UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Modelos de lenguaje para la representación de estados en planificación automática

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: Dorado Javier, Víctor

Tutor/a: Onaindia de la Rivaherrera, Eva

Director/a Experimental: Aso Mollar, Ángel

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Modelos de lenguaje para la representación de estados en planificación automática

TRABAJO FIN DE MÁSTER

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Víctor Dorado Javier

Tutor: Eva Onaindia de la Rivaherrera

Curso 2023-2024

Resum

Històricament, els models de llenguatge s'han utilitzat per a resoldre una gran quantitat de desafiaments en el camp de l'aprenentatge automàtic. No obstant això, en els últims anys, gràcies a l'aparició de noves arquitectures de xarxes neuronals com els *transformers*, millores en les tècniques d'entrenament i un increment en el poder computacional, la popularitat d'esta mena de models està en auge.

L'objectiu d'este treball és aplicar tècniques de modelatge del llenguatge a problemes de planificació amb intel·ligència artificial. Específicament, es pretén entrenar una sèrie de models *transformer* per a convertir estats del món descrits en llenguatge *PDDL* en format text a vectors característics que puguen ser usats posteriorment en tasques de planificació. Els dominis de *blocksworld*, *logistics* i *floortile* s'han utilitzat com a base per a l'entrenament i avaluació dels models.

Els vectors característics generats tenen com a finalitat substituir tècniques tradicionals de representació d'estats, com la codificació *one-hot*, millorant així el rendiment del procés de planificació i reduint la petjada de memòria.

Paraules clau: Planificació Automàtica, Representació d'estats, Models de Llenguatge, Cerca

Resumen

Históricamente, los modelos de lenguaje se han utilizado para resolver una gran cantidad de desafíos en el campo del aprendizaje automático. Sin embargo, en los últimos años, gracias a la aparición de nuevas arquitecturas de redes neuronales como los *transformers*, mejoras en las técnicas de entrenamiento y un incremento en el poder computacional, la popularidad de este tipo de modelos está en auge.

El objetivo de este trabajo es aplicar técnicas de modelado del lenguaje a problemas de planificación con inteligencia artificial. Específicamente, se pretende entrenar una serie de modelos *transformer* para convertir estados del mundo descritos en lenguaje *PDDL* en formato texto a vectores característicos que puedan ser utilizados posteriormente en tareas de planificación. Los dominios de *blocksworld*, *logistics* y *floortile* se han utilizado como base para el entrenamiento y evaluación de los modelos.

Los vectores característicos generados tienen como objetivo sustituir técnicas tradicionales de representación de estados, como la codificación *one-hot*, mejorando así el rendimiento del proceso de planificación y reduciendo la huella de memoria.

Palabras clave: Planificación Automática, Representación de estados, Modelos de Lenguaje, Búsqueda

Abstract

Historically, language models have been used to solve a variety of machine learning challenges. However, in recent years, thanks to the emergence of new neural network architectures such as *transformers*, improvements in training techniques, and an increase in computational power, the popularity of this type of models is booming.

The goal of this work is to apply language modeling techniques to artificial intelligence planning problems. Specifically, we intend to train a set of *transformer* models to convert *PDDL* states in text format into feature vectors that can later be used in planning

tasks. The domains *blocksworld*, *logistics* and *floortile* were used as a basis for training and evaluation of the models.

The generated feature vectors are intended to replace traditional state representation techniques, such as *one-hot* encoding, thus improving the performance of the planning process and reducing the memory footprint.

Key words: Automatic Planning, State Embeddings, Language Models, Search

Índice general

Apéndices

Índice de figuras

Índice de tablas

"El que controla el pasado -decía el slogan del Partido-, controla también el futuro. El que controla el presente, controla el pasado."

- George Orwell, *1984*

CAPÍTULO 1
Introducción

El propósito de este capítulo es transmitir una visión general de la motivación y los objetivos de este trabajo de fin de máster, así como presentar la estructura de la memoria.

1.1 Motivación

Una gran variedad de problemas requiere de la aplicación de un proceso de búsqueda para encontrar una solución y en muchos de ellos la solución no es fácilmente alcanzable, especialmente si buscamos una solución óptima. Algunos ejemplos incluyen las rutas de reparto para una flota de vehículos, la secuencia de movimientos de contenedores de carga o la coordinación logística en sistemas con múltiples recursos y medios de transporte.

Un esquema de resolución para este tipo de problemas es mediante procesos de búsqueda en un espacio de estados, donde un nodo del espacio de búsqueda representa un estado o situación del problema, y las aristas entre dos nodos representan la transformación de un estado del problema en otro mediante la aplicación de una acción.

Un algoritmo de búsqueda en un espacio de estados requiere una representación adecuada de los posibles estados del mundo. Tradicionalmente, se utilizan representaciones simbólicas para llevar a cabo estas tareas de búsqueda o planificación pero existen ciertas tareas que requieren del uso de modelos neuronales y estos sí que necesitan representaciones vectoriales densas para poder funcionar adecuadamente.

Cuando el espacio de búsqueda es muy grande, las representaciones simbólicas no son apropiadas porque los algoritmos de búsqueda no pueden manejar tanta información de forma eficiente. Esto es especialmente relevante en problemas de gran tamaño donde los estados tienen que representar información de un gran número de objetos y relaciones entre ellos.

Otra cosa a tener en cuenta es que, en un proceso de búsqueda, los diferentes estados del espacio suelen ser muy similares, al menos para aquellos estados que se encuentren próximos entre sí, debido a que los cambios que produce una acción en el estado se limitan únicamente a una pequeña parte del estado, haciendo que la representación simbólica sea una forma muy ineficiente de representar la información de los estados.

En esos casos, se usan modelos neurosimbólicos para generar dichas representaciones vectoriales densas. Sin embargo, estos métodos suelen ser costosos en términos de cómputo y tamaño, lo que incrementa el coste de su utilización. En el campo de la planificación automática, es fundamental tener una representación eficiente y compacta de los estados del mundo para optimizar el proceso de búsqueda de soluciones.

La representación de los estados juega un papel muy importante, ya que estas representaciones son utilizadas por los sistemas de planificación para navegar el conjunto de estados posibles. Representaciones tradicionales como la codificación *one-hot* son intuitivas y fáciles de entender, pero pueden ser ineficientes en cuanto al espacio que requiere almacenarlos y el cómputo necesario para generarlos, especialmente cuando se trabaja con dominios muy complejos o problemas de gran tamaño.

1.2 Objetivos

El objetivo de este trabajo es explorar el uso de modelos de lenguaje, específicamente los modelos *transformer* [29], como alternativa a otros métodos de representación de estados en planificación automática.

Para acometer los objetivos del trabajo, se realizará una tarea de pre-entrenamiento sobre una serie de modelos *transformer* con el objetivo de que estos tengan la capacidad de entender estados descritos mediante un lenguaje simbólico, concretamente el lenguaje PDDL (*Planning Domain Description Language*) [14, 4] que se utiliza habitualmente para modelar problemas de planificación. De este modo, el objetivo es convertir estados modelados simbólicamente en representaciones vectoriales densas.

Los vectores obtenidos podrán ser utilizados posteriormente en tareas de planificación, con la expectativa de mejorar el rendimiento de los algoritmos de planificación y reducir la huella de memoria de los mismos. Los dominios PDDL que se han utilizado en estos experimentos son "mundo de bloques" o *blocksworld* en inglés, *logistics* y *floortile*.

1.3 Estructura de la memoria

La memoria se estructura de la siguiente manera:

- **Capítulo 1 Introducción**: Presenta la motivación, el objetivo y la estructura del trabajo.
- **Capítulo 2 Estado de la cuestión**: Revisión de la literatura y conceptos relevantes en planificación, modelos neurosimbólicos y modelos de lenguaje.
- **Capítulo 3 Diseño de la solución**: Descripción detallada del diseño y arquitectura del sistema, incluyendo la generación, formato y análisis de los datos.
- **Capítulo 4 Pruebas experimentales**: Detalles sobre los experimentos realizados, incluyendo hardware y software utilizados, modelos entrenados, ajuste de hiperparámetros y resultados obtenidos.
- **Capítulo 5 Conclusiones**: Limitaciones y desafíos que hemos encontrado y posibles líneas de trabajo futuro.

CAPÍTULO 2
Estado de la cuestión

En este capítulo haremos una revisión de la literatura existente y los conceptos clave relacionados con la planificación automática, los modelos neurosimbólicos y los modelos de lenguaje.

Comenzaremos con una introducción a la planificación automática, explicando su importancia y aplicaciones. Luego seguiremos con los modelos neurosimbólicos, destacando los avances recientes y su relevancia en el contexto de la planificación. Y finalmente, trataremos los modelos de lenguaje, con un enfoque particular en su evolución y aplicación en la representación de estados. Este capítulo establece la base teórica y el marco conceptual sobre el cual se ha desarrollado este trabajo.

2.1 Planificación

La planificación automática se refiere al problema de identificar el conjunto de acciones que, aplicadas a un estado inicial, resultan en un estado final donde se satisface el cumplimiento de uno o varios objetivos [7]. En planificación clásica se trabaja con diferentes tipos de modelos de planificación como los basados en autómatas regulares, gramáticas libres de contexto o máquinas de estado finito. Uno de los más populares es el modelo de acciones STRIPS [3], considerado como el modelo de planificación de referencia que ha dado lugar a muchos de los trabajos de investigación en planificación automática. STRIPS es también el nombre con el que se denomina al lenguaje de planificación de referencia que posteriormente evolucionó al conocido PDDL. Para el propósito de este trabajo, que es explorar el uso de modelos de lenguaje para la representación de estados de planificación, asumiremos el modelo de planificación STRIPS.

Una tarea de planificación se describe con dos elementos principales, el dominio y el problema. Un **dominio de planificación** se define como $D = \langle P, O \rangle$, donde P es un conjunto de predicados de primer orden que se utiliza para describir las propiedades de los objetos del dominio y sus relaciones, y O es el conjunto de operadores o esquemas de acciones. Un operador $o \in O$ se define por una tripleta $\langle \text{Pre}(o), \text{Add}(o), \text{Del}(o) \rangle$, donde $\text{Pre}(o), \text{Add}(o), \text{Del}(o) \subseteq P$ son las precondiciones, efectos positivos y efectos negativos del operador o, respectivamente.

Un conocido dominio de planificación es el mundo de bloques o *blocksworld* (en inglés) que puede encontrarse en el Apéndice A. Este dominio describe un conjunto de bloques dispuestos sobre una mesa en una determinada configuración y las operaciones que pueden realizarse en el dominio son desapilar un bloque de otro bloque, apilar un bloque sobre otro, coger un bloque de la mesa y dejar un bloque, mediante un brazo de robot. El conjunto P se compone de cinco predicados: el predicado binario `(on ?x ?y)`

para indicar que un bloque está encima de otro bloque; los predicados unarios `(ontable ?x)`, `(clear ?x)`, `(holding ?x)` que representan si un bloque está encima de la mesa, si un bloque está libre (no tiene bloques encima y el robot no lo tiene), y si el bloque lo tiene el robot, respectivamente; y el predicado sin argumentos `(handempty)` para indicar que el brazo de robot no está sujetando un bloque.

Un **problema de planificación** asociado a un dominio $D = \langle P, O \rangle$ se define como una tupla $PB = \langle F, A, I, G \rangle$ donde F es el conjunto de hechos o literales (*facts*) que resultan de instanciar los predicados P del dominio con los objetos del problema; A es el conjunto de acciones que resulta de instanciar los operadores O del dominio con los objetos del problema; I es el estado inicial del problema y G es el conjunto de objetivos que se desea alcanzar. Un **estado de planificación** $s \subseteq F$ es un conjunto de literales positivos. Asumimos la semántica de negación por fallo o de mundo cerrado de modo que el conjunto de literales que no aparece explícitamente en un estado s se asume que son falsos, es decir, que no se satisfacen en s.

Una acción $a \in A$ se define como $\langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$, donde $\text{pre}(a), \text{add}(a), \text{del}(a) \subseteq F$. El conjunto $\text{pre}(a)$ son las precondiciones de la acción a, es decir, el conjunto de literales que deben ser ciertos en un estado s para que la acción sea aplicable en s; $add(a)$ son los efectos positivos de la acción, esto es, el conjunto de literales que se afirman en el estado resultante tras la ejecución de a; y $del(a)$ son los efectos negativos, es decir, el conjunto de literales que se eliminan en el estado resultante tras la aplicación de la acción a. Formalmente, la aplicación de una acción a en un estado s genera un nuevo estado $s' = (s \setminus del(a)) \cup add(a)$.

Por ejemplo, para un problema del dominio del *mundo de bloques* donde se definen tres objetos de tipo bloque, `A`, `B` y `C`, el conjunto de literales F del problema sería el siguiente:

```
(ontable A)(ontable B)(ontable C)
(clear A)(clear B)(clear C)
(on A B)(on B A)(on A C)(on C A)(on B C)(on C B)
(holding A)(holding B)(holding C)
(handempty)
```

Dos ejemplos de acciones del conjunto A del problema se muestran a continuación, donde se puede observar los literales que conforman las precondiciones de las acciones, los literales que forman el conjunto $add(a)$ (literales que no llevan el constructor `not` delante), y los literales que forman el conjunto $del(a)$ (literales que llevan el constructor `not` delante del nombre del literal):

```
(:action pickup A
  :precondition (and (clear A)(ontable A)(handmepty))
  :effect (and (holding A) (not (clear A))(not (ontable A))(not (handempty)))
)
(:action stack A B
  :precondition (and (holding A)(clear B))
  :effect (and (on A B) (handempty)(not (holding A))(not (clear AB)))
)
```

Finalmente, se muestran dos posibles estados para un problema de planificación con tres bloques `A`, `B` y `C`. El primer estado representa una torre donde el bloque `B` está encima del bloque `A`, y el `A` encima de `C`. En el segundo estado, el bloque `A` está encima de la mesa, y el bloque `C` está encima de `B`.

```
(ontable C)(on A C)(on B A)(clear B)(handempty)

(ontable A)(ontable B)(clear A)(on C B)(clear C)(handempty)
```

Una solución para un problema de planificación $PB = \langle F, A, I, G \rangle$ es una secuencia de acciones o plan $\langle a_1, a_2, \ldots, a_n \rangle$, $a_i \in A$, tal que la aplicación de $\langle a_1, a_2, \ldots, a_n \rangle$ al estado inicial del problema I genera un estado s que satisface G, es decir, el estado s contiene todos los literales de G ($G \subseteq s$).

Los problemas de planificación se abordan principalmente desde una perspectiva de búsqueda en el espacio de estados. En esencia, un problema de planificación consiste en encontrar caminos o planes dentro de un espacio de búsqueda, siendo un criterio habitual de optimalidad minimizar el número de acciones del plan. La búsqueda suele estar guiada por funciones heurísticas que facilitan la operación computacionalmente. Nos limitamos a la planificación clásica, la cual considera solo acciones instantáneas sin duración, información perfecta y planificación secuencial.

2.2 Modelos neurosimbólicos

Los modelos neurosimbólicos combinan métodos simbólicos y neuronales para aprovechar las fortalezas de ambos. En planificación, estos modelos son capaces de representar y razonar sobre estados complejos. Los métodos simbólicos ofrecen una estructura clara y explícita para representar conocimientos y reglas, mientras que los modelos neuronales pueden gestionar grandes volúmenes de datos y aprender patrones complejos.

Los recientes avances en modelos neurosimbólicos han mejorado su capacidad para abordar problemas de planificación. Por ejemplo, las redes neuronales de grafos (GNN) [31] han demostrado ser muy efectivas, facilitando la transferencia de conocimiento entre distintas partes del grafo y mejorando la capacidad del modelo para generalizar a nuevos problemas.

Las GNN son modelos diseñados para trabajar con la estructura de datos de grafo, donde los vértices representan entidades y las aristas representan relaciones entre las mismas. El objetivo de las GNN es aprender representaciones vectoriales a nivel de grafo, vértice o arista por medio de propagar información contextual acorde con la conectividad del grafo.

En el caso de las representaciones a nivel de vértice, en cada iteración todos los vértices actualizan sus representaciones en función de la información que reciben de los vértices vecinos, este mecanismo recibe el nombre de propagación de mensajes (*message passing*).

También cabe destacar que, durante el proceso de propagación de mensajes, intervienen una serie de perceptrones multicapa (MLP) [18] para permitir a la GNN aprender como propagar la información. En conjunto, todo este mecanismo permite a la GNN identificar y aprender relaciones complejas sobre partes diferentes del grafo.

Actualmente, las GNNs continúan siendo uno de los mecanismos más utilizados en modelos neurosimbólicos. Uno de los algoritmos de GNN más destacados es el algoritmo de Hector Geffner [24], que se puede observar en la Figura 2.1 y que utiliza una representación de los estados en forma de grafo para poder generar una representación. El hecho de que estos grafos puedan llegar a tener un tamaño considerable y que el proceso de aprendizaje de las GNNs no sea altamente paralelizable, limita bastante la escalabilidad del algoritmo para problemas de gran envergadura.

Estos avances han creado nuevas oportunidades para aplicar modelos neurosimbólicos en tareas de planificación, mejorando tanto la eficiencia como la precisión de los algoritmos de planificación. No obstante, aún quedan desafíos por resolver, como la escalabilidad de estos modelos a problemas de gran tamaño.

Input: Relational struct. $\mathcal{R} = (\mathfrak{D}, R_1, \ldots, R_m)$ [states s)]
Output: $\boldsymbol{v} \in \mathbb{R}^q$ of dimension q [value $V(s)$]

// Partial random initialization
$\boldsymbol{s}_o^{(0)} \sim \mathbf{0}^{k/2}\mathcal{N}(0,1)^{k/2}$ for each object $o \in \mathfrak{D}$;
for $i \in \{1, \ldots, L\}$ **do**
  for *atom* $p := R(o_1, \ldots, o_n)$ *with* $\bar{o} \in R$ **do**
    // Generate messages $p \to o_j$
    $(\boldsymbol{m}_{p,o_j})_j := \mathbf{MLP}_R(\boldsymbol{s}_{o_1}^{(i-1)}, \ldots, \boldsymbol{s}_{o_n}^{(i-1)})$;
  for $o \in O$ **do**
    // Aggregate messages and update
    $\boldsymbol{s}_o^{(i)} := \mathbf{MLP}_U\left(\boldsymbol{s}_o^{(i-1)}, \mathbf{agg}(\{\!\!\{\boldsymbol{m}_{p,o} \mid o \in p\}\!\!\})\right)$;
// Final Readout
$\boldsymbol{v} := \mathbf{MLP}_2\left(\sum_{o\in\mathfrak{D}} \mathbf{MLP}_1(\boldsymbol{s}_o^L)\right)$

Figura 2.1: Algoritmo de Geffner que define una función paramétrica entrenable que permite transformar estructuras relacionales en un vector característico

2.3 Modelos de lenguaje

El objetivo de los modelos de lenguaje es aprender una distribución de probabilidad sobre un conjunto de elementos condicionada a una secuencia de elementos, normalmente un prefijo. Un ejemplo es, en el caso del lenguaje natural, aprender una distribución de probabilidad para la siguiente palabra dado un prefijo. Hemos puesto el ejemplo del lenguaje natural, pero esta misma mecánica se puede aplicar a cualquier tipo de datos que podamos representar como una secuencia ordenada de elementos.

Volviendo al caso del lenguaje natural, los modelos de lenguaje son muy útiles, por ejemplo, para tareas de generación de texto [11], donde estimar la probabilidad de la secuencia y la de los siguientes tókenes es clave para la fase de generación, ya que esta distribución, si está bien aprendida, va a determinar la calidad del texto generado. Existen múltiples formas de utilizar la distribución de probabilidad para generar las palabras, como un muestreo aleatorio sobre el conjunto de k palabras más probables o incluso una búsqueda en haz (*beam search*) [5, 23].

La forma más sencilla e intuitiva de implementar un modelo de lenguaje es contando las frecuencias de las palabras dadas cada posible n-grama previo, donde un n-grama se refiere a una secuencia de n palabras contiguas $(p_{i-n+1}, ..., p_i)$. Es decir, que este modelo de lenguaje se trataría de un mapa donde las claves son los diferentes n-gramas que se han encontrado en los datos y los valores son distribuciones de probabilidad sobre las diferentes palabras que se han contabilizado para cada n-grama. A pesar de ser intuitivo, rápidamente podemos encontrar inconvenientes con esta implementación, como los n-gramas o palabras que no encontramos en el conjunto de entrenamiento o la limitación en cuanto al tamaño de los n-gramas a considerar.

Por este motivo, actualmente se utilizan arquitecturas neuronales para implementar los modelos de lenguaje, los cuales no sufren los problemas mencionados anteriormente. Aún así tienen otro tipo de inconvenientes como un mayor coste computacional, tanto en entrenamiento como en inferencia, y también requieren más datos para ser entrenados.

Aunque en la práctica, en la actualidad existen formas de superar estos inconvenientes, ya que disponemos unidades de procesamiento gráfico (GPU, *graphics processing unit*) que pueden paralelizar operaciones de forma masiva (y los modelos neuronales se benefician mucho de esto [16]) y también disponemos de cantidades de información muy grandes (en cuanto al texto, podemos hablar de escala de internet).

Ahora bien, el objetivo de este trabajo es generar representaciones de estados en formato de texto para planificación automática, lo cual es también otra tarea en la que los modelos de lenguaje neuronales rinden muy bien.

El motivo de esto es que, de forma interna, estos modelos de lenguaje neuronales aprenden una representación vectorial para cada uno de los posibles elementos de las secuencias[1]. Estas representaciones vectoriales, tras entrenar el modelo, consiguen capturar muchas de las relaciones que se pueden encontrar, incluso semánticas, y pueden ser agregadas para generar una representación vectorial de la secuencia [10].

[1]Cuando hablamos de texto, nos referimos a estos elementos de la secuencia como tókenes, que son las unidades en las que se divide la secuencia y que el modelo toma como entrada y produce predicciones

CAPÍTULO 3
Diseño de la solución

En este capítulo, describiremos el diseño y la arquitectura del sistema desarrollado. Detallaremos la generación y el formato de los datos utilizados, y discutiremos la arquitectura del sistema, enfatizando el diseño del tokenizador personalizado y la configuración del modelo *transformer*.

3.1 Datos

Para comenzar, en esta sección vamos a comentar todo lo relativo a los datos utilizados en los experimentos que posteriormente se describirán. Pero antes de comenzar, mencionar que los dominios PDDL que se han tenido en cuenta son *blocksworld*, *logistics* y *floortile*, los cuales son dominios utilizados en la competición internacional de planificación (IPC) [26] y que también se pueden encontrar en los Apéndices A, B y C (respectivamente).

3.1.1. Formato de los datos

El objetivo del formato elegido es otorgar al modelo la posibilidad de aprender las relaciones entre estados y acciones, al contrario que si solo se hubiera entrenado con un único estado por cada muestra. Con esto en mente, las muestras que se han generado y utilizado para entrenar se tratan de tripletas de la forma $\langle estado, acción, estado \rangle$.

Como se ha comentado anteriormente, tenemos que generar estas muestras a modo de texto, así que hemos diseñado una plantilla que nos va a servir para informar al modelo sobre los diferentes elementos de la tripleta, la cual se puede encontrar en la Figura 3.1

```
<ST>estado-1<AC><PRE>precondicion-accion<EFF>efecto-accion<ST>estado-2
```

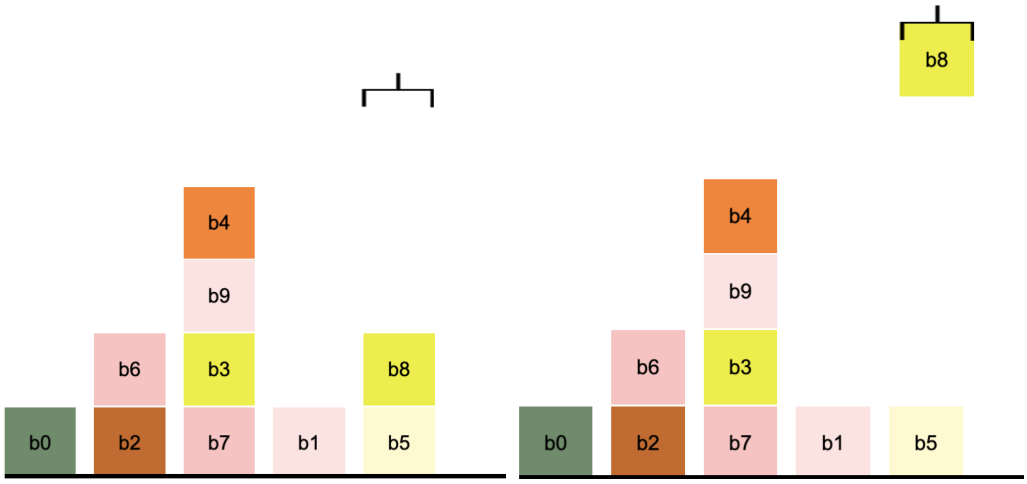
Figura 3.1: Plantilla utilizada para generar las muestras de datos en formato de tripleta

Para rellenar esta plantilla, únicamente tenemos que incluir las secuencias de literales que componen cada una de las partes: estado inicial, precondiciones y efectos de la acción, y estado resultante tras la aplicación de la acción en el estado inicial. En la Figura 3.2 se puede observar una muestra de ejemplo extraída del corpus del dominio *blocksworld*.

Los estados de la muestra responden a dos configuraciones de un problema de planificación con 10 bloques. El estado inicial s, la aplicación de la acción `unstack(b8,b5)` en el estado s, y el estado resultante s' de la muestra de la Figura 3.2 se muestran gráficamente en las Figuras 3.3a y 3.3b.

<ST>clear(b4),ontable(b0),clear(b1),ontable(b2),on(b9,b3),ontable(b7),on(b3,b7),on(b4,b9),clear(b0),clear(b6),ontable(b1),clear(b8),on(b8,b5),ontable(b5),on(b6,b2),handempty()<AC><PRE>on(b8,b5),clear(b8),handempty()<EFF>holding(b8),clear(b5),not-clear(b8),not-handempty(),not-on(b8,b5)<ST>clear(b4),ontable(b0),clear(b1),on(b6,b2),ontable(b2),clear(b5),on(b9,b3),ontable(b7),holding(b8),on(b3,b7),on(b4,b9),clear(b0),clear(b6),ontable(b1),ontable(b5)

Figura 3.2: Ejemplo de muestra de datos proveniente del dominio *blocksworld*



(a) Estado inicial s **(b)** Estado resultante s'

Figura 3.3: Representación visual de los estados presentes en la Figura 3.2

3.1.2. Generación de los datos

Para generar los datos que hemos utilizado, el proceso que se ha seguido es partir de un problema inicial y comenzar una exploración exhaustiva en anchura (*breadth first search*, BFS) [1], consiguiendo de esta forma generar todos los posibles caminos que tienen de origen el estado inicial del problema.

Para aquellos problemas de talla reducida o dominios de complejidad reducida, donde el número de acciones posibles en cada paso de exploración no es elevado, es decir, aquellos que tienen un número de objetos reducido, ha sido posible generar el espacio de búsqueda del problema planificación al completo, ya que el tamaño del espacio de búsqueda va a depender de la talla del problema.

Sin embargo, para otros problemas de mayor tamaño o un dominio con mayor complejidad, el número de posibles acciones en cada paso de exploración crece considerablemente, y generar el espacio de búsqueda completo se hace inviable. Así, para estos casos hemos tenido que introducir un límite de transiciones a generar por cada problema, el cual hemos configurado a 500 transiciones.

En cuanto al número de exploraciones llevado a cabo en cada dominio, es importante comentar que no se ha explorado el mismo número de problemas en todos ellos. Esto se ha hecho para contrarrestar el hecho de que algunos dominios tienen un abanico de acciones posibles mucho mayor en cada estado. Es por esto que hemos elegido el siguiente número de problemas iniciales: 250 para *blocksworld*, 350 para *logistics* y 150 para *floortile*.

Si se hubiera usado el mismo número de problemas iniciales, dependiendo del número de acciones posibles en cada estado y para un mismo límite de transiciones a generar, podría haber diferencias significativas en la profundidad alcanzada en los diferentes dominios y, por lo tanto, algunos dominios tendrían un conjunto de muestras insuficiente

considerando que tienen una complejidad mayor. Los valores elegidos son únicamente una aproximación elegida de forma manual.

Cabe mencionar que se podrían haber utilizado otras técnicas de exploración del espacio como, por ejemplo, una exploración aleatoria o *random-walk*, pero esto no nos habría proporcionado una visión exhaustiva y completa del espacio de búsqueda e incluso podría haber introducido algunos sesgos (*bias*) exploratorios en los datos.

Para ilustrar un posible problema con los sesgos, si el espacio de estados hubiera sido explorado mediante una exploración en profundidad (*depth first search*, DFS) [27], dependiendo de la acción elegida para generar el siguiente estado, podría darse el caso de que todas las muestras generadas hicieran referencia a la misma acción o a un subconjunto de las acciones disponibles, entorpeciendo así la capacidad de los modelos de aprender información completa sobre el dominio.

Finalmente, algo que queremos destacar es el reducido coste de generar muestras mediante este proceso, lo cual es de gran utilidad si en un futuro se desea extender este trabajo incrementando masivamente la escala de los datos (lo cual no es el objetivo del trabajo actual).

3.1.3. Análisis de los datos

Hemos realizado un análisis preliminar sobre los datos de entrenamiento para los tres diferentes dominios. Concretamente, se han tomado las muestras de entrenamiento y se ha construido un grafo con ellas, donde los vértices son los diferentes estados y las aristas representan la transición de un estado a otro por medio de una acción. Sobre ese grafo hemos tomado las siguientes medidas, las cuales podemos encontrar en la Tabla 3.1:

- el número de muestras únicas (o lo que es lo mismo, el número de transiciones únicas)
- el número de estados únicos
- el número de componentes conexas

Adicionalmente, se ha generado una gráfica para cada dominio donde se puede tener una visión general sobre la componente conexa con el mayor número de estados del grafo (para no saturar el gráfico). Cabe destacar que, cuando nos referimos a una componente conexa, nos referimos al subconjunto de vértices (y aristas que los conectan) del grafo tal que todas ellos tienen al menos un camino que los conecta.

El análisis de componentes conexas es importante porque nos puede dar una visión, por una parte, sobre como de aislados o separados estaban los diferentes problemas de exploración iniciales y, por otra, sobre el nivel de conexión del espacio de estados (por ejemplo, si desde cualquier estado se puede visitar cualquier otro o si hay grupos de estados ocultos detrás de unas transiciones muy especificas).

Dominio	Transiciones	Transiciones únicas	Estados únicos	Comp. conexas
Blocksworld	308.149	265.496	157.450	148
Logistics	1.391.097	1.384.280	387.495	416
Floortile	313.341	86.870	35.782	11

Tabla 3.1: Tabla con las medidas obtenidas tras analizar los datos de entrenamiento

En cuanto a las medidas presentes en la Tabla 3.1, un detalle que se repite en las medidas de los tres dominios es el hecho de que tanto el número de transiciones únicas como el número de estados son menores que el número total de transiciones.

En el caso de las transiciones únicas, el hecho de que en los datos generados tengamos algunas muestras duplicadas significa que diferentes procesos de exploración (que han tenido inicio en problemas diferentes) han llegado a generar las mismas transiciones, es decir, que a partir de cierto punto se han explorado los mismos caminos. Por ejemplo, en el caso del dominio *blocksworld*, esto se puede deber a que diferentes problemas iniciales compartían el mismo número de bloques, lo cual garantiza que todos esos estados iniciales están conectados por medio de al menos un camino.

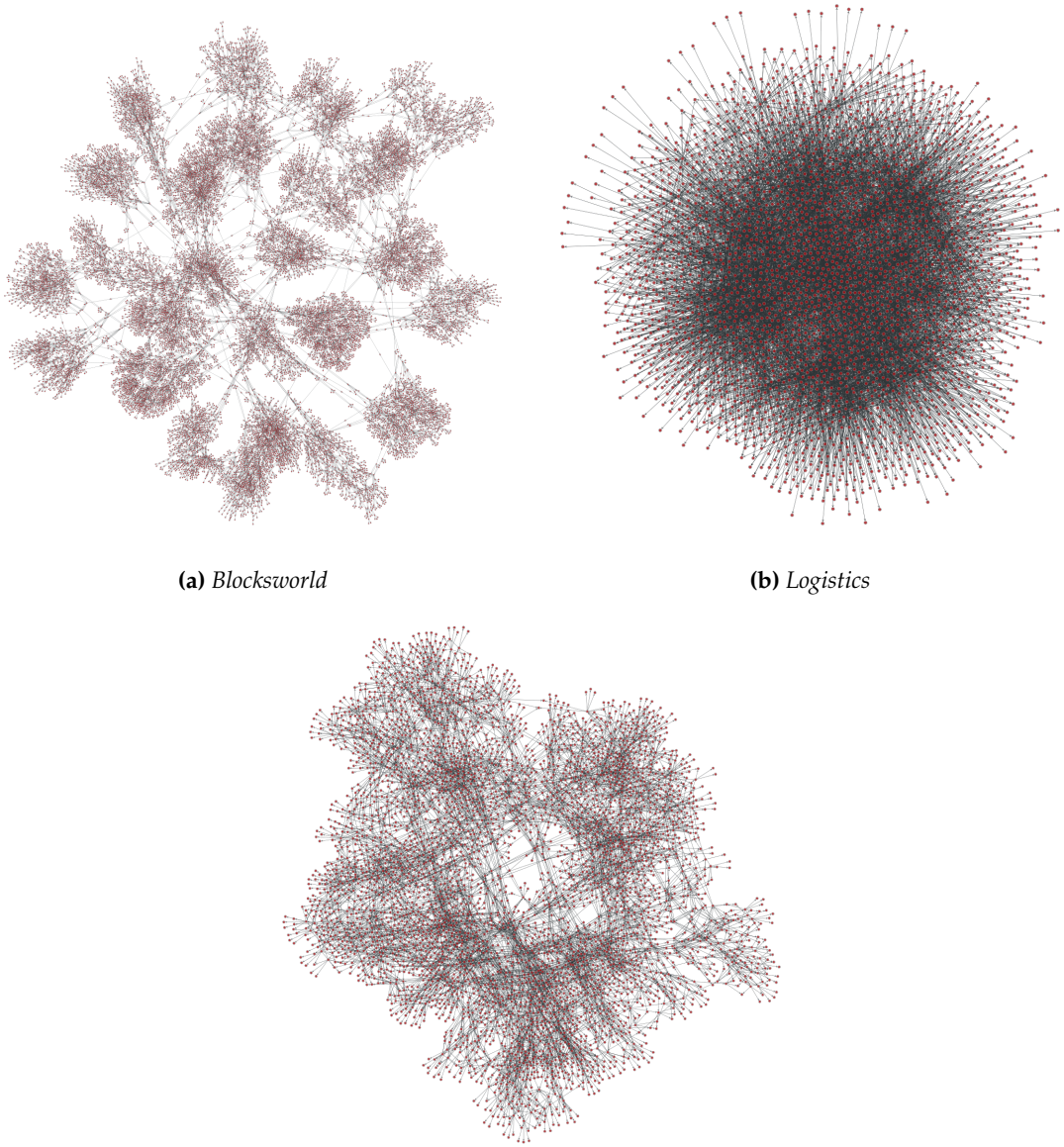
En el caso de los estados únicos, este hecho que acabamos de comentar también es una de las causas, pero en los estados hay una causa adicional. Como ya hemos explicado, las muestras que hemos generado se tratan de tripletas en la forma $\langle estado, acción, estado \rangle$ generadas a partir de una exploración en BFS. Lo que esto nos garantiza es que, para una muestra $\langle s, a, s' \rangle$, en muchos casos (ya que esto no se cumple para aquellas muestras al límite de la exploración BFS) también se habrá generado una muestra $\langle s', a', s'' \rangle$, la cual contiene un duplicado del estado s'. Es decir, a pesar de que cada muestra contiene dos estados, un conjunto de n muestras cuyas transiciones forman un camino contendrán un total de $n + 1$ estados únicos.

En la tabla 3.1 se puede observar que el número de muestras de los dominios es distinto, especialmente el dominio *logistics*, donde el número de muestras es significativamente mayor que en los conjuntos de datos de los otros dos dominios. Esto realmente no tiene ningún motivo técnico y es únicamente debido a que los dominios no se han explorado uniformemente, siendo el dominio *logistics* para el cual se han generado mas muestras, como ya hemos comentado en el apartado anterior.

Otra observación es el hecho de que en los datos de *floortile* existe un gran número de muestras que están duplicadas, como indica la gran diferencia entre el número de muestras (transiciones) y el número de transiciones únicas. Como comparación, el modelo *blocksworld* tiene aproximadamente el mismo número de muestras total que el dominio *floortile*, pero el número de muestras únicas también es mucho mayor.

Finalmente, otro dato llamativo es el número de componentes conexas presente en los datos del dominio *floortile*. Como indica la tabla, este dominio solo contiene 11 componentes conexas. Este reducido número de componentes conexas se puede enlazar bien con el comentario anterior sobre el número de muestras duplicadas. El hecho de que haya muestras duplicadas nos indica que muchas exploraciones en BFS con diferente puntos de partida acaban generando el mismo espacio de estados. Un análisis más detallado sobre las componentes conexas de los dominios se ofrece al final de la sección.

Finalizado este breve análisis sobre las medidas tomadas, ahora vamos a presentar las diferentes gráficas mencionadas al principio del apartado para cada uno de los dominios. En las Figuras 3.4a, 3.4b, y 3.4c se pueden encontrar las visualizaciones de la mayor componente conexa para cada uno de los dominios.



(a) *Blocksworld*

(b) *Logistics*

(c) *Floortile*

Figura 3.4: Visualización de los grafos formados por la componente conexa con mayor número de estados en las muestras de cada dominio

Anteriormente se comentaba la importancia del análisis del grafo de estados y su utilidad para conocer la conectividad del espacio de estados para un determinado dominio. En la Figura 3.4b se puede observar que los estados del dominio *logistics* se encuentran muy densamente conectados. Pero por otra parte, en la Figura 3.4a del dominio *blocksworld* (y también en menor medida en la Figura 3.4c para el dominio *floortile*), se puede ver como existen diferentes agrupaciones de estados las cuales están conectadas entre sí solamente por un número reducido de transiciones.

Esto se entiende muy bien al explicar algunas particularidades de cada dominio. En el caso del dominio *logistics*, el problema a resolver es que disponemos de un número de camiones, aviones, paquetes y ciudades, y el objetivo es conseguir que cada paquete llegue a una ciudad concreta. Con esto en mente, si tenemos 10 diferentes camiones y 10 diferentes aviones, de forma intuitiva ya podemos ver que hay muchas formas de llegar a un mismo estado. Y no solo eso, sino que además el orden de las acciones no es tan

relevante ni tiene tanto efecto sobre el estado, ya que, por ejemplo, la localización de uno de los camiones no afecta al resto de camiones ni aviones. Esto explica por que el grafo de *logistics* tiene una densidad tan alta de transiciones.

En cambio, la contraparte al dominio *logistics* es el dominio *blocksworld*, ya que en *blocksworld* la posición de un bloque concreto sí que puede afectar mucho a otros bloques. Intuitivamente, en una pila de bloques, la posición del más superior tiene un efecto sobre los bloques inferiores, los cuales no se pueden mover hasta que se retiren los bloques superiores. Esta relación tan fuerte entre los bloques de una misma pila explica lo que se ha comentado antes, que existen diferentes agrupaciones de estados las cuales solo se pueden conectar cuando un número de dependencias se resuelven.

Además de todo esto, también hay que considerar el posible número de acciones que se pueden tomar desde un estado. En *blocksworld*, en el mejor de los casos (cuando todos los bloques se encuentran sobre la tabla), únicamente se pueden tomar tantas acciones diferentes como bloques contenga el problema (una acción `pick-up` por cada bloque), pero esto no siempre se cumple ya que normalmente encontramos pilas de bloques. Pero por otra parte, en *logistics* siempre existe la opción de mover cualquiera de los camiones o aviones a cualquiera de las ciudades que lo permitan, el abanico de acciones posibles es mucho mayor.

Finalmente, también hay una particularidad en uno de los dominios, concretamente en *floortile*. En el resto de dominios, las acciones son siempre reversibles, en *blocksworld* se pueden coger y dejar bloques, y en *logistics* se pueden transportar los paquetes a una ciudad y de nuevo a la ciudad original, pero esto no ocurre en *floortile*. En este dominio, el objetivo es mover un robot en una cuadrícula y pintar las celdas de la cuadrícula formando un patrón concreto.

Así como las acciones de mover al robot sí que son reversibles, las acciones de pintar una celda no son reversibles. Esto hace que las acciones de pintar una celda, cuando son aplicadas, creen una barrera en el espacio de estados que no se puede superar, ya que no se puede volver a ningún estado donde la celda pintada no lo esté. Por último, otra de las implicaciones de las acciones no reversibles es que se pueden alcanzar estados que no tienen salida (*dead ends*), es decir, que desde estos estados no se puede aplicar ninguna acción, y esto también restringe mucho el número de posibles soluciones al problema.

3.2 Arquitectura del sistema

En esta sección, vamos a hablar de las dos piezas principales que componen el diseño del modelo que hemos utilizado en nuestros experimentos: el tokenizador y el modelo *transformer*.

3.2.1. Tokenizador

El tokenizador [8] es el componente que se encarga de transformar cadenas de texto plano a secuencias de números enteros (lo que también denominamos *token ids*), que es el tipo de entrada que los modelos *transformer* necesitan. Conceptualmente, un tokenizador divide las cadenas de texto en unidades de información más pequeñas que a continuación se pueden transformar en números enteros por medio de un índice (el cual también denominamos vocabulario).

Existen muchas estrategias para dividir las cadenas de texto. La más intuitiva es dividir la cadena de texto por los divisores de palabra (típicamente espacios en blanco) y tomar las palabras como la unidad de información.

La división de la cadena de texto inicial no depende en ningún sentido del vocabulario o índice (en el ejemplo de división de la cadena por palabra, la división simplemente se hace dividiendo por los divisores de palabra o espacios en blanco), sino que la dependencia es justo en el otro sentido. Para generar el vocabulario primero tenemos que tomar un conjunto de muestras de ejemplo, el vocabulario se construye tomando el conjunto de tókenes únicos que aparecen en las muestras divididas y asignando un identificador numerico arbitrario a cada uno.

El principal problema de esta estrategia intuitiva es que el tokenizador resultante es muy sensible a palabras (tókenes) que no pertenezcan al vocabulario, ya que el vocabulario se construye mediante un conjunto de muestras de ejemplo, pero durante inferencia puede ocurrir que aparezcan tókenes que no se hayan observado anteriormente.

Una forma de mitigar este problema es asignar un valor predefinido que represente a todos los tókenes desconocidos (típicamente `<UNK>` o similar), pero esto es solo una mitigación, ya que los *token ids* resultantes no estarían capturando toda la información que se encontraba en la cadena original.

Por este motivo, existen otras estrategias más avanzadas para dividir la cadena original en tókenes, con los objetivos de reducir al máximo el número de tókenes que se encuentren fuera del vocabulario y generar una representación densa con toda la información de la cadena original.

Un ejemplo de estrategia para dividir la cadena es la codificación de pares de bytes (*byte pair encoding* en inglés) [6], la cual fue diseñada originalmente como un algoritmo de compresión de datos, pero que también ha resultado ser bastante útil cuando se utiliza en un tokenizador. Arquitecturas de modelo como *GPT* [19] o *RoBERTa* [12] utilizan este tipo de tokenizador.

La necesidad de estas nuevas estrategias de tokenización deriva de que los datos que se utilizan tradicionalmente tienen mucha variabilidad, ya que pueden ser texto en lenguaje natural en una gran variedad de idiomas, texto que represente algún tipo de código fuente o incluso texto que contiene errores gramaticales. En estos casos es muy necesario ser lo más flexible posible mientras se mantiene una representación relevante y densa.

En nuestro caso, los datos con los que vamos a trabajar, como ya hemos comentado anteriormente, se tratan de cadenas de texto que se van a adherir a un formato muy concreto, que es aquel que el dominio PDDL va a definir. Por eso, para nuestro trabajo hemos desarrollado un nuevo tipo de tokenizador que considera algunas de las reglas sintácticas de PDDL para poder generar una representación mucho más densa sin perder nada de información.

En nuestro tokenizador, lo que hacemos es dividir las cadenas por los siguientes caracteres: , ( ) <>. Al dividir, para los caracteres < y > hacemos la concatenación de estos símbolos con el token posterior o anterior, respectivamente, de forma que mantenemos los tókenes de la estructura de la tripleta intactos. Pero en el caso del resto de caracteres, simplemente se eliminan. Por ejemplo, la tripleta que habíamos propuesto antes como ejemplo en la Figura 3.2 quedaría de la siguiente forma tras dividirla en tókenes:

```
[<ST>,clear,b4,ontable,b0,clear,b1,ontable,b2,on,b9,b3,ontable,b7,on,b3,b7,
on,b4,b9,clear,b0,clear,b6,ontable,b1,clear,b8,on,b8,b5,ontable,b5,on,b6,b2
,handempty,<AC>,<PRE>,on,b8,b5,clear,b8,handempty,<EFF>,holding,b8,clear,b5
,not-clear,b8,not-handempty,not-on,b8,b5,<ST>,clear,b4,ontable,b0,clear,b1,
on,b6,b2,ontable,b2,clear,b5,on,b9,b3,ontable,b7,holding,b8,on,b3,b7,on,b4,
b9,clear,b0,clear,b6,ontable,b1,ontable,b5]
```

Figura 3.5: Ejemplo de muestra de datos tokenizada proveniente del dominio *blocksworld*

Como se puede observar en la Figura 3.5, tras dividir una de nuestras muestras utilizando nuestro propio tokenizador, conseguimos una representación mucho más densa de la información, ya que pasamos de 450 caracteres en la cadena original a 91 tókenes en la cadena tokenizada, mientras que al mismo tiempo también conseguimos mantener la misma cantidad de información.

3.2.2. Modelo *transformer*

Es posible implementar un modelo de lenguaje por medio de diferentes arquitecturas neuronales, pero para este trabajo hemos elegido la arquitectura *transformer* porque tiene claras ventajas frente a otras arquitecturas.

Otra arquitectura tradicional que se puede utilizar para implementar un modelo de lenguaje son las redes neuronales recurrentes (RNN) [22], pero esta arquitectura tiene dos grandes inconvenientes. Por una parte, los modelos RNN tienen una capacidad muy limitada de identificar y mantener relaciones entre tókenes muy separados dentro de la secuencia de entrada. Esto es debido a que las RNNs procesan los tókenes de entrada de forma secuencial, así que a medida que aumenta la longitud de la secuencia de entrada, el modelo pierde capacidad para almacenar apropiadamente la información según avanza en la inferencia. Además, el procesamiento de la secuencia de entrada de forma secuencial también limita la eficiencia del modelo, ya que la predicción del token en posición n depende de la predicción del token en la posición $n-1$. Esto es un factor muy limitante en entrenamiento, debido a que nos fuerza a hacer múltiples llamadas a la función del modelo (usualmente la función `forward`) para que el modelo visite todos los elementos de la secuencia de entrada.

En cualquier caso, los modelos basados en la arquitectura *transformer* introducen varios mecanismos que alivian los inconvenientes descritos. La principal ventaja de esta arquitectura es que es capaz de procesar la secuencia de entrada (los *token ids*) entera y de forma simultanea en paralelo. Esto se consigue a través de las dos siguientes técnicas:

1. La primera técnica es la codificación posicional (*positional encodings*), la cual es necesaria porque el modelo procesa la secuencia al completo, así que esto es una forma de informar al modelo sobre el orden de los tókenes. Esto permite al modelo conservar el contexto del orden de entrada.

2. El segundo mecanismo son las capas de atención (*attention*). En una RNN, la predicción de un token depende de un subconjunto de los tókenes generados anteriormente y de la información que almacene en las celdas de memoria, lo cual limita la visión hacia atrás que tiene el modelo al predecir. En cambio, las capas de atención sirven para que el modelo sea capaz de analizar todos los tókenes generados anteriormente y pueda asignar un peso a cada uno de ellos en función de como de relevante es para el token actual.

De esta forma, un modelo *transformer* es capaz de considerar cualquier token previo de la secuencia sin importar como de distante sea del token actual. Por ejemplo, en una oración larga, puede llegar a considerar un token o palabra al principio de la oración como importante a la hora de generar un token que aparezca por el final.

Gracias a esta forma de procesar la información y a los nuevos mecanismos que se introducen, los modelos basados en arquitectura *transformer* se posicionan como unos de los mejores candidatos cuando se espera que las secuencias de entrada sean extensas, pero además permiten ser paralelizados a un mayor nivel, lo que mejora la eficiencia no solo en entrenamiento, sino que también en inferencia.

Además, este tipo de modelos neuronales representan el conocimiento, como ya mencionamos anteriormente, manteniendo de forma interna una representación vectorial para cada uno de los tókenes que se encuentran en el vocabulario (a estas representaciones también se les denomina *embeddings*).

Al mismo tiempo, estas representaciones se encuentran incorporadas en la función de perdida del modelo de tal forma que, al entrenar el modelo, también se ajustan los *embeddings* como parte del algoritmo de optimización.

Al final del proceso de entrenamiento, estas representaciones han demostrado capturar bastante bien los significados y relaciones entre los diferentes tókenes, así que son muy útiles a la hora de crear representaciones densas de secuencias de tókenes.

Para este trabajo, hemos elegido una arquitectura que es una variante de *BERT* (*Bidirectional Encoder Representations from Transformers*) [2], nos hemos decantado por *RoBERTa* (*Robustly Optimized BERT Pretraining Approach*). Todos los modelos que mencionaremos más adelante han sido generados a partir de un modelo *RoBERTa* con los pesos inicializados de forma aleatoria. En el capítulo 4 comentaremos las diferentes configuraciones que se han explorado.

CAPÍTULO 4
Pruebas experimentales

En este capítulo presentamos los experimentos realizados para evaluar el rendimiento de los modelos entrenados. Detallaremos las métricas de evaluación utilizadas, así como los resultados obtenidos para cada modelo entrenado en los distintos dominios de planificación. También discutiremos las configuraciones de hardware y software utilizadas, el ajuste de hiperparámetros y las metodologías de evaluación.

4.1 Evaluación

En esta sección vamos a describir cuales han sido los procedimientos utilizados para evaluar los diferentes modelos que hemos entrenado a lo largo de la experimentación.

4.1.1. Métricas de entrenamiento

Existen principalmente dos alternativas a la hora de llevar a cabo una tarea de preentrenamiento sobre un modelo *transformer*: modelado de lenguaje causal (*causal language modeling*, CLM) [15] y modelado de lenguaje por enmascaramiento (*masked language modeling*, MLM) [15].

Cuando se entrena un modelo por CLM, la función de perdida se calcula en base a como de bien el modelo es capaz de, dado un prefijo de tókenes, predecir el siguiente token. Este tipo de entrenamiento resulta muy útil cuando el modelo se va a utilizar en una tarea de generación de texto, ya que el modelo no tiene información sobre los tókenes futuros y tiene que predecir únicamente en base a un prefijo. Un ejemplo de modelo que usa este tipo de entrenamiento es *GPT-2* [20].

Por otra parte, cuando se entrena un modelo por MLM, la función de perdida se calcula en base a como de bien el modelo es capaz de, dada una muestra de entrenamiento con algunos de los tókenes enmascarados con un valor común (normalmente `<MASK>`), predecir los tókenes correctos que deberían de ir en el lugar de los tókenes de mascara. Las muestras de entrada se enmascaran en función a un parámetro que controla la probabilidad de cada token de que sea enmascarado, lo que se traduce en una ampliación práctica del número de muestras, ya que para una misma muestra de entrada existen muchas posibles formas de generar una máscara para la misma.

El entrenamiento por MLM es muy útil para aquellas tareas donde el modelo tiene que tener una buena comprensión contextual de una secuencia completa, ya que al tener visibilidad completa sobre la muestra (exceptuando la máscara), el modelo aprende mejor las relaciones contextuales de la muestra completa. Dos modelos que utilizan este tipo de modelado del lenguaje son *BERT* y *RoBERTa*.

Debido a que en este trabajo nuestro objetivo es ser capaces de generar representaciones vectoriales densas de cadenas de entrada, resulta apropiado utilizar MLM como la estrategia de modelado a utilizar. Y, como ya habíamos dicho, *RoBERTa* es uno de los modelos que se entrenan muy bien por MLM, así que lo hemos elegido como modelo base para todos los modelos que hemos entrenado.

Durante el entrenamiento por MLM, se va a utilizar la función de perdida de entropía cruzada (*cross entropy*) [13] la cual se define con la ecuación que aparece en la Figura 4.1 y nos sirve para medir la diferencia entre dos distribuciones de probabilidad.

En la ecuación, M representa el número de clases del problema de clasificación que, en el caso de los modelos de lenguaje, es el conjunto de tókenes posibles (los tókenes que el vocabulario contiene), $y_{o,c}$ es un indicador binario que representa si la clase c es la predicción correcta para la observación o, y $p_{o,c}$ es la probabilidad que da el modelo para la clase c dada la observación o.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Figura 4.1: Ecuación que define la función de perdida de entropía cruzada para la tarea de clasificación multi-clase

En MLM, la predicción de qué token colocar en cada uno de los espacios de la máscara se puede tratar como un problema de clasificación multi-clase, donde el valor de referencia es el token original antes de aplicar la máscara y las diferentes clases son el conjunto de posibles tókenes. Es por eso que en la Figura 4.1 hemos elegido la variante de entropía cruzada para clasificación multi-clase.

4.1.2. Métricas de modelado del lenguaje

Cuando se habla de métricas de modelado del lenguaje, una de las métricas más clásicas es de la perplejidad (PPL) o *perplexity* [9], y es por eso que también la hemos utilizado en este trabajo como medida de la calidad de los modelos. De forma intuitiva, la perplejidad mide cómo de acertado y con cuanta confianza el modelo es capaz de predecir correctamente el siguiente token dado un prefijo de tókenes. El rango de la medida va desde 1 (incluido) hasta infinito, siendo 1 el mejor valor posible donde el modelo es capaz de predecir el siguiente token con un acierto del 100 % y una confianza del 100 %.

Formalmente, la perplejidad se define con la ecuación que aparece en la Figura 4.2. En la misma, X representa una secuencia de tókenes, t es la longitud de la secuencia X, y $p_\theta(x_i|x_{<i})$ representa la probabilidad que da el modelo al token x_i dado el prefijo $x_{<i}$.

$$PPL(X) = exp\left\{ -\frac{1}{t} \sum_{i}^{t} \log p_\theta(x_i|x_{<i}) \right\}$$

Figura 4.2: Ecuación que define la medida de perplejidad

Además de la métrica de perplejidad y la métrica de entrenamiento descrita en el apartado anterior, también hemos implementado una nueva métrica para evaluar las representaciones vectoriales, la hemos denominado métrica de consistencia.

Cuando en la literatura se intenta ilustrar como los modelos de lenguaje y representaciones vectoriales pueden capturar información semántica del lenguaje natural, en muchas ocasiones se propone un ejemplo donde, si el modelo es capaz de identificar esas relaciones, se cumple que $emb(rey) - emb(chico) + emb(chica) = emb(reina)$, donde $emb(x)$ se refiere a la representación vectorial de la palabra x. Si dado un lenguaje natural se pudieran extraer todas las posibles operaciones e igualdades que deberían de ser verdaderas, esto nos podría servir para evaluar como de consistente es el modelo en representar estas relaciones.

Lo que en este trabajo hemos hecho es aplicar la misma idea a las muestras, donde tenemos un conjunto de transiciones de un estado inicial (s) a otro resultante (s') junto con la acción que facilita la transición de uno a otro (a). Utilizando la misma nomenclatura que en el ejemplo anterior, lo que evaluamos en nuestro caso es que $emb(s) + emb(a) = emb(s')$, de nuevo, donde $emb(x)$ se refiere a la representación vectorial del estado o acción x.

Un inconveniente que nos encontramos en ambos ejemplos es que, sin importar cómo de preciso sea el modelo capturando las relaciones semánticas o de contexto, es muy poco probable que consigamos una igualdad exacta entre el vector resultante del lado izquierdo de la igualdad y el vector esperado, ya que estamos hablando de representaciones vectoriales de alta dimensionalidad y con valores numéricos decimales.

Para superar este problema, calculamos la distancia (en nuestro caso hemos usado la distancia coseno) entre los vectores del lado izquierdo y derecho de la igualdad y comprobamos que esta distancia es menor que la distancia entre el lado izquierdo de la igualdad y la representación de un estado aleatorio. Extendiendo esta idea, podemos comparar contra un número mayor de estados aleatorios y calcular un porcentaje de acierto.

El razonamiento detrás de esta métrica es que, si el modelo es realmente capaz de aprender las relaciones entre estados y acciones, el resultado de esta métrica sobre los modelos entrenados debería de ser mayor que si la aplicamos sobre un modelo de referencia o *baseline*.

4.1.3. Cobertura de problemas en la tarea de planificación

Adicionalmente a estas métricas, también hemos tomado la mejor de las configuraciones para cada modelo y hemos evaluado el modelo resultante al ser aplicado en una serie de tareas de planificación. Para el algoritmo de planificación, se ha escogido un algoritmo de aprendizaje por refuerzo (*reinforcement learning*, RL) [25], específicamente *Proximal Policy Optimization* (PPO) [21], el cual puede consumir como entrada las codificaciones vectoriales densas que nuestros modelos generan.

El objetivo del algoritmo PPO es, dado un problema concreto, conseguir aprender una política que consiga resolver el problema, donde la política es una red neuronal que decide que acción tomar en cada paso de la planificación. De esta forma, cuando el entrenamiento de la política por PPO finaliza, lo que hemos hecho es evaluar si la política entrenada para cada problema era capaz de encontrar una secuencia de acciones que lleven a una solución. Si la solución se alcanza dentro de un número límite de pasos, se considera que el problema está cubierto, y se considera que no está cubierto si no se llega a la solución dentro del límite de pasos.

Concretamente, para cada modelo hemos tratado de resolver una batería de 20 problemas y hemos calculado la cobertura sobre los mismos, es decir, que porcentaje de problemas han podido ser resueltos con respecto al total que se ha tratado de resolver. Para evaluar el modelo entrenado con los tres dominios de forma conjunta, en lugar de

la batería de 20 problemas de un único dominio, hemos juntado las tres baterías para formar una combinada de 60 problemas.

También hemos realizado los mismos experimentos en la tarea de planificación mediante el uso de una representación *one-hot* y hemos utilizado los valores de cobertura obtenidos como *baseline*. Al igual que ocurría con el modelo entrenado con todos los dominios, la representación *one-hot* también se ha evaluado con la batería conjunta de 60 problemas.

4.2 Experimentos

A continuación, en esta sección vamos a describir la metodología seguida durante la experimentación, describiendo el entorno de experimentación que hemos utilizado y los diferentes modelos e hiperparámetros que hemos explorado.

4.2.1. Hardware utilizado

Para llevar a cabo los experimentos que se van a describir más adelante, hemos utilizado equipamiento especial proporcionado por el departamento. Específicamente, el equipamiento se trata de un servidor con acceso remoto en el cual se ejecutaba el sistema operativo Ubuntu 22.04 LTS y el cual disponía de una CPU Intel(R) Core(TM) i9-12900KF de 12ª Generación, 33 GB de memoria RAM y una GPU Nvidia GeForce RTX 3090.

Tener acceso a un servidor con GPU ha facilitado mucho la tarea en cuanto al tiempo que los experimentos han tardado en ser completados (incluyendo entrenamiento y otras tareas de evaluación), ya que hemos trabajado con modelos neuronales los cuales pueden ser acelerados significativamente por medio de una GPU moderna.

4.2.2. Software utilizado

En cuanto al software utilizado, cabe mencionar que todo el código implementado se ha escrito en *Python* [28], ya que es uno de los lenguajes con mayor soporte de librerías para inteligencia artificial y mayor soporte de comunidad.

Las principales librerías que hemos utilizado son *PyTorch* [17], *Transformers* [30] y *graph-tool*, además de otras librerías que ya van incorporadas en las instalaciones base de *Python*.

También cabe destacar que se ha utilizado otro software de código abierto (*open-source* en inglés) para el manejo de los experimentos: *MLflow* [32]. Éste ha sido una pieza clave para poder almacenar y ordenar los experimentos realizados, así como también almacenar todas las métricas generadas en cada uno de los pasos de entrenamiento y evaluación.

4.2.3. Modelos entrenados

Como ya hemos comentado en el capítulo anterior, disponemos de datos para tres conocidos dominios de planificación. Para cada uno de estos dominios se han entrenado diferentes modelos de lenguaje, de tal forma que cada modelo individual tiene conocimiento solamente de uno de los dominios.

Adicionalmente, también hemos entrenado una serie de modelos mediante el uso de muestras extraídas de todos los dominios, con el objetivo de que estos modelos tengan

información y sean capaces de representar estados de cualquiera de los tres dominios de forma indistinta.

Cabe mencionar que los datos utilizados para los experimentos son el conjunto de transiciones únicas generadas para cada dominio, conjuntos que se han dividido en tres partes, del 80 %, 10 % y 10 %, las cuales se han usado para entrenamiento, validación y evaluación (respectivamente).

4.2.4. Ajuste de hiperparámetros

Para cada uno de los dominios que hemos considerado y para el conjunto de datos con muestras combinadas, hemos realizado una búsqueda exhaustiva en cuadrícula con tal de optimizar una serie de hiperparámetros. El objetivo de esta metodología es poder encontrar la combinación de hiperparámetros que nos sirva para obtener el mejor modelo posible.

Una cosa que cabe destacar es que, al tratarse de una búsqueda exhaustiva y al tener que entrenar un modelo para cada posible combinación de hiperparámetros, el número de hiperparámetros y el conjunto de posibles valores para cada uno de ellos va a afectar drásticamente al tiempo requerido para completar la exploración, así que hemos tratado de mantener el número de posibles combinaciones a un número manejable. A continuación, en la tabla 4.1 se pueden observar los diferentes hiperparámetros explorados y sus posibles valores.

Hiperpárametro	Valores
`mlm_probability`	$\{0,15,\ 0,05\}$
`num_attention_heads`	$\{6,\ 12\}$
`num_hidden_layers`	$\{5,\ 8\}$

Tabla 4.1: Tabla con hiperparámetros explorados

A continuación vamos a proporcionar más detalles acerca de cada uno de los hiperparámetros:

- `mlm_probability` se encarga de controlar la probabilidad de que un token sea enmascarado durante el entrenamiento MLM. Valores mayores de este hiperparámetro implican que la secuencia de entrada se va a enmascarar en más medida, lo cual puede hacer que el modelo tarde más en aprender las relaciones pero que llegue a generalizar mejor. También cabe mencionar que este hiperparámetro no tiene ningún efecto sobre el tamaño del modelo, solo afecta a la parte de procesamiento de las muestras antes de ser pasadas al modelo.

- `num_attention_heads` controla el número de cabezales que va a tener el modelo *transformer* en cada una de sus capas de atención. Incrementar este valor puede mejorar la capacidad del modelo de aprender relaciones contextuales entre los diferentes tókenes, pero a su vez también aumenta el número de parámetros del modelo.

- `num_hidden_layers` controla el número de capas ocultas que va a tener el modelo *transformer*. Básicamente, a mayor número de capas ocultas, más profundidad va a tener la red neuronal resultante, lo que va a incrementar la capacidad del modelo de identificar y aprender patrones complejos de los datos, pero, igual que ocurría con `num_attention_heads`, incrementar el valor de este hiperparámetro también va a incrementar el número de parámetros del modelo.

Estos hiperparámetros y valores nos generan un espacio de 8 posibles combinaciones, las cuales tenemos que aplicar a cada uno de los conjuntos de datos que queremos utilizar, es decir, a cada uno de los dominios y al conjunto compuesto de datos de todos los dominios, con lo que al final hemos entrenado un total de 32 modelos.

4.3 Resultados

Finalmente, en esta sección vamos a mostrar y describir los resultados que hemos obtenido tras realizar la experimentación descrita en la sección anterior. Vamos a dividir esta presentación de resultados por el dominio utilizado y vamos a presentar el mismo conjunto de métricas y visualizaciones para todos ellos. Finalmente, también presentaremos los valores de cobertura de problemas obtenidos en los experimentos realizados sobre la tarea de planificación.

4.3.1. Modelo *blocksworld*

En cuanto a los resultados obtenidos para el dominio *blocksworld*, en la tabla 4.2 se pueden encontrar los identificadores de cada modelo junto con la configuración y las métricas obtenidas en cada uno.

	Parametros			Metricas		
model-id	mlm-prob	att-heads	hidden	train-loss	test-loss	perplexity
valuable-stoat-645	0,150	6	8	0,369	0,025	1,025
zealous-slug-492	0,150	12	5	0,491	0,033	1,033
brawny-lynx-893	0,150	12	8	0,479	0,033	1,033
suave-dolphin-275	0,150	6	5	0,531	0,035	1,036
sincere-mare-808	0,050	12	5	0,713	0,105	1,111
merciful-wasp-101	0,050	6	5	0,852	0,107	1,113
rare-ray-342	0,050	6	8	0,752	0,123	1,131
defiant-snipe-538	0,050	12	8	0,772	0,127	1,136
random-baseline	-	-	-	-	10,489	35907,376

Tabla 4.2: Tabla de resultados para el dominio *blocksworld*

Como se puede observar en los resultados, las métricas obtenidas con los modelos entrenados son significativamente mejores y la perplejidad obtenida en los modelos entrenados es bastante cercana a 1, que se trata del mejor valor que se puede obtener. Por otro lado, entre todas las configuraciones evaluadas, realmente no hay indicaciones estadísticamente significativas como para decir que una configuración es mejor que otra, para poder encontrar estas diferencias probablemente necesitaríamos extender de forma significativa el espacio de posibles configuraciones evaluadas.

Como se mencionó anteriormente en el apartado 4.1.2, también hemos evaluado los modelos en una nueva métrica que hemos denominado consistencia. En la gráfica de la Figura 4.3 se puede observar como varia el valor de la métrica en función del número de estados aleatorios contra los que se comparan.

Como se esperaba, todos los modelos (incluyendo la *baseline*) tienen obtienen un resultado similar sobre la métrica de consistencia cuando el número de estados aleatorios a comparar es bajo. Sin embargo, se pueden observar dos tendencias claramente diferenciadas entre los modelos entrenados y la *baseline* según se aumenta el número de estados

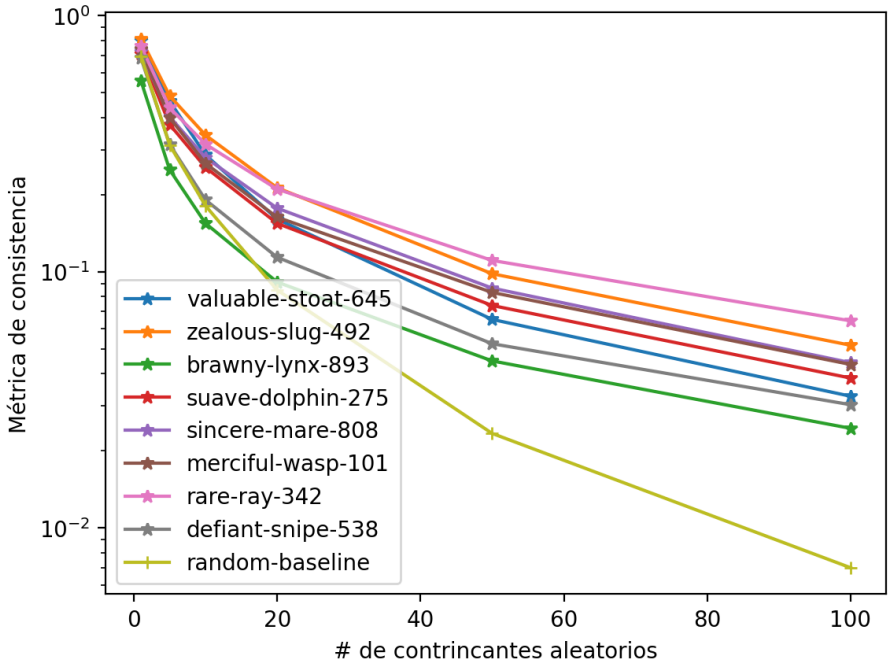


Figura 4.3: Gráfica mostrando la métrica de consistencia (en eje y) en función del número de estados contrincantes aleatorios (en eje x) para los modelos de *blocksworld*

aleatorios a comparar: la calidad del modelo *baseline* se reduce drásticamente con el aumento de estados aleatorios, mientras que los modelos entrenados ven una disminución de la calidad más ligera.

4.3.2. Modelo *logistics*

En cuanto a los resultados obtenidos para el dominio *logistics*, en la tabla 4.3 se pueden encontrar los identificadores de cada modelo junto con la configuración y las métricas obtenidas en cada uno.

	Parametros			Metricas		
model-id	mlm-prob	att-heads	hidden	train-loss	test-loss	perplexity
brawny-hawk-363	0,150	12	8	0,372	0,006	1,006
able-pig-880	0,150	12	5	0,452	0,012	1,012
bedecked-mink-814	0,050	6	8	0,708	0,015	1,015
luminous-perch-679	0,050	12	8	0,595	0,016	1,016
upbeat-cod-430	0,150	6	8	0,526	0,016	1,016
glamorous-goat-169	0,150	6	5	0,629	0,020	1,020
thundering-crab-892	0,050	6	5	0,642	0,021	1,021
orderly-fly-264	0,050	12	5	0,631	0,022	1,023
random-baseline	-	-	-	-	10,592	39804,827

Tabla 4.3: Tabla de resultados para el dominio *logistics*

En cuanto a la interpretación de los resultados obtenidos para estos modelos, las observaciones que podemos hacer son las mismas que para los modelos de *blocksworld*. Aun así, un detalle importante a destacar sobre los experimentos de *logistics* es que las métri-

cas obtenidas son ligeramente mejores cuando las comparamos con los experimentos de otros dominios.

Esto puede deberse a que, como comentamos en el apartado 3.1.3, la cantidad de datos disponibles para el dominio de *logistics* era significativamente mayor y la uniformidad de los mismos también era mayor. Esto ha podido llevar a que estos modelos hayan sido capaces de aprender mejor las cualidades y relaciones que se pueden encontrar dentro de las muestras del dominio.

A continuación, al igual que con los otros experimentos, en la gráfica de la Figura 4.4 se puede observar como varía el valor de la métrica en función del número de estados aleatorios que se comparan.

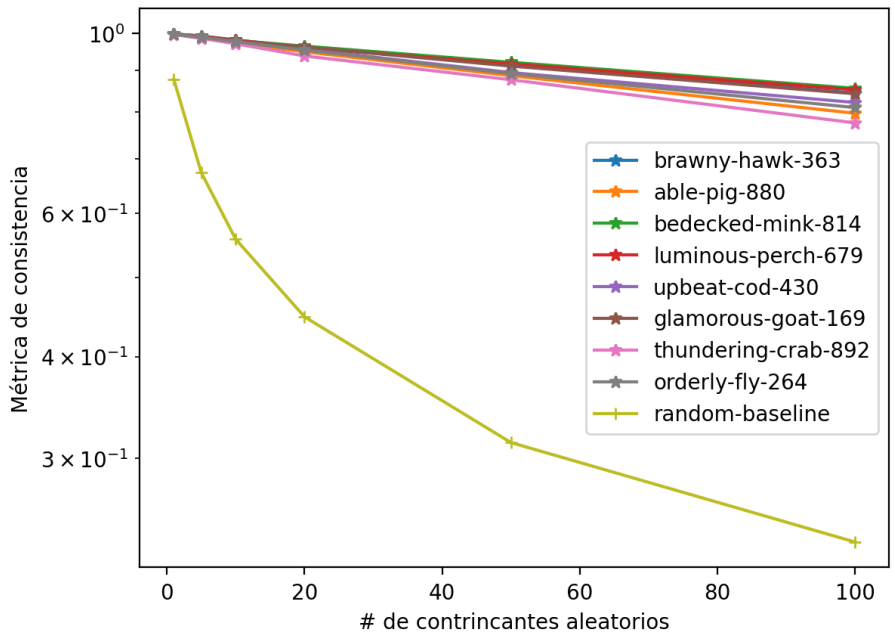


Figura 4.4: Gráfica mostrando la métrica de consistencia (en eje y) en función del número de estados contrincantes aleatorios (en eje x) para los modelos de *logistics*

De nuevo y de la misma forma que ocurre en el anterior experimento, todos los modelos (incluyendo la *baseline*) tienen obtienen un resultado similar sobre la métrica de consistencia cuando el número de estados aleatorios a comparar es bajo. Sin embargo, se pueden observar dos tendencias claramente diferenciadas entre los modelos entrenados y la *baseline* según se aumenta el número de estados aleatorios a comparar: la calidad del modelo *baseline* se reduce drásticamente con el aumento de estados aleatorios, mientras que los modelos entrenados ven una disminución de la calidad más ligera.

En cuanto a la interpretación de la gráfica, de nuevo, se pueden observar las mismas dos tendencias que mencionamos en los resultados de los experimentos con *blocksworld*. La única peculiaridad es que la diferencia entre la tendencia de los modelos entrenados y la tendencia de la *baseline* está mucho más pronunciada para este dominio, al igual que ocurría con las métricas de la Tabla 4.4, lo cual también puede haber sido causado por la mayor cantidad y calidad de las muestras de datos.

4.3.3. Modelo *floortile*

En cuanto a los resultados obtenidos para el dominio *floortile*, en la tabla 4.4 se pueden encontrar los identificadores de cada modelo junto con las configuración y las métricas obtenidas en cada uno.

	Parametros			Metricas		
model-id	mlm-prob	att-heads	hidden	train-loss	test-loss	perplexity
righteous-stoat-331	0,150	12	5	0,292	0,011	1,012
sedate-hound-123	0,150	6	8	0,305	0,013	1,013
intrigued-midge-815	0,150	12	8	0,408	0,013	1,013
salty-stag-647	0,150	6	5	0,419	0,017	1,017
nervous-lamb-968	0,050	12	8	0,434	0,022	1,022
bustling-crab-661	0,050	6	8	0,595	0,027	1,028
burly-cub-692	0,050	6	5	0,739	0,035	1,036
thundering-toad-844	0,050	12	5	0,616	0,036	1,036
random-baseline	-	-	-	-	10,251	28310,310

Tabla 4.4: Tabla de resultados para el dominio *floortile*

En cuanto a la interpretación de los resultados obtenidos para estos modelos, las observaciones que podemos hacer son exactamente las mismas que para los modelos de *blocksworld*. A continuación, al igual que con los otros experimentos, en la gráfica de la Figura 4.5 se puede observar como varía el valor de la métrica en función del número de estados aleatorios que se comparan.

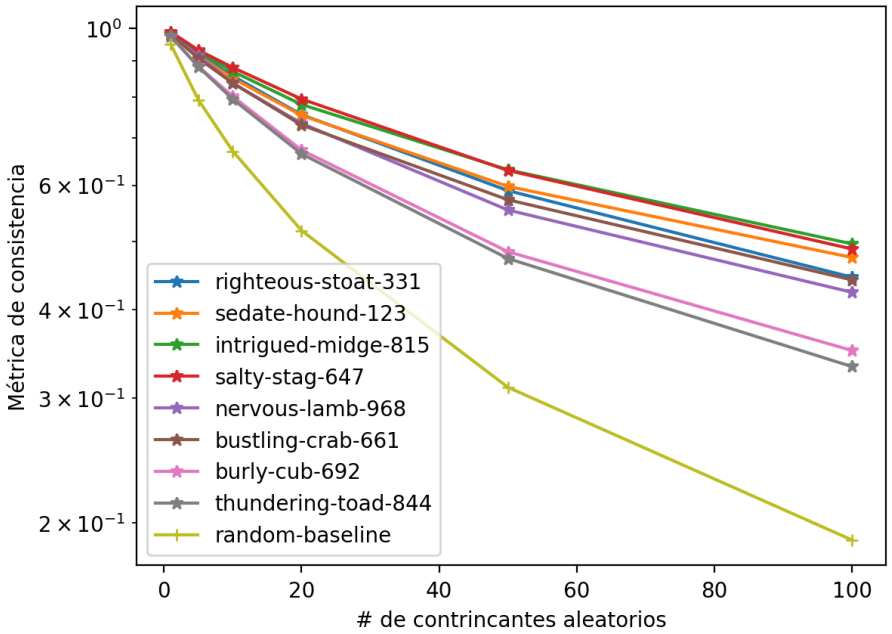


Figura 4.5: Gráfica mostrando la métrica de consistencia (en eje y) en función del número de estados contrincantes aleatorios (en eje x) para los modelos de *floortile*

La interpretación de esta gráfica es la misma que para los otros experimentos: se pueden observar dos tendencias claramente diferenciadas entre los modelos entrenado y la *baseline*.

4.3.4. Modelo multi-dominio

En cuanto a los resultados obtenidos para los modelos multi-dominio, en la tabla 4.5 se pueden encontrar los identificadores de cada modelo junto con las configuración y las métricas obtenidas en cada uno.

	Parametros			Metricas		
model-id	mlm-prob	att-heads	hidden	train-loss	test-loss	perplexity
hilarious-rat-632	0,050	12	8	0,636	0,024	1,024
auspicious-trout-843	0,150	12	8	0,639	0,024	1,024
gifted-deer-312	0,150	6	5	0,515	0,027	1,027
unruly-snake-708	0,150	6	8	0,626	0,028	1,028
traveling-hen-73	0,150	12	5	0,510	0,032	1,033
placid-goose-407	0,050	6	5	0,641	0,033	1,034
sassy-colt-215	0,050	6	8	0,898	0,045	1,046
efficient-donkey-36	0,050	12	5	0,791	0,045	1,046
random-baseline	-	-	-	-	10,544	37936,238

Tabla 4.5: Tabla de resultados para el modelo multi-dominio

En cuanto a la interpretación de los resultados obtenidos para estos modelos, las observaciones que podemos hacer son exactamente las mismas que para los modelos de *blocksworld*. A continuación, al igual que con los otros experimentos, en la gráfica de la Figura 4.6 se puede observar como varía el valor de la métrica en función del número de estados aleatorios que se comparan.

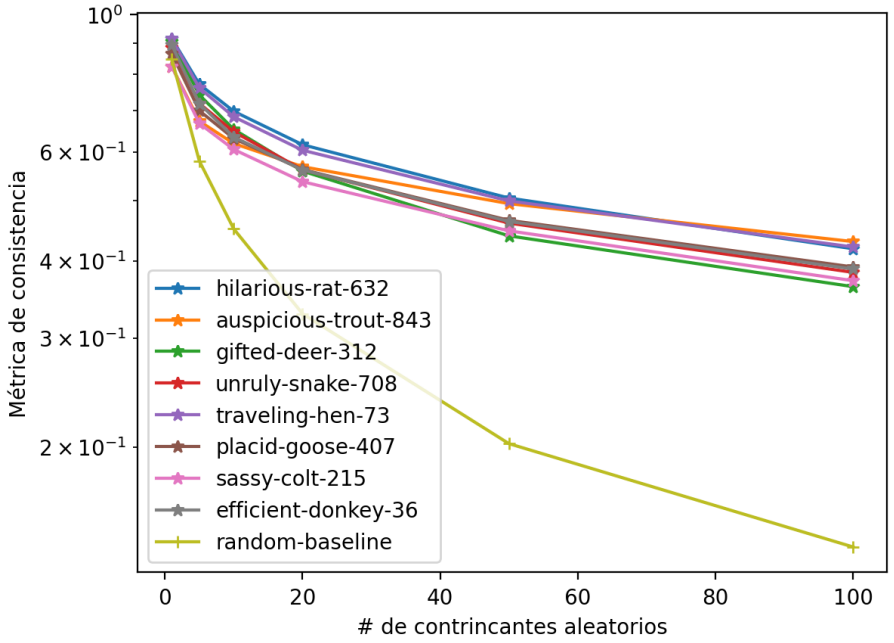


Figura 4.6: Gráfica mostrando la métrica de consistencia (en eje y) en función del número de estados contrincantes aleatorios (en eje x) para los modelos con dominios combinados

La interpretación de esta gráfica es la misma que para los otros experimentos: se pueden observar dos tendencias claramente diferenciadas entre los modelos entrenado y la *baseline*. Aunque un detalle a destacar sobre este experimento es que, al igual que ocurría en el experimento de *logistics*, este experimento con las muestras combinadas de todos los dominios también ha podido verse beneficiado por la mayor cantidad y calidad de los datos disponibles para *logistics*, ya que de nuevo se puede observar esa mayor diferencia en la tendencia de los modelos entrenados y la *baseline*.

4.3.5. Cobertura de problemas en la tarea de planificación

Como ya comentamos en la sección 4.1.3, también hemos realizado una evaluación de los mejores modelos de cada categoría sobre la tarea de planificación. Hemos elegido estas configuraciones porque han obtenido los mejores valores para la métrica de perplejidad dentro de sus respectivas categorías, son: *valuable-stoat-645* para *blocksworld*, *brawny-hawk-363* para *logistics*, *righteous-stoat-331* para *floortile* y *hilarious-rat-632* para la categoría de modelos de dominio conjunto. A continuación, en la Tabla 4.6 se pueden encontrar los resultados de esta evaluación.

	Dominios		
	Blocksworld	*Logistics*	*Floortile*
codificación *one-hot*	95 %	95 %	55 %
modelos individuales	95 %	100 %	50 %
modelo conjunto	90 %	100 %	50 %

Tabla 4.6: Tabla de resultados de cobertura de problemas

Como se puede observar en la Tabla 4.6, las coberturas de las diferentes representaciones evaluadas son bastante similares y se podría decir que no hay ninguna diferencia estadísticamente significativa entre ellas, al menos en cuanto a la cobertura.

A pesar de que no hay ninguna diferencia en la cobertura, cabe mencionar que las representaciones densas que se extraen de los modelos entrenados en este trabajo tiene otra serie de ventajas frente a la codificación *one-hot* que no hemos podido llegar a medir y comparar. Por ejemplo, la ventaja principal de los modelos aquí entrenados es que las representaciones densas tienen una dimensionalidad fija, pero las representaciones en codificación *one-hot* crecen en tamaño significativamente según el número de literales y su complejidad aumenta.

4.3.6. Interpretación de resultados

En términos generales, se puede decir que los experimentos han sido un éxito, ya que se ha conseguido alcanzar el objetivo propuesto inicialmente: pre-entrenar una serie de modelos *transformer* con el objetivo de que aprendan a representar información simbólica en representaciones vectoriales densas.

No solo eso, sino que también hemos confirmado que las representaciones generadas se pueden utilizar en un proceso de planificación y alcanzar una cobertura de problemas equivalente a la que se alcanza con una representación *one-hot*, mientras que al mismo tiempo se mantiene una representación vectorial densa de dimensionalidad fija.

CAPÍTULO 5
Conclusiones

En el capítulo final, resumiremos las limitaciones o desafíos que nos hemos encontrado durante el desarrollo de este trabajo y presentaremos algunas de las posibles líneas de trabajo futuro, sugiriendo posibles mejoras y nuevas aplicaciones de los modelos de lenguaje en planificación automática.

5.1 Limitaciones

En cuanto a limitaciones que hemos encontrado en este trabajo, la más clara era el tiempo necesario para entrenar cada uno de los modelos. En nuestros experimentos, teníamos una política de entrenamiento de parada temprana, la cual finalizaba el entrenamiento de los modelos si las métricas de validación empeoraban durante algunos pasos de evaluación seguidos (el número de pasos durante los cuales las métricas de validación tienen que empeorar se le suele denominar paciencia), por lo que es posible que los modelos todavía tengan capacidad de seguir aprendiendo.

Aún con esto, cada modelo puede tardar alrededor de 2 o 3 horas para finalizar el entrenamiento, motivo por el cual hemos tratado de mantener el número de configuraciones evaluadas a un número reducido. Con una mayor cantidad de tiempo disponible y una mayor cantidad de hardware, sería posible extender ese conjunto de configuraciones de hiperparámetros y encontrar modelos con métricas aún mejores, lo cual es ideal.

Otra limitación o desafío que hemos encontrado es la falta de literatura en el ámbito de la evaluación de representaciones vectoriales dada una estructura discreta de referencia, la cual en nuestro caso era el grafo generado a partir del espacio de estados. Para superar este desafío, hemos diseñado e implementado la métrica de consistencia, pero lo ideal habría sido disponer de una métrica utilizada extensivamente en la literatura, de la misma forma que lo es la perplejidad.

5.2 Trabajo futuro

Como se menciona al inicio del documento, hemos alcanzado el objetivo del trabajo mediante una tarea de pre-entrenamiento, pero una posible línea de trabajo futuro podría ser evaluar si se puede hacer un entrenamiento específico a una tarea (también denominado *fine-tuning*) con el objetivo de poder incorporar estos modelos en otras partes del proceso de planificación. Por ejemplo, se podría hacer un *fine-tuning* de los modelos para que sean capaces de aprender como de lejos se encuentra un estado de otro, con el objetivo de usar estos modelos como heurísticas en el proceso de planificación.

Otra línea de trabajo podría ser realizar el mismo ejercicio que hemos realizado en este trabajo pero con modelos de tamaño mucho mayor, con lo que se denominan actualmente *large language models* (LLM). Utilizar esta nueva clase de modelos requeriría de muchos más datos de entrenamiento (dependiendo de la estrategia a seguir para el entrenamiento) ya que contienen un número de parámetros mucho mayor, pero al mismo tiempo permitirían al modelo aprender una mayor cantidad de información relativa a los dominios y con una complejidad mayor, llegando incluso a poder extender la idea presentada en este trabajo del modelo de dominios combinados y entrenar el modelo con tantos dominios como sea posible recopilar.

Bibliografía

[1] Alan Bundy and Lincoln Wallen. *Breadth-First Search*, pages 13–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[3] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[4] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[5] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017.

[6] Matthias Gallé. Investigating the effectiveness of BPE: The power of shorter sequences. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1375–1381, Hong Kong, China, November 2019. Association for Computational Linguistics.

[7] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2004.

[8] Gregory Grefenstette. *Tokenization*, pages 117–133. Springer Netherlands, Dordrecht, 1999.

[9] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63, 08 2005.

[10] Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. On the sentence embeddings from pre-trained language models, 2020.

[11] Junyi Li, Tianyi Tang, Wayne Xin Zhao, and Ji-Rong Wen. Pretrained language models for text generation: A survey, 2021.

[12] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[13] Anqi Mao, Mehryar Mohri, and Yutao Zhong. Cross-entropy loss functions: Theoretical analysis and applications, 2023.

[14] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language, 1998.

[15] Nicolo Micheletti, Samuel Belkadi, Lifeng Han, and Goran Nenadic. Exploration of masked and causal language modelling for text generation, 2024.

[16] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[18] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[19] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[20] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[22] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, March 2020.

[23] Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. A thorough examination of decoding methods in the era of llms, 2024.

[24] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. *CoRR*, abs/2109.10129, 2021.

[25] C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis lectures on artificial intelligence and machine learning. Morgan & Claypool, 2010.

[26] Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, et al. The 2023 international planning competition, 2024.

[27] Robert Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, 1971.

[28] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[30] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021.

[32] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

APÉNDICE A
Dominio *blocksworld*

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 4 op-blocks world
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
         (ontable ?x)
         (clear ?x)
         (handempty)
         (holding ?x)
         )

    (:action pick-up
        :parameters (?x)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
        (and (not (ontable ?x))
        (not (clear ?x))
        (not (handempty))
        (holding ?x)))

    (:action put-down
        :parameters (?x)
        :precondition (and (holding ?x))
        :effect
        (and (not (holding ?x))
        (clear ?x)
        (handempty)
        (ontable ?x)))

    (:action stack
        :parameters (?x ?y)
        :precondition (and (holding ?x) (clear ?y))
        :effect
        (and (not (holding ?x))
        (not (clear ?y))
        (clear ?x)
        (handempty)
        (on ?x ?y)))

    (:action unstack
        :parameters (?x ?y)
        :precondition (and (on ?x ?y) (clear ?x) (handempty))
        :effect
        (and (holding ?x)
        (clear ?y)
        (not (clear ?x))
```

```
49          (not (handempty))
50          (not (on ?x ?y)))))
```

APÉNDICE B
Dominio *logistics*

```
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates   (OBJ ?obj)
          (TRUCK ?truck)
                 (LOCATION ?loc)
    (AIRPLANE ?airplane)
                 (CITY ?city)
                 (AIRPORT ?airport)
    (at ?obj ?loc)
    (in ?obj1 ?obj2)
    (in-city ?obj ?city))

  ; (:types )   ; default object

(:action LOAD-TRUCK
  :parameters
   (?obj
    ?truck
    ?loc)
  :precondition
   (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
   (at ?truck ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?truck)))

(:action LOAD-AIRPLANE
  :parameters
   (?obj
    ?airplane
    ?loc)
  :precondition
   (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
   (at ?obj ?loc) (at ?airplane ?loc))
  :effect
   (and (not (at ?obj ?loc)) (in ?obj ?airplane)))

(:action UNLOAD-TRUCK
  :parameters
   (?obj
    ?truck
    ?loc)
  :precondition
   (and (OBJ ?obj) (TRUCK ?truck) (LOCATION ?loc)
        (at ?truck ?loc) (in ?obj ?truck))
  :effect
   (and (not (in ?obj ?truck)) (at ?obj ?loc)))

(:action UNLOAD-AIRPLANE
```

```
  :parameters
   (?obj
    ?airplane
    ?loc)
  :precondition
   (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
        (in ?obj ?airplane) (at ?airplane ?loc))
  :effect
   (and (not (in ?obj ?airplane)) (at ?obj ?loc)))

(:action DRIVE-TRUCK
  :parameters
   (?truck
    ?loc-from
    ?loc-to
    ?city)
  :precondition
   (and (TRUCK ?truck) (LOCATION ?loc-from) (LOCATION ?loc-to) (CITY ?city)
   (at ?truck ?loc-from)
   (in-city ?loc-from ?city)
   (in-city ?loc-to ?city))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

(:action FLY-AIRPLANE
  :parameters
   (?airplane
    ?loc-from
    ?loc-to)
  :precondition
   (and (AIRPLANE ?airplane) (AIRPORT ?loc-from) (AIRPORT ?loc-to)
  (at ?airplane ?loc-from))
  :effect
   (and (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to)))
)
```

APÉNDICE C
Dominio *floortile*

```
;;Created by Tomas de la Rosa
;;Domain for painting floor tiles with two colors

(define (domain floor-tile)
(:requirements :typing)
(:types robot tile color - object)

(:predicates
    (robot-at ?r - robot ?x - tile)
    (up ?x - tile ?y - tile)
    (down ?x - tile ?y - tile)
    (right ?x - tile ?y - tile)
    (left ?x - tile ?y - tile)

    (clear ?x - tile)
                (painted ?x - tile ?c - color)
    (robot-has ?r - robot ?c - color)
                (available-color ?c - color)
                (free-color ?r - robot)
                )



(:action change-color
  :parameters (?r - robot ?c - color ?c2 - color)
  :precondition (and (robot-has ?r ?c) (available-color ?c2))
  :effect (and (not (robot-has ?r ?c)) (robot-has ?r ?c2)
               )
)


(:action paint-up
  :parameters (?r - robot ?y - tile ?x - tile ?c - color)
  :precondition (and (robot-has ?r ?c) (robot-at ?r ?x) (up ?y ?x) (clear ?y))
  :effect (and (not (clear ?y)) (painted ?y ?c)
               )
)


(:action paint-down
  :parameters (?r - robot ?y - tile ?x - tile ?c - color)
  :precondition (and (robot-has ?r ?c) (robot-at ?r ?x) (down ?y ?x) (clear ?y)
      )
  :effect (and (not (clear ?y)) (painted ?y ?c)
)
)


```

```
; Robot movements
(:action m_up
  :parameters (?r - robot ?x - tile ?y - tile)
  :precondition (and (robot-at ?r ?x) (up ?y ?x) (clear ?y))
  :effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
               (clear ?x) (not (clear ?y))
               )
)


(:action m_down
  :parameters (?r - robot ?x - tile ?y - tile)
  :precondition (and (robot-at ?r ?x) (down ?y ?x) (clear ?y))
  :effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
               (clear ?x) (not (clear ?y))
               )
)

(:action m_right
  :parameters (?r - robot ?x - tile ?y - tile)
  :precondition (and (robot-at ?r ?x) (right ?y ?x) (clear ?y))
  :effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
               (clear ?x) (not (clear ?y))
         )
)

(:action m_left
  :parameters (?r - robot ?x - tile ?y - tile)
  :precondition (and (robot-at ?r ?x) (left ?y ?x) (clear ?y))
  :effect (and (robot-at ?r ?y) (not (robot-at ?r ?x))
               (clear ?x) (not (clear ?y))
               )
)

)
```