La comparación la haremos sobre las 225 instancias de cada familia con un presupuesto de 15000 expansiones máximas por instancia. Esto significa que cada instancia individual debe ser resuelta en menos de 15000 expansiones, de lo contrario, diremos que no pudo ser resuelta. Esta es la única limitación que hay, ya que no tenemos restricciones de tiempo ni memoria. Cabe aclarar que todas las instancias podrían resolverse si no existieran restricciones en la cantidad de expansiones, tiempo ni memoria. Además, dado que ejecutar todas estas instancias demanda mucho tiempo de cómputo, asumiremos que si una instancia falla para $n = n_0$ y $k = k_0$, entonces fallará para todo $n = n_0$ y $k > k_0$.

En base a estas reglas, mediremos la cantidad de instancias resueltas para cada heurística y la cantidad total de expansiones realizadas. Llamaremos CRL a la heurística basada en aprendizaje por refuerzos con la extensión de los features personalizados, y RL a la que no tiene dicha extensión.

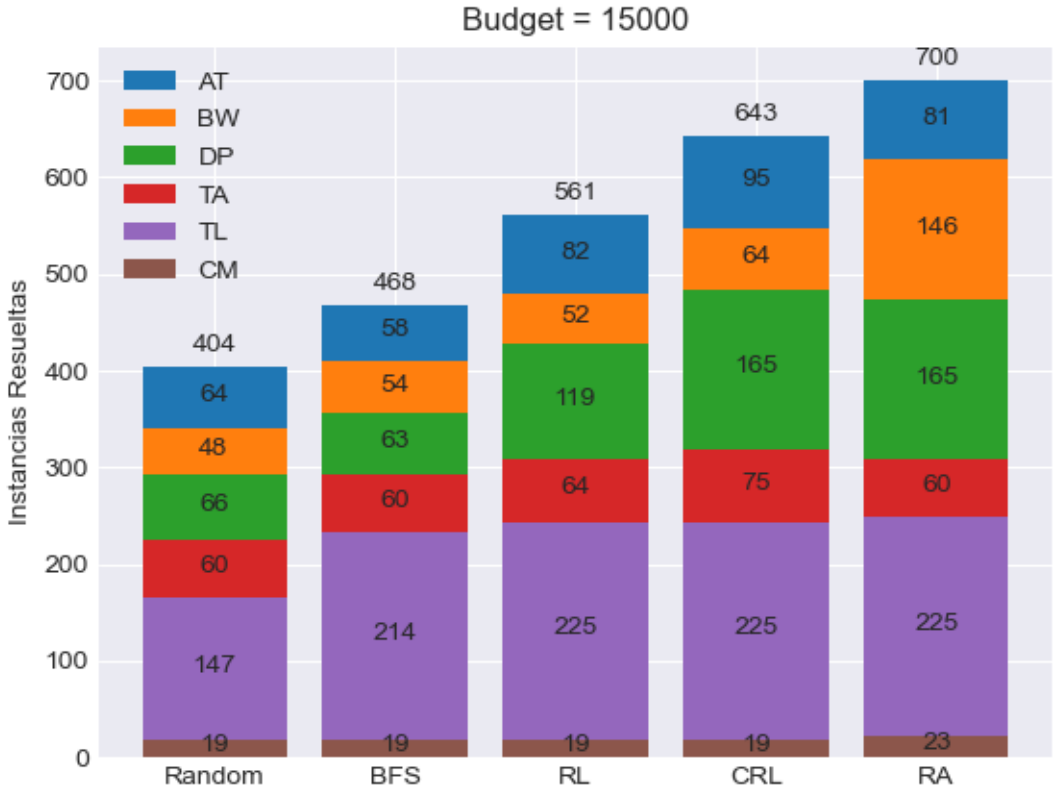

Figura 4: Cantidad de instancias resueltas para cada heurística usando un máximo de 15000 expansión por cada instancia. Arriba de cada barra se muestra el total de instancias resueltas.

La primera pregunta que queremos responder, es si la heurística RL es capaz de resolver el problema de síntesis en su versión GR(1), de manera más eficiente que heurísticas básicas como Random o BFS. Podemos observar que esto se logra, ya que ambas versiones de la heurística RL resuelven más instancias que las heurísticas simples. Esto ocurre en todas las familias de instancias, con una excepción: `CM`. En esta familia, salvo RA, todas las heurísticas resuelven la misma cantidad de instancias. Esto se debe a que, al ser una familia donde el tamaño de la planta crece tan rápido, resolver la siguiente instancia es exponencialmente más difícil. En las siguientes subsecciones profundizaremos más sobre este hecho.

5.1. Observando el impacto de agregar información de dominio

La segunda pregunta que nos interesa responder, es si agregar información específica del dominio puede ayudar a resolver de manera más óptima un problema. Para eso, ahora nos centraremos en comparar RL con CRL. Con esta idea en mente, observamos en la figura 4 que CRL claramente resuelve más instancias. Introduciendo estos features vemos que podemos resolver 82 instancias mas que cuando no le dábamos esta infracción extra a la heurística RL. Si analizamos cada familia de instancias por separado, vemos que CRL claramente ofrece una mejora en las familias `TA`, `AT`, `DP` y `BW`, mientras que en `TL` y `CM` resuelve la misma cantidad de instancias.

Comenzando por `TL`, esta familia es la más fácil de resolver, por lo que resulta relativamente sencillo resolver las 225 instancias. Esto se debe al crecimiento lineal del director con respecto a ambos parámetros, n y k. Es importante destacar que la velocidad de crecimiento del director es tan lenta que tampoco vale la pena aumentar el tamaño de las instancias para esta familia. Por ejemplo, `TL-15-15` puede ser resuelto en 36 expansiones, y hemos observado que `TL-100-100` puede ser resuelto en menos de 1000 expansiones. Tanto RL como CRL pueden resolver las 225 instancias, por lo que en ese aspecto no podemos observar ninguna diferencia. Sin embargo, al analizar la cantidad de expansiones realizadas, encontramos que CRL logra reducir la cantidad de expansiones necesarias a menos de la mitad. Además, hemos observado que CRL, para un valor fijo de n, resuelve todas las instancias en la misma cantidad de expansiones, ignorando por completo el valor de k. Esto tiene sentido ya que aumenta el valor de k, incrementa la capacidad de almacenamiento de todos los buffers, sin embargo solo necesitamos almacenar un elemento en cada buffer, haciendo que ese incremento en la capacidad no influya a la hora de resolver el problema.

`DP` es la familia de instancias que muestra más mejoras cuando se resuelve con la heurística CRL. Esto se refleja tanto en la cantidad de instancias resueltas como en la cantidad de expansiones realizadas. Al analizar detenidamente el agente obtenido por CRL, observamos que expande los filósofos en orden, lo cual era precisamente la idea que teníamos en mente para mejorar la resolución de este problema.

Por otro lado, en `TA` se observa una mejora en la cantidad de instancias resueltas. Es importante destacar que en esta familia de instancias cuando $n \geq 5$ se vuelve muy difícil de resolver el problema, debido al crecimiento exponencial del tamaño de la planta con respecto a n, esto no pasa cuando el parámetro k crece, ya que este hace que la planta crezca de manera lineal. Por ese mismo motivo, las instancias donde $n < 5$ sean sencillas de resolver para todo valor de k. Es por este motivo que todas las heurísticas pueden resolver las primeras 60 instancias de `TA`, sin embargo, es difícil resolver las siguientes instancias. A pesar de eso, se observa que ambas heurísticas basadas en RL pueden resolver más de esas 60 instancias iniciales, y utilizando features custom podemos llegar a las 15 instancias donde $n = 5$, cosa que no podríamos lograr sin estos features custom.

Las familias de instancias AT y BW son otros casos donde se ven un aumento de instancias resueltas cuando usamos CRL. Finalmente, CM es una familia muy particular para las heurísticas basadas en aprendizaje por refuerzos, ya que, a pesar de ser un problema simple de entender y de plantear una estrategia ganadora, el rápido crecimiento exponencial de su director hace que los episodios con los que entrenamos al agente sean mucho más largos que en otras familias de instancias. Mientras en otras familias la duración promedio de los episodios de entrenamiento estaba en las 80 expansiones, en CM la duración promedio es de 2000 expansiones, teniendo los episodios mas largos de todas la familias de instancias. Pensamos que la longitud de los episodios, junto con el hecho de que el reward buscado por el agente este al finalizar el episodio, hace que sea difícil realizar acciones buenas cuando se resuelve el problema con la política ϵ-greedy y por lo tanto el agente no puede mejorar. Creemos que por ese motivo el agente no aprende de manera eficaz y en ambas heurísticas basadas en RL tiene un desempeño muy similar a la heurística random.

Pensamos que como al agente le cuesta aprender en este ambiente, no importa mucho qué features usemos, ya que de todas formas no podrá encontrar una manera efectiva de usarlos. De hecho, agregar más features podría ser perjudicial, ya que estaríamos ampliando la información que le damos al agente. La idea de que no importa que features le demos al agente, se ve reflejada en los resultados, donde efectivamente RL y CRL resuelven la misma cantidad de instancias. Posiblemente necesitemos otro tipo de mejora en la heurística RL si queremos mejorar en esta familia de instancias.

De esta manera, si bien es difícil medir estas mejora de menera exacta en cada caso, hemos visto que con excepción de CM, todas las familias de instancias muestran una mejora cuando se amplía la información que contienen los features vectors con información especifica del dominio del problema.

5.2. Comparando la heurística RL con RA

Ahora analizaremos la diferencia en los resultados entre CRL y RA. Esto resulta interesante dado que RA es la heurística más óptima conocida hasta el momento y muchos de los features de CRL fueron diseñados observando el funcionamiento de RA. Para llevar a cabo esta comparación, examinaremos cada familia de instancias por separado. En este análisis, no solo nos importa quien resolvió mas instancias, si no que también queremos si en algún caso una heurística resuelve instancias distintas que la otra ya que al usar la información de la planta de maneras tan distintas, podría ocurrir el caso de que ambas resuelvan problemas de distintas características.

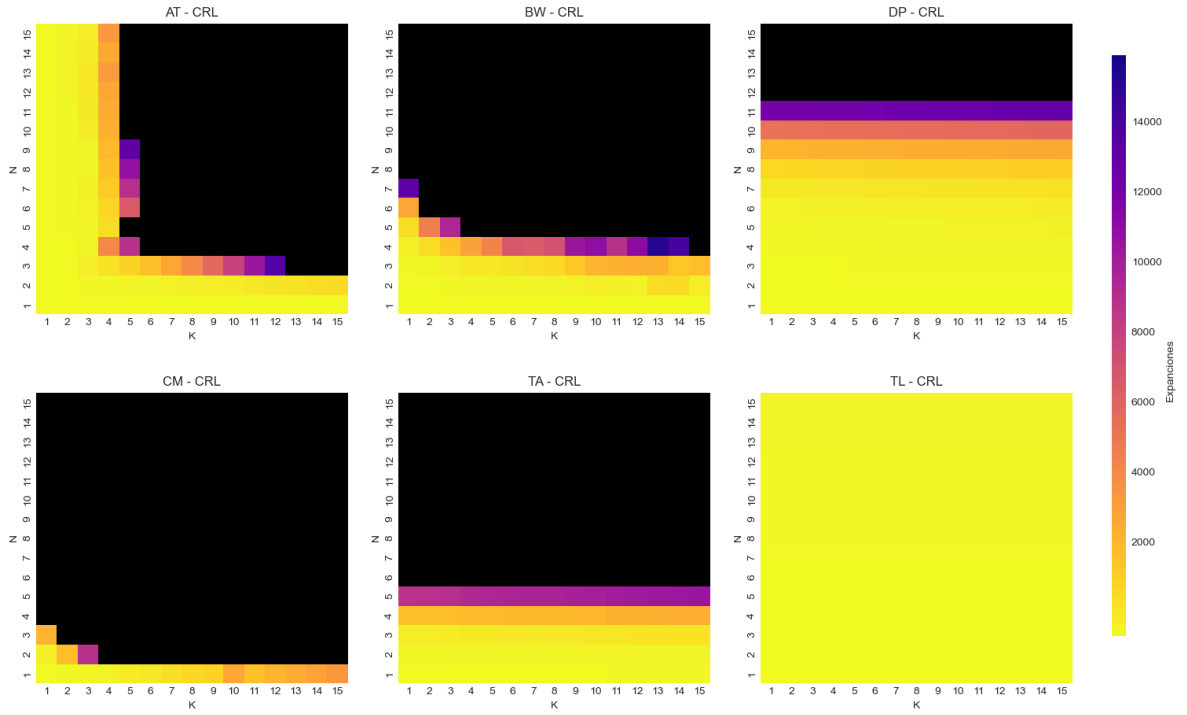

Figura 5: Instancias resueltas con la heurística CRL. Mientras mas oscuro, mas expansiones son necesarias para resolver una instancia. En negro las instancias que no pudieron ser resueltas en menos de 15000 expansiones.

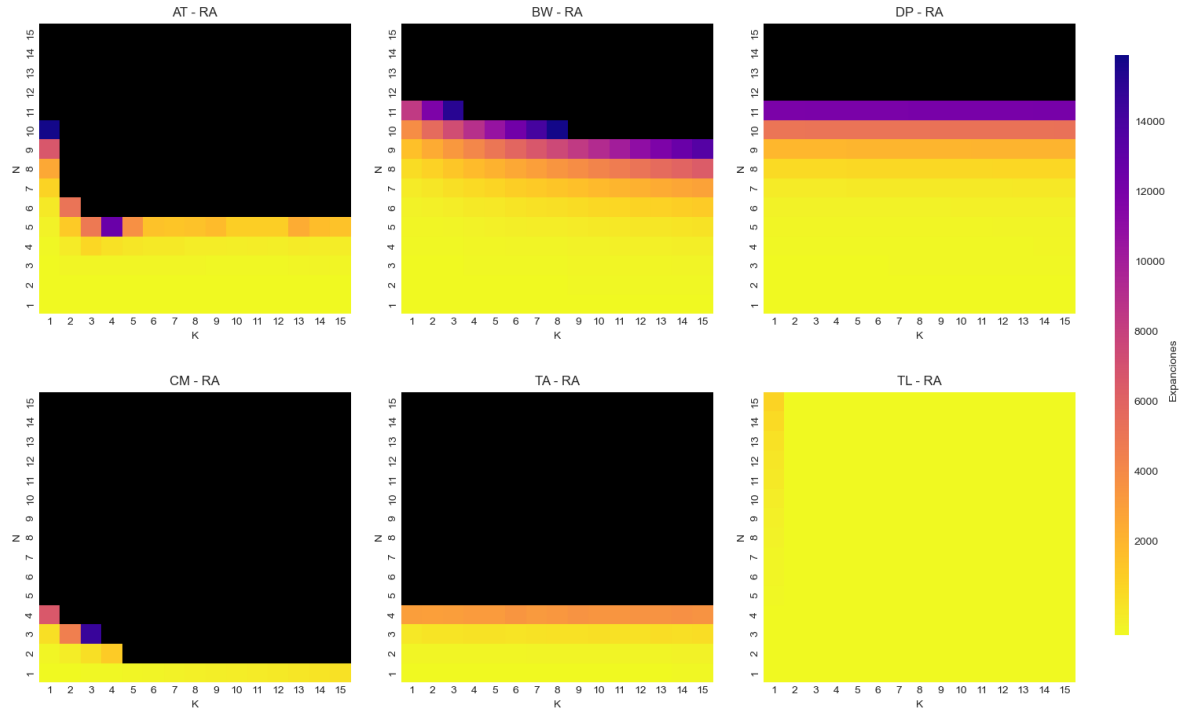

Figura 6: Instancias resueltas con la heurística RA. Mientras mas oscuro, mas expansiones son necesarias para resolver una instancia. En negro las instancias que no pudieron ser resueltas en menos de 15000 expansiones.

Observando la figura 4, lo primero que podemos ver, es que hay 2 familias en las que CRL resuelve mas instancias que RA (`TA` y `AT`), 2 familias donde CRL y RA resuelven la misma cantidad de instancias (`TL` y `DP`) y finalmente 2 familias donde RA resuelve mas instancias (`BW` y `CM`). Cada uno de estos casos se deben a múltiples motivos, a continuación expondremos y explicaremos estos motivos.

Primero, veamos el caso donde CRL resuelve más instancias que RA, lo cual ocurre en las familias `TA` y `AT`. En el caso de `TA`, como mencionamos anteriormente, el parámetro n es crucial para determinar la dificultad del problema. Observamos que RA no puede resolver ninguna instancia con $n \geq 5$, mientras que CRL puede resolver todas las instancias correspondientes con $n = 5$. Aunque esto pueda parecer poco significativo, dado el crecimiento exponencial, significa que estamos resolviendo instancias mucho más grandes que antes.

Por otro lado, `AT` es otro caso donde CRL resuelve más instancias que RA. Esto es notablemente interesante, ya que en las figuras 5 y 6, vemos que es el único caso donde las instancias resueltas por estas dos heurísticas son fundamentalmente distintas. Podemos observar que CRL es mejor para resolver instancias con un bajo valor de k pero cualquier valor de n, mientras que RA no puede resolver las instancias con un valor de n mayor a 5. Esto contrasta notablemente con el hecho de que si $n > k$, este problema

no tiene solución. Por lo tanto, vemos que CRL puede resolver fácilmente las instancias donde el problema no tiene un director solución, mientras que a RA le resulta mucho más complicado enfrentarse a ese tipo de escenario. Por el contrario, RA tiene más facilidad para resolver las instancias que sí tienen solución. Esto se debe a que RA busca cerrar ciclos, sin importar si la instancia es resoluble o no, mientras que RL tiene una aproximación más exploratoria. También, es importante aclara que el parámetro n hace que la planta crezca de manera exponencial, mientras que el parámetro k hace que crezca de manera lineal, por lo tanto las instancias que resuelve CRL tienen plantas mucho mas grandes y por ende mas difíciles que las instancias que resuelve RA.

Veamos los casos donde CRL y RA resuelven la misma cantidad de instancias, que son `TL` y `DP`. En el caso de `TL`, como ya hemos observado, es relativamente sencillo resolver las 225 instancias de este problema, este es el caso tanto para CRL como para RA. Respecto a `DP`, ocurre lo mismo, donde ambas heurísticas resuelven la misma cantidad de instancias. Como tenemos este empate, observamos la cantidad total de expansiones que ambas heurísticas realizan para resolver todas las instancias que estas resuelven. Sin embargo al hacer esto obtenemos resultados similares. Vemos que en ambos casos RA necesita menos expansiones que CRL, sin embargo la diferencia es pequeña, haciendo que no sea significativa.

Esta pequeña diferencia puede deberse a que CRL implementa una estrategia sintetizada a partir de su aprendizaje en una instancia en particular, lo que significa que pequeños cambios que no pudo generalizar adecuadamente podrían llevarlo a realizar algunas pocas expansiones innecesarias. Por otro lado, RA es más directo, ya que las estimaciones que hace priorizan el cierre de ciclos por encima de todo. También un detalle a aclarar, es que podemos observar como a RA le resulta difícil resolver `TL` cuando $k = 1$, mientras que CRL no tiene diferencia con respecto al parámetro k. Esto puede deberse a que en este caso las estimaciones para las acciones `get` no son tan precisas, haciendo que la heurística explore caminos innecesarios. Por ese motivo, la dificultad de RA para resolver `TL` con $k = 1$, podría hacer que CRL gane una ventaja sobre RA si agrandamos el benchmark a instancias mas grandes.

Por último, analizaremos por qué RA resuelve más instancias en `BW` y `CM`, lo que creemos para estos casos, es que se debe a razones diferentes en cada una de estas familias. En `CM`, observamos que tanto RL como CRL no pueden aprender una política óptima debido a la larga duración de los episodios, tal como remarcamos anteriormente. En contraste, RA no tiene este problema ya que no requiere un proceso de aprendizaje y no tiene repercusión la duración de los episodios, ya que esto no es algo influyente a la hora de hacer las estimaciones que hace RA. Por otro lado, en `BW`, si bien CRL puede aprender una política buena, la solución de este tipo de problema es muy grande y debe ser muy precisa. Si la heurística comete algunos errores y expande unas pocas transiciones incorrectas, entonces el único director posible a encontrar termina siendo de un tamaño similar al de la planta entera. De esta manera, el equivocarse se penaliza mucho, lo que hace que las heurísticas como CRL, que pueden tener dificultades para

generalizar de manera muy precisa, no sean óptimas para este tipo de problemas. En contraposición, RA es una heurística que va directamente a cerrar ciclos y resulta ser más precisa cuando esta estrategia es ganadora. Finalmente, es importante aclarar que la heurística de RA mejoro mucho en `BW` cuando introducimos las mejoras en el analizáis de la estructura de la planta propuestas por Pazos. Sin esta mejora en particular, RA tenia un desempeño peor que CRL.

6. Conclusiones

En este trabajo buscamos entender el impacto, sobre la heurística basada en RL, de agregar features nuevas con información específica sobre el problema de control GR(1) que se desea resolver. Para eso adaptamos la implementación Non-Blocking de dicha heurística para la versión GR(1) del problema y pensamos nuevas maneras de abstraer la información de los estados de la exploración parcial para ser consumidos por un agente de aprendizaje por refuerzos.

Los nuevos features usan información particular los índices de las acciones, índices que antes eran ignorados por completo y que por lo que observamos, juegan un rol fundamental a la hora de decidir la mejor transición a expandir. Con esta novedosa manera de abstraer información, diseñamos nuevos features para cada familia de instancias del benchmark seleccionado.

Analizando los resultados, vimos que en todos los casos se observa una mejora al usar estos nuevos features vectors, salvo por el caso particular de `CM`. En este caso en especial, pensamos que esto se debe a la extensión de los episodios, lo que impide que el agente pueda aprender la heurística de forma correcta, de hecho, vemos que este problema también estaba presente en la heurística basada en RL original. Comparando la heurística con custom features contra RA, vimos que de las 6 familias de instancias, tenemos dos casos donde CRL supera a RA, dos casos donde se resuelven la misma cantidad de instancias y dos donde RA resuelve más instancias. El caso más interesante fue el de `AT`, donde las instancias que resuelven estas dos heurísticas son fundamentalmente distintas. En `AT` observamos que CRL es mejor para resolver instancias donde no existe director posible que controle la planta, mientras que en `CM` vimos que RA es mejor para resolver instancias donde los episodios de entrenamiento tienen una larga duración. Además, en instancias como `DP` pudimos mejorar el rendimiento de la heurística RL, empatando la cantidad de instancias resueltas de RA. Con estos resultados podemos concluir que estos nuevos features mejoran el desempeño de la heurística basada en RL, sin embargo es importante notar que para llegar a ellos necesitamos entender muy bien cada dominio de cada problema, algo que no siempre podría ser sencillo de hacer.

Por otro lado, no pudimos obtener ningún resultado positivos al entrenar en instancias más grandes o al implementar un esquema curricular de entrenamiento. Creemos que esto se puede deber a la larga duración de las instancias más grandes y que la frontera de expansión crece y por lo tanto crece la cantidad de acciones por las que debe decidir el agente, haciendo una tarea difícil el encontrar buenas expansiones de las que pueda aprender el agente o bien, a que no esta bien escalonada la dificultad de las muestras de entrenamiento en el caso de curriculum learning. Esto, junto con que los tiempos de entrenamiento se alargan por estar resolviendo instancias grandes, nos indican que no es recomendable usar estas modificaciones en el proceso de entrenamiento, por lo menos con el algoritmo de DQN tal como lo planteamos en este trabajo, bajo los ambientes que usamos.

Vemos que la duración de un episodio es un problema recurrente en varias propuestas de mejora. Por lo tanto, pensamos que es un buen punto a mejorar en futuros trabajos. Como ya mencionamos antes, esto podría ser de gran ayuda para resolver CM de manera eficiente o para sacar mayor provecho de técnicas como curriculum learning. Para esto, una posible idea es cambiar la función de recompensa por una función que intente medir de manera aproximada qué tan lejos está el agente de terminar el episodio. Esto nos permite truncar la duración de los episodios de entrenamiento a lapsos más manejables para el agente.

También se podría cambiar la función de recompensa en base a un director ya computado previamente, de manera que la recompensa sea mayor cuando el agente expanda transiciones pertenecientes a ese director. Si bien esto tiene el problema de que el director de referencia podría no ser minimal, esto nos da un punto de partida para mejorar la función de recompensa y por ende, mejorar el proceso de entrenamiento.

Otra línea a futuro que podría ser interesante explorar, es la de extender las familias de instancias del benchmark usado en este trabajo, con el objetivo de aprender cómo la heurística basada en RL trabaja bajo diferentes contextos. Por ejemplo, sería muy interesante observar qué pasa si se entrena RL en una instancia no resoluble u observar cómo trabajan todas las heurísticas en problemas con distinta estructura.

Otra razón por la que esta línea es interesante es porque creemos que algunas de las familias de instancias que ya tenemos, no tienen mucho margen de mejora con el budget que manejamos o bien ya podemos resolverlas fácilmente. Este es el caso de TL la cual es una familia de instancia fácil de resolver y al tener un crecimiento lineal en su controlador no sirve de mucho aumentar los valores de n y k. Otras familias de instancias como DP y CM, si bien no se pueden resolver todas las 225 instancias, sabemos que superar el desempeño de RA es muy difícil debido al crecimiento exponencial de estas instancias. Por ese motivo, sería útil llegar a una cota teórica de la cantidad de expansiones necesarias para resolver cada una de estas instancias y de esta manera, saber qué tan lejos estamos de una heurística perfecta en cada uno de estos casos.

Otra incorporación prometedora podría ser añadir fases a la exploración que realiza la heurística RL. Esta idea surge de la tesis de Gagliardi [25], donde propone separar la exploración que hace RA en cuatro etapas: encontrar un estado marcado, cerrar un ciclo, garantizar la controlabilidad del ciclo y propagar el estado de *Goal*. Esta metodología mejora la heurística RA en ciertos contextos, lo que nos lleva a pensar que se podría mejorar la heurística RL entrenando modelos específicos para resolver cada una de estas etapas o bien indicándole al agente en que etapa se encuentra de la exploración. También existe la posibilidad de que a la heurística RL le resulte más conveniente definir otras fases de la exploración, ya que no cuenta con la misma información de entrada que RA.

En este trabajo expusimos una serie de features diseñados a medida para cada familia de instancias del benchmark. Para esto, necesitamos estudiar individualmente cada una de estas familias y analizar, en cada caso, qué hace a una buena estrategia de exploración. Al hacer esto, encontramos que algunas familias de instancias se parecen en ciertos aspectos. Por ejemplo, la estrategia que creemos óptima para DP y para BW es similar en ambos casos: explorar por completo una entidad y luego pasar a la siguiente, así hasta explorar todas las entidades. Es por eso que algunos de estos features propuestos en este trabajo podrían generalizarse para algunas familias de instancias. También se podrían diseñar nuevos features con la idea de que sean generales para cualquier tipo de instancia del benchmark.

Por último, existen trabajos en el área de *general planning*[26], un área muy cercana a la síntesis de controladores, en donde también se utiliza aprendizaje por refuerzos para resolver problemas similares y se proponen ideas parecidas a las expuestas en este trabajo con resultados muy prometedores. Si bien no son exactamente iguales, se puede hacer un paralelismo entre algunos de los features que nosotros propusimos y lo que llaman *roles*. Es interesante explorar las alternativas propuestas e intentar adaptarlas a nuestros problemas.

Referencias

[1] P. J. Ramadge y W. M. Wonham. «Supervisory Control of a Class of Discrete Event Processes.» En: (1987).

[2] A. Pnueli y R. Rosner. «On the Synthesis of a Reactive Module. In Proc. of the Symp. on Principles of Programming Languages». En: (1989).

[3] Dana Nau Malik Ghallab y Paolo Traverso. «Automated Planning: Theory Practice.» En: (2004).

[4] Ciolek D. Duran M. Zanollo F. Pazos N. Braier J. Braberman V. D'Ippolito N. y Uchitel S. «On-the-fly informed search of non-blocking directed controllers». En: (2023).

[5] https://mtsa.dc.uba.ar/.

[6] Tomas Delgado. «Exploration Policies for On-the-Fly Controller Synthesis: A Reinforcement Learning Approach». En: (2023).

[7] Dimitra Giannakopoulou Jeff Magee. «Fluent Model Checking for Event-based Systems». En: (2004).

[8] Roderick Bloema Barbara Jobstmann Nir Pitermanc Amir Pnueli Yaniv Sa'ar. «Synthesis of Reactive(1) designs». En: (2011).

[9] Nicolas D'ippolito Victor Braberman Nir Piterman y Sebastian Uchitel. «The modal transition system control problem.» En: (2012).

[10] Florencia Zanollo y Matias Duran. En: (2021).

[11] Nicolas Pazos Mendez. «Ready Abstraction: A heuristic for the on-the-fly synthesis technique of non-blocking directors». En: (2023).

[12] Sutton y Barto. «Reinforcement learning: An introduction.» En: (2018).

[13] Long-Ji Lin. «Reinforcement learning for robots using neural networks.» En: (1992).

[14] Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller. «Playing Atari with Deep Reinforcement Learning». En: (2013).

[15] J. Magee y J. Kramer. «Concurrency: State Models and Java Programs». En: (1999).

[16] F. T. Durso y C. A. Manning. «Air traffic control.» En: (2008).

[17] R. K. L. Ko. «A Computer Scientist's Introductory Guide to Business Process Management (BPM).» En: (2009).

[18] P. J. Ramadge y W. M. Wonham. «Supervisory Control of a Class of Discrete Event Processes.» En: (1987).

[19] D. Lehmann y M. O. Rabin. «On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem.» En: (1981).

[20] B. Srivastava y J. Koehler. «Web service composition-current solutions and open problems». En: (2003).

[21] W. M. Wonham. «Notes on Control of Discrete-Event Systems. Dep. of Electrical and Comp. Engineering». En: (1999).

[22] Yoshua Bengio - Jerome Louradour - Ronan Collobert - Jason Weston. «Curriculum Learning.» En: (2009).

[23] Petru Soviany - Radu Tudor Ionescu - Paolo Rota - Nicu Sebe. «Curriculum Learning: A Survey.» En: (2021).
[24] https://github.com/darioturco/mtsa.
[25] Sebastian Uchitel Sebastian Zudaire Hernan Gagliardi. «Definici´on y an´alisis de etapas de exploraci´on on-the-fly para s´ıntesis de controladores del tipo non-blocking». En: (2022).
[26] Siddharth Srivastava Rushang Karia. «Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning.» En: (2021).