Universidad ORT Uruguay
Facultad de Ingeniería

Mejora del tiempo de ejecución de algoritmos de inferencia regular

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Alejandro Rodríguez Reche - 223303

Tutores: Franz Mayr, Sergio Yovine

2021

Declaración de Autoría

Yo, Alejandro Rodríguez Reche, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto Final de Ingeniería en Sistemas;

- Cuando he consultado el trabajo publicado por otros, lo he atribuído con claridad;

- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;

- En la obra, he acusado recibo de las ayudas recibidas;

- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente que fue contribuido por otros, y qué fue contribuído por mí;

- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Alejandro Rodríguez Reche

30-09-2021

Dedicado con mucho amor a mi familia y amigos, quienes fueron, son y serán siempre mi espada y mi escudo. Los de acá, y los de Arriba. No hubiera podido llegar hasta acá sin ustedes. Infinitamente GRACIAS.

Agradezco enormemente también a mis tutores, Mág. Franz Mayr y Dr. Sergio Yovine, por el constante e implacable apoyo en este desafío, en este contexto de pandemia mundial.

Abstract

El presente trabajo se centra en la mejora de los tiempos de ejecución del algoritmo de *machine learning L** de Angluin.

Se implementan distintas variantes del algoritmo, que involucran la búsqueda de la mejora de los tiempos en distintas etapas de la ejecución. También se evalúan estas variantes en el contexto de redes neuronales, tomando como referencia el algoritmo *Bounded-L** propuesto por Franz Mayr para el empleo de *L** en este contexto. Como resultado, se propone un algoritmo cuyos tiempos de ejecución en los casos de referencia de redes demuestran ser consistentemente mejores que los de *Bounded-L** para casos de estudio relevantes.

Se realiza un estudio sobre tecnologías de paralelización, distribución y optimización de código Python, en pos de evaluar técnicas candidatas para su aplicación en este algoritmo dentro del *framework* utilizado.

Palabras clave

Inteligencia Artificial; Redes neuronales recurrentes; Lenguajes regulares; Autómata finito determinista; Aprendizaje aproximadamente correcto, Inferencia regular; Explicabilidad.

Índice general

1 Introducción 8

2 Preliminares 10
- 2.1 Lenguajes Regulares 10
- 2.2 *L** 12
  - 2.2.1 Procedimiento de L* 14
  - 2.2.2 Ejemplo de ejecución 14
- 2.3 Probably Approximately Correct Learning 18
  - 2.3.1 *L** basado en PAC 19
- 2.4 Redes Neuronales Recurrentes 19
  - 2.4.1 PAC-Learning para RNNs 20
  - 2.4.2 *Bounded-L** 20
- 2.5 Planteamiento del problema 21

3 Propuesta de solucion 22
- 3.1 *L*Col* 23
  - 3.1.1 Ejemplo de ejecución 23
  - 3.1.2 Implementación de *L*Col* y primeros experimentos 25
- 3.2 Análisis de oportunidades de mejora 27
  - 3.2.1 Evaluación de oportunidades de paralelismo 27
  - 3.2.2 Elección de primera opción de paralelismo a implementar 29
  - 3.2.3 Experimentos con técnica Abbadingo One 29
  - 3.2.4 *BatchMQ*: Próximos pasos. 31
- 3.3 *Bounded-L*Col BatchMQ* 32
  - 3.3.1 Algoritmo 32

4 Experimentos realizados 34
- 4.1 Abbadingo One 34
- 4.2 Dificultades durante la ejecución 35
- 4.3 Problemas de implementación 36
- 4.4 Nuevos casos de experimentación 40
- 4.5 Resultados obtenidos 41

5 Investigación de posibles soluciones de paralelismo 45

5.1 Análisis de tecnologías de paralelismo en Python 45
5.2 *BatchMQ* y Ray: Primeros experimentos 46

6 Lecciones aprendidas 48
6.1 Abbadingo One: Limitaciones 48
6.2 Limitaciones de las tecnologías de
Python 49

7 Conclusiones 52
7.1 Trabajo a futuro 53

8 Referencias Bibliográficas 54

9 Anexos 57
9.1 Anexo 1: Análisis de tecnologías de paralelismo en Python 57
9.1.1 *Multithreading* 57
9.1.2 *Multiprocessing* 58
9.1.3 Ray 59
9.1.4 Dask 59
9.1.5 Modin 60
9.1.6 Cython 60
9.1.7 Algunas interrogantes 61
9.1.8 Pasos a seguir: Sugerencia 62
9.2 Anexo 2: Experimentos realizados con *Bounded-L*Col BatchMQ* 63
9.2.1 Experimento 1 63
9.2.2 Experimento 2 65
9.2.3 Experimento 3 67
9.2.4 Experimento 4 70

1 Introducción

Este proyecto se inserta en el marco de las actividades de investigación de la cátedra de Inteligencia Artificial y Big Data. Tiene como punto de partida los resultados publicados en [1, 2]. Es pertinente entonces, describir brevemente ese trabajo en la medida que define el marco conceptual de este proyecto.

El área abarcada es la de inteligencia artificial explicable [3], con foco en las redes neuronales como modelos predictivos. Si bien las redes neuronales gozan de un gran potencial para realizar predicciones acertadas sobre problemas complejos, como por ejemplo clasificación de imágenes, su forma de funcionar es poco trans- parente en el sentido de que no son amenas a proveer explicaciones humanamente comprensibles de las razones por las cuales llegan a una determinada conclusión [4].

En este contexto, el trabajo de la cátedra se ubica en el marco del uso de redes neuronales recurrentes (RNN) para el análisis de secuencias (*logs* y trazas de ejecu- ción de software, comportamientos de individuos, clasificación de textos, análisis de sentimiento, entre otros). En este caso, se plantea un problema de *language membership*. Una vez entrenada con un conjunto de secuencias, la red neuronal “aprende” un lenguaje y es capaz de discernir entre las palabras que pertenecen a ese lenguaje y las que no. A partir de esto, se pretende dar explicación a las respuestas de la red planteándose como pregunta: ¿Cuál es el lenguaje aprendido por la red?

Para lograrlo, la idea es plantear el interrogante como un problema de apren- dizaje de tipo “caja negra” con el objetivo de inferir del comportamiento de la red un autómata que caracterice de la mejor manera posible el lenguaje de la red. Para ello se hace uso del *framework* **PAC** (Probably Approximately Correct) **Learning** en un contexto de aprendizaje activo, extendiendo apropiadamente el algoritmo $L*$ de Angluin [5]. La base de este último es aprender interrogando a un Teacher que tiene conocimiento de la red neuronal mediante preguntas de membresía (una secuencia dada, ¿pertenece al lenguaje definido por la red?) y de equivalencia me-

dida por un test estadístico. El algoritmo desarrollado, llamado *Bounded L** [1], es capaz de extraer un autómata finito determinista, que reconoce un lenguaje que se aproxima al de la red neuronal recurrente con una confianza dada.

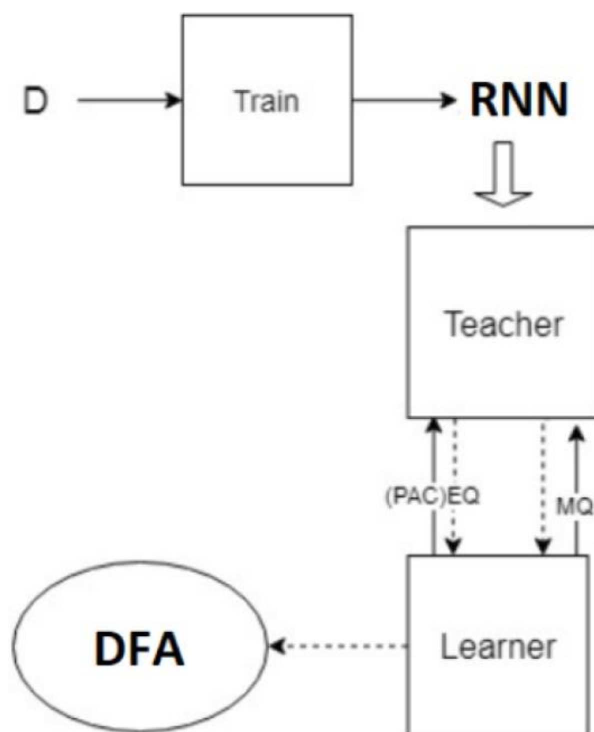


Figura 1.1: Estructura del framework de la Cátedra.

En la cátedra se está desarrollando un *framework* que permite integrar diferentes algoritmos para la explicabilidad de RNNs basados en el principio de aprendizaje activo en un contexto *"black-box"*. La aplicación de estos algoritmos en la práctica sugiere que existen oportunidades para mejorar la *performance* de los mismos [6]. El propósito de este proyecto de tesis de ingeniería es explorar y explotar estas oportunidades. En particular, a través de la optimización, uso adecuado de estructuras de datos, y paralelización.

2 Preliminares

2.1 Lenguajes Regulares

Los lenguajes regulares son aquellos que pueden representarse mediante un autómata finito determinista o DFA [7]. Un DFA se define como una tupla

$$M : (Q, \Sigma, \delta, q_0, F)$$

donde:

- Q representa el conjunto de estados.
- Σ representa el alfabeto.
- δ es la función de transición entre los estados.
- q_0 es el estado inicial del autómata.
- F es el conjunto de estados finales del mismo.

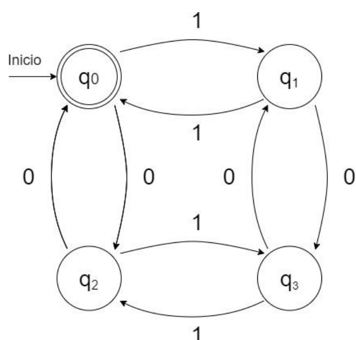
Un ejemplo de lenguaje regular sencillo [6] es el lenguaje definido como:

$$L = \{w \mid w \text{ tiene un número par de 0's y un número par de 1's}\}$$

donde:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- δ la función de transición representada gráficamente en la figura 1.2.

- q_0 es el estado inicial, indicado con una flecha entrante.
- $F = \{q_0\}$ es el conjunto de estados finales, conteniendo únicamente a q_0, indicado con una doble circunferencia.



(a) Representación gráfica.

	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

(b) Tabla de transición asociada.

Figura 2.1: Ejemplo de autómata.

2.2 L*

Tal como propone Dana Angluin [6], el algoritmo *L** consiste en el aprendizaje de un lenguaje regular mediante la interacción con un *Minimum Adequate Teacher* (MAT), quien expone una interfaz con dos operaciones: una de membres´ıa, y otra de equivalencia. El algoritmo, al finalizar su ejecución, devuelve el DFA mínimo que representa el lenguaje aprendido.

Las operaciones de la interfaz provista por el Teacher son las siguientes:

Membership Query (MQ): Dada una secuencia finita de símbolos, el Teacher devuelve *True* si la secuencia pertenece al lenguaje, y *False* en casocontrario.

Equivalence Query (EQ): Dado el lenguaje inferido hasta el momento, devuelve *True* si el mismo es equivalente al lenguaje objetivo, y devuelve un contraejemplo en caso contrario. Siendo un **contraejemplo** una secuencia que pertenece al lenguaje objetivo, pero no al lenguaje inferido, o viceversa.

La idea general del algoritmo es ir construyendo el DFA mencionado de manera iterativa mediante interacciones con el Teacher a través de sus operaciones.
Para ello se construye una *Observation Table* (también llamada **OT**), que consiste en una tabla que almacena la información de las consultas realizadas hasta el momento y habilita a construir un DFA a partir de ella.

Dentro de la Observation Table se puede ver la siguiente información, como se expone en [7]:

- Un conjunto finito no vacío de secuencias **cerradas por prefijo *(prefix- closed)***, que implica que cada prefijo de cada miembro también pertenece al conjunto de prefijos.

- Análogamente para sufijos, un conjunto finito no vacío de secuencias **cerra- das por sufijo *(suffix-closed)***, donde cada sufijo de sus miembros también pertenece al conjunto de sufijos.

- Una función que a cada secuencia le asigna el valor 1 o 0 dependiendo de si pertenece o no al lenguaje objetivo.

Para representar esta información, la Observation Table se estructura de la siguiente manera:

- El conjunto superior de filas, a quien Colin de la Higuera le llama **RED** [6], que contiene el primer conjunto cerrado por prefijo mencionado anteriormente.
- El conjunto inferior de filas, análogamente llamado **BLUE**, que contiene la unión de este conjunto de elementos con los símbolos del alfabeto del lenguaje.
- Las columnas representan el conjunto cerrado por sufijo.
- Cada celda representa la función de mapeo mencionada, que indica la pertenencia o no pertenencia de la secuencia al lenguaje.

Para que la información provista por la **OT** permita construir un DFA, las propiedades que se busca que se cumplan en ella son:

- **Clausura**: Una tabla es **cerrada** si, para cada fila perteneciente a **BLUE**, existe una fila igual en el conjunto **RED**.
- **Consistencia**: Una tabla es **consistente** si, para cada fila perteneciente a **RED**, para cada símbolo $e \in \Sigma$, si $OT[v] = OT[w]$ entonces $OT[v.e] = OT[w.e]$.
- **Completitud**: Una tabla es **completa** si no contiene agujeros, es decir, si cada una de sus celdas es no vacía.

Dicho esto, la idea es construir una **OT** que sea consistente, cerrada y completa, para poder así construir una hipótesis (DFA). Ésta se le envía al Teacher como *equivalence query*. En caso de equivalencia, el algoritmo termina, pues encontró un DFA equivalente al lenguaje objetivo, y en caso contrario utiliza el contraejemplo devuelto por el Teacher para actualizar la información existente en la OT.

Luego de actualizar la OT, se realizan *membership queries* para que la tabla esté cerrada y completa, y el proceso se repite hasta que la respuesta de la *equivalence query* sea positiva.

2.2.1 Procedimiento de L*

Antes de comenzar el proceso iterativo, la OT se inicializa añadiendo una fila al conjunto **RED** para la palabra vacía λ, y una fila al conjunto **BLUE** para cada $e \in \Sigma$.

En el primer paso de la iteración, se chequea si la tabla está cerrada. Si no lo está, el algoritmo toma una fila de **BLUE** que no exista en **RED** y la mueve a este conjunto, y asimismo agrega a **BLUE** filas conformadas por la secuencia encontrada concatenada con cada símbolo $e \in \Sigma$.

El siguiente paso es asegurar la consistencia de la tabla. Para lograr esto [7], el algoritmo expande el conjunto de sufijos original con la letra que hace que sus extensiones correspondientes sean diferentes, es decir, un elemento $e \in \Sigma$ tal que $OT[v] = OT[w]$ *pero* $OT[v.e] = OT[w.e]$. Esto permite diferenciar las dos palabras que tenían el mismo valor de fila.

Finalmente, luego que la **OT** cumple ambas propiedades, se procede como se comentó inicialmente: Se construye el DFA que representa el lenguaje inferido hasta el momento, y se le pregunta al Teacher mediante una EQ si este lenguaje es equivalente al lenguaje objetivo. En caso afirmativo, la ejecución termina y devuelve el DFA en cuestión. De lo contrario, el Teacher devuelve una secuencia de contraejemplo, y ésta se utiliza para actualizar la **OT**. Este procedimiento se realiza agregando cada prefijo del contraejemplo a **RED** y, para cada uno de estos prefijos, se agrega a **BLUE** su concatenación con cada símbolo $e \in \Sigma$ (dado que la concatenación no es un prefijo). Hecho esto, se pasa a una nueva iteración.

2.2.2 Ejemplo de ejecución

Un caso de ejemplo sencillo de la ejecución de algoritmo es el que propone De la Higuera en [6].

Se comienza con la tabla vacía, siendo RED = λ y EXP = λ, donde EXP representa el conjunto de experimentos que serán utilizados para la corrida del algoritmo.

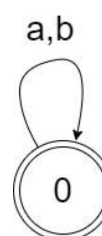
Se realiza una primera *membership query* con la secuencia vacía λ. La respuesta es *True*, luego *a* y *b* se agregan a BLUE, y se realizan nuevamente las *membership queries* *a* y *b*.

Supóngase que la respuesta es nuevamente *True*. La tabla correspondiente es cerrada, completa y consistente, por lo que se puede construir un autómata a partir de ella para realizar una *equivalence query*.

Para poder construir un automata a partir de la tabla [7], los estados son representados por cada fila única en **RED**. Los estados finales son aquellos correspondientes a las filas v donde $OT[v][\lambda] = 1$, mientras que los estados de rechazo son aquellos donde para cada fila w se cumple que $OT[w][\lambda] = 0$. Por último, la función de transición se define como $\delta(q_v, a) = w$ si $OT[va] = OT[w]$. El autómata correspondiente a la tabla de la figura 2.2a se muestra en la figura 2.2b.

	λ
λ	1
a	1
b	1

(a) Una tabla consistente.



(b) Autómata asociado a la tabla (a).

Figura 2.2: Consistencia.

Ahora, supóngase que el Teacher retorna el contraejemplo *abb*. La tabla se actualiza a la de la figura 2.3a. La tabla es cerrada, entonces los agujeros se llenan a través de *membership queries*, produciendo la tabla 2.3b.

Pero ahora la tabla no es consistente, puesto que $OT[a] = OT[ab]$ pero $OT[ab] = OT[abb]$. Esto se debe al experimento b, por lo que se agrega a EXP y la tabla en la figura 2.3c se debe completar. Se obtiene la tabla de la figura 2.3d que es cerrada, completa y consistente, y puede traducirse en un DFA a ser enviado mediante una *equivalence query*.

	λ
λ	1
a	1
ab	
abb	0
b	1
aa	
aba	
abba	
abbb	

(a) Tabla después de la **EQ**, retornando *abb* (que no está en el lenguaje).

	λ
λ	1
a	1
ab	1
abb	0
b	1
aa	1
aba	1
abba	0
abbb	0

(b) Se realizan **MQ**s: La tabla no es consistente.

	λ	b
λ	1	1
a	1	1
ab	1	0
abb	0	0
b	1	
aa	1	
aba	1	
abba	0	
abbb	0	

(c) Se agrega una columna con el experimento ”b”.

	λ	b
λ	1	1
a	1	1
ab	1	0
abb	0	0
b	1	0
aa	1	1
aba	1	1
abba	0	0
abbb	0	0

(d) Luego de llenar los agujeros, la tabla es cerrada y consistente.

Figura 2.3: Ejecución de $L*$.

Suponiendo que aquí la **EQ** dé *True*, el algoritmo termina, resultando el siguiente autómata:

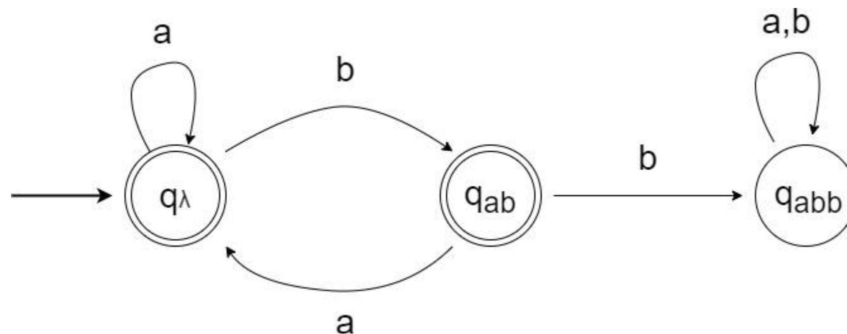


Figura 2.4: DFA mínimo luego de finalizar la ejecución de $L*$.

2.3 Probably Approximately Correct Learning

Tal como se expone en [7], hasta este momento se vio que $L*$ tiene la manera de comparar el DFA que representa al lenguaje inferido contra el lenguaje objetivo. Sin embargo, esto no siempre sucede así. Existen ocasiones donde no se puede acceder a una comparación de modelos deterministas y equivalencias binarias entre los mismos (son equivalentes, o no lo son). Por lo cual, para este tipo de casos, es necesario otro tipo de criterio de comparación para lograr el mismo objetivo.

Uno de los criterios que se utilizan en estos casos es el de **aprendizaje probablemente aproximadamente correcto** o **PAC** por su sigla en inglés. A grandes rasgos, este mecanismo consiste en la comparación de los dos lenguajes (el inferido y el objetivo) mediante un test estadístico, y poder determinar el resultado de su equivalencia según si la probabilidad de la misma pertenece o no a un determinado intervalo de confianza.

Para una mejor comprensión acerca de como trabaja un algoritmo en este contexto, es pertinente mencionar algunas definiciones:

- La **diferencia simétrica** $L_1 \oplus L_2$ entre dos lenguajes L_1 y L_2 es el conjunto de secuencias que pertenecen solamente a uno de los dos lenguajes.
- El **error de predicción** de L_1 con respecto a L_2 es la probabilidad de que una secuencia pertenezca a su diferencia simétrica.
- Dado $\epsilon \in (0,1)$, se dice que L_1 es **ϵ-aproximadamente correcto** con respecto a L_2 si el error de predicción es menor a ϵ.

- Un **oráculo** es una entidad cuya responsabilidad es tomar una determinada secuencia del alfabeto, y etiquetarla como **positiva** o **negativa** según si pertenece o no al lenguaje objetivo.

Dicho esto, un algoritmo de *PAC-learning* se encarga de tomar un parámetro de **aproximación** $\epsilon \in (0,1)$, un parámetro de **confianza** $\delta \in (0,1)$, y un **oráculo**, y en caso de terminar devuelve un lenguaje ϵ-aproximadamente correcto con respecto al lenguaje objetivo, con probabilidad mayor o igual a $1 - \delta$.

Los algoritmos de *PAC-learning* también pueden contar con *queries* de membresía y equivalencia, como se vieron **MQ** y **EQ** respectivamente para L^*, con la diferencia de que la **EQ** esta vez realiza un *test* de equivalencia aproximada al momento de comparar ambos lenguajes, utilizando un conjunto lo suficientemente grande de secuencias de muestra generadas y etiquetadas por el oráculo. La respuesta del oráculo es positiva si cada una de estas secuencias, o bien pertenece a ambos lenguajes, o bien no pertenece a ninguno.

2.3.1 L^* basado en PAC

En este contexto, el Learner L^* es exactamente igual. La diferencia radica en que el Teacher esta vez utiliza el mencionado test estadístico para la comparación de los lenguajes. Se puede profundizar sobre la completa descripción del procedimiento en [7], pero a grandes rasgos la idea es la siguiente: Cada vez que se llama a la **EQ**, se debe extraer una muestra de un tamaño lo suficientemente grande como para asegurar una confianza **total** del algoritmo de al menos $1 - \delta$.

Supóngase que una **EQ** representa una iteración i. Para poder cumplir con la propiedad deseada, se extrae una muestra S_i de tamaño r_i, donde:

$$r_i = \left\lceil \frac{1}{\epsilon}(i \ln 2 - \ln \delta) \right\rceil \tag{2.1}$$

2.4 Redes Neuronales Recurrentes

Las redes neuronales recurrentes, también llamadas **RNN**s por su sigla en inglés, son un caso particular de redes neuronales que, como se detalla en [7], son utilizadas para procesar secuencias de datos. Pueden interpretarse como un sistema dinámico, cuyo estado actual depende de ciertos parámetros, algunos dependientes del tiempo (como por ejemplo su estado anterior), y otros independientes.

En el marco del presente trabajo, así como en el trabajo citado, se hace foco

en las RNNs que procesan una secuencia entera de datos y producen una salida binaria única. El propósito de ello es que esta familia de RNNs puede utilizarse para clasificar secuencias como el **oráculo** mencionado previamente, sobre si pertenecen o no a cierto lenguaje. Se define el **lenguaje** de una RNN como el conjunto de secuencias que son etiquetadas **positivamente** por la red neuronal.

2.4.1 PAC-Learning para RNNs

Un problema, llegado a este punto, es saber qué tipo de lenguaje es el reconocido por una red neuronal. Independientemente de la respuesta de la red, no se sabe a primera vista sobre el tipo de lenguaje que se trata (por ejemplo, si es regular o no), o si se puede representar mediante algún tipo de autómata. Por lo que el problema que se plantea en [7] es si es posible construir un DFA tal que el lenguaje representado por el mismo sea (ϵ–δ)aproximadamente correcto con respecto al lenguaje reconocido por una RNN.

La idea básica, para solucionar esto, es utilizar $L*$ realizando **MQ**s a un Teacher que contiene una RNN. Como se subrayó anteriormente, $L*$ devuelve un DFA que es una ϵ-aproximación del lenguaje objetivo, con una probabilidad de al menos $1 - \delta$ en caso de terminar su ejecución. De hecho, está probado que $L*$ terminará su ejecución siempre y cuando el lenguaje sea un lenguaje regular. Sin embargo, puesto que las RNNs son más expresivas que los DFAs, no hay garantía de que $L*$ termine su ejecución cuando se utilizan RNNs. Puede que no exista un DFA que represente al lenguaje definido por la red.

2.4.2 Bounded-L*

Para abordar este problema es que, en su trabajo [7], Franz Mayr propone una solución que consiste en una variación del algoritmo $L*$, llamada ***Bounded-L****, que consiste en imponer restricciones a la cantidad de iteraciones que se permitan en su ejecución. Se utilizan dos medidas que suelen utilizarse a la hora de definir la complejidad de un algoritmo de *PAC-learning* [7]:

1. El **número máximo de estados** del autómata a ser aprendido.
2. El **largo máximo de secuencia**s utilizadas para llamar a la **MQ**.

De esta manera, el algoritmo *Bounded-L** terminará ya sea con una **EQ** exitosa, o porque una de estas dos restricciones fue alcanzada. En caso de éxito, el DFA obtenido como resultado ha demostrado ser una ϵ aproximación de la red neuronal, con una probabilidad de al menos $1 - \delta$. En caso contrario, de haber finalizado por alguna de las cotas superiores definidas, se obtendrá como resul-

tado el último DFA propuesto por el Learner, el cual no necesariamente es una ϵ-aproximación con una confianza de al menos $1 - \delta$, porque falló el último test de la **EQ**. Sin embargo, este resultado puede utilizarse para analizar la relación entre este autómata obtenido y la red. Un estudio en profundidad sobre esta relación es el que se realiza en [7].

2.5 Planteamiento del problema

En la práctica, en el *framework* escrito en Python con que cuenta la Cátedra de Inteligencia Artificial y Big Data, la aplicación del algoritmo $L*$ sugiere la existencia de oportunidades que permitirían mejorar su *performance* [6] debido a los costes computacionalmente altos que implican los procesamientos de datos de sus ejecuciones, especialmente a medida que aumenta la complejidad de los casos de uso. Por lo que se plantea la siguiente interrogante: En el contexto de la presente implementación de la Cátedra, ¿es posible poder mejorar la eficiencia de los tiempos de ejecución del algoritmo $L*$, y/o de su versión alternativa *Bounded-$L*$*? En ese caso, ¿mediante qué técnicas o cambios a realizar? ¿Tendrían un leve o un alto impacto de cambio a nivel de implementación?

El resto del presente trabajo está dedicado a responder estas preguntas.

3 Propuesta de solución

Se propone abordar el problema de la reducción de los tiempos de ejecución del algoritmo *L**. Para ello, se busca hacer foco en buscar oportunidades de optmización y paralelismo.

El presente capítulo consta de las siguientes secciones:

En la sección 3.1, se propone el uso del algoritmo *L*Col* [8], una variante optimizada del algoritmo *L** de Angluin, que será el puntapié inicial para la búsqueda de mejores tiempos de ejecución.

La sección 3.2 consiste en una evaluación de posibles candidatos de paralelización; esto es, secciones del código del algoritmo cuya implementación pudiera llevarse a cabo de manera paralelizada, en una relación costo-beneficio de esfuerzo razonable.

En la sección 3.3, luego de haber realizado dicha investigación y análisis, se procede a poner en práctica las decisiones tomadas y lecciones aprendidas durante dicho proceso, para poder trabajar en la implementación de una versión alternativa del algoritmo *L** que sea más eficiente en tiempos de ejecución, para casos de prueba relevantes, utilizando autómatas de tamaños mayores a 64 estados y entrenando redes neuronales a partir de los mismos.

3.1 L*Col

En el trabajo publicado por Oded Maler y Amir Pnueli [8], los autores proponen una variante del algoritmo *L** propuesto por Angluin, al cual le llaman L^{ω}. En un comentario realizado a pie de página, se deja plasmada una propiedad que resulta de particular interés para el presente trabajo:

"In fact, Angluin uses an additional notion of a consistent table, but this can be eliminated by a slight modification of L: instead of adding the prefixes of a counter-example to S, add their suffixes to E. This way the table is always consistent."*

Lo que esta propiedad dice es directo: Si en vez de agregar los prefijos de los contraejemplos a las filas de la Observation Table, se agregan los sufijos a las columnas de la misma, la Observation Table se mantiene siempre consistente. Esto quiere decir que, en contraposición al algoritmo *L** que debe ejecutar en cada iteración una función que asegure que la tabla preserve su consistencia, el algoritmo L^{ω} (de aquí en más, *L*Col*, por *columns*) mantiene la tabla siempre consistente sin tener que ejecutar operaciones adicionales.

Es por esto que, llegado a este punto, surge la siguiente interrogante: En caso de

sustituir exitosamente la presente implementación de *L** por una implementación de su variante “en columnas”, ¿se ganaría eficiencia en tiempos de ejecución al realizar una operación menos en cada iteración? ¿Podría lograrse que las corridas sean más rápidas para distintos casos de prueba?

En el marco de la búsqueda y análisis de oportunidades de mejora en el proyec- to, se propuso iniciar evaluando el algoritmo *L*Col*, el cual resultó ser la opción más interesante de abordar debido a esta propiedad encontrada por los autores, prometiendo en principio mejorar la eficiencia de *L** en tiempo de ejecución al disminuir el total de la cantidad de operaciones.

3.1.1 Ejemplo de ejecución

Obsérvese análogamente el mismo ejemplo de la sección 2.2.2, pero esta vez para el algoritmo *L*Col*.

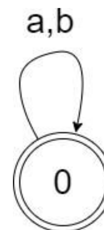
Se comienza con la tabla vacía, siendo RED = λ y EXP = λ, donde EXP representa el conjunto de experimentos que serán utilizados para la corrida del algoritmo.

Se realiza una primera *membership query* con la secuencia vacía λ. La respuesta es *True*, luego *a* y *b* se agregan a BLUE, y se realizan nuevamente las *membership queries* *a* y *b*.

Supóngase que las respuestas son nuevamente *True*. La tabla correspondiente es cerrada, consistente y completa, por lo que se puede construir un autómata a partir de ella para realizar una *equivalence query*. El autómata correspondiente a la tabla de la figura 3.1a se muestra en la figura 3.1b.

	λ
λ	1
a	1
b	1

(a) Una tabla consistente.



(b) Autómata asociado a la tabla (a).

Figura 3.1: OT y autómata asociado.

Ahora, supóngase que el Teacher retorna el contraejemplo *abb*. Hasta este puto, la ejecución es similar al mismo caso visto para *L**. Pero esta vez, la tabla se actualiza a la de la figura 3.2a, donde se agregan a sus columnas los sufijos del contraejemplo *abb*. Se

prosigue a completar la tabla, llenando los agujeros a través de *membership queries*, produciendo la tabla 3.2b que ahora es completa.

Pero ahora la tabla no está cerrada, por lo que se mueve la fila *b* de **BLUE** a **RED**, y se agregan a **BLUE** las concatenaciones de cada uno de los elementos de las filas de **RED** (*a* y *b*) con cada uno de los símbolos del alfabeto (*a, b*), resultando la tabla como la de la figura 3.2c, que se debe completar. Se obtiene la tabla de la figura 3.2d que es cerrada, completa, y también puede verse que se mantiene consistente sin necesidad de realizar una verificación aparte dentro del algoritmo, puesto que siendo *OT* [λ] = *OT* [*a*] se cumple que *OT* [$\lambda.a$] = *OT* [*aa*] y *OT* [$\lambda.b$] = *OT* [*ab*]. En este contexto, la tabla puede traducirse en un DFA a serenviado mediante una *equivalence query*.

	λ	b	bb	abb
λ	1			0
a	1			
b	1			

(a) Tabla después de la **EQ**, retornando *abb* (que no está en el lenguaje).

	λ	b	bb	abb
λ	1	1	0	0
a	1	1	0	0
b	1	0	0	0

(b) Se realizan **MQ**s: La tabla no está cerrada.

	λ	b	bb	abb
λ	1	1	0	0
a	1	1	0	0
b	1	0	0	0
aa				
ab				
ba				
bb				

(c) Se mueve la fila a **RED** y se agregan las filas correspondientes a **BLUE**.

	λ	b	bb	abb
λ	1	1	0	0
a	1	1	0	0
b	1	0	0	0
aa	1	1	0	0
ab	1	0	0	0
ba	1	1	0	0
bb	1	0	0	0

(d) Luego de llenar los agujeros, la tabla es cerrada y consistente.

Figura 3.2: Ejecución de *L*Col*.

Siguiendo el mismo ejemplo y suponiendo nuevamente que aquí la EQ dé *True*, el algoritmo termina, resultando el mismo autómata de la figura 2.4.

3.1.2 Implementación de *L*Col* y primeros experimentos

A partir de esta propiedad, se procedió a la implementación de *L*Col*, para ser testeado, en primera instancia, con los mismos casos de prueba que su contrapar- te *L**. El experimento que se realizó fue el siguiente: Para ambos algoritmos, se ejecutaron los mismos tests utilizando un total de 108 autómatas compuestos por varios grupos, entre ellos, mas no excluyentemente, los autómatas de Tomita, que aparecen en diversos trabajos [9][10][11], y los autómatas de Omlin-Giles [12]. A su vez, para cada autómata, se ejecutaron veinte corridas del mismo, se calculó el tiempo de cada una, y luego se calculó la mediana de todos los tiempos, junto con el IQR, que sirve como indicador de dispersión.

Este procedimiento se puede ver en el siguiente pseudo-código:

```
1        Empezar
2            Para cada algoritmo en { L*, L*Col }:
3                Para cada autómata en {TomitasGrammar, OmlinGilesAutomata,
4                                       OtherAutomata}:
5                    Para cada contador en { 1, ... , 20 }:
6                        Calcular_tiempo_de_ejecución[contador]
7        Calcular_estadisticas_de_tiempos()
8        Fin
9
```

En pos de obtener una primera percepción de la relación entre los resultados de ambos algoritmos para las mismas entradas, se analizaron los datos que se obtuvieron como salida de dichas ejecuciones. Se muestra a continuación un fragmento de los primeros resultados:

Instance	**LStar (s)**	**LStarCol (s)**
Simpler ab prefixed automaton	0.0016 ± 0.00005	**0.0014 ± 0.00009**
Tomita's grammar 1 automaton	0.0006 ± 0.00008	0.0006 ± 0.00007
Tomita's grammar 2 automaton	0.0017 ± 0.00011	**0.0015 ± 0.00016**
Tomita's grammar 3 automaton	0.0039 ± 0.00044	**0.002 ± 0.00014**
Tomita's grammar 4 automaton	0.0021 ± 0.00023	**0.0019 ± 0.00014**
Tomita's grammar 5 automaton	0.0032 ± 0.00019	**0.0027 ± 0.00024**
Tomita's grammar 6 automaton	0.0017 ± 0.00012	**0.0015 ± 0.0001**
Tomita's grammar 7 automaton	0.0027 ± 0.00022	**0.0023 ± 0.00014**
Uneven number of as	0.0008 ± 0.00015	**0.0007 ± 0.00008**
Uneven number of as and symbols	0.002 ± 0.00008	**0.0017 ± 0.00009**
Uneven number of as or symbols	0.002 ± 0.00006	**0.0017 ± 0.00015**
Uneven number of symbols	0.0008 ± 0.00009	**0.0007 ± 0.0000**5
Zero ending automaton	0.0007 ± 0.00005	0.0007 ± 0.00008
a automaton	0.0017 ± 0.00007	**0.0016 ± 0.00011**
a ending automaton	0.0008 ± 0.00005	**0.0007 ± 0.00007**
a ending with cs automaton	0.0008 ± 0.00005	0.0008 ± 0.00012
a or b Automaton	0.0005 ± 0.00005	0.0006 ± 0.00018
ab automaton	0.0017 ± 0.0001	**0.0015 ± 0.00015**
ab with cs automaton	0.002 ± 0.0001	0.002 ± 0.00024

Como se puede apreciar, se encontró que ya en este experimento, en la mayoría de los casos *L*Col* demora un tiempo promedio igual o menor a *L** para un mismo tipo de autómata. Sin embargo, debido a la varianza que tienen los resultados, no se puede hablar de una mejora concluyente. De todas maneras, son suficiente para asegurar que *L*Col* no empeora los tiempos con respecto a *L**, sino que logra mantenerlos o incluso puede llegar a mejorarlos.

Con el objetivo de buscar diferencias aún más significativas en las mejoras de tiempos de ejecución, se decidió profundizar en el camino de la implementación y mejora del algoritmo *L*Col*, ya que estos resultados podrían ser el puntapié inicial de la verificación de la hipótesis planteada sobre *L*Col* versus *L** en contextos más generales que los tratados hasta ahora, como por ejemplo mayor cantidad de autómatas, mayor cantidad de ejecuciones para cada autómata, autómatas de tamaños mayores, entre otros. Se decide entonces, pues, comenzar a estudiar posibles oportunidades de paralelismo a partir de la implementación de *L*Col*.

3.2 Análisis de oportunidades de mejora

3.2.1 Evaluación de oportunidades de paralelismo

Para el estudio de distintos candidatos a oportunidades de paralelismo, se procedió a la especificación de los mismos en pseudocódigo, donde dentro de cada sentencia *forall* se indica la porción de código a ser eventualmente paralelizada.

En pos de continuar con la mejora mencionada del algoritmo *L*Col*, se decidió iniciar la búsqueda de estos candidatos dentro de la estructura del mismo algoritmo. Luego de un análisis de las diferentes fases del algoritmo, los candidatos considerados fueron los siguientes:

Candidato #1: *_update_observation_with(counterexample)*

Como su nombre lo indica, esta función se encarga de actualizar la Observation Table con el contraejemplo que encuentra el Teacher cuando la respuesta de la Equivalence Query (EQ) de pertenencia al lenguaje es negativa.

Su pseudocódigo se ve a continuación:

```
1    _update_observation_table_with(self,   counterexample)
2    {
3        suffixes = counterexample.getSuffixes()
4        forall sequence in suffixes
5        {
6            self._add_to_suffixes(sequence)
7        }
8    }
9
```

Candidato #2: _close(self)

Esta funcion se encarga de cerrar la tabla [6], uno de los pasos previos a realizar la **EQ**. A continuación se muestra su especificación:

```
1   _close(self)
2   {
3       while True
4       {
5           blueSequence = self._get_closedness_violation_sequence()
6           If blueSequence is None
7           {
8               return
9           }
10          self._move_from_blue_to_red(blueSequence)
11          forall symbol in self.symbols
12          {
13              newBlueSequence = blueSequence + symbol
14              self._add_to_blue(newBlueSequence)
15          }
16      }
17  }
18
```

Candidato #3:_get_filled_row_for(self, sequence: Sequence)

Finalmente, este tercer candidato tiene como objetivo llenar una fila de la **OT** para una determinada secuencia que se le pasa como parámetro. La especificación correspondiente es la que le sigue:

```
1   _get_filled_row_for(self, sequence: Sequence): list
2   {
3       requiredSuffixes = self._observation_table.exp
4       row = []
5       forall suffix in requiredSuffixes
6       {
7           result = self._teacher.membership_query(sequence + suffix)
8           row.append(result)
9       }
10      return row
11  }
12
```

3.2.2 Elección de primera opción de paralelismo a implementar

Con el análisis de estos tres candidatos, se resolvió y se decidió ir, como primera opción de implementación de paralelismo, por el **candidato #3**, dado que la sentencia ***for*** de su estructura parece prometer ser una buena oportunidad para la ejecución de sus sentencias de manera paralela en lugar de secuencial. El estudio y discusión sobre este candidato derivó en ir por el camino de la **implementación de la *Membership Query (MQ)* en *batch*** (**BatchMQ** de aquí en más).

La implementación de *L*Col con BatchMQ* a grandes rasgos consiste en lo siguiente: En lugar de que el Learner le pregunte al Teacher una por una si las secuencias pertenecen o no al lenguaje, el Learner le realiza una única consulta al Teacher, y le pasa en la misma todas las secuencias en una estructura como parámetro. El Teacher resuelve la pertenencia o no pertenencia de cada una de las secuencias, y le devuelve al Learner los resultados todos juntos. De esta manera, esa comunicación entre ambas clases se da una vez sola en cuanto a pertenencia de lenguaje se trata, en vez de múltiples interacciones de pregunta-respuesta para cada una de las secuencias.

3.2.3 Experimentos con técnica Abbadingo One

En el presente nuevo experimento, para utilizar casos de prueba más representativos, se generaron 40 DFAs de tamaño nominal n de 8 estados, 40 DFAs de tamaño 16, 40 de tamaño 32, y 40 de tamaño 64 estados, utilizando la técnica de generación de DFAs Abbadingo One, que será explicada en detalle en la sección 4.1.

Los resultados obtenidos se muestran a continuación:

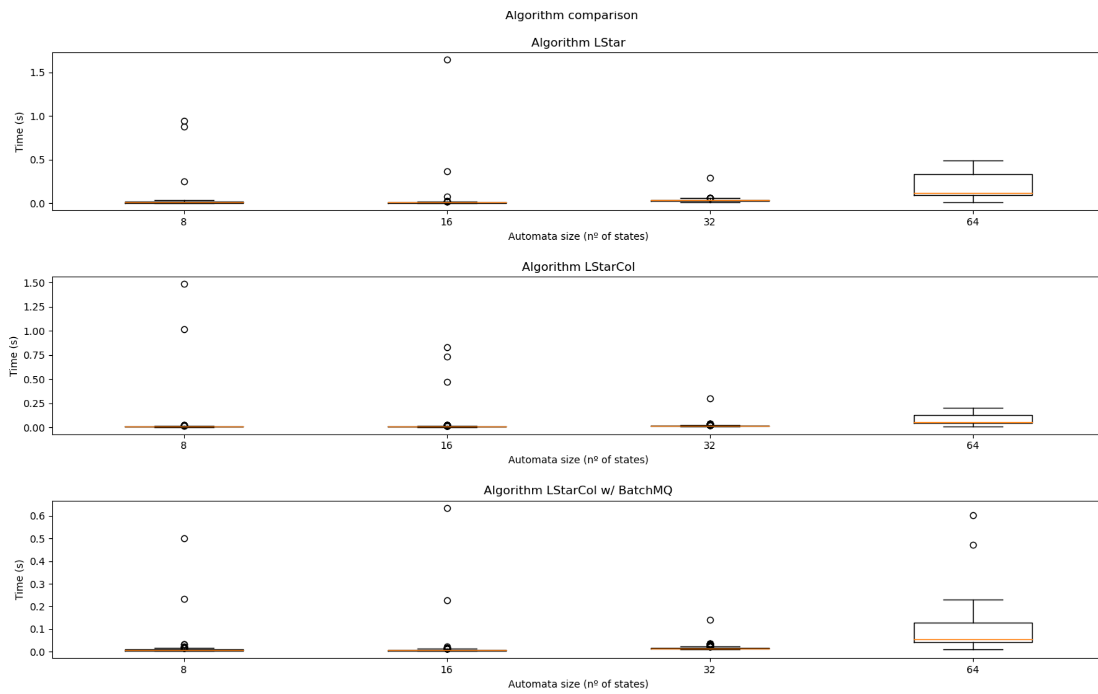


Figura 3.3: Prueba de *L**, *L*Col* y *L*Col Batch MQ* mediante Abbadingo One.

Esta grafica muestra el resultado general que se obtuvo, que permite ver a grandes rasgos el grado de tendencia de mejora de los algoritmos a medida que la complejidad de los datos de prueba aumenta. Sin embargo, observando medidas de media y dispersión de los valores promedio de estas gráficas, se pudo ver que se necesita mayor experimentación para obtener conclusiones firmes. A continuación se realiza un análisis de estos valores, para los distintos tamaños de autómatas estudiados, y se comparan los resultados que se obtienen utilizando éstos como casos de prueba para cada uno de los tres algoritmos.

Size (states)	***L**** **(s)**	***L*Col*** **(s)**	***L*Col BatchMQ*** **(s)**
8	0.0098 ± 0.0078	0.0064 ± 0.0033	**0.0063 ± 0.0032**
16	0.0072 ± 0.0036	0.0049 ± 0.0021	**0.0049 ± 0.0019**
32	0.0205 ± 0.0065	0.0159 ± 0.0033	**0.0155 ± 0.0032**
64	0.0685 ± 0.1617	0.0403 ± 0.0667	**0.0393 ± 0.0668**

Se puede observar que en tamaño 8 no hay mayor diferencia, ya que si al promedio de *L*Col* o *L*Col BatchMQ* se le suma el correspondiente IQR, el resultado

se superpone con los valores de *L**, por lo que la diferencia no es significativa. Para tamaño 16 estados, si bien *L*Col* y *L*Col BatchMQ* muestran tanto tiempo promedio como varianza menores que *L**, de todas maneras los valores siguen cercanos. Sucede lo mismo para los tamaños 32 y 64 estados. Pese a las mejoras en los tiempos promedio, de momento hay demasiada dispersión como para realizar afirmaciones más precisas sobre mejoras.

3.2.3.1 Conclusiones del experimento

Los valores obtenidos indican que, para cada uno de los casos de autómatas de tamaño 8, 16, 32 y 64 estados, el tiempo promedio de la ejecución del algoritmo *L*Col* parecer´ıa ser menor que la de *L**, y al mismo tiempo, el algoritmo *L*Col BatchMQ* pretende presentar tiempos aún menores incluso que los de *L*Col*. Esto muestra resultados de interés, ya que las hipótesis de ganancia de los tiempos de ejecución de las nuevas versiones mejoradas del algoritmo *L** aparentan manifestarse en este marco experimental. No solamente esto, sino que además las diferencias en las medianas de los tiempos son cada vez más notorias. Sin embargo, se habla aquí de una <u>aparente</u> mejora en los tiempos de ejecución, ya que, pese a la comparación de los valores de las medianas, los elevados valores de la métrica de dispersión IQR no permiten sacar conclusiones significativas. Como se menciona más adelante, a futuro se busca intentar mejorar el marco de experimentación para reducir la dispersión de las métricas.

3.2.4 *BatchMQ*: Próximos pasos.

Una de las cosas más importantes que se pretende realizar en la siguiente etapa del proyecto, en pos de la búsqueda de mejores tiempos de ejecución, es la **integración de BatchMQ con <u>redes neuronales</u>**. Se busca poder comparar, en algunos casos de referencia de redes, cómo funcionan *L**, *L*Col* y *L*Col BatchMQ* interactuando con redes dentro del Teacher. Y se espera que en los tiempos de ejecución haya una mejora más grande que la que se registró hasta ahora.

3.3 Bounded-L*Col BatchMQ

En el marco de continuar los pasos mencionados en el cap´ıtulo anterior, se procedió a comparar estos algoritmos entre sí en el contexto de redes neuronales, utilizando el mismo conjunto de pruebas. Esto implicó que, como se explicó antes, hubiera que buscarle una solución al problema de que los algoritmos tal vez nunca terminaran su ejecución. Es por esto que, de la misma manera que se plantea el algoritmo *Bounded-L** en [7], en este trabajo se combina la implementación de este algoritmo con *L*Col* y *L*Col BatchMQ*, las variantes de *L** implementadas previamente.

3.3.1 Algoritmo

De manera similar a como se detalla *Bounded-L** en [7], el algoritmo *Bounded-L*Col BatchMQ* se describe a continuación:

Algoritmo 1: *Bounded-L*Col BatchMQ*

Input: MaxQueryLength, MaxStates, ϵ, δ
Output: DFA

```
1   Lstar-Initialise;
2   repeat
3       while OT is not closed do
4           if OT is not closed then
5               OT, QueryLengthExceeded ← LstarColBatchMQ-Close(OT);
6           end
7       end
8       if not QueryLengthExceeded then
9           A ← Lstar-BuildAutomaton(OT);
10          Answer ← EQ(A);
11          MaxStatesExceeded ← STATES(A) > MaxStates;
12          if Answer = Yes and not MaxStatesExceeded then
13              OT ← Lstar-UseEQ(OT, Answer);
14          end
15      end
16      BoundReached ← QueryLengthExceeded or MaxStatesExceeded;
17  until Answer = Yes or BoundReached;
18  return A;
19
```

La *observation table* **OT** se inicializa con *Lstar-Initialise* de la misma manera que para *L**. Este procedimiento consiste en crear la estructura de la **OT** como propone Angluin. Luego comienza el proceso de construcción de hipótesis.

Si la **OT** no está cerrada, se agrega otra columna con el procedimiento *LStar-Close*. Este procedimiento llama a la **MQ** para llenar los agujeros en la **OT**. El largo de estas *queries* puede exceder el largo máximo definido en *MaxQueryLength*, en cuyo caso la bandera *QueryLengthExceeded* se setea en True. La forma en la que se llama a la **MQ** dentro de *LstarColBatchMQ-Close* var´ıa respecto a como presenta De la Higuera su contraparte *Lstar-Close* en [6]:

Algoritmo 1.1: *LstarColBatchMQ-Close*

Input: OT
Output: OT updated.

```
1    for s ∈ BLUE such that ∀u ∈ RED OT[s]    OT[u] do
2        RED ← RED ∪ {s};
3        BLUE ← BLUE \ {s};
4        for a ∈ Σ do
5            BLUE ← BLUE ∪ {s.a}
6        OT[u][e] ← BatchMQ(ue ∀u, e ∈ Σ* such that OT[u][e] is a hole)
7    return OT;
8
```

Como se puede apreciar, este algoritmo presenta algunas diferencias frente a *Bounded-L**:

En primer lugar, *Bounded-L*Col BatchMQ* no verifica la consistencia de la **OT**. Esto se debe a la propiedad encontrada por Maler y Pnueli detallada en la sección 3.1, donde se expone que la tabla se mantiene siempre consistente.

En segundo lugar, las **MQ**s se realizan de manera diferente: En lugar de resolverlas una por una, el Learner le env´ıa al Teacher todas las **MQ**s en una sola consulta, y el Teacher las resuelve internamente, tal como se explicó previamente para *L*Col BatchMQ*. Luego de haberlas resuelto en su totalidad, el Teacher le devuelve al Learner todas las respuestas en una sola estructura mediante una única respuesta.

4 Experimentos realizados

Como se mencionó al final de la sección 3.2.4, se espera que los tiempos de ejecución de *Bounded-L*Col BatchMQ* sean menores a los de *Bounded-L** y *Bounded-L*Col*. Esto se debe a que las redes neuronales aceptan consultas en *batch* naturalmente [13], dado que a una red neuronal le toma un trabajo similar procesar una consulta que un *batch* de consultas (hasta ciertos tamaños de *batch*, dado que a mayor tamaño de *batch*, mayor es el uso de memoria RAM). En este caso, cada una de las partes a ejecutar de manera independiente corresponde a una *Membership Query*.

4.1 Abbadingo One

A la nueva implementación de la **MQ** en *batch*, se decidió acompañarla con un cambio en el marco experimental como se mencionó previamente. Un contexto donde pudieran emplearse casos de prueba más representativos, con variedad de autómatas en cantidad, tamaño, cantidad de corridas de cada uno, entre otros aspectos.

Para ello, se propone la evaluación de los algoritmos tenidos hasta ahora en el caso de generación de DFAs con la técnica Abbadingo One [14]. Esta técnica consiste en la generación de automatas finitos deterministas de manera aleatoria mediante una serie de pasos:

1. Se comienza eligiendo un conjunto de estados donde se elige uno al azar para que sea el estado inicial. El tamaño del conjunto de estados es de $5n/4$, donde n es el tamaño nominal deseado para el algoritmo.
2. Para cada estado, se decide si son estados finales, con probabilidad 0,5.

3. Se selecciona de manera uniforme sobre todos los estados y, para cada uno de ellos, sobre todos los símbolos, y se define cada uno de los resultados de la función de transición.

4. Los estados que resultan inalcanzables desde el estado inicial se eliminan. De esta manera, se generan DFAs de **tamaño nominal** $\boldsymbol{n}$. Esto quiere decir que su distribución de tamaños se da alrededor de n. La profundidad de los automatas es aproximadamente $2log_2(n) - 2$. De la misma forma que en [14], para asegurar profundidades de exactamente $2log_2(n)-2$ el proceso se repite hasta que la profundidad alcanzada sea exactamente igual a la deseada.

4.2 Dificultades durante la ejecución

El método de experimentación había adquirido robustez llegado a este punto, donde el flujo de trabajo entre una prueba y otra hab´ıa comenzado a ganar mayor fluidez entre los análisis de un experimento y la obtención de los resultados del siguiente. Sin embargo, cada vez que se corr´ıa un nuevo experimento, se estaban generando nuevamente todos los autómatas necesarios, y también se volvían a entrenar cada una de las redes desde cero. Esto incurr´ıa en costos de tiempo muy grandes, donde sumado al tiempo de ejecución de las pruebas luego de haber generado todos los datos, cada experimento demoraba en el orden de varias horas, e incluso días. En consecuencia, a medida que se fueran agregando casos más complejos, este problema iba a escalar de manera exponencial.

Ante esto, se planteó la posibilidad de implementar un sistema donde se pudiera dividir el experimento en dos partes independientes: la primera parte para generar los autómatas y las redes de prueba y **persistirlos en archivos**, y la segunda con el objetivo de **leer esos archivos** para tomar esos autómatas y redes persistidos, y utilizarlos para ejecutar los experimentos. Fue as´ı que se crearon respectivamente las rutinas ***generate_and_persist.py*** y ***load_and_run.py***, donde se fragmentó el experimento original para ganar eficiencia en los tiempos de ejecución de las pruebas.

1. ***generate_and_persist.py*** : Dado un *array* de enteros, crea una cantidad n de autómatas (en estos experimentos, $n = 40$) de cada uno de los tamaños pertenecientes al *array* mencionado, entrena una red neuronal con cada autómata, y persiste estos autómatas y redes en archivos en un determinado directorio.

2. ***load_and_run.py*** : Dados dos directorios, utiliza uno de ellos para cargar los datos de los autómatas en memoria, y el otro análogamente para las redes neuronales. Luego de esta etapa, se ejecuta el experimento normalmente.

4.3 Problemas de implementación

Los desafíos comentados previamente se dieron en torno a la estructura del marco experimental, el cual fue evolucionando, siendo modificado, y se le fue agregando complejidad con el paso del tiempo y los avances en la investigación. Este marco experimental, si bien mostró dificultades que fueron posteriormente resueltas, tuvieron su origen dentro de este mismo proyecto de investigación.

Sin embargo, también fueron surgiendo en el proceso otros inconvenientes en materia de implementación, que se dieron de manera inherente a la estructura del proyecto *Model Extraction Framework* en sí (el *framework* con que cuenta la cátedra). Estos involucraron lo siguiente:

En el marco del repositorio de código del mismo, para mantener un buen flujo de trabajo y evitar solapamiento de ambientes, los experimentos locales fueron realizados en la misma rama *parallel_l_star* donde se trabajó a lo largo del proyecto, y los experimentos a ejecutar en CECOFI, el centro de cómputos de FI-ORT que cuenta con nodos con GPU, fueron realizados en una rama llamada *exp_ort_abbadingo one*, dedicada exclusivamente a esta labor.

En el proceso de implementación de los experimentos en la rama *parallel_l_star*, involucrando la generación de autómatas, el entrenamiento de redes neuronales con los mismos y las ejecuciones de las pruebas de los tres algoritmos con estos datos, se utilizaban muestras de datos y casos de prueba sencillos para acelerar el *debugging*.

En la rama *exp ort abbadingo one*, a diferencia de los valores utilizados para el *debugging* en ambiente local (rama *parallel_l_star*), los valores de parámetros y demás variables utilizadas eran los efectivamente necesarios para realizar el experimento en el servidor de la universidad. Por lo tanto, antes de actualizar la rama *exp_ort_abbadingo_one* con los últimos cambios de la rama *parallel l star*, había que realizar cambios de valores de variables y parámetros en diversos archivos del proyecto.

A modo de ejemplo y sin ser exhaustivos, algunas variables necesarias, mas no todas, y algunos ejemplos de sus valores para cada ambiente, se listan a continuación:

	Ambiente local (*parallel_l_star*)	**Ambiente remoto** (*exp_ort_abbadingo_one*)
Tamaños de autómatas a generar con Abbadingo One	6, 7, 8	52, 64, 65
Cantidad de autómatas a generar por tamaño	2	20
Nombre de archivo de resultados de ejecuciones	*results_abbadingo_one.csv*	*cecofi_ab1_RNNs_52-64-65.csv*
Nombre de archivo de agregación de resultados	*aggResults_abbadingo_one.csv*	*cecofi_agg_ab1_RNNs_52-64-65.csv*
Nombre de archivo de ploteo de gráficas comparativas de los tres algoritmos	*plot_results.png*	*cecofi_plot_52-64-65.png*

El hecho de tener que cambiar una gran cantidad de variables, dispersas a su vez en numerosos archivos a lo largo del proyecto, resultó ser un problema que iba escalando a medida que se llevaban a la práctica pruebas más complejas, derivando en varios posibles inconvenientes. Entre ellos, la no actualización correcta de una o más de estas variables, detectar valores sin cambiar al momento de la ejecución en el servidor (después de haber hecho *commit, merge* y *push* en el repositorio),

obtener resultados incongruentes, entre otros. Las dependencias del experimento se daban de la siguiente manera:

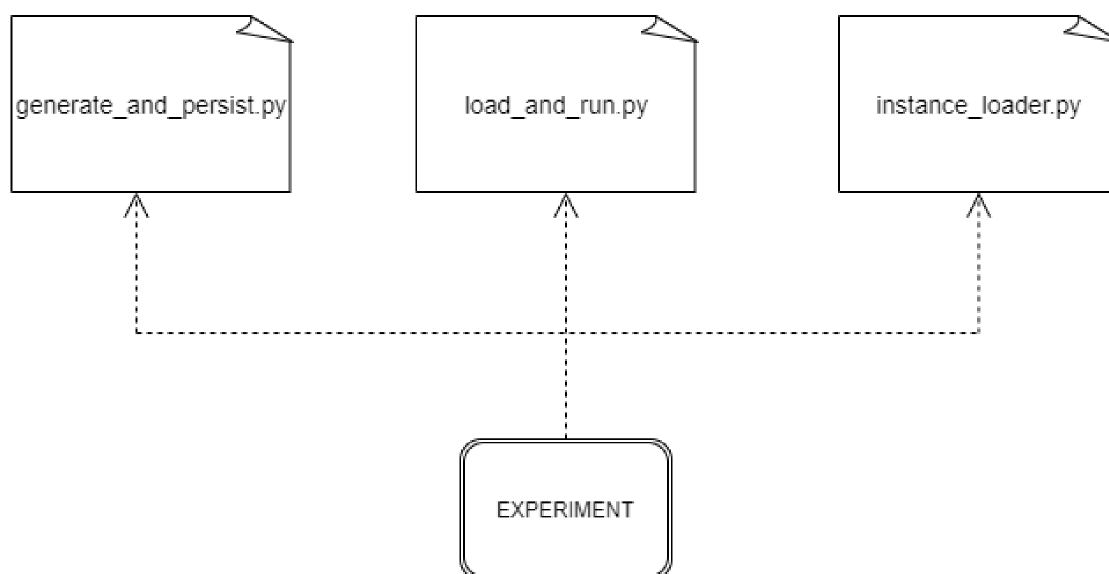


Figura 4.1: Dependencias del marco experimental al inicio.

Para ello, se decidió buscar alguna solución que fuera lo suficientemente flexible como para poder incorporar cualquier cantidad de parámetros experimentales de una manera eficiente y disminuir al mínimo el porcentaje de errores introducidos, pero siendo lo menos invasivo posible para el *framework*. El mismo ha sido y sigue siendo objetivo de numerosos aportes de múltiples trabajos, y esto muchas veces conlleva diferencias en configuraciones, entre otras. A modo de ejemplo, se pudo apreciar que en el archivo **.gitignore** del repositorio exist´ıan ya numerosas extensiones de archivos de entorno como **.env** y **.venv**, por lo que se optó no irrumpir con las implementaciones existentes. Por otro lado, para evitar mayor acoplamiento con terceros, se decidió que la solución implicara no necesitar el uso e instalación de librerías nuevas.

Es por ello que se decidió crear un archivo de entorno o ambiente, que contenga de manera centralizada todas los parámetros en cuestión, cuyos valores varíen según el ambiente de prueba sin tener que ser modificados manualmente. Este archivo, llamado ***ENV.py***, toma el valor de la varible booleana *isLocal* existente *en otro archivo* llamado ***ENV_VAR.py***, y según este valor le asigna distintos valores

a las variables que contiene. De esta manera, los archivos que las requieren, importan y utilizan estos valores acorde al ambiente correspondiente. Esto demostró ser muy útil en la práctica, donde las ejecuciones de los experimentos pasaron a ser mucho más consistentes, rápidas y con mucha menos introducción de errores.

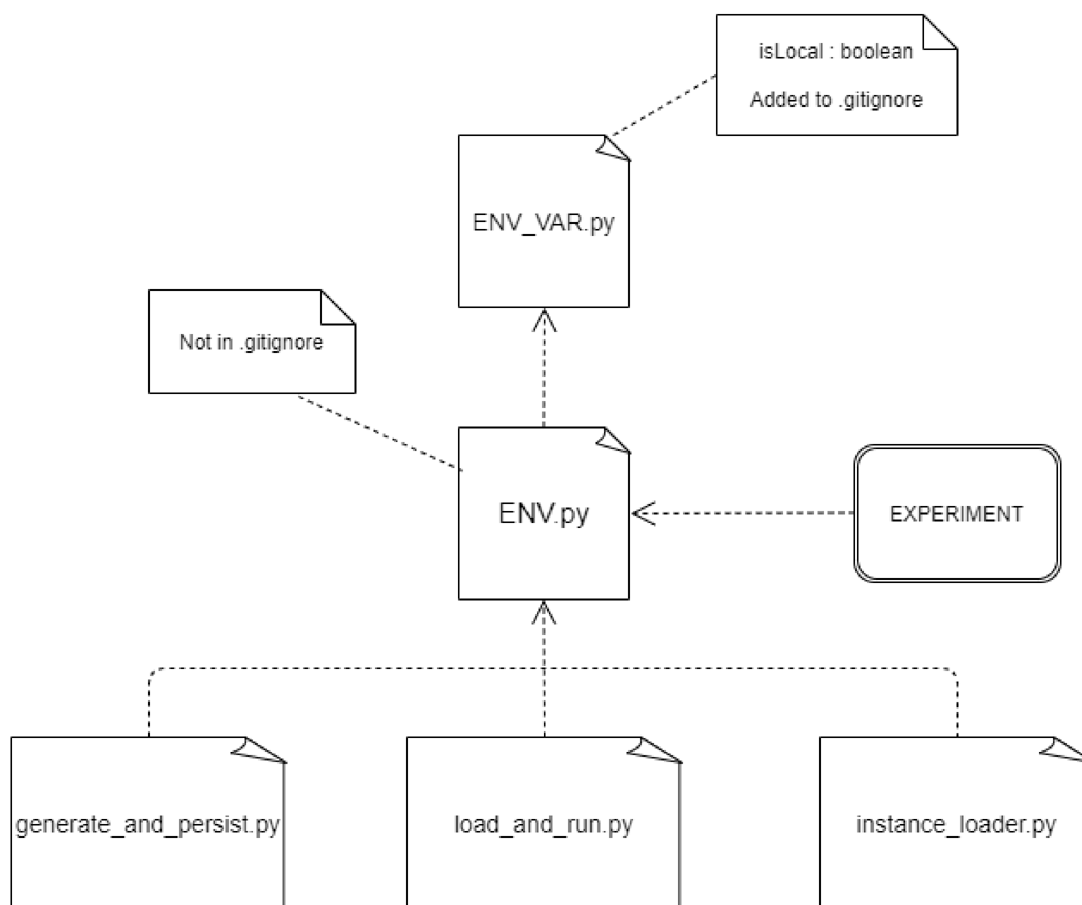


Figura 4.2: Solución al problema de dependencias del marco experimental.

Nótese que esta solución es sencilla, de bajo acoplamiento y cumple con el objetivo de la independencia de los ambientes de prueba. Los archivos implementados son archivos Python y no introducen ninguna dependencia de librer´ıas externas. Al mismo tiempo, los valores de los parámetros del marco experimental pasan a depender únicamente de un solo archivo, en lugar de depender de tres archivos o más. Claramente presenta algunas limitantes, por ejemplo el hecho de soportar la distinción de únicamente dos ambientes. Esto se podría generalizar a más am-

bientes, sustituyendo la variable booleana de *ENV_VAR.py* por otra que pueda admitir más valores, y modificar el código pertinente en *ENV.py*. No constituía el principal objetivo de este proyecto, y a la fecha se desconoce si esta generalización aportaría valor suficiente.

Una pregunta fácil de hacerse es: ¿por qué la necesidad de un archivo independiente para la variable *isLocal*? ¿Por qué no simplemente guardarla en el archivo *ENV.py* junto con las demás? En un principio se realizó de esta manera. El motivo de la separación de la variable *isLocal* a otro archivo se debe a que existen ocasiones donde los parámetros de experimentación necesitan ser modificados, guardados y actualizados en el repositorio, ya sea de forma definitiva o para realizar pruebas temporales. Por lo que no es recomendable agregar este archivo a *.gitignore*. En cambio, el valor de *isLocal* se mantiene siempre constante según la rama del repositorio en cuestión (*isLocal = True* para (*parallel_l_star*), *isLocal = False* para *export_abbadingo_one*), y agregar este archivo a *.gitignore* permite que al hacer cambio o *checkout* entre ramas, el valor de esta variable cambie acorde, y en consecuencia cambien los valores de todas las variables existentes en *ENV.py* sin necesidad de ninguna sustitución manual de valores.

4.4 Nuevos casos de experimentación

Según los experimentos realizados en los casos con y sin redes, los cambios verdaderamente apreciables en los tiempos de ejecución no comenzaban a verse sino a partir del tamaño **64 estados**, por lo que se retiraron los tamaños 8, 16 y 32 estados del marco experimental, en pos de emplear tiempo de ejecución y de análisis en casos más representativos.

Al mismo tiempo, como se verá más adelante en la sección 6.1, la técnica Abbadingo One mostró limitaciones a la hora de generar autómatas de tamaño 128 y 256 estados, siendo los autómatas de 64 estados los de mayor tamaño generados hasta el momento. Sin embargo, se planteó lo siguiente: ¿qué sucedería si se pudiera utilizar esta técnica para otros tamaños de autómatas, no necesariamente tan grandes como los anteriores pero que igualmente fueran representativos? ¿Ocurrirá algún fenómeno con la técnica en algún entorno del tamaño 64?

Fue entonces que se decidió hacer el siguiente experimento: utilizar el script *generate_and_persist* para generar autómatas de ciertos tamaños que estuvieran alrededor de 64 estados. Se inició con 64 ± 12 estados, es decir, tamaños 52 y 76 estados respectivamente.

Como era de esperarse, los autómatas de tamaño 52 lograron generarse de manera exitosa en tiempos razonables. Sin embargo, a los efectos de poder analizar estos resultados con el equipo en una fecha determinada, los autómatas de tamaño 76 no lograron ser generados. Se observó un fenómeno similar al de sus contrapartes de 128 y 256 estados.

En este punto, lo que se optó por hacer fue ir aumentando los tamaños de automatas en pequeños pasos e intentar la generación de los mismos, a la vez que se realizaban experimentos con los datos correspondientes a los tamaños ya existentes y generados exitosamente. Para poder realizar estos experimentos, se utilizó una *Raspberry Pi* conectada al servidor CECOFI de ORT, donde se procedía a correr la rutina *generate_and_persist* con nuevos tamaños de autómata, y luego de finalizada su ejecución se proseguía con la ejecución de la rutina *load_and_run*, donde a los datos existentes se les añadía los recientemente generados.

Dicho esto, se realizaron varias iteraciones de generar datos nuevos e incorporarlos a los ya existentes para aumentar la cantidad y diversidad de los mismos en el marco experimental.

En la próxima sección se muestran los resultados de cada uno de estos experimentos.

4.5 Resultados obtenidos

Se muestran a continuación los resultados experimentales de las ejecuciones descriptas anteriormente. Las mismas se describen siguiendo el mismo formato de mediana e IQR para cada tamaño de autómata con cada uno de los algoritmos utilizados, así como también otras métricas que se decidió agregar para análisis futuro, en pos de obtener mayor información para poder realizar análisis más descriptivos de los resultados obtenidos.

Las nuevas métricas agregadas a los experimentos son:

- Cantidad de estados a la salida (media e IQR).
- Cantidad de **EQ**s realizadas (media e IQR).
- Porcentaje de veces que el *bounded* falló por exceder la cantidad máxima de estados.
- Porcentaje de veces que el *bounded* falló por exceder el largo máximo permitido de la **MQ**.

Otras métricas deseadas pero que no llegaron a poder obtenerse en tiempo y forma son:

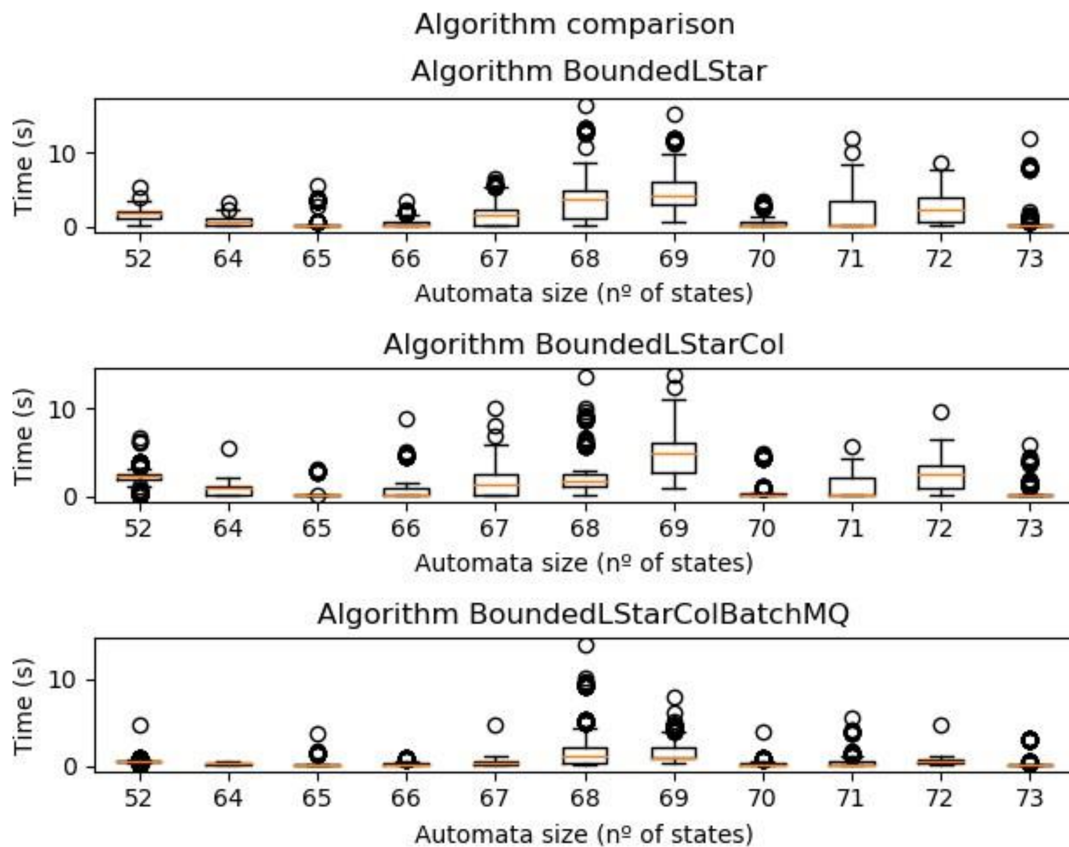
- Error de la red contra el DFA original.
- Cantidad promedio de errores que tuvo el último test.
- Medidas de ϵ y ϵ^*[7] de *PAC-Learning.*

A continuación se muestra el experimento realizado con los resultados obtenidos más relevantes. Una lista más detallada de los experimentos que siguen estas tendencias se encuentra en el Anexo 2.

Tamaños utilizados: 52, 64, hasta 71 estados.

Gráfica de los resultados:

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
67	Time (s)	1.4817 ± 2.12139	1.3819 ± 2.61188	**0.4256 ± 0.46727**
67	Output size (states)	3.5 ± 3	3.5 ± 3	3.5 ± 3
67	Nº of EQs	2.5 ± 2	2 ± 2	2 ± 2
67	Exceeded nº states (%)	0	0	0
67	Exceeded max MQ length (%)	0	0	0
68	Time (s)	3.9096 ± 5.07638	1.7631 ± 1.45113	**1.2543 ± 1.76370**
68	Output size (states)	8 ± 8.5	2 ± 2.5	12.5 ± 21.25
68	Nº of EQs	2 ± 1.25	1 ± 1	3 ± 1.25
68	Exceeded nº states (%)	0	0	0
68	Exceeded max MQ length (%)	0	0	0
69	Time (s)	3.4441 ± 3.18127	5.0022 ± 2.59761	**0.9712 ± 1.34976**
69	Output size (states)	7.5 ± 2.25	7 ± 3.25	8.5 ± 26.25
69	Nº of EQs	4 ± 1	2 ± 2	4 ± 1
69	Exceeded nº states (%)	0	0	0
69	Exceeded max MQ length (%)	0	0	0
70	Time (s)	0.1704 ± 0.60271	**0.1640** ± 0.25614	0.1647 ± **0.13664**
70	Output size (states)	2 ± 1.25	2 ± 1.25	2 ± 1.25
70	Nº of EQs	1 ± 0.25	1 ± 0.25	1 ± 0.25
70	Exceeded nº states (%)	0	0	0
70	Exceeded max MQ length (%)	0	0	0
71	Time (s)	0.1392 ± 3.46779	0.1323 ± 1.96141	**0.1308 ± 0.45903**
71	Output size (states)	1 ± 4.75	1 ± 1	1 ± 4.75
71	Nº of EQs	1 ± 1	1 ± 0	1 ± 1
71	Exceeded nº states (%)	0	0	0
71	Exceeded max MQ length (%)	0	0	0



En el presente experimento, lo interesante en el caso de 68 y 69 estados, es que por la selección aleatoria de la muestra de acuerdo a la ecuación 2.1, termina realizando más **EQ**s pero aún así su velocidad sigue siendo muy superior. Se puede observar en general que *Bounded-L*Col BatchMQ* es más eficiente que los otros algoritmos. Puede apreciarse una varianza grande en la cantidad de estados de los autómatas de salida, lo cual se conjetura que puede deberse a errores en el aprendizaje de las redes. Este aspecto escapa al alcance de este proyecto y las mejoras en los aprendizajes no constituye uno de sus objetivos.

5 Investigación de posibles soluciones de paralelismo

En el presente capítulo, se expone la investigación y el análisis de eventuales soluciones posibles que se podrían tomar en cuenta para la paralelización y/u otras formas de optimización del código de *L**. Se realizó un estudio de estas técnicas en el contexto del lenguaje Python, junto con ventajas y desventajas de cada una, en función de qué tan buenas candidatas puedan ser para el problema presente.

En el marco de uno de los objetivos del proyecto, se hizo un estudio (ver Anexo 1) en el campo de las tecnologías disponibles hoy en día para la programación concurrente, paralela y distribuida de algoritmos en Python. A continuación se muestran los *insights* y conclusiones extraídas luego del mismo, que condujeron a la decisión de enfocarse en la paralelización mediante el envío de **MQ** en *batch* (*BatchMQ*).

5.1 Análisis de tecnologías de paralelismo en Python

A la fecha del estudio de tecnologías de paralelismo, algunos puntos referentes a la realidad del algoritmo objetivo *L**, respecto al alcance del proyecto, son los siguientes:

1. Como ya se vio y se continúa en investigación, existen potencialmente múltiples oportunidades de paralelismo relativos a distintas etapas del algoritmo *L**. Por lo que sería conveniente incluir, entre las alternativas a utilizar, al-

guna que cumpla con características de genérica aplicación, es decir, que se pueda aplicar a distintos casos y no solamente a casos muy específicos (ejemplo de caso muy específico sería optimizar el uso del constructor *DataFrame* de Pandas [15]).

2. Existen algunas técnicas, que aunque sean específicas, parecen ser de sencilla implementación, por lo que se pueden incluir en pos de mejorar esos aspectos puntuales, que a corto y/o largo plazo puedan hacer la diferencia en su utilización. Un ejemplo de esto es la mencionada utilización de la librería Pandas. Al día de hoy, su uso es muy específico. Sin embargo, podría tenerse en consideración la optimización de esta librería en caso de requerirse mayor uso de la misma.

3. Python, como se menciona en algunas fuentes como [16], es un lenguaje de programación muy potente, y además, muy amigable para el programador. Sin embargo, debido a su implementación subyacente *single-threaded* (de un único hilo de ejecución), resulta ser en comparación más lento que otros lenguajes, por lo que también puede ser de utilidad la optimización de código general, no necesariamente de componentes que deban ejecutarse de manera paralela y/o distribuida. Un ejemplo para poder realizar esta tarea es la librería Cython.

5.2 *BatchMQ* y Ray: Primeros experimentos

En el marco de comenzar a trabajar con versiones paralelas de la implementación del algoritmo *L*Col BatchMQ* para resolver las **MQ**s en paralelo, y tomando en cuenta los estudios de las diversas tecnologías, se propuso realizar la transición de *L*Col BatchMQ* a una primera versión paralela de este algoritmo, mediante el uso de la librería Ray. El resultado inmediatamente derivó en que, en la práctica, esta librería presentó un muy fuerte acoplamiento con los distintos módulos del proyecto. Se pudo observar que, en contraposición a lo que la documentación prometía al momento de hablar de casos sencillos, el impacto de cambio sobre todo el proyecto era muy grande, dado que no solamente era menester modificar las clases inmediatamente necesarias, sino todas aquellas clases que fueran dependencias de estas.

Dado que los primeros experimentos con Ray no resultaron estar del todo

alineados con lo que se esperaba en teoría, donde la documentación destacaba a priori la fácil puesta en marcha de Ray a un proyecto, el alto impacto que demostró tener sobre el proyecto en los intentos de incorporarse al mismo hizo que se optara por pausar la implementación de Ray, ya que excedía el alcance del proyecto.

Se pudo ver que cada técnica tiene sus ventajas y desventajas. Algunas de ellas son particularmente útiles para determinados problemas, de baja complejidad y que no requieren mayores requisitos de memoria, y otros puede que sean de programación de más alto nivel y que eventualmente utilicen más recursos pero que están pensadas para resolver problemas de mayor escala.

6 Lecciones aprendidas

6.1 Abbadingo One: Limitaciones

Luego de haber logrado exitosamente la incorporación de esta técnica de generación de DFAs aleatorios al proyecto realizando inicialmente pruebas menores que utilizaban 20 autómatas de cada uno de los tamaños 8, 16 y 32 estados, en un principio la idea fue pasar a utilizar 40 autómatas para cada uno de los tamaños de 8, 16, 32, 64, 128 y 256 estados, en pos de buscar analizar casos más complejos y obtener resultados más representativos. Sin embargo, en las primeras pruebas con estos tamaños y números, las cuales se realizaron en computadoras con distintas capacidades de procesamiento y memoria RAM (una PC de uso doméstico y el servidor CECOFI de ORT), la generación de algoritmos se comportó de la siguiente manera:

- Los autómatas de 8 y 16 estados, se generaban muy rápidamente.
- Los autómatas de 32 estados, demoraron ligeramente más tiempo, pero de todas maneras eran décimas de segundo cada uno.
- Los autómatas de 64 estados, demoraron un promedio de 46 segundos cada uno en ser creados.
- La creación del primer autómata de 128 estados utilizando esta técnica, jamás llegó a culminar en un tiempo razonable.

El tiempo mayor que se le permitió al programa para crear un autómata de 128 estados fue de más de cinco horas. Hecho esto, se decidió, como primera medida, dejar de lado los casos correspondientes a estos dos tamaños utilizando los recursos

hasta aquí disponibles dado el tiempo de ejecución en semanas que esto llevaría, como mínimo. En su lugar, la segunda medida que se tomó fue restringir el estudio a los tamaños 8, 16, 32 y 64 estados, los cuales permitieron realizar los estudios aquí expuestos.

Después del análisis de estos resultados, no se descarta la posibilidad de que esta técnica, debido a su implementación subyacente, no sea escalable a medida que el tamaño de autómatas se vuelve cada vez mayor. Si bien la probabilidad de crear un autómata con un tamaño dado es no nula por construcción, no se tienen garantías respecto a lo que puede llevar el encontrarlo. Ni siquiera aumentando los recursos moviendo el marco de experimentación a la infraestructura de ORT (CECOFI) se logró culminar la generación de uno de estos autómatas en tiempo razonable.

Pese a las limitaciones de esta técnica, un gran problema que existe a la fecha de este trabajo, es la falta de *benchmarks* o puntos de referencia en este área: No existe actualmente un estándar o conjunto de *benchmarks* en la comunidad de investigadores que se pueda utilizar para la prueba y comparación de algoritmos de *machine learning*. Es por eso que hubo que elegir alguna técnica arbitraria para este trabajo. La elección de *Abbadingo One* tuvo que ver con lo sencilla que es su implementación, y con tener una versión de la misma ya existente dentro del *framework* que solamente hubo que adaptar a las necesidades presentes.

6.2 Limitaciones de las tecnologías de Python

Tras analizar, se buscó una relación aún más estrecha entre las características de cada una de las tecnologías candidatas y el alcance y los objetivos del proyecto. Algunas conclusiones que se extrajeron de esta revisión son los siguientes:

- En *multithreading* [17], el uso de una sola CPU no permite el paralelismo real, ya que los hilos se tienen que turnar para acceder al recurso. Por lo que su implementación no brindaría una ganancia considerable, menos aún teniendo en cuenta el costo de implementación dado el aumento en la complejidad del código. Se realizaron algunos intentos experimentales de implementación que no resultaron relevantes, menos aún fructíferos, para el proyecto.
- Con respecto a *multiprocessing* [17], se observó que no es tan sencillo comunicar procesos, dado que se necesita una estructura concurrente que los vincule,

como por ejemplo la Observation Table (**OT**) del proyecto, y para ello la función a llamar en cada proceso no debería ser un método de objeto sino de clase. Más allá de estas consideraciones, al igual que en *multithreading*, los intentos experimentales que se hicieron para evaluar la incorporación de esta técnica al proyecto demostraron que no brindaba valor suficiente como para justificar la dificultad en la implementación.

- En cuanto a Dask [18], pese a ser una solución de más alto nivel que las dos técnicas comentadas anteriormente, la citada documentación oficial parece indicar que es una herramienta que está muy orientada a colecciones. Con lo cual difiere del abordaje que se desea tomar.
- Modin [19], de igual manera, parece ser similar a Dask en cuanto a ser orientado a colecciones, y no es lo que se busca principalmente mejorar como objetivo de proyecto.
- De Cython [16] se puede decir que suena interesante para evaluar, **como técnica de optimización**, pero **no hace paralelismo en sí mismo** sino que pretende optimizar la ejecución de código Python a nivel general.
- Las técnicas analizadas que son para optimizar problemas más específicos, como Modin, podrían haberse considerado para casos muy puntuales del proyecto como por ejemplo el de la optimización de la librería Pandas. En un principio, al momento de la realización de la investigación de las diferentes técnicas de paralelismo, una de las hipótesis planteadas en las conclusiones fue que el uso de Pandas iba a incrementar con el paso del tiempo. Sin embargo, al momento de realizarse la revisión del documento del análisis, esta hipótesis se descartó. Se observó también que Pandas se usa para analizar datos, pero es especialmente lenta como estructura. Por propias experiencias pasadas, el uso de Pandas para otros ámbitos distinto al mencionado puede ser perjudicial.
- También es de común acuerdo que puede llegar a ser de utilidad, a futuro, la optimización de código en un plano general, no necesariamente en términos de paralelismo. Al comentarse este punto, vuelve a recordarse la posibilidad de la implementación de Cython como se dijo en un punto anterior.
- De las tecnologías de paralelismo para Python analizadas, en cuanto a técnicas de aplicación general, se llegó al acuerdo de que la opción aparentemente más propicia para implementar sería la librería Ray [20], ya que sus características permitirían poder modificar, en principio mínimamente, el código, y mediante el uso adecuado de las distintas entidades que Ray propone (a

modo de ejemplo, la especificación de *"tasks"* para funciones que se podrán ejecutar de manera paralela, *"actors"* análogamente para clases, entre otros) se podría lograr un mejor aprovechamiento de los recursos computacionales y ejecutar algoritmos de manera paralela en varias CPUs simultáneamente.

7 Conclusiones

Se presentó el algoritmo de *PAC-Learning Bounded-L*Col BatchMQ* para el aprendizaje de autómatas, que es una variante del algoritmo *Bounded-L** donde el Learner interactúa con un Teacher que contiene una red neuronal recurrente, y las *membership queries* se realizan en *batch* aprovechando la capacidad de procesamiento en *batch* que tiene esta familia de redes neuronales para ciertos tamaños de datos. Se pudo probar, para casos representativos, que este algoritmo fue consistentemente mejor que *Bounded-L**.

El algoritmo *Bounded-L*Col* no fue necesariamente mejor que *Bounded-L**, pues hubo casos de estudio donde su mejora no escaló. Por lo que no hay evidencia suficiente para afirmar que se mejoran los tiempos de ejecución solamente agregando la variación en columnas. La versión en *BatchMQ* también redujo drásticamente la varianza, lo cual es razonable ya que entre otros aspectos se redujo la cantidad de interacciones entre el Learner y el Teacher. Sin embargo, la combinación de ambas variantes parece ser la dirección más apropiada para seguir el camino de la búsqueda de mejor *performance* de este Learner, ya que se trata de una mejora algorítmica sin estar atada a una tecnología en particular, pudiendo aplicarse en el contexto de otros lenguajes de programación.

Se realizó también un estudio de técnicas de paralelismo, concurrencia y distribución existentes en Python, junto con sus ventajas y desventajas, y también su eventual grado de aplicabilidad en el proyecto *Model Extraction Framework* para la mejora de los tiempos de ejecución del algoritmo *L** y su variante *Bounded-L**. Como resultado, ninguna de las propuestas que existen de paralelización para Python parecieron adecuadas para resolver este problema, no al menos en el contexto de este proyecto. Sin embargo, no se descarta la posibilidad de que en otro contexto y alcance puedan realizarse experimentos más profundos con algunas de estas técnicas, especialmente de aquellas cuyos caminos quedaron abiertos en este trabajo, como por ejemplo el uso de Ray para la paralelización/distribución o de Cython para optimización.

Con respecto a la utilización de Python para el proyecto en sí, no corresponde decir que se trata de un problema. Si bien las debidas justificaciones sobre por qué se optó desarrollar el proyecto *Model Extraction Framework* en Python esca- pan a los objetivos de este trabajo, algunos de los motivos que se saben son lo amena que resulta la implementación para quien trabaja en el mismo, el vasto ecosistema de librerías para análisis de datos y las comunidades alrededor de las mismas, entre otros. El hecho de que Python no demostrara tener buen soporte de paralelización para casos de estudio como el del presente trabajo no implica que estos motivos no sean sólidos. Sin embargo, en algún caso podría llegar a evaluarse el re-implementar todo el proyecto y migrarlo a alguna otra tecnología que presente un mejor soporte *built-in* de paralelismo. Pero para esto se recomienda que hayan suficientes argumentos de peso para dicha migración, puesto que, debido al tamaño y la complejidad del proyecto, requeriría una inversión muy grande en recursos (tiempo de desarrollo, *testing*, entre otros).

7.1 Trabajo a futuro

Así como el algoritmo *L*Col* fue el candidato elegido para iniciar la mejora de los tiempos de ejecución de *L**, otro algoritmo que también fue inicialmente considerado es el algoritmo *TTT* [21]. Pese a esto, fue casi inmediatamente descartado debido a la complejidad que se notó que presentaba, en relación al alcance del proyecto. *L*Col*, por otra parte, además de la diferencia en la dificultad de implementación respecto a *TTT* también presentaba similitudes en el contexto de tratarse de una variación de una implementación ya existente en el *framework*.

En una etapa más avanzada del proyecto, se reevaluó la posibilidad de incorporar pruebas con este algoritmo en la medida que los tiempos permitieran, pero se decidió profundizar en el camino ya trazado de la integración con redes neuronales.

Con respecto a esto, también se consideró en su momento y se deja como trabajo futuro, evaluar algunas técnicas generativas utilizadas en otros trabajos como NFA Learning [22].

8 Referencias Bibliográficas

[1] F. Mayr and S. Yovine, “Regular inference on artificial neural networks,” in *Machine Learning and Knowledge Extraction*, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds. Cham: Springer International Publishing, 2018, pp. 350–369.

[2] F. Mayr, R. Visca, and S. Yovine, “On-the-fly black-box probably approximately correct checking of recurrent neural networks,” in *Machine Learning and Knowledge Extraction*, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds. Cham: Springer International Publishing, 2020, pp. 343–363.

[3] D. Gunning, “Explainable Artificial Intelligence (XAI)”, 2017. [Online]. Available: https://www.cc.gatech.edu/~alanwags/DLAI2016/(Gunning)%20IJCAI-16%20DLAI%20WS.pdf

[4] T. Lei, R. Barzilay, and T. Jaakkola, “Rationalizing neural predictions,” in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 107–117. [Online]. Available: https://aclanthology.org/D16-1011

[5] D. Angluin, “Learning regular sets from queries and counterexamples,” *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540187900526

[6] C. de la Higuera, *Grammatical Inference - Learning Automata and Grammars*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] F. Mayr, “Regular inference over recurrent neural networks as a method for black box explainability”, 2016. [Online]. Available: https://bibliotecas.ort.edu.uy/bibid/90543

[8] O. Maler and A. Pnueli, “On the learnability of infinitary regular sets,” Inf. Comput., Tech. Rep., 1995.

[9] Y. Bengio and P. Frasconi, “An EM approach to learning sequential behavior,” 07 1994.

[10] Q. Wang, K. Zhang, A. Ororbia II, X. Xing, X. Liu, and C. Lee Giles, “A comparison of rule extraction for different recurrent neural network models and grammatical complexity,” *CoRR*, vol. abs/1801.05420, 2018. [Online]. Available: http://arxiv.org/abs/1801.05420

[11] Massachusetts Institute of Technology, “An empirical evaluation of rule extraction from recurrent neural networks,” *Neural Comput.*, vol. 30, no. 9, 2018. [Online]. Available: https://doi.org/10.1162/neco a 01111

[12] C. Omlin and C. Giles, “Constructing deterministic finite-state automata in recurrent neural networks,” *Journal of the ACM*, vol. 43, no. 6, pp. 937–972, Nov. 1996.

[13] I. Goodfellow, Y. Bengio, and A. Courville, “Deep Learning. MIT Press”, 2016. [Online]. Available: http://www.deeplearningbook.org.

[14] K. J. Lang, B. A. Pearlmutter, and R. A. Price, “Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm,” in *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, ser. Lecture Notes in Computer Science, V. G. Honavar and G. Slutzki, Eds., vol. 1433. Springer, 1998, pp. 1–12.

[15] Pandas. “Pandas documentation”, Jan, 2021. [Online]. Available: https: //pandas.pydata.org/docs/

[16] Infoworld. “What is Cython? Python at the speed of ”, Jan, 2021. [Online]. Available: https://www.infoworld.com/article/3250299/ what-is-cython-python-At-the-speed-of-c.html

[17] Guru99. “Multithreading vs Multiprocessing: What’s the difference?”, Jan, 2021. [Online]. Available: https://www.guru99.com/ difference-between-multiprocessing-and-multithreading.html

[18] Dask. Jan, 2021. [Online]. Available: https://docs.dask.org/en/latest/

[19] Modin. “Scale your pandas workflow by changing a single line of code”, 2021. [Online]. Available: https://modin.readthedocs.io/en/latest/

[20] Ray. "What is Ray?", 2021. [Online]. Available: https://docs.ray.io/en/latest/

[21] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning," in *Bonakdarpour B., Smolka S.A. (eds) Runtime Verification: Lecture Notes in Computer Science, vol 8734*. Springer, Cham, 2014.

[22] P. M. Morales Jaurena and S. Uriarte Güimil, "Non-deterministic automata inference from Recurrent Neural Networks. Montevideo: Universidad ORT Uruguay", 2020. [Online]. Available: https://sisbibliotecas.ort.edu.uy/cgi-bin/ koha/opac-retrieve-file.pl?id=c0794ac7daa2e7eb51a8fbfea4c9184c

[23] Real Python. "Python Global Interpreter Lock", Jan, 2021. [Online]. Available: https://realpython.com/python-gil/

[24] Wikipedia. "CPython", 2021]. [Online]. Available:

https://es.wikipedia.org/wiki/CPython

[25] Towards Data SCience. "10x Faster Parallel Python Without Python Multiprocessing", 2021. [Online]. Available: https://towardsdatascience.com/ 10x-faster-parallel-python-without-Python-multiprocessing-e5017c93cce1

[26] Python. "Pyrex", Jan, 2021. [Online]. Available: https://wiki.python.org/moin/ Pyrex

[27] Data Revenue. "Scaling Pandas: Comparing Dask, Ray, Modin, Vaex, and RAPIDS", Feb, 2021]. [Online]. Available: https://www.datarevenue.com/en-blog/ pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray

[28] Geng yong. "Ray VS Dask", Feb, 2021. [Online]. Available: https://coolgeng. medium.com/ray-vs-dask-d0154a774f2a

[29] Ray-project. "Comparison to dask", Feb, 2021. [Online]. Available: https: //github.com/ray-project/ray/issues/642

9 Anexos

9.1 Anexo 1: Análisis de tecnologías de paralelismo en Python

A continuación se muestra un resumen de un estudio sobre distintas técnicas de paralelismo y optimizacion en Python, que sirvió de inspiración para la toma de algunas decisiones tomadas durante este proyecto.

9.1.1 Multithreading

El *multithreading* [17] es una técnica de ejecución de programas que permite que un mismo proceso o programa en ejecución (corriendo en una sola CPU) pueda tener múltiples hilos o *threads* de ejecución que son "hijos" de ese proceso, compartiendo su misma memoria y recursos de CPU. Las aplicaciones multi-hilo permiten la concurrencia de dos o más *threads* que se ejecutan de manera concurrente.

A continuación se destacan algunas ventajas y desventajas [17] del uso de *multithreading*. Las ventajas destacadas son:

- Los hilos comparten la misma dirección de espacio.
- Son livianos, lo cual deja baja huella en la memoria.
- El costo de comunicación entre hilos es bajo.
- El acceso al estado de memoria desde otro contexto es más fácil.
- Es una opción ideal para aplicaciones sujetas a entrada/salida.
- Los hilos son más rápidos de iniciar que los procesos, y también más rápidos al cambiar de tareas.

- Todos los hilos comparten un *pool* de memoria del proceso, que es muy beneficioso.

Sin embargo, algunas desventajas que esta tecnolog´ıa presenta son:

- El sistema de *multithreading* no es interrumpible.
- El código usualmente es más difícil de entender y se incrementa dramáticamente el potencial de ocurrencia de *race conditions*.
- Si no se sigue una cola de comandos y un modelo de env´ıo o *pump* de mensajes, entonces se vuelve una necesidad el uso manual de la sincron´ıa, lo cual aumenta la dificultad de implementación.

9.1.2 Multiprocessing

El *multiprocessing* [17] o multiprocesamiento es una técnica de ejecución de programas que consiste en utilizar no una, sino dos o más CPUs para realizar una determinada tarea. Las CPUs son agregadas al sistema para incrementar la velocidad de cómputo del mismo. Cada CPU tiene su propio conjunto de registros y memoria principal.

A continuación se destacan algunas ventajas y desventajas [17] del uso de *multiprocessing*. Las ventajas destacadas son:

- La ventaja más grande de un sistema multiprocesador es que ayuda a realizar más trabajo en un tiempo más corto.
- El código usualmente es bastante directo de entender e implementar.
- Toma ventaja de múltiples CPUs y núcleos.
- Ayuda a evitar limitaciones del Python GIL [23] para CPython [24].
- Remueve primitivas de sincronización a menos que se utilice memoria compartida.
- Los procesos hijos son usualmente interrumpibles.
- Este tipo de sistemas debería ser utilizado cuando se requiere muy alta velocidad para procesar un gran volumen de datos.

- Los sistemas de *multiprocessing* ahorran dinero en comparación con los monoprocesadores, dado que los procesadores pueden compartir periféricos y suministros de energía.

Algunas desventajas:

- La comunicación entre procesos o IPC (*Inter-Process Communication*) es algo más complicada que la comunicación entre hilos, con algo más de sobrecarga u *overhead* de trabajo para realizarla.
- Deja una mayor huella de memoria.

9.1.3 Ray

Ray [20, 25] es un *framework* rápido y simple que provee una API simple y universal para construir y ejecutar aplicaciones distribuídas. Cumple esta misión mediante dos características:

1. Provee primitivas simples para construir y ejecutar aplicaciones distribuídas, permitiendo a los usuarios paralelizar código de máquina singular, con pocos o nulos cambios en el código.
2. Incluye un amplio ecosistema de aplicaciones, librerías y herramientas que funcionan sobre el núcleo de Ray para posibilitar la implementación de aplicaciones complejas.

9.1.4 Dask

Dask [18] es una librería flexible para computación paralela en Python. Está compuesto por dos partes:

1. *Scheduling* dinámico de tareas optimizado para computación.
2. Colecciones de “Big Data” como arreglos paralelos, *dataframes* y listas que extienden interfaces comunes como *NumPy*, Pandas, o iteradores de Python a ambientes distribuídos.

Dask enfatiza las siguientes virtudes:

- Familiar: Provee versiones paralelas de los conocidos arreglos de *NumPy* y los objetos *DataFrame* de Pandas.
- Flexible: Provee una interfaz de planificación de tareas para cargas de trabajo más personalizadas e integración con otros proyectos.
- Nativo: Habilita la computación distribuida en Python puro con acceso al stack PyData.
- Rápido: Opera con bajo *overhead*, baja latencia, y serialización mínima necesaria para algoritmos numéricos rápidos.
- Escalabilidad hacia arriba: Funciona perfectamente en clusters con miles de núcleos.
- Escalabilidad hacia abajo: Es trivial setearlo para correr en una laptop en un solo proceso.
- Responsivo: Diseñado con la computación interactiva en mente, provee *feedback* y diagnósticos rápidamente para asesorar a las personas.

9.1.5 Modin

"Scale your pandas workflow by changing a single line of code".

Modin [19] utiliza Ray o Dask para proveer una forma de acelerar las notebooks, scripts y librerías de Pandas. Como ninguna otra librería de *dataframes*, Modin provee integración sin problemas y compatibilidad con código de Pandas ya existente. Incluso el uso del constructor DataFrame es idéntico.

9.1.6 Cython

Cython [16] es un compilador estático de optimización para el lenguaje Python y su versión extendida Cython (basado en *Pyrex* [26]). Permite escribir extensiones de C para Python tan fácil como Python mismo. Cython provee la combinación del poder de Python y C para poder:

- escribir código Python que se puede traducir a código C o C++, y viceversa, en cualquier punto.

- fácilmente modificar código Python legible en plena eficiencia de C agregando declaraciones de tipos estáticos, también en sintaxis de Python.
- utilizar un nivel de debugging de la combinación de códigos fuente, para poder encontrar bugs en el código de Python, Cython y C.
- interactuar eficientemente con grandes conjuntos de datos, por ejemplo con arreglos multidimensionales de *NumPy*.
- rápidamente construir aplicaciones dentro del amplio, maduro y ampliamente utilizado ecosistema de CPython.
- integrar nativamente con código y datos existentes de librerías y aplicaciones legado, de alta y de baja eficiencia.

9.1.7 Algunas interrogantes

9.1.7.1 ¿Ray o Dask?

Distintas fuentes [27, 28] sostienen que en realidad se tratan de herramientas bastante similares en un montón de aspectos. Al parecer, la diferencia principal está en la estrategia de planificación o *scheduling* que utilizan ambas herramientas para compartir el trabajo entre diferentes procesadores, como se discute también en un hilo en *GitHub* [29] donde los creadores de ambas herramientas comentan las diferencias entre ellas. Otro punto, mas no menor, es que si se considera también la implementación de Modin, cabe recordar que Modin puede funcionar utilizando tanto el motor de Ray como el de Dask. Sin embargo, pese a todo, la documentación oficial de Dask parece indicar que Dask está más centrado en la mejoras en la performance de distintas librerías de Python de público conocimiento en la ciencia de datos, como *NumPy*, Pandas y ScikitLearn.

9.1.7.2 ¿Modin en lugar de Dask?

Dask provee también optimizaciones para las distintas herramientas de Pandas, al igual que Modin. Dask, al mismo tiempo, provee un conjunto mayor de otras posibles herramientas de optimización, caracterizándose por su énfasis en la mejora de las tareas en el mundo de la ciencia de datos. Por lo que, de momento, no se descarta la implementación de Dask en el proyecto. De hecho, como ya

se mencionó, se podría utilizar tanto esta herramienta como Ray. Sin embargo, se propone implementar Modin por el mero hecho de su facilidad de instalación y configuración dentro del proyecto. Según las fuentes, Dask, no obstante, puede llevar algunas modificaciones extra en el código para su adecuado funcionamiento.

9.1.8 Pasos a seguir: Sugerencia

En base a los puntos considerados recientemente, y teniendo en cuenta la naturaleza y objetivo del proyecto, luego de estudiadas estas opciones se propusieron algunos pasos a seguir:

1. En pos de alcanzar el paralelismo y eventualmente también la distribución del algoritmo, resulta indispensable la implementación de una técnica que pueda utilizarse de manera general. Se propone utilizar las herramientas Ray o Dask para este propósito.

2. Para la optimización de las herramientas de Pandas, se propone también emplear el uso de Modin, que además tiene la característica de poderse instalar de manera realmente sencilla y directa.

3. Para la optimización de código Python, que puede ser tanto referente a regiones que involucren paralelismo como a cualquier parte, se puede considerar la implementación de Cython para acelerar su velocidad de ejecución. No es una solución "mágica" y presenta también sus limitantes [16], pero teniendo éstas en cuenta, se podría incluir en el proceso de optimización.

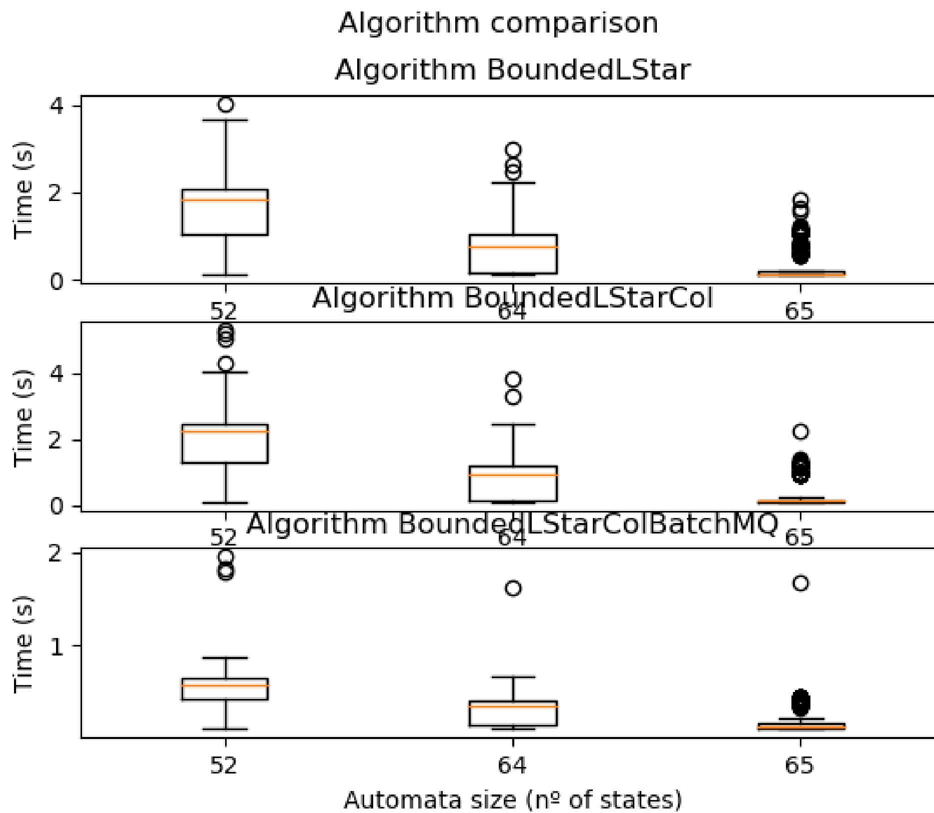
9.2 Anexo 2: Experimentos realizados con *Bounded-L*Col BatchMQ*

9.2.1 Experimento 1

Tamaños utilizados: 52, 64 y 65 estados.

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
52	Time (s)	1.8504 ± 1.05467	2.2454 ± 1.15859	**0.5825 ± 0.22302**
52	Size (output)	4.5 ± 1.25	4.5 ± 1.25	4.5 ± 1.25
52	Nº of EQs	2.5 ± 1	3 ± 1	3 ± 1
52	Exceeded nº states	0	0	0
52	Exceeded max MQ length	0	0	0
64	Time (s)	0.7420 ± 0.89756	0.9419 ± 1.02266	**0.3571 ± 0.25984**
64	Size (output)	3 ± 1.5	3 ± 1.5	3 ± 1.5
64	Nº of EQs	2 ± 1	2 ± 1	2 ± 1
64	Exceeded nº states	0	0	0
64	Exceeded max MQ length	0	0	0
65	Time (s)	0.1317 ± 0.06386	0.1251 ± 0.05167	**0.1244 ± 0.05065**
65	Size (output)	1 ± 1	1 ± 1	1 ± 1
65	Nº of EQs	1 ± 0	1 ± 0	1 ± 0
65	Exceeded nº states	0	0	0
65	Exceeded max MQ length	0	0	0

Gráfica de los resultados:



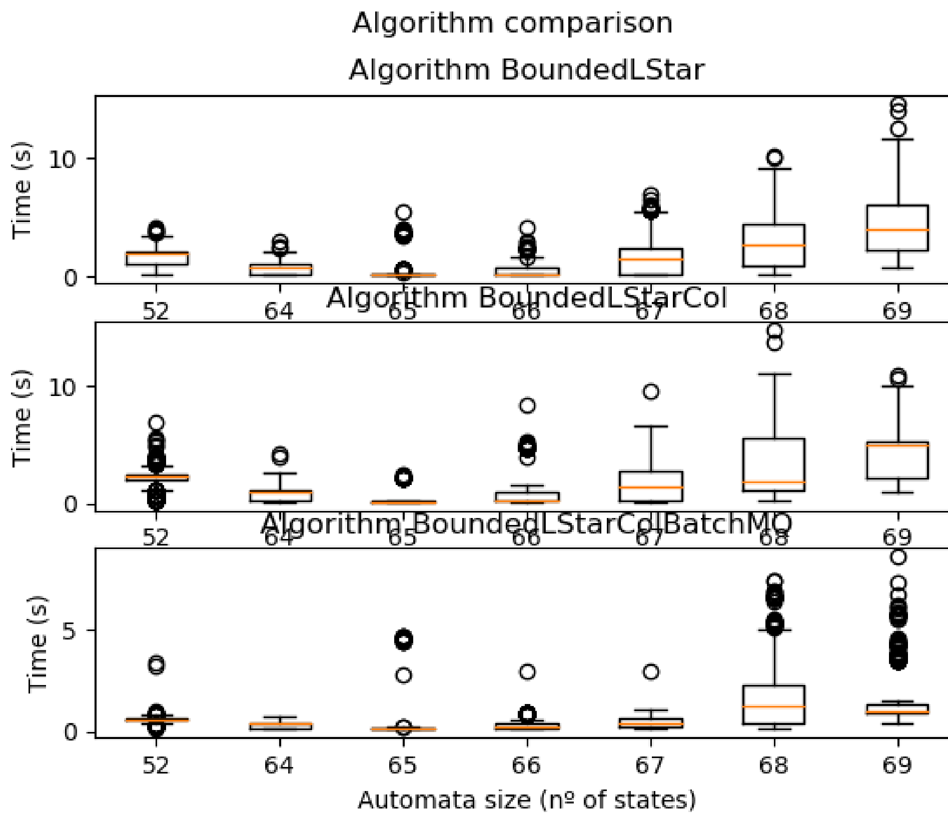
A diferencia de la aparente mejora vista en los primeros experimentos mostrados en la sección 3.2.3, en estos primeros resultados con los autómatas de mayor tamaño, la mejora que se aprecia en los tiempos de ejecución es significativa. En primer lugar, se puede ver lo antes mencionado sobre la mejora de *Bounded-L*Col BatchMQ* en la varianza, dado que para todos los casos es radicalmente menor que la de sus contrapartes. Al mismo tiempo, para el tamaños 52 y 64 estados se puede ver una mejora de más de tres y hasta casi cuatro veces más rápido respectivamente. No obstante, para el tamaño 65 la mejora en los tiempos no fue significativa.

9.2.2 Experimento 2

Tamaños utilizados: 52, 64, hasta 69 estados.

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
52	Time (s)	1.8946 ± 0.98697	2.3112 ± 0.54687	**0.5992 ± 0.11386**
52	Size (output)	5 ± 1	5 ± 1	5 ± 1
52	Nº of EQs	3 ± 1	3 ± 0	3 ± 0
52	Exceeded nº states	0	0	0
52	Exceeded max MQ length	0	0	0
64	Time (s)	0.7569 ± 0.88724	0.9590 ± 1.00746	**0.3654 ± 0.24619**
64	Size (output)	3 ± 1.25	3 ± 1.25	3 ± 1.25
64	Nº of EQs	2 ± 1	2 ± 1	2 ± 1
64	Exceeded nº states	0	0	0
64	Exceeded max MQ length	0	0	0
65	Time (s)	0.1263 ± 0.04767	**0.1236 ± 0.04120**	0.1254 ± 0.04175
65	Size (output)	1 ± 0.25	1 ± 0.25	1 ± 0.25
65	Nº of EQs	1 ± 0	1 ± 0	1 ± 0
65	Exceeded nº states	0	0	0
65	Exceeded max MQ length	0	0.05	0
66	Time (s)	0.1829 ± 0.58825	**0.1816** ± 0.80331	0.1846 ± **0.19925**
66	Size (output)	2 ± 1	2 ± 1	2 ± 1
66	Nº of EQs	1 ± 1	1 ± 1	1 ± 1
66	Exceeded nº states	0	0	0
66	Exceeded max MQ length	0	0	0
67	Time (s)	1.4303 ± 2.13606	1.3483 ± 2.60645	**0.4297 ± 0.47603**
67	Size (output)	3.5 ± 3	3.5 ± 3	3.5 ± 3
67	Nº of EQs	2.5 ± 2	2 ± 2	2 ± 2
67	Exceeded nº states	0	0	0
67	Exceeded max MQ length	0	0	0
68	Time (s)	2.5972 ± 3.48955	1.9047 ± 4.34824	**1.2165 ± 1.85897**
68	Size (output)	7 ± 8.5	3.5 ± 4.5	12 ± 20.25
68	Nº of EQs	2 ± 1.25	2 ± 1	3 ± 2
68	Exceeded nº states	0	0	0
68	Exceeded max MQ length	0.2	0.6	0
69	Time (s)	4.0249 ± 3.83407	5.0069 ± 3.26316	**0.9983 ± 0.43174**
69	Size (output)	7 ± 2.25	7 ± 5	8.5 ± 8
69	Nº of EQs	3 ± 1.25	2 ± 3	4 ± 1
69	Exceeded nº states	0	0	0
69	Exceeded max MQ length	0.3	0.5	0

Gráfica de los resultados:



En los resultados de este experimento se puede ver que la mejora en los tamaños 52 y 64 estados se mantiene, a razón de iguales proporciones que anteriormente. Análogamente, se mantiene la despreciable diferencia para el tamaño 65 estados.

Para el caso de 66 estados, en principio hay una leve desmejora frente a *Bounded-L** y *Bounded-L*Col*, pero además de ser casi despreciable se puede ver también que la varianza para estos dos algoritmos es bastante mayor que para *Bounded-L*Col BatchMQ*, con lo cual la desmejora no es concluyente.

Para los casos de 67, 68 y 69 estados, nuevamente se ven mejoras significativas en los tiempos promedio. Y si bien las varianzas de todos los casos aumentan, la del tercer algoritmo sigue siendo radicalmente menor que la de los primeros dos.

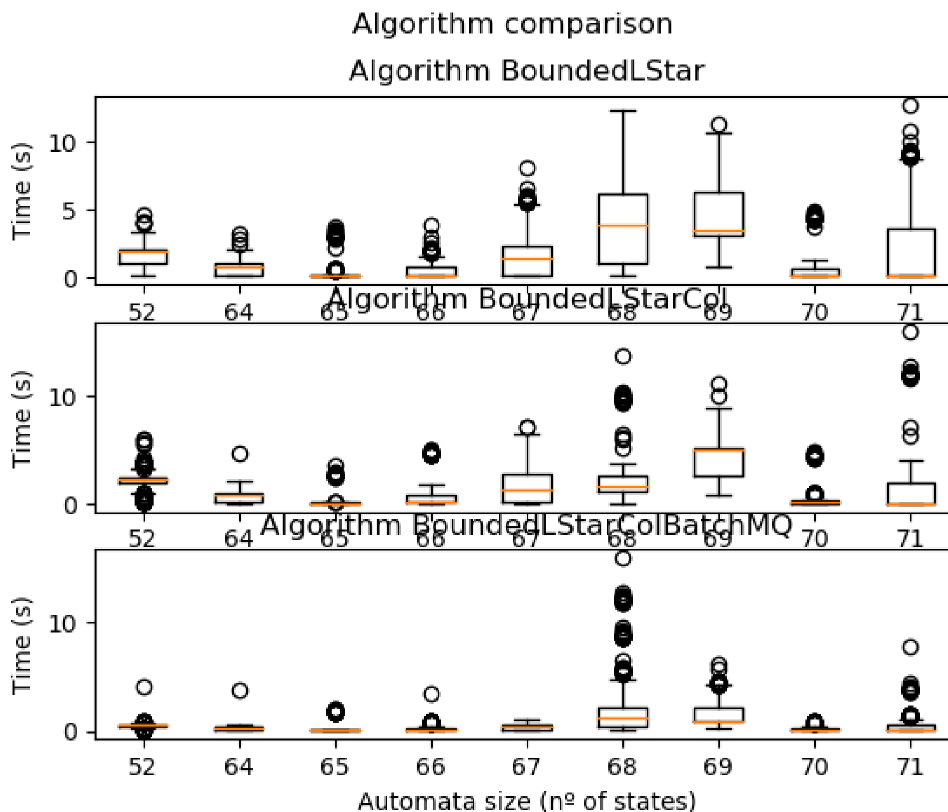
9.2.3 Experimento 3

Tamaños utilizados: 52, 64, hasta 71 estados.

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
52	Time (s)	1.8918 ± 1.00988	2.3034 ± 0.55485	**0.5893 ± 0.11293**
52	Output size (states)	5 ± 1	5 ± 1	5 ± 1
52	Nº of EQs	3 ± 1	3 ± 0	3 ± 0
52	Exceeded nº states (%)	0	0	0
52	Exceeded max MQ length (%)	0	0	0
64	Time (s)	0.7458 ± 0.88474	0.9462 ± 0.93806	**0.3623 ± 0.23944**
64	Output size (states)	3 ± 1.25	3 ± 1.25	3 ± 1.25
64	Nº of EQs	2 ± 1	2 ± 1	2 ± 1
64	Exceeded nº states (%)	0	0	0
64	Exceeded max MQ length (%)	0	0	0
65	Time (s)	0.1277 ± 0.04927	0.1262 ± 0.04049	**0.1210 ± 0.04157**
65	Output size (states)	1 ± 0.25	1 ± 0.25	1 ± 0.25
65	Nº of EQs	1 ± 0	1 ± 0	1 ± 0
65	Exceeded nº states (%)	0	0	0
65	Exceeded max MQ length (%)	0	0	0
66	Time (s)	0.1838 ± 0.58775	0.1808 ± 0.78525	**0.1782 ± 0.20606**
66	Output size (states)	2 ± 1	2 ± 1	2 ± 1
66	Nº of EQs	1 ± 1	1 ± 1	1 ± 1
66	Exceeded nº states (%)	0	0	0
66	Exceeded max MQ length (%)	0	0	0
67	Time (s)	1.4817 ± 2.12139	1.3819 ± 2.61188	**0.4256 ± 0.46727**
67	Output size (states)	3.5 ± 3	3.5 ± 3	3.5 ± 3
67	Nº of EQs	2.5 ± 2	2 ± 2	2 ± 2
67	Exceeded nº states (%)	0	0	0
67	Exceeded max MQ length (%)	0	0	0
68	Time (s)	3.9096 ± 5.07638	1.7631 ± 1.45113	**1.2543 ± 1.76370**
68	Output size (states)	8 ± 8.5	2 ± 2.5	12.5 ± 21.25
68	Nº of EQs	2 ± 1.25	1 ± 1	3 ± 1.25
68	Exceeded nº states (%)	0	0	0
68	Exceeded max MQ length (%)	0	0	0

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
69	Time (s)	3.4441 ± 3.18127	5.0022 ± 2.59761	**0.9712 ± 1.34976**
69	Output size (states)	7.5 ± 2.25	7 ± 3.25	8.5 ± 26.25
69	Nº of EQs	4 ± 1	2 ± 2	4 ± 1
69	Exceeded nº states (%)	0	0	0
69	Exceeded max MQ length (%)	0	0	0
70	Time (s)	0.1704 ± 0.60271	**0.1640** ± 0.25614	0.1647 ± **0.13664**
70	Output size (states)	2 ± 1.25	2 ± 1.25	2 ± 1.25
70	Nº of EQs	1 ± 0.25	1 ± 0.25	1 ± 0.25
70	Exceeded nº states (%)	0	0	0
70	Exceeded max MQ length (%)	0	0	0
71	Time (s)	0.1392 ± 3.46779	0.1323 ± 1.96141	**0.1308 ± 0.45903**
71	Output size (states)	1 ± 4.75	1 ± 1	1 ± 4.75
71	Nº of EQs	1 ± 1	1 ± 0	1 ± 1
71	Exceeded nº states (%)	0	0	0
71	Exceeded max MQ length (%)	0	0	0

Gráfica de los resultados:



En el presente experimento, como se puede ver en cada caso, cada uno de los tamaños 52 hasta 69 inclusive mantienen sus correspondientes cambios respecto a los experimentos anteriores. En los nuevos casos de estudio, los tamaños 70 y 71 estados muestran que en los tiempos promedio no hay una aparente mejora relevante, dado que la diferencia es muy pequeña. Y si bien la varianza para *Bounded-L*Col BatchMQ* es notoriamente menor como en los casos anteriores, es suficiente para que exista solapamiento entre los rangos de valores de los tiempos entre los distintos algoritmos (similar a los experimentos iniciales vistos en la sección 3.2.3), con lo cual aquí la mejora tampoco es concluyente.

Como se vio en la sección 4.5, lo interesante en el caso de 68 y 69 estados es que por cuestiones aleatorias termina haciendo más **EQ**s, pero aún así su velocidad sigue siendo muy superior. Puede apreciarse una varianza grande en la cantidad de estados de los autómatas de salida, lo cual se conjetura que puede deberse a errores en el aprendizaje de las redes. Este aspecto escapa al alcance de este proyecto.

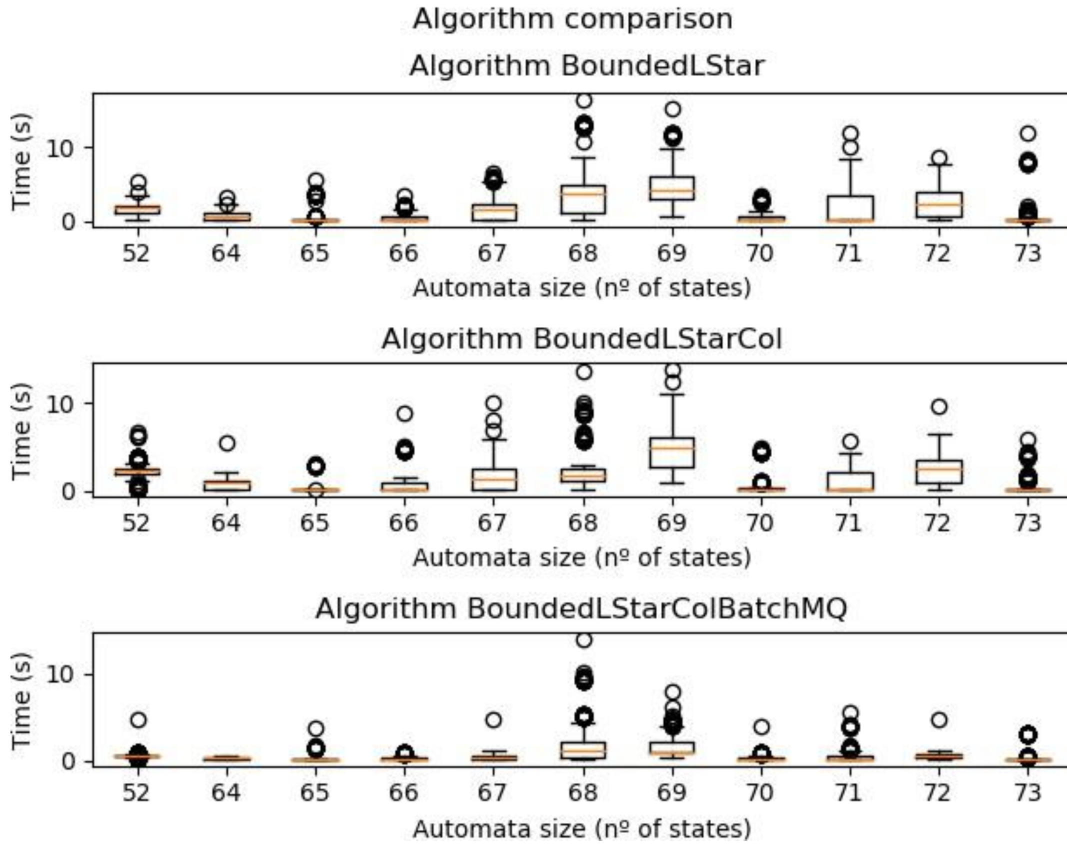
9.2.4 Experimento 4

Tamaños utilizados: 52, 64, hasta 73 estados.

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
52	Time (s)	1.8806 ± 1.02398	2.2914 ± 0.56074	**0.5892** ± **0.11530**
52	Output size (states)	5 ± 1	5 ± 1	5 ± 1
52	Nº of EQs	3 ± 1	3 ± 0	3 ± 0
52	Exceeded nº states (%)	0	0	0
52	Exceeded max MQ length (%)	0	0	0
64	Time (s)	0.7461 ± 0.87243	0.9410 ± 0.97874	**0.3641** ± **0.24934**
64	Output size (states)	3 ± 1.25	3 ± 1.25	3 ± 1.25
64	Nº of EQs	2 ± 1	2 ± 1	2 ± 1
64	Exceeded nº states (%)	0	0	0
64	Exceeded max MQ length (%)	0	0	0
65	Time (s)	0.1258 ± 0.04718	0.1268 ± 0.04010	**0.1208** ± **0.04268**
65	Output size (states)	1 ± 0.25	1 ± 0.25	1 ± 0.25
65	Nº of EQs	1 ± 0	1 ± 0	1 ± 0
65	Exceeded nº states (%)	0	0	0
65	Exceeded max MQ length (%)	0	0	0
66	Time (s)	0.1825 ± 0.58461	**0.1736** ± 0.78466	0.1798 ± **0.20811**
66	Output size (states)	2 ± 1	2 ± 1	2 ± 1
66	Nº of EQs	1 ± 1	1 ± 1	1 ± 1
66	Exceeded nº states (%)	0	0	0
66	Exceeded max MQ length (%)	0	0	0
67	Time (s)	1.7027 ± 2.15149	1.3161 ± 2.40768	**0.4244** ± **0.45870**
67	Output size (states)	3.5 ± 3	3.5 ± 3	3.5 ± 3
67	Nº of EQs	2.5 ± 2	2 ± 2	2 ± 2
67	Exceeded nº states (%)	0	0	0
67	Exceeded max MQ length (%)	0	0	0
68	Time (s)	2.5045 ± 5.58882	1.7408 ± 1.74905	**1.1970** ± **1.82176**
68	Output size (states)	8 ± 7.75	2 ± 4	12 ± 21.5
68	Nº of EQs	2 ± 1.5	1 ± 1	3 ± 1.25
68	Exceeded nº states (%)	0	0	0
68	Exceeded max MQ length (%)	0	1	0
69	Time (s)	3.1759 ± 2.45409	4.8189 ± 2.03064	**0.9255** ± **1.20110**
69	Output size (states)	7 ± 2	7 ± 4.25	8.5 ± 23.5
69	Nº of EQs	4 ± 2	2.5 ± 2.25	4 ± 1
69	Exceeded nº states (%)	0	0	0
69	Exceeded max MQ length (%)	0	0	0
70	Time (s)	0.1685 ± 0.54772	0.1649 ± 0.78916	**0.1627** ± **0.12210**
70	Output size (states)	2 ± 1.25	2 ± 1.25	2 ± 1.25
70	Nº of EQs	1 ± 0.25	1 ± 0.25	1 ± 0.25
70	Exceeded nº states (%)	0	0	0
70	Exceeded max MQ length (%)	0	0	0

Size	Metric	BL*	BL*Col	BL*Col-BatchMQ
71	Time (s)	0.1425 ± 3.08404	**0.1307** ± 1.88319	0.1318 ± **0.50315**
71	Output size (states)	1 ± 3.25	1 ± 3.25	1 ± 4.75
71	Nº of EQs	1 ± 1	1 ± 0	1 ± 1
71	Exceeded nº states (%)	0	0	0
71	Exceeded max MQ length (%)	0	0	0
72	Time (s)	2.2586 ± 3.17502	2.5484 ± 2.94593	**0.6302** ± **0.40703**
72	Output size (states)	5 ± 3.25	5 ± 3.25	5 ± 3.25
72	Nº of EQs	3 ± 2.25	3 ± 1.25	3 ± 1.25
72	Exceeded nº states (%)	0	0	0
72	Exceeded max MQ length (%)	0	0	0
73	Time (s)	0.1286 ± 0.07739	**0.1239** ± 0.05332	0.1264 ± **0.05225**
73	Output size (states)	1 ± 1	1 ± 1	1 ± 1
73	Nº of EQs	1 ± 0	1 ± 0	1 ± 0
73	Exceeded nº states (%)	0	0	0
73	Exceeded max MQ length (%)	0	0	0

Gráfica de los resultados:



Finalmente, para este nuevo experimento, nuevamente las tendencias de mejora o mantenimiento de los tiempos según corresponda, se mantiene para todos los casos mencionados, en los tamaños de 52 a 71 estados.

Para el caso de tamaño 72 estados, vuelve a verse una amplia mejora en el tiempo promedio de ejecución de *Bounded-L*Col BatchMQ*, con una dispersión también mucho menor que la de sus contrapartes. Sin embargo, para el tamaño 73 estados una vez más se percibe una semejanza en los valores obtenidos, tanto en los tiempos promedio como en las varianzas.